CodeIgniter 4.6.3 Tutorial: The Ultimate Comprehensive Guide to Mastering the Framework (Final Enhanced Edition)

Prerequisites: PHP 8.1+ (8.2+ recommended for better type safety and performance), Composer, a web server (Apache/Nginx/Docker), and basic PHP/OOP knowledge. For databases, set up MySQL, PostgreSQL, or SQLite. Use VS Code with Intelephense for CI-specific autocompletion.

Version Confirmation: As of October 16, 2025, 4.6.3 is the latest stable, with enhancements in validation rules, DB failover, and security headers.

1. Introduction to CodeIgniter

Detailed Explanation: CodeIgniter (CI) is an open-source, lightweight PHP framework designed for rapid development of dynamic web applications and APIs. It strictly follows the Model-View-Controller (MVC) architectural pattern to separate concerns: **Models** handle data access and business logic (e.g., database queries, validation rules), **Views** manage presentation (HTML templates, avoiding script logic to keep them clean), and **Controllers** act as the glue (processing HTTP requests, interacting with models, and rendering views or returning responses). Why lightweight? The core footprint is under 2MB, with minimal dependencies, making it faster and easier to deploy than bloated frameworks like Laravel or Symfony—ideal for shared hosting, prototypes, or high-performance needs. Version 4.x overhauled CI3 with modern features: namespaces for better organization, Composer for package management, PSR compliance for interoperability, and support for PHP 8+. Specifically, 4.6.3 includes bug fixes for Query Builder edge cases, improved session handling in clustered environments, and enhanced CSP (Content Security Policy) for security. Under the hood, CI uses lazy loading (configs/services load on demand) for speed, achieving 1000+ requests/second in benchmarks. Common pitfalls: Misunderstanding MVC leads to bloated controllers—always delegate data to models. Performance impact: Minimal abstraction layers mean low overhead, but custom extensions can add if not careful.

Why Choose CI4?: It's developer-friendly with zero-configuration defaults, built-in security features (automatic XSS escaping, CSRF tokens), and extensibility via events/filters without requiring a full ORM or queue system (add via packages for modularity). Great for CRUD apps, RESTful APIs, or simple sites like blogs/contact managers.

Examples:

- **Basic Use Case (MVC Flow)**: A user requests `/contacts` → Controller fetches user-specific contacts from Model → Model queries DB → Controller passes data to View → View renders HTML list.
- **Intermediate Scenario**: Building an API endpoint where Controller validates input, Model processes data with transactions, and response is JSON.
- **Advanced Real-World**: A multi-module e-commerce app with 'cart' module (own MVC) for reusable checkout logic, integrated with external APIs via HTTP Client library.

Best Practices: Embrace MVC for scalability—models for logic, controllers slim (under 100 lines/method), views pure. Start with the welcome page as a template. Follow PSR-12 coding style for readability.

2. Installation and Setup

Detailed Explanation: Installation leverages Composer for dependency resolution (e.g., fetching Laminas Escaper for security, PHPUnit for testing). The `appstarter` template provides a bare-bones structure for production; `framework` includes demo code for learning. Post-install, `.env` file overrides config classes for environment-specific settings (e.g., DB creds in prod vs. dev). Why .env? It keeps secrets out of version control and allows toggling behaviors like error display (detailed in dev, hidden in prod). Pitfalls: Incompatible PHP version (below 8.1) causes fatal errors; missing extensions like `intl` (for locales) or `mbstring` (for strings) break features—verify with `php -m`. For clean URLs, configure server rewrites to route through `public/index.php`. Performance: Composer autoloader optimizes in prod (`--optimize-autoloader`).

Steps with Examples:

1. **Basic Composer Install** (For a new project):

   ```
   composer create-project codeigniter4/appstarter myapp --prefer-dist
   cd myapp
   ```

   This downloads 4.6.3, sets up folders, and installs core deps. Why `--prefer-dist`? Faster for production (pre-built archives).

2. **Configure .env** (For dev/prod switching): Copy `env` to `.env` and edit:

   ```
   # General
   app.baseURL = 'http://localhost:8080'
   CI_ENVIRONMENT = development  # Shows errors; change to 'production' for logging only

   # Database (MySQL example)
   database.default.hostname = localhost
   database.default.database = myapp_db
   database.default.username = root
   database.default.password =
   database.default.DBDriver = MySQLi
   ```

   Why? Overrides `Database.php` without modifying code.

3. **Run the Built-in Server** (For quick testing):

```
php spark serve --port 8080
```

   Access `http://localhost:8080`—see the default welcome page. Pitfall: This is dev-only; use Apache/Nginx for prod.

4. **Intermediate: Manual Install from Git** (For customizing core):

```
git clone https://github.com/codeigniter4/CodeIgniter4.git myapp
cd myapp
composer install
```

   Useful for forking/contributing.

5. **Advanced: Docker Containerization** (For portable, reproducible envs): Create `Dockerfile`:

```
FROM php:8.2-apache
RUN apt-get update && apt-get install -y libzip-dev unzip libicu-dev libonig-dev
RUN docker-php-ext-install pdo_mysql zip intl mbstring
COPY . /var/www/html
RUN chown -R www-data:www-data /var/www/html/writable /var/www/html/public
```

   `docker-compose.yml` (with DB):

```
version: '3'
services:
  app:
    build: .
    ports:
      - "8080:80"
    volumes:
      - .:/var/www/html
    depends_on:
      - db
  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: myapp_db
    ports:
      - "3306:3306"
```

   Run: `docker-compose up -d`. Access app at `localhost:8080`. Why? Isolates env, easy scaling.

Best Practices: Always use `.env` for secrets. Run `php spark env:check` to validate setup. For upgrades: `composer update codeigniter4/framework --with-dependencies`. Add `.htaccess` for Apache:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php/$1 [L]
```

3. Project Structure and Modules

Detailed Explanation: CI's directory layout promotes organization and security: `public/` is the only exposed folder (web root), preventing direct access to app code. `app/` contains your custom MVC elements; `system/` is the immutable framework core. Namespaces (e.g., `App\Controllers`) enable PSR-4 autoloading, avoiding class name collisions in large projects. Modules extend this with Hierarchical MVC (HMVC)—self-contained units (e.g., an 'auth' module with its own routes/controllers/models/views) for modularity. Why modules? They allow reusing features across apps or separating concerns (e.g., admin panel as a module). Under the hood, modules register via autoload PSR-4 mappings. Pitfall: Forgetting to register namespaces causes "Class not found" errors—use `php spark namespaces:list` to debug. Performance: Modules add slight overhead (extra autoloading), but negligible for most apps.

Key Folders with Examples:

- **app/Config/**: Holds config classes like `App.php` (base URL, session settings). Example: Edit `Routes.php` for URL mappings.
- **app/Controllers/**: Request handlers. Example: Create `app/Controllers/Contact.php` for `/contacts` logic.
- **app/Models/**: Data layers. Example: `ContactModel.php` for DB ops on contacts table.
- **app/Views/**: Templates. Example: `contacts/index.php` for listing UI.
- **app/Helpers/**: Functions like `format_phone()`. Example: Custom helper for phone masking.
- **app/Libraries/**: Classes like `MyEmailer`. Example: Extend Email library for custom SMTP.
- **app/Filters/**: Middleware like auth checks.

- **public/**: `index.php` (entry point), `assets/` (CSS/JS/images). Example: Place `style.css` here for public access.
- **writable/**: Runtime files (logs, cache, sessions, uploads). Example: Sessions stored here in file driver—secure with .htaccess deny.
- **tests/**: PHPUnit files. Example: `unit/ContactModelTest.php`.
- **vendor/**: Composer deps—ignore in Git.
- **system/**: Framework—don't modify; override via services.

Modules Example (Basic to Advanced):

- **Basic**: Create `app/Modules/Auth/Controllers/Login.php`.
- **Intermediate Registration**: In `app/Config/Autoload.php`:

  ```
  public array $psr4 = [
      'Modules\Auth' => APPPATH . 'Modules/Auth',
  ];
  ```

  Add module routes in `Modules/Auth/Config/Routes.php`:

  ```
  $routes->get('login', 'Login::index');
  ```

  Why? Auto-discovers if `Feature.php` `$discoverIn` includes modules.
- **Advanced**: A 'payment' module with own DB config override, models for transactions, and filters for API keys.

Best Practices: Set web root to `public/` in server config for security. Use modules for >5 features to keep `app/` clean. Git ignore writable/vendor.

4. Configuration in Depth

Detailed Explanation: Configurations are PHP classes in `app/Config/`, allowing OOP features like inheritance and methods (e.g., dynamic values). `.env` parses first for overrides, enabling per-environment tweaks without code changes. Why classes? Type-safe, extendable (e.g., custom `Database` subclass). Lazy loading means they're instantiated only when accessed, saving memory/perf. Environments (`development/production/testing`) alter behavior: Dev shows errors/stack traces; prod logs them silently. Pitfall: Typo in .env ignores override—debug with `php spark config:show App`. Performance: Minimal, but large configs (e.g., many routes) can slow startup—cache in prod.

Key Files with Examples (Basic to Advanced):

- **App.php** (Core settings):

  ```
  namespace Config;

  class App extends \CodeIgniter\Config\BaseConfig {
      public string $baseURL = 'http://localhost:8080/';  // For absolute URLs
      public string $indexPage = '';  // Empty for clean URLs (requires rewrites)
      public bool $CSPEnabled = true;  // Enables Content Security Policy headers
      public array $CSP = [ 'default-src' => "'self'"];  // Whitelist sources to prevent XSS
      public string $defaultLocale = 'en';  // For i18n
      public string $sessionDriver = 'CodeIgniter\Session\Handlers\FileHandler';  // File/Database/Redis
      public int $sessionExpiration = 7200;  // 2 hours
  }
  ```

  Why CSP? Blocks unauthorized scripts; add nonces for inline JS.

- **Database.php** (Connections):

  ```
  namespace Config;

  class Database extends \CodeIgniter\Config\BaseConfig {
      public array $default = [
          'DSN'      => '',  // e.g., 'mysql:host=localhost;dbname=myapp'
          'hostname' => 'localhost',
          'username' => 'root',
          'password' => '',
          'database' => 'myapp_db',
          'DBDriver' => 'MySQLi',  // Alternatives: Postgre, SQLite3, SQLSRV
          'DBPrefix' => 'ci_',
          'pConnect' => false,  // Persistent connections for perf in high-load
          'DBDebug'  => (ENVIRONMENT !== 'production'),  // Show queries/errors in dev
          'cacheOn'  => false,  // Enable query caching
          'failover' => [  // Backup connection
              ['hostname' => 'backup.host', 'database' => 'myapp_db'],
          ],
      ];
  }
  ```

Why failover? Automatic switch on primary failure for HA.

- **Autoload.php** (Pre-loads):

```
namespace Config;

class Autoload extends \CodeIgniter\Config\BaseConfig {
    public array $psr4 = ['App' => APPPATH];  // Namespace mappings
    public array $helpers = ['url', 'form', 'cookie'];  // Auto-load for global access
    public array $libraries = ['session'];  // Pre-instantiate
}
```

Why auto-load? Convenience, but avoid overload for perf.

- **Filters.php** (Middleware):

```
public array $aliases = ['auth' => \App\Filters\AuthFilter::class];
public array $globals = [
    'before' => ['csrf' => ['except' => 'api/*']],  // Global CSRF except APIs
];
```

- **Validation.php** (Rulesets):

```
public array $contact = [
    'name'  => 'required|min_length[3]|alpha_space',
    'email' => 'required|valid_email|is_unique[contacts.email]',
];
```

Why rulesets? Reusable validation groups.

Advanced Custom Config: Create `app/Config/MyApp.php`:

```
class MyApp extends BaseConfig {
    public string $apiKey = 'secret';  // Access via config('MyApp')->apiKey
}
```

Override in .env: `MyApp.apiKey = override`.

Best Practices: Use .env for all secrets/prod changes. Extend configs for app-specific logic. Run `php spark config:validate` to check syntax.

5. Routing: From Basics to Advanced

Detailed Explanation: Routes in `app/Config/Routes.php` define how URLs map to controllers/methods, supporting HTTP verbs (GET/POST/PUT/DELETE) for RESTful design. Placeholders capture dynamic segments (e.g., IDs); regex for custom matching. Groups organize with shared prefixes/namespaces/filters. Closures for quick prototypes without controllers. Why verbs? Enforces API standards (e.g., POST for create). Under the hood, routes are compiled/cached for speed. Pitfall: Ambiguous routes (e.g., overlapping params) cause wrong matching—order from specific to general. Performance: Many routes slow parsing; use resources/groups to consolidate.

Examples (Basic to Advanced):

- **Basic Static Route**:

```
$routes->get('/', 'Home::index');  // Homepage calls Home controller's index method
```

Why? Default entry point.

- **Intermediate with Parameters and Options**:

```
$routes->get('contact/(:num)', 'Contact::show/$1', ['as' => 'contact.show']);  // /contact/42 → show(42); named for redirect()->route('contact.show', [42])
$routes->get('search/(:alpha)/page/(:num)', 'Search::index/$1/$2', ['filter' => 'auth']);  // Alpha for letters only; auth filter requires login
$routes->match(['get', 'post'], 'form', 'Form::handle');  // Handles both verbs
```

Why params? Dynamic URLs like user profiles.

- **Advanced Resources and Groups**:

```
$routes->resource('contacts', ['controller' => 'Contact', 'placeholder' => '(:num)', 'websafe' => true]);  // Auto-generates 7 CRUD routes: index, show, new, create, edit, update, delete; websafe escapes params
$routes->group('api/v1', ['namespace' => 'App\Api', 'filter' => 'rate:5,60'], function ($routes) {  // Namespace for organization, filter for 5 req/min throttling
```

```
    $routes->post('auth/login', 'Auth::login', ['as' => 'api.login']);
});
```

Why resources? DRY code for standard APIs.

- **CLI-Only Route** (Advanced for scripts):

```
$routes->cli('backup/db', 'Backup::db');  // Run via php spark backup:db
```

Best Practices: Use named routes for refactoring. Add filters for auth/security. Debug with `php spark routes` (lists all). For large apps, split routes into module files.

6. Controllers: The Brain of Your App

Detailed Explanation: Controllers are the central hub, extending `CodeIgniter\Controller` (or `BaseController` for custom init). They receive requests via `$this->request` (unified GET/POST/JSON/files), process logic (call models, validate, manage sessions), and return responses (views, JSON, redirects). Constructors allow dependency injection (e.g., auto-load models/services). Traits add reusable behaviors (e.g., `ResponseTrait` for APIs). Why extend? Shared helpers/filters across controllers. Under the hood, controllers are instantiated per request for isolation. Pitfall: Putting business logic here makes testing hard—keep <50 lines/method, delegate to models/services. Performance: Slim controllers = faster execution; heavy ones bottleneck.

Examples (Basic to Advanced):

- **Basic Direct Response**:

```
<?php
namespace App\Controllers;
use CodeIgniter\Controller;

class Home extends Controller {
    public function index() {
        return 'Welcome!';  // Direct string response
    }
}
```

Why? Simple pages without views.

- **Intermediate with Input, Validation, and Session**:

```
class Contact extends Controller {
    protected $contactModel;

    public function __construct() {
        $this->contactModel = model('App\Models\ContactModel');  // DI
    }

    public function create() {
        helper('form');  // Load helper
        if ($this->request->getMethod() === 'post') {
            $rules = ['name' => 'required|min_length[3]', 'email' => 'required|valid_email'];
            if (! $this->validate($rules)) {
                return view('contacts/create', ['validation' => $this->validator]);
            }
            $data = $this->request->getPost();  // Sanitized input
            $data['user_id'] = auth()->id();  // From auth
            $this->contactModel->insert($data);
            \Config\Services::session()->setFlashdata('success', 'Contact added!');
            return redirect()->to('/contacts');
        }
        return view('contacts/create');
    }
}
```

Why? Handles form POST, validates, uses session flash for messages.

- **Advanced API with Traits and Filters**:

```
use CodeIgniter\API\ResponseTrait;
use CodeIgniter\RESTful\ResourceController;

class ApiContact extends ResourceController {
    use ResponseTrait;
    protected $modelName = 'App\Models\ContactModel';
    protected $format = 'json';
```

```
    public function index() {
        $contacts = $this->model->where('user_id', auth()->id())->findAll();
        return $this->respond($contacts, 200);  // JSON with status
    }

    public function show($id = null) {
        $contact = $this->model->find($id);
        if (!$contact || $contact->user_id !== auth()->id()) {
            return $this->failNotFound('Contact not found or unauthorized');
        }
        return $this->respond($contact);
    }
}
```

Why? Auto-handles CRUD for APIs; integrate with auth filter in routes.

Best Practices: Use `$this->response->setJSON($data)` for APIs. Handle exceptions with try-catch. For auth, check in constructor or filters.

7. Views: Dynamic Presentation Layer

Detailed Explanation: Views are PHP files in `app/Views/` for rendering output, receiving data from controllers as arrays. They support plain PHP or Parser templating (placeholders like {name}). Layouts use extend/section for shared structures (e.g., headers/footers). Escaping with `esc()` prevents XSS by HTML-encoding output. Partials (`include`) for reusable snippets. Why no heavy logic? Keeps views testable/maintainable—put in helpers/controllers. Under the hood, views are compiled to PHP for speed. Pitfall: Unescaped user input = security holes. Performance: Many includes slow rendering; cache full pages if static.

Examples (Basic to Advanced):

- **Basic Data Rendering**: `app/Views/contact/show.php`

```
<!DOCTYPE html>
<html lang="en">
<head><title>Contact Details</title></head>
<body>
    <h1><?= esc($contact->name) ?></h1>  // Escapes HTML
    <p>Email: <?= esc($contact->email) ?></p>
    <p>Phone: <?= esc($contact->phone) ?></p>
</body>
</html>
```

Controller: `return view('contact/show', ['contact' => $model->find(1)]);` Why? Simple, secure output.

- **Intermediate Parser Templating** (For dynamic placeholders): `app/Views/email/template.php`

```
Dear {name},

Your contact {contact_name} has been updated.

Thanks, {app_name}
```

Controller:

```
$parser = \Config\Services::parser();
$data = ['name' => 'User', 'contact_name' => 'Grok', 'app_name' => 'MyApp'];
$html = $parser->setData($data)->render('email/template');
// Send via email library
```

Why? Easier than PHP echoes for emails/reports.

- **Advanced Layouts and Sections** (For DRY code): `app/Views/layouts/main.php`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title><?= esc($title ?? 'Default Title') ?></title>
    <?= $this->include('partials/head') ?>  // Partial for CSS/JS
</head>
<body>
    <?= $this->include('partials/header') ?>
    <main>
        <?= $this->renderSection('content') ?>  // Child inserts here
    </main>
```

```
    <?= $this->include('partials/footer') ?>
</body>
</html>
```

Child View: `app/Views/contacts/index.php`

```
<?= $this->extend('layouts/main') ?>
<?= $this->section('content') ?>
    <h1>Contacts List</h1>
    <ul>
        <?php foreach ($contacts as $contact): ?>
            <li><?= esc($contact->name) ?> - <a href="/contacts/<?= $contact->id ?>">View</a></li>
        <?php endforeach; ?>
    </ul>
    <?php if (session()->getFlashdata('success')): ?>
        <p class="success"><?= esc(session()->getFlashdata('success')) ?></p>
    <?php endif; ?>
<?= $this->endSection() ?>
```

Why? Reuses boilerplate; integrate sessions for messages.

Best Practices: Always esc() user data. Use third-party like Twig via Composer for complex templating. Add conditionals sparingly (e.g., auth checks for buttons).

8. Models, Database Handling, and Basic Querying

Detailed Explanation: Models extend `CodeIgniter\Model` to encapsulate data logic, providing magic CRUD methods (find/insert/update/delete). Query Builder is a fluent, driver-agnostic API for building safe SQL (auto-binds params against injection). Migrations version schema changes (up/down methods for rollback); seeds populate test data. Transactions ensure all-or-nothing ops. Multiple connections for read/write splitting. Why Builder? Readable, portable (switch DB without code changes). Pitfall: Forgetting $allowedFields allows mass assignment attacks. Performance: Builder adds minor overhead vs raw SQL, but safer; cache queries for repeats.

Examples (Basic to Advanced):

- **Basic Model Setup**: `app/Models/ContactModel.php`

```
<?php
namespace App\Models;
use CodeIgniter\Model;

class ContactModel extends Model {
    protected $table = 'contacts';
    protected $primaryKey = 'id';
    protected $allowedFields = ['name', 'email', 'phone', 'user_id'];  // Whitelist for security
    protected $validationRules = ['name' => 'required|min_length[3]'];  // Auto-validate on save
    protected $useTimestamps = true;  // Auto created_at/updated_at
    protected $createdField = 'created_at';
    protected $updatedField = 'updated_at';
}
```

Why? Base for CRUD.

- **Intermediate Query Builder**: Controller usage:

```
$db = \Config\Database::connect();
$query = $db->table('contacts')
            ->select('id, name, email')
            ->join('users', 'users.id = contacts.user_id', 'inner')  // Relations via join
            ->where('contacts.user_id', auth()->id())
            ->where('name LIKE', '%Grok%')
            ->orderBy('name', 'ASC')
            ->limit(10, 0)  // Pagination: 10 items, offset 0
            ->get();
$contacts = $query->getResultArray();  // Or getResultObject(), getRow()
```

Why? Fluent, secure queries.

- **Advanced Migrations, Seeds, and Transactions**: Migration: `php spark migrate:create CreateContactsTable`

```
public function up() {
    $this->forge->addField([
        'id' => ['type' => 'INT', 'auto_increment' => true],
        'name' => ['type' => 'VARCHAR', 'constraint' => 255],
        'email' => ['type' => 'VARCHAR', 'constraint' => 255],
```

```
        'phone' => ['type' => 'VARCHAR', 'constraint' => 50, 'null' => true],
        'user_id' => ['type' => 'INT'],
        'created_at' => ['type' => 'DATETIME', 'null' => true],
        'updated_at' => ['type' => 'DATETIME', 'null' => true],
    ]);
    $this->forge->addPrimaryKey('id');
    $this->forge->addForeignKey('user_id', 'auth_users', 'id', 'CASCADE', 'CASCADE');
    $this->forge->createTable('contacts');
}

public function down() {
    $this->forge->dropTable('contacts');
}
```

Run: `php spark migrate`.

Seed: `php spark make:seeder ContactSeeder`

```
public function run() {
    $data = [
        ['name' => 'Grok', 'email' => 'grok@x.ai', 'user_id' => 1],
    ];
    $this->db->table('contacts')->insertBatch($data);
}
```

Run: `php spark db:seed ContactSeeder`.

Transaction:

```
$db->transBegin();
try {
    $model->insert(['name' => 'New']);
    if (rand(0,1)) throw new \Exception('Fail');
    $db->transCommit();
} catch (\Exception $e) {
    $db->transRollback();
}
```

Why? Atomicity for consistency.

Best Practices: Use $validationRules for model-level checks. Index columns for frequent wheres. Switch to raw SQL only for complex queries.

9. ORM-Like Features: Entities for Object-Relational Mapping

Detailed Explanation: Entities extend `CodeIgniter\Entity\Entity` to map DB rows to objects, adding ORM flavors like type casting (e.g., bool/int/date), mutators (transform on set), accessors (on get), and dirty tracking (changed fields). Integrate with models via $returnType for auto-hydration. Why? Makes data OOP (e.g., $contact->phone = '123'; auto-formats). No built-in relationships (no lazy loading)—simulate with methods/joins. Under the hood, entities use array attributes with magic getters/setters. Pitfall: Casting mismatches throw exceptions—define carefully. Performance: ~10% slower than arrays for large datasets; use asArray() for bulk.

Examples (Basic to Advanced):

- **Basic Entity**: `app/Entities/Contact.php`

```
<?php
namespace App\Entities;
use CodeIgniter\Entity\Entity;

class Contact extends Entity {
    protected $datamap = ['fullName' => 'name'];  // Alias properties
    protected $dates = ['created_at', 'updated_at'];  // To DateTime objects
    protected $casts = ['user_id' => 'integer', 'is_favorite' => 'boolean'];
}
```

  Why? Auto-converts types on hydration.

- **Intermediate Mutators/Accessors**: Add to entity:

```
public function setEmail(string $email): self {
    $this->attributes['email'] = strtolower($email);  // Normalize
    return $this;
}
```

```
public function getDisplayName(): string {
    return ucfirst($this->name) . ' (' . $this->email . ')';  // Computed
}
```

Usage: `$contact = $model->find(1); echo $contact->displayName;`

- **Advanced Integration and Dirty Tracking**: Model: `protected $returnType = 'App\Entities\Contact';`

```
$contact = $model->find(1);
$contact->name = 'Updated Grok';
if ($contact->hasChanged('name')) {  // True
    $model->save($contact);  // Updates only changed fields
}
```

Simulate Relation:

```
public function getUser() {
    return model('UserModel')->find($this->user_id);
}
```

Usage: `$contact->user->username;`

Best Practices: Use for domain logic (e.g., password hashing mutator). For full ORM, add Doctrine via Composer. Test casting with various data types.

10. Helpers: Utility Functions Galore

Detailed Explanation: Helpers are non-OOP function collections for common tasks, loaded manually (`helper('url')`) or auto (`Autoload.php`). They're global once loaded, making them easy for views/controllers. Why? Quick utils without classes (e.g., url() for links). Pitfall: Global namespace pollution—load only needed; custom helpers for app-specific. Performance: Negligible, as functions are lightweight.

Examples (Basic to Advanced, Covering Key Helpers):

- **Basic URL Helper**: Load: `helper('url');`

```
echo site_url('contacts');  // http://localhost:8080/contacts
echo anchor('contacts/1', 'View Contact', ['class' => 'link']);  // <a href="..." class="link">View Contact</a>
redirect('login');  // HTTP redirect
```

Why? Absolute URLs prevent path issues.

- **Intermediate Cookie Helper** (Secure storage): Load: `helper('cookie');`

```
set_cookie('theme', 'dark', 86400 * 30, '/', 'example.com', true, true, 'Strict');  // 30 days, secure, HttpOnly, SameSite=Strict
$theme = get_cookie('theme') ?? 'light';  // Fallback
delete_cookie('theme');
```

Why? Client-side prefs; config in Cookie.php for defaults.

- **Advanced Form and Other Helpers**: Form: `helper('form');`

```
echo form_open('contacts/store', ['id' => 'contactForm']);  // <form action="..." method="post">
echo form_input('name', 'Default Name', ['placeholder' => 'Enter name']);
echo form_submit('submit', 'Save');
echo form_close();
```

Array: `helper('array'); dot_array_search('user.name', ['user' => ['name' => 'Grok']]);` // 'Grok' Date: `helper('date'); echo humanize(now() - 3600);` // '1 hour ago' Filesystem: `helper('filesystem'); write_file(WRITEPATH . 'log.txt', 'Data');` Custom: `app/Helpers/phone_helper.php`

```
function format_phone($phone) { return preg_replace('/(\d{3})(\d{3})(\d{4})/', '($1) $2-$3', $phone); }
```

Load: `helper('phone'); echo format_phone('1234567890');` // (123) 456-7890

Best Practices: Auto-load frequent ones. Use for views (e.g., form in templates). Avoid stateful logic—use libraries.

11. Libraries: Powerful Classes

Detailed Explanation: Libraries are OOP classes accessed as singletons via `\Config\Services::libraryName()`. They're for complex, reusable functionality with state (e.g., Email with config). CI provides many; extend or create custom. Why singletons? Shared instance per request for efficiency. Pitfall: Misconfig (e.g., wrong Email driver) fails silently—debug with printDebugger(). Performance: Lazy-instantiated; heavy libs (e.g., Images) impact if misused.

Examples (Basic to Advanced, Covering Key Libraries):

- **Basic Session Library** (State management): Config: `App.php` $sessionDriver = 'DatabaseHandler'; (migrate ci_sessions table)

```
$session = \Config\Services::session();
$session->set('user_id', 42);  // Persistent
$session->setFlashdata('message', 'Success!');  // One-request
$id = $session->get('user_id') ?? null;
if ($session->has('user_id')) echo 'Logged in';
$session->remove('temp');
$session->destroy();  // Logout
$session->regenerate();  // New ID vs fixation
```

  Why? Server-side state; database driver for clusters.

- **Intermediate Email Library** (Sending): Config: `Email.php` $protocol = 'smtp'; $SMTPHost = 'smtp.example.com';

```
$email = \Config\Services::email();
$email->setFrom('no-reply@myapp.com', 'My App');
$email->setTo('user@example.com');
$email->setCC('cc@example.com');
$email->setSubject('Welcome');
$email->setMessage('<h1>Hello!</h1>');
$email->attach(WRITEPATH . 'report.pdf', 'Report.pdf');
if ($email->send()) echo 'Sent!';
else echo $email->printDebugger();  // Errors
```

  Why? Reliable mailing; alternatives like Sendmail.

- **Advanced Caching, HTTP Client, Pagination, Images**: Caching: Config `Cache.php` $handler = 'redis';

```
$cache = \Config\Services::cache();
if ($data = $cache->get('contacts')) return $data;  // Hit
$data = $model->findAll();  // Miss
$cache->save('contacts', $data, 300);  // TTL 5 min
$cache->delete('contacts');
```

  HTTP Client (CURLRequest):

```
$client = \Config\Services::curlrequest(['base_uri' => 'https://api.example.com']);
$response = $client->get('/data', ['query' => ['key' => 'value'], 'auth' => ['user', 'pass', 'basic']]);
$body = $response->getBody();  // JSON
$client->post('/submit', ['json' => ['field' => 'value']]);
```

  Why? API calls; handles headers/certs.

  Pagination:

```
$pager = \Config\Services::pager();
$total = $model->countAllResults();
$pager->store('default', $total, 10, $this->request->getVar('page') ?? 1);
$contacts = $model->paginate(10);  // Auto-limits
echo $pager->links('default', 'bootstrap_full');  // <ul class="pagination">...
```

  Why? Easy list splitting.

  Images (Manipulation): Config `Image.php` $handler = 'gd'; // Or imagick

```
$image = \Config\Services::image();
$image->withFile(PUBLICPATH . 'photo.jpg')
      ->resize(200, 200, true, 'height')  // Maintain aspect
      ->crop(100, 100, 'center')
      ->rotate(90)
      ->convert(IMAGETYPE_PNG)
      ->save(PUBLICPATH . 'thumb.png');
```

Why? On-the-fly thumbnails.

Best Practices: Register custom in `Services.php`. Use for integrations (e.g., AWS S3 via custom file handler).

12. Forms, Validation, and Input Handling

Detailed Explanation: `$this->request` unifies input sources (GET/POST/JSON/files/cookies). Validation uses rules (built-in like required/valid_email, custom callbacks) for security/data integrity. File uploads validate size/type/move securely. Why? Prevents bad data/injections. Pitfall: Inline validation duplicates—use config groups. Performance: Validation adds checks; batch for large forms.

Examples (Basic to Advanced):

- **Basic Input**:

```
$name = $this->request->getVar('name');  // GET/POST, escaped
$json = $this->request->getJSON(true);  // Array from body
```

- **Intermediate Validation**: `Config/Validation.php`:

```
public array $contact = [
    'name' => 'required|min_length[3]|alpha_numeric_spaces',
    'email' => 'required|valid_email|is_unique[contacts.email,id,{id}]',  // Unique except self
];
```

Controller:

```
if (! $this->validate('contact')) {
    return view('form', ['validation' => $this->validator->getErrors()]);
}
// Valid: process
```

View: `<?= validation_show_error('name') ?>` // Per-field

- **Advanced File Uploads and Custom Rules**: Rules: 'uploaded[file]|max_size[file,1024]|ext_in[file,jpg,png]'

```
$file = $this->request->getFile('photo');
if ($file->isValid() && ! $file->hasMoved()) {
    $newName = $file->getRandomName();
    $file->move(PUBLICPATH . 'uploads', $newName);
}
```

Custom Rule in Validation.php:

```
public function adult_age($age): bool {
    return $age >= 18;
}
```

Use: 'age' => 'adult_age'

Best Practices: Whitelist fields with getPost(['fields']). Integrate with models for save validation.

13. Sessions and Cookies

Detailed Explanation: Sessions store server-side state (e.g., user ID), using cookies for ID tracking. Drivers (File/Database/Redis/Memcached) for flexibility—file for simple, database for shared servers (auto-creates ci_sessions table). Expiration/regeneration prevent fixation/hijacking. Cookies for client-side (non-sensitive like themes). Why separate? Sessions secure; cookies visible. Pitfall: Large sessions slow—store minimal. Performance: Database driver queries per request; Redis faster.

Examples (Basic to Advanced):

- **Basic Session**:

```
$session = \Config\Services::session();
$session->set('cart', ['item1' => 2]);
$cart = $session->get('cart');
$session->setTempdata('token', 'abc', 300);  // Expires in 5 min
```

Why? User state across pages.

- **Intermediate Cookie**:

```
helper('cookie');
set_cookie('pref', 'dark', time() + 86400, '/', '', false, false, 'Lax');  // 1 day
$pref = get_cookie('pref');
```

- **Advanced Regeneration and Database Driver**: Config: $sessionDriver = 'DatabaseHandler';

```
$session->regenerate(true);  // New ID on login
$session->markAsFlashdata('message');  // Convert to flash
```

Best Practices: Regenerate on privilege change. Use cookies for remember-me tokens (hashed, DB-validated).

14. Security Features

Detailed Explanation: CI provides out-of-box protections: CSRF tokens in forms, XSS escaping via esc(), CSP headers to block unauthorized resources, Honeypot for bots, Throttler for rate-limiting. Why? Prevents common attacks like injections/cross-site. Pitfall: Custom output without esc() exposes XSS. Performance: Minimal; CSP adds headers.

Examples (Basic to Advanced):

- **Basic CSRF/XSS**: Form: `<?= csrf_field() ?>` // Hidden token View: `esc($userInput, 'html')`

- **Intermediate CSP**: App.php: $CSPEnabled = true; $CSP['script-src'] = ["'self'", 'trusted.com'];

- **Advanced Honeypot/Throttler**: Filters.php: $globals['before'] = ['honeypot'];

```
$throttler = \Config\Services::throttler();
if ($throttler->check('login', 3, 60)) {  // 3/min
    // Allow
} else {
    // Block
}
```

Best Practices: Always enable in prod. Use Shield for auth security.

15. Authentication and Authorization

Detailed Explanation: No built-in auth for modularity—use official Shield package (`composer require codeigniter4/shield`). It handles registration, login, 2FA, password reset, roles/groups, permissions, JWT for APIs. Sessions store logged-in state; cookies for remember-me. Why package? Customizable without core bloat. Pitfall: Custom auth risks weak hashing—Shield uses bcrypt. Performance: DB queries for user lookup.

Examples (Basic to Advanced):

- **Setup**: `php spark shield:setup` (migrations/tables).

- **Basic Login**: Controller:

```
if (auth()->attempt(['email' => $post['email'], 'password' => $post['password']], $remember)) {
    return redirect('dashboard');
}
```

- **Advanced Roles/Perms**: Assign: `auth()->user()->addGroup('admin');` Check: `if (auth()->user()->can('contacts.edit')) { ... }`

Best Practices: Use filters for protected routes. Enable email activation.

16. API Development

Detailed Explanation: Use ResourceController for auto-CRUD APIs. Supports JSON/XML, status codes. Integrate throttler/auth filters. Why? Quick REST. Pitfall: No validation = bad data.

Examples:

- `routes->resource('api/contacts');` Controller: Extend ResourceController, respond().

Best Practices: Use fail() for errors.

17. Caching

Detailed Explanation: Caches data/queries/pages for speed. Drivers: File/Redis. Why? Reduces DB hits.

Examples:

- `$cache->save('key', $data, 60);`

Best Practices: Invalidate on updates.

18. Testing

Detailed Explanation: PHPUnit-integrated with traits for unit/feature/DB tests. Why? Catch bugs.

Examples:

- Unit: assertEquals(model->find(1), expected).

Best Practices: Cover 80% code.

19. CLI Tools (Spark)

Detailed Explanation: Spark for commands/generators. Why? Automation.

Examples:

- `php spark make:controller Contact`

Best Practices: Custom commands for cron.

20. Extending CodeIgniter

Detailed Explanation: Events for hooks, filters for intercept, services for overrides.

Examples:

- Event: `Events::on('login', fn() => log());`
- Filter: Implement before/after.

Best Practices: Use for plugins.

21. Logging and Error Handling

Detailed Explanation: Log levels; custom handlers.

Examples:

- `log_message('info', 'User logged in');`

Best Practices: Custom error views.

22. Internationalization and Localization

Detailed Explanation: Lang files for translations.

Examples:

- `lang('Contacts.name')`

Best Practices: Detect from request.

23. Deployment and Best Practices

Detailed Explanation: Prod env, optimize.

Examples: Composer no-dev.

24.Sample Project: Building a Secure Contact Manager App from Zero to Publish

This is a complete, standalone sample project for CodeIgniter 4.6.3: A web app where users register/login (via Shield for auth), then perform full CRUD on personal contacts (name, email, phone). Contacts are tied to the logged-in user for security (ownership checks). It demonstrates 80% of CI features: MVC, models/entities, validation, sessions/flashdata, routes/filters, forms/CSRF, migrations/seeds, and deployment.

Key Features:

- Auth: Register, login, logout, remember-me (session + cookie).
- CRUD: Add/view/edit/delete contacts (user-specific).
- Security: Validation, CSRF, esc(), ownership checks.
- Integrations: Entities for casting, sessions for messages, helpers for forms.
- Database: MySQL (adaptable to others).

Prerequisites: PHP 8.1+, Composer, MySQL (or SQLite). Total setup time: ~30 min to run locally.

Step 1: Project Setup from Zero

1. **Install CodeIgniter Starter**:

   ```
   composer create-project codeigniter4/appstarter contactapp --prefer-dist
   cd contactapp
   ```

   This creates the folder structure with v4.6.3.

2. **Configure Environment**:

   - Copy `env` to `.env`.
   - Edit `.env`:

     ```
     CI_ENVIRONMENT = development
     app.baseURL = 'http://localhost:8080'

     # Database
     database.default.hostname = localhost
     database.default.database = contactdb
     database.default.username = root
     database.default.password =
     database.default.DBDriver = MySQLi
     database.default.DBPrefix =
     ```

   - Create MySQL DB: `CREATE DATABASE contactdb;`

3. **Install Shield for Authentication**:

   ```
   composer require codeigniter4/shield
   php spark shield:install  # Or shield:setup in older; runs migrations for auth tables (users, tokens, etc.)
   php spark migrate --all  # Apply Shield migrations
   ```

   Shield adds tables like `auth_identities`, `auth_logins`. Config in `app/Config/Auth.php` (defaults fine for now).

4. **Auto-Load Essentials**: Edit `app/Config/Autoload.php`:

```
public array $helpers = ['url', 'form', 'auth', 'cookie'];  // For forms, URLs, auth checks
```

5. **Enable Filters and Sessions**:
   - Sessions auto-start. Config in `App.php`: `public string $sessionDriver = 'CodeIgniter\Session\Handlers\DatabaseHandler';` (for persistence; migrate ci_sessions if needed: see docs).
   - Filters: Edit `app/Config/Filters.php`:

```
public array $aliases = [
    'csrf'     => \CodeIgniter\Filters\CSRF::class,
    'toolbar'  => \CodeIgniter\Filters\DebugToolbar::class,
    'honeypot' => \CodeIgniter\Filters\Honeypot::class,
    'login'    => \CodeIgniter\Shield\Filters\SessionAuth::class,  // Shield's session filter
];
public array $globals = [
    'before' => ['csrf', 'honeypot'],
    'after'  => ['toolbar'],
];
```

6. **Run Dev Server**:

```
php spark serve
```

   Base URL: `http://localhost:8080`.

Step 2: Database Schema - Migrations and Seeds

1. **Create Contacts Table Migration**:

```
php spark migrate:create CreateContactsTable
```

   Edit the generated file `app/Database/Migrations/*_CreateContactsTable.php`:

```
<?php
namespace App\Database\Migrations;
use CodeIgniter\Database\Migration;

class CreateContactsTable extends Migration {
    public function up() {
        $this->forge->addField([
            'id' => [
                'type' => 'INT',
                'constraint' => 11,
                'unsigned' => true,
                'auto_increment' => true,
            ],
            'name' => [
                'type' => 'VARCHAR',
                'constraint' => 255,
            ],
            'email' => [
                'type' => 'VARCHAR',
                'constraint' => 255,
            ],
            'phone' => [
                'type' => 'VARCHAR',
                'constraint' => 50,
                'null' => true,
            ],
            'user_id' => [
                'type' => 'INT',
                'unsigned' => true,
            ],
            'created_at' => [
                'type' => 'DATETIME',
                'null' => true,
            ],
            'updated_at' => [
                'type' => 'DATETIME',
                'null' => true,
            ],
        ]);
        $this->forge->addPrimaryKey('id');
        $this->forge->addForeignKey('user_id', 'auth_identities', 'user_id', 'CASCADE', 'CASCADE');  // Link to Shield users
```

```
        $this->forge->createTable('contacts');
    }

    public function down() {
        $this->forge->dropTable('contacts');
    }
}
```

Run: `php spark migrate` (creates table).

2. **Optional Seed for Test Data**:

```
php spark make:seeder ContactSeeder
```

Edit `app/Database/Seeds/ContactSeeder.php`:

```
<?php
namespace App\Database\Seeds;
use CodeIgniter\Database\Seeder;

class ContactSeeder extends Seeder {
    public function run() {
        $data = [
            ['name' => 'Test Contact', 'email' => 'test@example.com', 'phone' => '1234567890', 'user_id' => 1],
        ];
        $this->db->table('contacts')->insertBatch($data);
    }
}
```

Run: `php spark db:seed ContactSeeder` (after creating a user).

Step 3: Models and Entities

1. **Contact Entity** (ORM-like):

```
php spark make:entity Contact
```

Edit `app/Entities/Contact.php`:

```
<?php
namespace App\Entities;
use CodeIgniter\Entity\Entity;

class Contact extends Entity {
    protected $datamap = [];
    protected $dates   = ['created_at', 'updated_at'];
    protected $casts   = [
        'user_id' => 'integer',
        'phone'   => '?string',  // Optional string
    ];

    // Mutator example: Normalize email
    protected function setEmail(string $email) {
        $this->attributes['email'] = strtolower(trim($email));
    }
}
```

2. **Contact Model**:

```
php spark make:model ContactModel -e Contact  # Links entity
```

Edit `app/Models/ContactModel.php`:

```
<?php
namespace App\Models;
use CodeIgniter\Model;

class ContactModel extends Model {
    protected $DBGroup          = 'default';
    protected $table            = 'contacts';
    protected $primaryKey       = 'id';
    protected $useAutoIncrement = true;
```

```
        protected $returnType       = \App\Entities\Contact::class;
        protected $useSoftDeletes   = false;
        protected $protectFields    = true;
        protected $allowedFields    = ['name', 'email', 'phone', 'user_id'];

        // Dates
        protected $useTimestamps = true;
        protected $dateFormat    = 'datetime';
        protected $createdField  = 'created_at';
        protected $updatedField  = 'updated_at';

        // Validation
        protected $validationRules      = [
            'name'  => 'required|min_length[3]|max_length[255]',
            'email' => 'required|valid_email|max_length[255]',
            'phone' => 'permit_empty|min_length[10]',
        ];
        protected $validationMessages   = [];
        protected $skipValidation       = false;

        // Custom: Get user's contacts
        public function getUserContacts(int $userId) {
            return $this->where('user_id', $userId)->findAll();
        }
    }
```

Step 4: Routes

Edit `app/Config/Routes.php`:

```
<?php
$routes->get('/', 'Home::index');

// Auth Routes (unprotected)
$routes->get('register', 'AuthController::register');
$routes->post('register', 'AuthController::register');
$routes->get('login', 'AuthController::login');
$routes->post('login', 'AuthController::login');
$routes->get('logout', 'AuthController::logout');

// Protected Contacts (apply login filter)
$routes->group('', ['filter' => 'login'], function ($routes) {
    $routes->get('contacts', 'ContactController::index');
    $routes->get('contacts/create', 'ContactController::create');
    $routes->post('contacts/create', 'ContactController::store');
    $routes->get('contacts/edit/(:num)', 'ContactController::edit/$1');
    $routes->post('contacts/update/(:num)', 'ContactController::update/$1');
    $routes->get('contacts/delete/(:num)', 'ContactController::delete/$1');
});
```

Step 5: Controllers

1. **AuthController** (Handles register/login/logout):

   ```
   php spark make:controller AuthController
   ```

   Edit `app/Controllers/AuthController.php`:

   ```
   <?php
   namespace App\Controllers;
   use CodeIgniter\Controller;
   use CodeIgniter\Shield\Authentication\Actions\EmailActivator;  // Optional for activation

   class AuthController extends Controller {
       protected $helpers = ['form', 'url'];

       public function register() {
           if ($this->request->getMethod() === 'post') {
               $rules = [
                   'username' => 'required|min_length[3]|is_unique[users.username]',
                   'email'    => 'required|valid_email|is_unique[auth_identities.secret]',
                   'password' => 'required|min_length[8]',
               ];
               if (! $this->validate($rules)) {
                   return view('auth/register', ['validation' => $this->validator]);
               }
   ```

```
        $users = auth()->getProvider();
        $user = new \CodeIgniter\Shield\Entities\User([
            'username' => $this->request->getPost('username'),
            'email'    => $this->request->getPost('email'),
            'password' => $this->request->getPost('password'),
        ]);
        $users->save($user);
        $user = $users->findById($users->getInsertID());
        $user->addGroup('user');  // Default group
        session()->setFlashdata('message', 'Registered successfully! Please login.');
        return redirect()->to('/login');
    }
    return view('auth/register');
}

public function login() {
    if ($this->request->getMethod() === 'post') {
        $rules = [
            'email'    => 'required|valid_email',
            'password' => 'required',
        ];
        if (! $this->validate($rules)) {
            return view('auth/login', ['validation' => $this->validator]);
        }
        $credentials = [
            'email'    => $this->request->getPost('email'),
            'password' => $this->request->getPost('password'),
        ];
        $remember = (bool) $this->request->getPost('remember');
        $auth = auth();  // Shield service
        if ($auth->attempt($credentials, $remember)) {
            return redirect()->to('/contacts');
        }
        return redirect()->back()->with('error', 'Invalid credentials');
    }
    return view('auth/login');
}

public function logout() {
    auth()->logout();
    return redirect()->to('/login')->with('message', 'Logged out');
    }
}
```

2. **ContactController** (CRUD logic):

```
php spark make:controller ContactController
```

Edit `app/Controllers/ContactController.php`:

```
<?php
namespace App\Controllers;
use App\Models\ContactModel;
use CodeIgniter\Controller;

class ContactController extends Controller {
    protected $contactModel;

    public function __construct() {
        $this->contactModel = new ContactModel();
    }

    public function index() {
        $userId = auth()->id();
        $data['contacts'] = $this->contactModel->getUserContacts($userId);
        $data['message'] = session()->getFlashdata('message');
        return view('contacts/index', $data);
    }

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

    public function store() {
        $post = $this->request->getPost();
        $post['user_id'] = auth()->id();
        if (! $this->contactModel->insert($post)) {
            return redirect()->back()->withInput()->with('errors', $this->contactModel->errors());
```

```
        }
        return redirect()->to('/contacts')->with('message', 'Contact added successfully!');
    }

    public function edit($id) {
        $contact = $this->contactModel->find($id);
        if (! $contact || $contact->user_id !== auth()->id()) {
            return redirect()->to('/contacts')->with('error', 'Unauthorized or not found');
        }
        $data['contact'] = $contact;
        return view('contacts/edit', $data);
    }

    public function update($id) {
        $contact = $this->contactModel->find($id);
        if (! $contact || $contact->user_id !== auth()->id()) {
            return redirect()->to('/contacts')->with('error', 'Unauthorized');
        }
        $post = $this->request->getPost();
        if (! $this->contactModel->update($id, $post)) {
            return redirect()->back()->withInput()->with('errors', $this->contactModel->errors());
        }
        return redirect()->to('/contacts')->with('message', 'Contact updated!');
    }

    public function delete($id) {
        $contact = $this->contactModel->find($id);
        if (! $contact || $contact->user_id !== auth()->id()) {
            return redirect()->to('/contacts')->with('error', 'Unauthorized');
        }
        $this->contactModel->delete($id);
        return redirect()->to('/contacts')->with('message', 'Contact deleted!');
    }
}
```

3. **HomeController** (Optional landing): Edit `app/Controllers/Home.php`:

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

Step 6: Views

Create folders `app/Views/auth/` and `app/Views/contacts/`. Use basic HTML—no external CSS for simplicity.

1. **auth/register.php**:

```
<!DOCTYPE html>
<html><head><title>Register</title></head><body>
<h2>Register</h2>
<?= form_open('/register') ?>
<?= csrf_field() ?>
<label>Username: <input type="text" name="username" value="<?= old('username') ?>"></label><br>
<?= validation_show_error('username') ?>
<label>Email: <input type="email" name="email" value="<?= old('email') ?>"></label><br>
<?= validation_show_error('email') ?>
<label>Password: <input type="password" name="password"></label><br>
<?= validation_show_error('password') ?>
<button type="submit">Register</button>
<?= form_close() ?>
<?php if (session()->getFlashdata('message')): ?><p><?= esc(session()->getFlashdata('message')) ?></p><?php endif; ?>
</body></html>
```

2. **auth/login.php**:

```
<!DOCTYPE html>
<html><head><title>Login</title></head><body>
<h2>Login</h2>
<?= form_open('/login') ?>
<?= csrf_field() ?>
<label>Email: <input type="email" name="email" value="<?= old('email') ?>"></label><br>
<label>Password: <input type="password" name="password"></label><br>
<label><input type="checkbox" name="remember"> Remember Me</label><br>
<button type="submit">Login</button>
<?= form_close() ?>
<?php if (session()->getFlashdata('error')): ?><p style="color:red;"><?= esc(session()->getFlashdata('error')) ?></p><?php endif; ?>
```

```
<a href="/register">Register</a>
</body></html>
```

3. **contacts/index.php** (List):

```
<!DOCTYPE html>
<html><head><title>My Contacts</title></head><body>
<h2>Contacts</h2> <a href="/logout">Logout</a> | <a href="/contacts/create">Add New</a>
<?php if ($message): ?><p><?= esc($message) ?></p><?php endif; ?>
<table border="1">
    <tr><th>Name</th><th>Email</th><th>Phone</th><th>Actions</th></tr>
    <?php foreach ($contacts as $contact): ?>
        <tr>
            <td><?= esc($contact->name) ?></td>
            <td><?= esc($contact->email) ?></td>
            <td><?= esc($contact->phone ?? 'N/A') ?></td>
            <td>
                <a href="/contacts/edit/<?= $contact->id ?>">Edit</a>
                <a href="/contacts/delete/<?= $contact->id ?>" onclick="return confirm('Delete?')">Delete</a>
            </td>
        </tr>
    <?php endforeach; ?>
</table>
</body></html>
```

4. **contacts/create.php**:

```
<!DOCTYPE html>
<html><head><title>Add Contact</title></head><body>
<h2>Add Contact</h2>
<?= form_open('/contacts/create') ?>
<?= csrf_field() ?>
<label>Name: <input type="text" name="name" value="<?= old('name') ?>"></label><br>
<label>Email: <input type="email" name="email" value="<?= old('email') ?>"></label><br>
<label>Phone: <input type="text" name="phone" value="<?= old('phone') ?>"></label><br>
<button type="submit">Save</button>
<?= form_close() ?>
<?php if (session()->getFlashdata('errors')): ?>
    <ul style="color:red;">
        <?php foreach (session()->getFlashdata('errors') as $error): ?>
            <li><?= esc($error) ?></li>
        <?php endforeach; ?>
    </ul>
<?php endif; ?>
<a href="/contacts">Back</a>
</body></html>
```

5. **contacts/edit.php** (Similar to create, pre-filled):

```
<!DOCTYPE html>
<html><head><title>Edit Contact</title></head><body>
<h2>Edit Contact</h2>
<?= form_open('/contacts/update/' . $contact->id) ?>
<?= csrf_field() ?>
<label>Name: <input type="text" name="name" value="<?= esc(old('name', $contact->name)) ?>"></label><br>
<label>Email: <input type="email" name="email" value="<?= esc(old('email', $contact->email)) ?>"></label><br>
<label>Phone: <input type="text" name="phone" value="<?= esc(old('phone', $contact->phone)) ?>"></label><br>
<button type="submit">Update</button>
<?= form_close() ?>
<?php if (session()->getFlashdata('errors')): ?>
    <ul style="color:red;">
        <?php foreach (session()->getFlashdata('errors') as $error): ?>
            <li><?= esc($error) ?></li>
        <?php endforeach; ?>
    </ul>
<?php endif; ?>
<a href="/contacts">Back</a>
</body></html>
```

Step 7: Testing Locally

1. Visit `http://localhost:8080/register` → Create account.
2. Login → Redirect to /contacts (empty list).
3. Add/edit/delete → See flash messages, validation errors.
4. Logout → Back to login; direct /contacts redirects via filter.

5. Debug: Check `writable/logs/` for errors.

Step 8: Production Touches

- Set `.env`: `CI_ENVIRONMENT = production` (hides errors).
- Optimize: `composer install --optimize-autoloader --no-dev`.
- Security: Ensure HTTPS for remember-me cookies.

Step 9: Deployment (Publish)

1. **Git Init and Push**:

   ```
   git init
   git add .
   git commit -m "Initial"
   ```

   Ignore: `.env`, `vendor/`, `writable/` in .gitignore.

2. **Free Host Example: Render.com (or Heroku/Hostinger)**:

   - Sign up, new Web Service → Link Git repo.
   - Runtime: PHP.
   - Build Command: `composer install --no-dev --optimize-autoloader && php spark migrate`.
   - Start Command: `heroku-php-apache2 public/` (or equivalent for Render).
   - Add Env Vars: DB creds, `app.baseURL = https://yourapp.onrender.com`.
   - Deploy → Auto-builds/migrates.
   - Access live URL.

3. **Alternative: Shared Hosting**:

   - Upload files via FTP (public/ as web root).
   - Import DB schema.
   - Run migrations via SSH if available.

Extensions Ideas: Add search (Query Builder whereLike), API endpoints (ResourceController), or email notifications (Email library).