REPUBLIC OF THE PHILIPPINES

DEPARTMENT OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

RUFINO JOHN E. AGUILAR

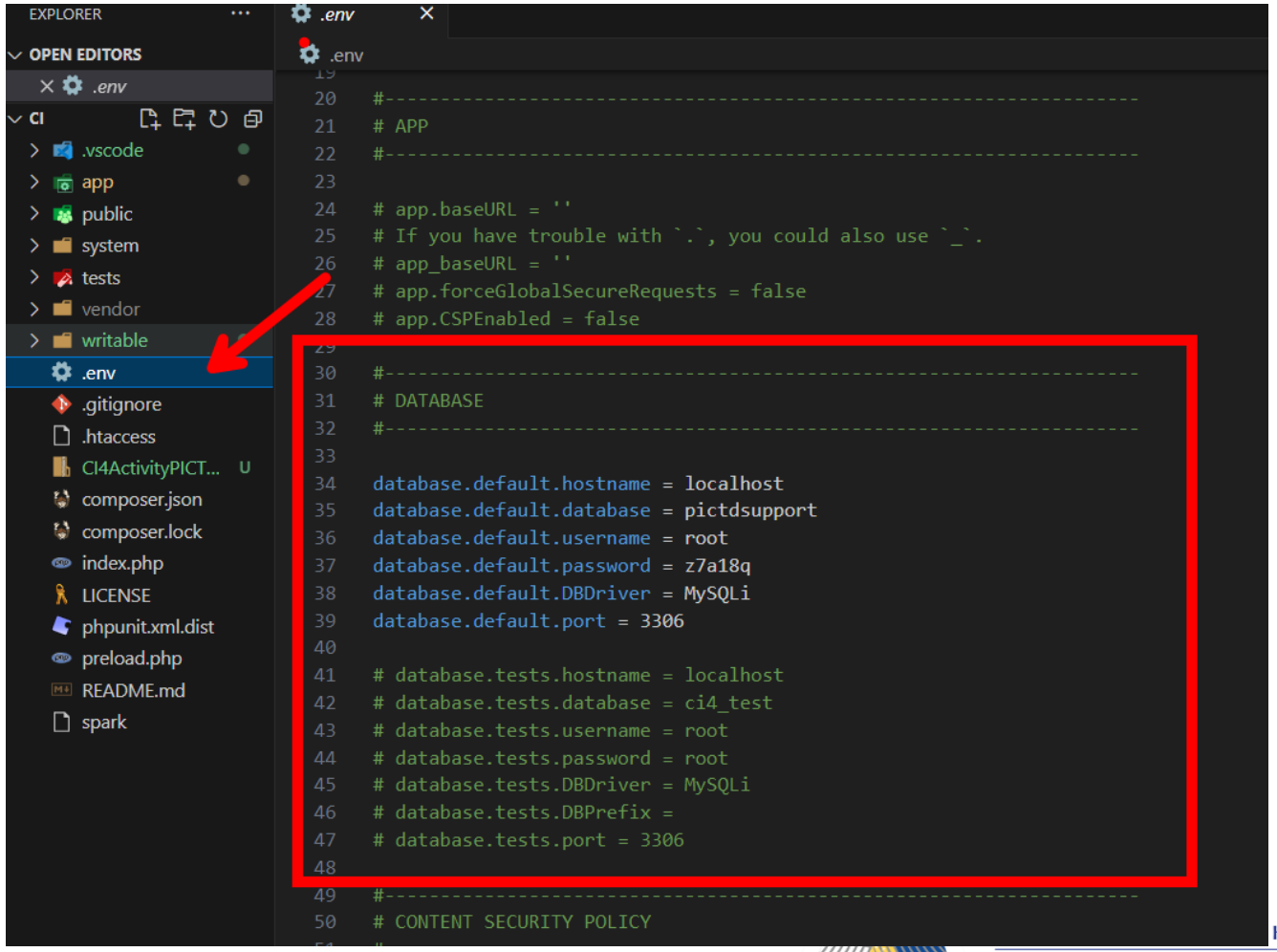
https://rufdev.github.io/

Module 2 : Database Interaction

- Database configuration.
- Working with multiple databases.
- Query Builder class for advanced database operations.
- Database migrations and seeding.

Database Configuration

- .env



Working with multiple database

- Production
- Staging
- Testing/Dev

```
database.default.hostname = localhost
database.default.database = pictdsupport
database.default.username = root
database.default.password = z7a18q
database.default.DBDriver = MySQLi
database.default.port = 3306
```

```php
public string $defaultGroup = 'default';

/**
 * The default database connection.
 */
public array $default = [
    'DSN'      => '',
    'hostname' => 'localhost',
    'username' => 'root',
    'password' => 'z7a18q',
    'database' => 'dborder',
    'DBDriver' => 'MySQLi',
    'DBPrefix' => '',
    'pConnect' => false,
    'DBDebug'  => true,
    'charset'  => 'utf8',
    'DBCollat' => 'utf8_general_ci',
    'swapPre'  => '',
    'encrypt'  => false,
    'compress' => false,
    'strictOn' => false,
    'failover' => [],
    'port'     => 3306,
];
```

Working with multiple database

A database is required for most web application programming. Currently supported databases are:

- MySQL via the `MySQLi` driver (version 5.1 and above only)
- PostgreSQL via the `Postgre` driver (version 7.4 and above only)
- SQLite3 via the `SQLite3` driver
- Microsoft SQL Server via the `SQLSRV` driver (version 2005 and above only)
- Oracle Database via the `OCI8` driver (version 12.1 and above only)

Query Builder class

Loading the Query Builder

The Query Builder is loaded through the `table()` method on the database connection. This sets the **FROM** portion of the query for you and returns a new instance of the Query Builder class:

```php
<?php

$db      = \Config\Database::connect();
$builder = $db->table('users');
```

Query Builder class

- Get

Get

$builder->get()

Runs the selection query and returns the result. Can be used by itself to retrieve all records from a table:

```php
<?php

$builder = $db->table('mytable');
$query   = $builder->get();  // Produces: SELECT * FROM mytable
```

The first and second parameters enable you to set a limit and offset clause:

```php
<?php

$query = $builder->get(10, 20);
/*
 * Executes: SELECT * FROM mytable LIMIT 20, 10
 * (in MySQL. Other databases have slightly different syntax)
 */
```

Query Builder class

- getResult()

```php
<?php

$query = $builder->get();

foreach ($query->getResult() as $row) {
    echo $row->title;
}
```

Query Builder class

- getCompletedSelect()

```php
<?php

$sql = $builder->getCompiledSelect();
echo $sql;
// Prints string: SELECT * FROM mytable
```

Query Builder class

- getWhere()

$builder->getWhere()

Identical to the `get()` method except that it permits you to add a "where" clause in the first parameter, instead of using the `$builder->where()` method:

```php
<?php

$query = $builder->getWhere(['id' => $id], $limit, $offset);
```

Query Builder class

- Select

Select

$builder->select()

Permits you to write the **SELECT** portion of your query:

```
<?php

$builder->select('title, content, date');
$query = $builder->get();
// Executes: SELECT title, content, date FROM mytable
```

Query Builder class

- RawSQL

RawSql

New in version 4.2.0.

Since v4.2.0, `$builder->select()` accepts a `CodeIgniter\Database\RawSql` instance, which expresses raw SQL strings.

```
<?php

use CodeIgniter\Database\RawSql;

$sql = 'REGEXP_SUBSTR(ral_anno,"[0-9]{1,2}([,.][0-9]{1,3})([,.][0-9]{1,3})") AS ral';
$builder->select(new RawSql($sql));
$query = $builder->get();
```

Query Builder class

- selectMax()
- selectMin()
- selectAvg()
- selectSum()
- selectCount()
- selectSubquery()

Query Builder class

- From

From

$builder->from()

Permits you to write the **FROM** portion of your query:

```php
<?php

$builder = $db->table('users');
$builder->select('title, content, date');
$builder->from('mytable');
$query = $builder->get();
// Produces: SELECT title, content, date FROM users, mytable
```

Query Builder class

- Subqueries

Subqueries

$builder->fromSubquery()

Permits you to write part of a **FROM** query as a subquery.

This is where we add a subquery to an existing table:

```php
<?php

$subquery = $db->table('users');
$builder  = $db->table('jobs')->fromSubquery($subquery, 'alias');
$query    = $builder->get();
// Produces: SELECT * FROM `jobs`, (SELECT * FROM `users`) `alias`
```

Use the `$db->newQuery()` method to make a subquery the main table:

```php
<?php

$subquery = $db->table('users')->select('id, name');
$builder  = $db->newQuery()->fromSubquery($subquery, 't');
$query    = $builder->get();
// Produces: SELECT * FROM (SELECT `id`, `name` FROM users) `t`
```

Query Builder class

- Join

Join

$builder->join()

Permits you to write the **JOIN** portion of your query:

```
<?php

$builder = $db->table('blogs');
$builder->select('*');
$builder->join('comments', 'comments.id = blogs.id');
$query = $builder->get();
/*
 * Produces:
 * SELECT * FROM blogs JOIN comments ON comments.id = blogs.id
 */
```

Multiple method calls can be made if you need several joins in one query.

If you need a specific type of **JOIN** you can specify it via the third parameter of the method. Options are: `left`, `right`, `outer`, `inner`, `left outer`, and `right outer`.

Query Builder class

- Where

Where

$builder->where()

This method enables you to set **WHERE** clauses using one of five methods:

Query Builder class

- Where

1. Simple key/value method

```php
<?php

$builder->where('name', $name);
// Produces: WHERE name = 'Joe'
```

If you use multiple method calls they will be chained together with **AND** between them:

```php
<?php

$builder->where('name', $name);
$builder->where('title', $title);
$builder->where('status', $status);
// WHERE name = 'Joe' AND title = 'boss' AND status = 'active'
```

Query Builder class

- Where

2. Custom key/value method

You can include an operator in the first parameter in order to control the comparison:

```php
<?php

$builder->where('name !=', $name);
$builder->where('id <', $id);
// Produces: WHERE name != 'Joe' AND id < 45
```

Query Builder class

- Where

3. Associative array method

```php
<?php

$array = ['name' => $name, 'title' => $title, 'status' => $status];
$builder->where($array);
// Produces: WHERE name = 'Joe' AND title = 'boss' AND status = 'active'
```

You can include your own operators using this method as well:

```php
<?php

$array = ['name !=' => $name, 'id <' => $id, 'date >' => $date];
$builder->where($array);
```

Query Builder class

- Where

4. Custom string

You can write your own clauses manually:

```
<?php

$where = "name='Joe' AND status='boss' OR status='active'";
$builder->where($where);
```

Query Builder class

- Where

5. RawSql

New in version 4.2.0.

Since v4.2.0, `$builder->where()` accepts a `CodeIgniter\Database\RawSql` instance, which expresses raw SQL strings.

```php
<?php

use CodeIgniter\Database\RawSql;

$sql = "id > 2 AND name != 'Accountant'";
$builder->where(new RawSql($sql));
```

Query Builder class

- Where

6. Subqueries

```php
<?php

// With closure
use CodeIgniter\Database\BaseBuilder;

$builder->where('advance_amount <', static function (BaseBuilder $builder) {
    $builder->select('MAX(advance_amount)', false)->from('orders')->where('id >', 2);
});
// Produces: WHERE "advance_amount" < (SELECT MAX(advance_amount) FROM "orders" WHERE "id" > 2)

// With builder directly
$subQuery = $db->table('orders')->select('MAX(advance_amount)', false)->where('id >', 2);
$builder->where('advance_amount <', $subQuery);
```

Query Builder class

- Where
  - orWhere()
  - whereIn()
  - orWhereIn()
  - whereNotIn()
  - orWhereNotIn()

Query Builder class

- Like

Like

$builder->like()

This method enables you to generate **LIKE** clauses, useful for doing searches.

Query Builder class

- Like

Like

$builder->like()

This method enables you to generate **LIKE** clauses, useful for doing searches.

Query Builder class

- Like

1. Simple key/value method

```
<?php

$builder->like('title', 'match');
// Produces: WHERE `title` LIKE '%match%' ESCAPE '!'
```

If you use multiple method calls they will be chained together with **AND** between them:

```
<?php

$builder->like('title', 'match');
$builder->like('body', 'match');
// WHERE `title` LIKE '%match%' ESCAPE '!' AND  `body` LIKE '%match%' ESCAPE '!'
```

Query Builder class

- Like

2. Associative array method

```php
<?php

$array = ['title' => $match, 'page1' => $match, 'page2' => $match];
$builder->like($array);
/*
 * WHERE `title` LIKE '%match%' ESCAPE '!'
 *     AND  `page1` LIKE '%match%' ESCAPE '!'
 *     AND  `page2` LIKE '%match%' ESCAPE '!'
 */
```

Query Builder class

- Like

3. RawSql

New in version 4.2.0.

Since v4.2.0, `$builder->like()` accepts a `CodeIgniter\Database\RawSql` instance, which expresses raw SQL strings.

```php
<?php

use CodeIgniter\Database\RawSql;

$sql    = "CONCAT(users.name, ' ', IF(users.surname IS NULL OR users.surname = '', '', users.surname))";
$rawSql = new RawSql($sql);
$builder->like($rawSql, 'value', 'both');
```

Query Builder class

- Like
  - orLike()
  - notLike()
  - orNotLike()
  - groupBy()
  - distinct()
  - having(), havingIn(), orHaving(), orHavingIn()
  - havingNotIn(), orhvingNotIn()

Query Builder class

- OrderBy

OrderBy

$builder->orderBy()

Lets you set an **ORDER BY** clause.

The first parameter contains the name of the column you would like to order by.

The second parameter lets you set the direction of the result. Options are `ASC`, `DESC` AND `RANDOM`.

```
<?php

$builder->orderBy('title', 'DESC');
// Produces: ORDER BY `title` DESC
```

Query Builder class

- Limit

Limit

$builder->limit()

Lets you limit the number of rows you would like returned by the query:

```
<?php

$builder->limit(10);
// Produces: LIMIT 10
```

Query Builder class

- Union

Union

$builder->union()

Is used to combine the result-set of two or more SELECT statements. It will return only the unique results.

```php
<?php

$builder = $db->table('users')->select('id, name')->limit(10);
$union   = $db->table('groups')->select('id, name');
$builder->union($union)->get();
/*
 * Produces:
 * SELECT * FROM (SELECT `id`, `name` FROM `users` LIMIT 10) uwrp0
 * UNION SELECT * FROM (SELECT `id`, `name` FROM `groups`) uwrp1
 */
```

Query Builder class

- Insert
- inertBatch()

Insert

$builder->insert()

Generates an insert string based on the data you supply, and runs the query. You can either pass an **array** or an **object** to the method. Here is an example using an array:

```php
<?php

use CodeIgniter\Database\RawSql;

$data = [
    'id'          => new RawSql('DEFAULT'),
    'title'       => 'My title',
    'name'        => 'My Name',
    'date'        => '2022-01-01',
    'last_update' => new RawSql('CURRENT_TIMESTAMP()'),
];

$builder->insert($data);
/* Produces:
    INSERT INTO mytable (id, title, name, date, last_update)
    VALUES (DEFAULT, 'My title', 'My name', '2022-01-01', CURRENT_TIMESTAMP())
*/
```

Query Builder class

- Update

Update

$builder->replace()

This method executes a **REPLACE** statement, which is basically the SQL standard for (optional) **DELETE** + **INSERT**, using *PRIMARY* and *UNIQUE* keys as the determining factor. In our case, it will save you from the need to implement complex logics with different combinations of `select()`, `update()`, `delete()` and `insert()` calls.

Example:

```php
<?php

$data = [
    'title' => 'My title',
    'name'  => 'My Name',
    'date'  => 'My date',
];

$builder->replace($data);
// Executes: REPLACE INTO mytable (title, name, date) VALUES ('My title', 'My name', 'My date')
```

Query Builder class

- Update
- updateBatch()

$builder->update()

Generates an update string and runs the query based on the data you supply. You can pass an **array** or an **object** to the method. Here is an example using an array:

```php
<?php

$data = [
    'title' => $title,
    'name'  => $name,
    'date'  => $date,
];

$builder->where('id', $id);
$builder->update($data);
/*
 * Produces:
 * UPDATE mytable
 * SET title = '{$title}', name = '{$name}', date = '{$date}'
 * WHERE id = $id
 */
```

Query Builder class

- Delete
- deleteBatch()

Delete

$builder->delete()

Generates a **DELETE** SQL string and runs the query.

```php
<?php

$builder->delete(['id' => $id]);
// Produces: DELETE FROM mytable WHERE id = $id
```

The first parameter is the where clause. You can also use the `where()` or `orWhere()` methods instead of passing the data to the first parameter of the method:

```php
<?php

$builder->where('id', $id);
$builder->delete();
/*
 * Produces:
 * DELETE FROM mytable
 * WHERE id = $id
 */
```

Models

- The CodeIgniter's Model provides convenience features and additional functionality that people commonly use to make working with a single table in your database more convenient.

- It comes out of the box with helper methods for much of the standard ways you would need to interact with a database table, including finding records, updating records, deleting records, and more.

Models

Accessing Models

Models are typically stored in the **app/Models** directory. They should have a namespace that matches their location within the directory, like `namespace App\Models`.

Models

You can access models within your classes by creating a new instance or using the `model()` helper function.

```php
<?php

// Create a new class manually.
$userModel = new \App\Models\UserModel();

// Create a shared instance of the model.
$userModel = model('UserModel');
// or
$userModel = model('App\Models\UserModel');
// or
$userModel = model(App\Models\UserModel::class);

// Create a new class with the model() function.
$userModel = model('UserModel', false);

// Create shared instance with a supplied database connection.
$db        = db_connect('custom');
$userModel = model('UserModel', true, $db);
```

Models

CodeIgniter's Model

CodeIgniter does provide a model class that provides a few nice features, including:

- automatic database connection
- basic CRUD methods
- in-model validation
- automatic pagination
- and more

Models

- Create Model

```
RufinoJohn@PICTD-RUFY MINGW64 /c/xampp/htdocs/ci (main)
$ php spark make:model Office
```

Models

- Model Template

```php
k?php

namespace App\Models;

use CodeIgniter\Model;

class Office extends Model
{
    protected $DBGroup          = 'default';
    protected $table            = 'offices';
    protected $primaryKey       = 'id';
    protected $useAutoIncrement = true;
    protected $returnType       = 'array';
    protected $useSoftDeletes   = false;
    protected $protectFields    = true;
    protected $allowedFields    = [];
```

Models

- Model Template

```
// Dates
protected $useTimestamps = false;
protected $dateFormat    = 'datetime';
protected $createdField  = 'created_at';
protected $updatedField  = 'updated_at';
protected $deletedField  = 'deleted_at';
```

Models

- Model Template

```
// Validation
protected $validationRules      = [];
protected $validationMessages   = [];
protected $skipValidation       = false;
protected $cleanValidationRules = true;
```

Models

- Model Template

```
    // Callbacks
    protected $allowCallbacks = true;
    protected $beforeInsert   = [];
    protected $afterInsert    = [];
    protected $beforeUpdate   = [];
    protected $afterUpdate    = [];
    protected $beforeFind     = [];
    protected $afterFind      = [];
    protected $beforeDelete   = [];
    protected $afterDelete    = [];
}
```

Models

- Finding Data
  - find()
  - findColumn()
  - findAll()
  - first()

```php
<?php

$user = $userModel->find($user_id);
```

```php
<?php

$user = $userModel->findColumn($column_name);
```

```php
<?php

$users = $userModel->where('active', 1)->findAll();
```

```php
<?php

$user = $userModel->where('deleted', 0)->first();
```

Models

- Saving Data
  - insert()

```php
<?php

$data = [
    'username' => 'darth',
    'email'    => 'd.vader@theempire.com',
];

// Inserts data and returns inserted row's primary key
$userModel->insert($data);

// Inserts data and returns true on success and false on failure
$userModel->insert($data, false);

// Returns inserted row's primary key
$userModel->getInsertID();
```

Models

- Saving Data
  - udpate()

```php
<?php

$data = [
    'username' => 'darth',
    'email'    => 'd.vader@theempire.com',
];

$userModel->update($id, $data);
```

Models

- Saving Data
  - save()

```php
<?php

// Defined as a model property
$primaryKey = 'id';

// Does an insert()
$data = [
    'username' => 'darth',
    'email'    => 'd.vader@theempire.com',
];

$userModel->save($data);

// Performs an update, since the primary key, 'id', is found.
$data = [
    'id'       => 3,
    'username' => 'darth',
    'email'    => 'd.vader@theempire.com',
];
$userModel->save($data);
```

Models

- Deleting Data
  - delete()

```php
<?php

$userModel->delete(12);
```

Models

- Model Validation

Validating Data

For many people, validating data in the model is the preferred way to ensure the data is kept to a single standard, without duplicating code. The Model class provides a way to automatically have all data validated prior to saving to the database with the `insert()`, `update()`, or `save()` methods.

Models

- Validation rules

Setting Validation Rules

The first step is to fill out the `$validationRules` class property with the fields and rules that should be applied. If you have custom error message that you want to use, place them in the `$validationMessages` array:

```php
<?php

namespace App\Models;

use CodeIgniter\Model;

class UserModel extends Model
{
    protected $validationRules = [
        'username'     => 'required|max_length[30]|alpha_numeric_space|min_length[3]',
        'email'        => 'required|max_length[254]|valid_email|is_unique[users.email]',
        'password'     => 'required|max_length[255]|min_length[8]',
        'pass_confirm' => 'required_with[password]|max_length[255]|matches[password]',
    ];
    protected $validationMessages = [
        'email' => [
            'is_unique' => 'Sorry. That email has already been taken. Please choose another.',
        ],
    ];
}
```

Models

- Validation rules
  - setValidationRule()
  - setValidationRules()
  - setValidationMessage()
  - setValidationMessages()

Models

- Getting Validation Result

Getting Validation Result

Now, whenever you call the `insert()`, `update()`, or `save()` methods, the data will be validated. If it fails, the model will return boolean **false**.

Models

- Getting Validation Result
  - errors()

Getting Validation Errors

You can use the `errors()` method to retrieve the validation errors:

```
<?php

if ($model->save($data) === false) {
    return view('updateUser', ['errors' => $model->errors()]);
}
```

This returns an array with the field names and their associated errors that can be used to either show all of the errors at the top of the form, or to display them individually:

```
<?php if (! empty($errors)): ?>
    <div class="alert alert-danger">
    <?php foreach ($errors as $field => $error): ?>
        <p><?= esc($error) ?></p>
    <?php endforeach ?>
    </div>
<?php endif ?>
```

Models

- Getting Validation Result
  - errors()

Getting Validation Errors

You can use the `errors()` method to retrieve the validation errors:

```
<?php

if ($model->save($data) === false) {
    return view('updateUser', ['errors' => $model->errors()]);
}
```

This returns an array with the field names and their associated errors that can be used to either show all of the errors at the top of the form, or to display them individually:

```
<?php if (! empty($errors)): ?>
    <div class="alert alert-danger">
    <?php foreach ($errors as $field => $error): ?>
        <p><?= esc($error) ?></p>
    <?php endforeach ?>
    </div>
<?php endif ?>
```

Models

- Query Builder + Model

Mixing Methods of Query Builder and Model

You can also use Query Builder methods and the Model's CRUD methods in the same chained call, allowing for very elegant use:

```php
<?php

$users = $userModel->where('status', 'active')
    ->orderBy('last_login', 'asc')
    ->findAll();
```

Database migrations and seeding

- Database Forge Class
  - createDatabase(‘db_name’)
    - php spark db:create foo
  - dropDatabase(‘db_name’)

Load the Forge Class as follows:

```php
<?php

$forge = \Config\Database::forge();
```

Database migrations and seeding

- Creating Table

- `unsigned` /true : to generate "UNSIGNED" in the field definition.
- `default` /value : to generate a default value in the field definition.
- `null` /true : to generate "null" in the field definition. Without this, the field will default to "NOT null".
- `auto_increment` /true : generates an auto_increment flag on the field. Note that the field type must be a type that supports this, such as integer.
- `unique` /true : to generate a unique key for the field definition.

Database migrations and seeding

- addFields($fields);

```php
<?php

$fields = [
    'id' => [
        'type'           => 'INT',
        'constraint'     => 5,
        'unsigned'       => true,
        'auto_increment' => true,
    ],
    'title' => [
        'type'       => 'VARCHAR',
        'constraint' => '100',
        'unique'     => true,
    ],
    'author' => [
        'type'       => 'VARCHAR',
        'constraint' => 100,
        'default'    => 'King of Town',
    ],
    'description' => [
        'type' => 'TEXT',
        'null' => true,
    ],
    'status' => [
        'type'       => 'ENUM',
        'constraint' => ['publish', 'pending', 'draft'],
        'default'    => 'pending',
    ],
];
```

Database migrations and seeding

- Adding Keys
  - addKey()
  - addPrimaryKey()
  - addUniqueKey()
- Adding Foreign Keys
  - addForeignKey()

```php
<?php

$forge->addKey('blog_id', true);
// gives PRIMARY KEY `blog_id` (`blog_id`)

$forge->addKey('blog_id', true);
$forge->addKey('site_id', true);
// gives PRIMARY KEY `blog_id_site_id` (`blog_id`, `site_id`)

$forge->addKey('blog_name');
// gives KEY `blog_name` (`blog_name`)

$forge->addKey(['blog_name', 'blog_label'], false, false, 'my_key_name');
// gives KEY `my_key_name` (`blog_name`, `blog_label`)

$forge->addKey(['blog_id', 'uri'], false, true, 'my_key_name');
// gives UNIQUE KEY `my_key_name` (`blog_id`, `uri`)
```

```php
<?php

$forge->addForeignKey('users_id', 'users', 'id');
// gives CONSTRAINT `TABLENAME_users_id_foreign` FOREIGN KEY(`users_id`) REFERENCES `users`(`id`)

$forge->addForeignKey(['users_id', 'users_name'], 'users', ['id', 'name']);
// gives CONSTRAINT `TABLENAME_users_id_foreign` FOREIGN KEY(`users_id`, `users_name`) REFERENCES `users`(`id`, `name`
```

Database migrations and seeding

- Creating a Table

Creating a Table

After fields and keys have been declared, you can create a new table with

```
<?php

$forge->createTable('table_name');
// gives CREATE TABLE table_name
```

Database migrations and seeding

- Dropping a Table

Dropping a Table

Execute a `DROP TABLE` statement and optionally add an `IF EXISTS` clause.

```php
<?php

// Produces: DROP TABLE `table_name`
$forge->dropTable('table_name');

// Produces: DROP TABLE IF EXISTS `table_name`
$forge->dropTable('table_name', true);
```

Database migrations and seeding

- Adding a Field to a Table
  - addColumn()

```php
<?php

$fields = [
    'preferences' => ['type' => 'TEXT'],
];
$forge->addColumn('table_name', $fields);
// Executes: ALTER TABLE `table_name` ADD `preferences` TEXT
```

Database migrations and seeding

- Dropping Field From a Table
  - dropColumn()

```php
<?php

$forge->dropColumn('table_name', 'column_to_drop'); // to drop one single column
```

Database migrations and seeding

- Modify a Field in a Table
  - modifyColumn()

```php
<?php

$fields = [
    'old_name' => [
        'name' => 'new_name',
        'type' => 'TEXT',
        'null' => false,
    ],
];
$forge->modifyColumn('table_name', $fields);
// gives ALTER TABLE `table_name` CHANGE `old_name` `new_name` TEXT NOT NULL
```

Database migrations and seeding

- Migration

Database Migrations

Migrations are a convenient way for you to alter your database in a structured and organized manner. You could edit fragments of SQL by hand but you would then be responsible for telling other developers that they need to go and run them. You would also have to keep track of which changes need to be run against the production machines next time you deploy.

Database migrations and seeding

- Migration File Names

Migration File Names

Each Migration is run in numeric order forward or backwards depending on the method taken. Each migration is numbered using the timestamp when the migration was created, in **YYYY-MM-DD-HHIISS** format (e.g., **2012-10-31-100537**). This helps prevent numbering conflicts when working in a team environment.

Prefix your migration files with the migration number followed by an underscore and a descriptive name for the migration. The year, month, and date can be separated from each other by dashes, underscores, or not at all. For example:

- 2012-10-31-100538_AlterBlogTrackViews.php
- 2012_10_31_100539_AlterBlogAddTranslations.php
- 20121031100537_AddBlog.php

Database migrations and seeding

- Create a Migration
  - Function up()
  - Function down()

```php
public function up()
{
    $this->forge->addField([
        'blog_id' => [
            'type'           => 'INT',
            'constraint'     => 5,
            'unsigned'       => true,
            'auto_increment' => true,
        ],
        'blog_title' => [
            'type'       => 'VARCHAR',
            'constraint' => '100',
        ],
        'blog_description' => [
            'type' => 'TEXT',
            'null' => true,
        ],
    ]);
    $this->forge->addKey('blog_id', true);
    $this->forge->createTable('blog');
}
```

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

Database migrations and seeding

- Command-Line Tools

migrate

Migrates a database group with all available migrations:

```
php spark migrate
```

You can use (migrate) with the following options:

- `-g` - to chose database group, otherwise default database group will be used.
- `-n` - to choose namespace, otherwise (App) namespace will be used.
- `--all` - to migrate all namespaces to the latest migration.

Database migrations and seeding

- Command-Line Tools

migrate

Migrates a database group with all available migrations:

```
php spark migrate
```

You can use (migrate) with the following options:

- `-g` - to chose database group, otherwise default database group will be used.
- `-n` - to choose namespace, otherwise (App) namespace will be used.
- `--all` - to migrate all namespaces to the latest migration.

Database migrations and seeding

- Command-Line Tools

rollback

Rolls back all migrations, taking the database group to a blank slate, effectively migration 0:

```
php spark migrate:rollback
```

You can use (rollback) with the following options:

- `-g` - to choose database group, otherwise default database group will be used.
- `-b` - to choose a batch: natural numbers specify the batch.
- `-f` - to force a bypass confirmation question, it is only asked in a production environment.

Database migrations and seeding

- Command-Line Tools

status

Displays a list of all migrations and the date and time they ran, or '–' if they have not been run:

```
php spark migrate:status

...

+---------------------+-------------------+----------------------+---------+---------------------+-------+
| Namespace           | Version           | Filename             | Group   | Migrated On         | Batch |
+---------------------+-------------------+----------------------+---------+---------------------+-------+
| App                 | 2022-04-06-234508 | CreateCiSessionsTable | default | 2022-04-06 18:45:14 | 2     |
| CodeIgniter\Settings | 2021-07-04-041948 | CreateSettingsTable  | default | 2022-04-06 01:23:08 | 1     |
| CodeIgniter\Settings | 2021-11-14-143905 | AddContextColumn     | default | 2022-04-06 01:23:08 | 1     |
+---------------------+-------------------+----------------------+---------+---------------------+-------+
```

You can use (status) with the following options:

- `-g` - to choose database group, otherwise default database group will be used.

Database migrations and seeding

- Command-Line Tools

make:migration

Creates a skeleton migration file in **app/Database/Migrations**. It automatically prepends the current timestamp. The class name it creates is the Pascal case version of the filename.

```
php spark make:migration <class> [options]
```

You can use (`make:migration`) with the following options:

- `--namespace` - Set root namespace. Default: `APP_NAMESPACE`.
- `--suffix` - Append the component title to the class name.

Database migrations and seeding

- Seeding

Database Seeding

Database seeding is a simple way to add data into your database. It is especially useful during development where you need to populate the database with sample data that you can develop against, but it is not limited to that. Seeders can contain static data that you don't want to include in a migration, like countries, or geo-coding tables, event or setting information, and more.

Database migrations and seeding

Database Seeders

Database seeders are simple classes that must have a `run()` method, and extend `CodeIgniter\Database\Seeder`. Within the `run()` the class can create any form of data that it needs to. It has access to the database connection and the forge through `$this->db` and `$this->forge`, respectively. Seed files must be stored within the **app/Database/Seeds** directory. The name of the file must match the name of the class.

```php
<?php

namespace App\Database\Seeds;

use CodeIgniter\Database\Seeder;

class SimpleSeeder extends Seeder
{
    public function run()
    {
        $data = [
            'username' => 'darth',
            'email'    => 'darth@theempire.com',
        ];

        // Simple Queries
        $this->db->query('INSERT INTO users (username, email) VALUES(:username:, :email:)', $data);

        // Using Query Builder
        $this->db->table('users')->insert($data);
    }
}
```

Database migrations and seeding

- Creating Seeder Files

Creating Seeder Files

Using the command line, you can easily generate seed files:

```
php spark make:seeder user --suffix
```

Database migrations and seeding

- Running Seeder File

Command Line Seeding

You can also seed data from the command line, as part of the Migrations CLI tools, if you don't want to create a dedicated controller:

```
php spark db:seed TestSeeder
```