WELCOME TO BOVIDAE FRAMEWORK DOCUMENTATION

Installation

Server Requirements

The Bovidae framework has a few system requirements; you will need to make sure your server meets the following requirements:

- PHP >= 7.0.0
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension
- Ctype PHP Extension
- JSON PHP Extension
- BCMath PHP Extension

Installing Bovidae

Bovidae utilizes Composer to manage its dependencies. So, before using Bovidae, make sure you have Composer installed on your machine.

First, download the Bovidae repository from Github:

Github URL:

Configuration

Public Directory

After downloading Bovidae, you should configure your web server's document / web root to be the public directory. The `index.php` in this directory serves as the front controller for all HTTP requests entering your application.

Directory Structure

Introduction

The default Bovidae application structure is intended to provide a great starting point for both large and small applications. Of course, you are free to organize your application however you like. Bovidae imposes almost no restrictions on where any given class is located - as long as Composer can autoload the class.

The Root Directory

The App Directory

The app directory, as you might expect, contains the core code of your application. We'll explore this directory in more detail soon; however, almost all of the classes in your application will be in this directory.

The Public Directory

The public directory contains the `index.php` file, which is the entry point for all requests entering your application and configures autoloading. This directory also houses your assets such as images, JavaScript, and CSS.

The Routes Directory

The routes directory contains all of the route definitions for your application. By default, route files are included with Bovidae: `web.php`

The `web.php` file contains routes, all of your routes will most likely be defined in the `web.php` file.

The Vendor Directory

The vendor directory contains your Composer dependencies.

Deployment

Introduction

When you're ready to deploy your Bovidae application to production, there are some important things you can do to make sure your application is running as efficiently as possible. In this document, we'll cover some great starting points for making sure your Bovidae application is deployed properly.

Routing

The most basic Bovidae routes accept a URI and a Closure, providing a very simple and expressive method of defining routes:

```
Route::add(“/foo”, “ControllerName@MethodName”);
```

All Bovidae routes are defined in your route files, which are located in the routes directory. These files are automatically loaded by the framework. The routes/web.php file defines routes that are for your web interface.

These routes are assigned the web middleware group, which provides features like session state and CSRF protection. The routes in routes/api.php are stateless and are assigned the api middleware group.

For most applications, you will begin by defining routes in your routes/web.php file. The routes defined in routes/web.php may be accessed by entering the defined route's URL in your browser. For example, you may access the following route by navigating to

http://your-app.test/user in your browser:

```
Route:add('/user', 'UserController@index');
```

Controllers

Introduction

Controllers can group related request handling logic into a single class. Controllers are stored in the app/controllers directory.

Defining Controller

Below is an example of a basic controller class. Note that the controller extends the base controller class included with Bovidae. The base class provides a few convenience methods, which may be used to attach to controller actions:

```
<?php

namespace App\Controllers;

use App\Model\User;

class UserController extends Controller
{
   /**
    * Show the profile for the given user.
    *
    * @param  int  $id
    * @return View
    */
   public function show($id)
   {
      View::renderTemplate(user/home.html');
   }
}
?>
```

You can define a route to this controller action like so:

```
Route::add('/user', 'UserController@show');
```

Now, when a request matches the specified route URI, the show method on the UserController class will be executed. Of course, the route parameters will also be passed to the method.

Controllers & Namespaces

It is very important to note that we did not need to specify the full controller namespace when defining the controller route. Since the loads your route files within a route group that contains the namespace, we only specified the portion of the class name that comes after the app/controllers portion of the namespace.

If you choose to nest your controllers deeper into the app/controllers directory, use the specific class name relative to the app/controllers root namespace. So, if your full controller class is app/controllers/photos/adminController you should register routes to the controller like so:

```
Route::add('foo', 'Photos\AdminController@method');
```

Views

Views contain the HTML served by your application and separate your

Controller/application logic from your presentation logic. Views are stored in the app/views directory. A simple view might look something like this:

```
<html>
  <body>
    <h1>Hello, {{ $name }}</h1>
  </body>
</html>
```

Since this view is stored at app/views/hello.html we may return it using the global Core\View Class helper like so:

```
View::renderTemplate("foo.html", array('name'=>'b'))
```

As you can see, the first argument passed to the view class corresponds to the name of the view file in the app/view directory. The second argument is an array of data that should be made

available to the view. In this case, we are passing the name variable, which is displayed in the view using template engine.

Passing Data to Views

As you saw in the previous examples, you may pass an array of data to views:

```
return View::renderTemplate('view_name', ['name' =>
```

Template Engine

Defining a Layout

Two of the primary benefits of using template engine are *template inheritance* and *sections*. To get started, let's take a look at a simple example. First, we will examine a "master" page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single template view:

```
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <div class="container">
      {% block content %} {% endblock %}
    </div> </body>
{% block scripts %}{% endblock %}
</html>
```

Extending a Layout

When defining a child view, use the "**extends**" directive to specify which layout the child view should "inherit". Views which extend a layout may inject content into the layout's sections using **"block"** directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using **"{% block %}"**:

```
{% extends "layouts/app.html" %}

{% block title %} Create Model {% endblock %}

{% block content %}

   <p>This is my body content.</p>

{% endblock %}
```

Bovidae ORM

The Bovidae ORM included with Bovidae provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table. Before getting started, be sure to configure a database connection in config/database.php

The easiest way to create a model instance is using Bovidae Generator Module which we will discuss later in this documentation. Now, let's look at an example User model, which we will use to retrieve and store information from our user database table.

```
<?php

namespace App\Models;

use Core\BovidaeORM;

use PDO;

class User extends BovidaeORM

{ }
```

Table Name

Note that we did not tell BovidaeORM which table to use for our user model. By convention, the name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Bovidae will assume the User model stores records in the user table. You may specify a custom table by defining a "table" property on your model:

```
<?php
namespace App\Models;
use Core\BovidaeORM;
use PDO;
class User extends BovidaeORM
{  protected static $table =
'my_users';
}
```

Primary Key

BovidaeORM will also assume that each table has a primary key column named id. You may define a protected $pk property to override this convention.

Generator

Bovidae Framework also have Generator module that provides a web based code generator for generating Model, Controllers and Views

Generate Model

Step 1: Ensure that Database connection properly configured by checking the database configuration file in `config/database.php` file.

Step2: Select Table from Table Drop Down, the model will automatically populate in model name input field.

Step3: Click on Create Button

Note: You can also provide table name and model name explicitly

Generate Controller

Step 1: Ensure that Database connection properly configured by checking the database configuration file in `config/database.php` file.

Step2: Select Table from Table Drop Down, the model will automatically populate in model name input field.

Step3: Click on Create Button

Note: You can also generate CRUD operations

Generate View

Step 1: Give path where you want to store the view file whether in `app/views/hello.html` or `app/views/foo/foo_/foo.html`

Step2: Input the view name

Step3: Click on Create Button

Generate All Models

Bovidae Framework also have Generator module that provides all models automatically generate on the screens

Generate All Models

Bovidae Framework also have Generator module that provides all controllers automatically generate on the screens.