HiveApi Documentation

Release 1.0.0

Johannes Schobel

Jul 11, 2018

Getting Started

CHAPTER 1

Software Requirements

- GIT
- PHP >= 7.1.3
- PHP Extensions:
  - OpenSSL PHP Extension
  - PDO PHP Extension
  - Mbstring PHP Extension
  - Tokenizer PHP Extension
  - BCMath PHP Extension (required when the Hash ID feature is enabled)
  - Intl Extension (required when you use the Localization Container)
- Composer
- Node (required for the API Docs generator feature)
- Web Server (Nginx is recommended)
- Database Engine (no specific Database Engine recommended)
- Cache Engine (Redis is recommended)
- Queues Engine (Beanstalkd is recommended)

CHAPTER 2

Installation

2.1 HiveApi Application Installation

HiveApi can be installed automatically with Composer (recommended way) or manually (via Git or via direct download):

2.1.1 1) Download

In the following, both methods are described in short:

1.A) Automatically via Composer

1. Clone the repo, install dependencies and setup the project:

Option 1: Latest stable release release:

```
composer create-project hiveapi/framework my-awesome-api
```

Option 2: Target a specific version:

```
composer create-project hiveapi/framework my-awesome-api ~major.minor
```

Option 3: Ongoing development branch `dev-master` (unstable):

> Heads up!
>
> This may provide (unstable) features from the upcoming releases. You may need to keep syncing your project with the upstream `master` branch and run `composer update` in order to apply changes!*

```
composer create-project hiveapi/framework my-awesome-api dev-master
```

1. Edit your `.env` variables to match with your environment (set database credentials, app url, . . .).
2. Continue from *2) Database Setup* below.

1.B) Manually

You can download the code directly from the repository as `.zip` file or clone the repository using `git` (recommended approach):

1. Clone the repository using `git`:

```
git clone https://github.com/hiveapi/framework.git
```

1. Install all dependency packages (including containers dependencies):

```
composer install
```

1. Create a new `.env` file by copying the provided `.env.example` file.

```
cp .env.example .env
```

Heads up!

heck all variables and edit accordingly!

1. Generate a random `APP_KEY`

```
php artisan key:generate
```

1. Delete the existing `.git` folder from the root directory and initialize your own one with `git init`.

2.1.2 2) Database Setup

1. Migrate the provided database by runing the migration artisan command:

```
php artisan migrate
```

1. Seed the database with the artisan command:

```
php artisan hive:seed:deploy
```

1. (optional) By default `HiveApi` seeds a “Super User”, given the default `admin` role (the role has no permissions set to it).

To give the `admin` role, access to all the seeded permissions in the system, run the following command at any time.

```
php artisan hive:permissions:toRole admin
```

If you are using Laradock, you need to run those commands from the `workspace` container, you can enter that container by running `docker-compose exec workspace bash` from the Laradock folder.

2.1.3 3) OAuth 2.0 Setup

1. Create encryption keys to generate secure access tokens and create “personal access” and “password grant” clients, which will be used to generate access tokens for your users or applications:

```
php artisan passport:install
```

2.1.4 4) Documentation Setup

If you are planning to use ApiDoc JS then proceed with this setup, else skip this and use whatever you prefer:

1. Install ApiDocJs using NPM or your favorite dependencies manager:

Install it Globally with `-g` or locally in the project without `-g`

```
npm install apidoc -g
```

or install it by just running `npm install` on the root of the project, after checking the `package.json` file on the root.

1. run `php artisan hive:docs`

Behind the scene `hive:docs` executes a command like this

```
apidoc -c app/Containers/Documentation/ApiDocJs/public -f public.php -i app -o public/
↪api/documentation
```

See the API Docs Generator page for more details.

2.1.5 5) Tests Setup

1. Open `.env.testing` and set up the environment variables correctly.
2. Open the `/tests/_data/presets/*` files and adapt the `urls` accordingly to fit your domains.
3. Run the tests

```
vendor/bin/codecept run
```

2.2 B) Development Environment Setup

You can run `HiveApi` on your favorite environment. Below you see how you can run it on top of Vagrant (using Laravel Homestead) or Docker (using Laradock).

We will see how to use both tools and you can pick one, or you can use other options like Larvel Valet, Laragon or even run it directly on your machine.

Heads up!

The ICANN has now officially approved `.dev` as a generic top level domain (gTLD). Therefore, it is **not** recommended to use `.dev` domains any more in your local development setup! The docs here has been changed to use `.develop` instead of `.dev`, however, you may change to `.localhost`, `.test`, or whatever suits your needs.

2.2.1 B.1) Using Docker (with Laradock)

`Laradock` is a Docker PHP development environment. It facilitate running PHP Apps on Docker.

1. Install Laradock.
2. Navigate into the `laradock` directory:

```
cd laradock
```

This directory contains a `docker-compose.yml` file. (From the `Laradock` project).

2.1) If you haven't done so, rename `env-example` to `.env`.

```
cp env-example .env
```

1. Run the Docker containers:

```
docker-compose up -d nginx mysql redis beanstalkd
```

1. Make sure you are setting the `Docker IP` as `Host` for the `DB` and `Redis` in your `.env` file.
2. Add the domain to the `hosts` file:

5.1) Open the hosts file on your local machine `/etc/hosts`.

We'll be using `hive.local` as local domain (you can change it if you want).

5.2) Map the domain and its subdomains to 127.0.0.1:

```
127.0.0.1  hive.local
127.0.0.1  api.hive.local
127.0.0.1  admin.hive.local
```

If you are using NGINX or Apache, make sure the **server_name** (in case of NGINX) or **ServerName** (in case of Apache) in your the server config file, is set to the following `hive.local api.hive.local admin.hive.local`. Also don't forget to set your **root** or **DocumentRoot** to the public directory inside hive (i.e., `hive/public`).

2.2.2 B.2) Using Vagrant (with Laravel Homestead)

1. Configure Homestead:

1.1) Open the Homestead config file:

```
homestead edit
```

1.2) Map the `api.hive.local` domain to the project public directory - Example:

```
sites:
    - map: api.hive.local
      to: /{full-project-path}/hive/public
```

1.3) You can also map other domains like `hive.local` and `admin.hive.local` to other web apps:

```
    - map: hive.local
      to: /{full-project-path}/clients/web/user
    - map: admin.hive.local
      to: /{full-project-path}/clients/web/admin
```

Note: in the example above the `/{full-project-path}/clients/web/xxx` are separate apps, who live in their own repositories and in different folder than the `HiveApi`. If your admins, users or other type of applications are within `HiveApi`, then you must point them all to the `HiveApi` project folder `/{full-project-path}/hive/public`. So in that case you would have something like this:

```
    - map: api.hive.local
      to: /{full-project-path}/hive/public
    - map: hive.local
      to: /{full-project-path}/hive/public
```

(continues on next page)

(continued from previous page)

```
    - map: admin.hive.local
      to: /{full-project-path}/hive/public
```

1. Add the domain to the hosts file:

2.1) Open the hosts file on your local machine `/etc/hosts`.

We'll be using `hive.local` as local domain (you can change it if you want).

2.2) Map the domain and its subdomains to the Vagrant IP Address:

```
192.168.10.10   hive.local
192.168.10.10   api.hive.local
192.168.10.10   admin.hive.local
```

If you are using NGINX or Apache, make sure the **server_name** (in case of NGINX) or **ServerName** (in case of Apache) in your the server config file, is set to the following `hive.local api.hive.local admin.hive.local`. Also don't forget to set your **root** or **DocumentRoot** to the public directory inside hive (i.e., `hive/public`).

2.3) Run the Virtual Machine:

```
homestead up --provision
```

If you see `No input file specified` on the sub-domains!try running this command `homestead halt && homestead up --provision`.

2.2.3 B.3) Using something else

If you're not into virtualization solutions, you can setup your environment directly on your machine. Check the software requirements list.

2.3 C) Play

Now let's see it in action

1.a. Open your web browser and visit:

- `http://hive.local` You should see an HTML page, with `HiveApi` in the middle.
- `http://admin.hive.local` You should see an HTML Login page.

1.b. Open your HTTP client and call:

- `http://api.hive.local/` You should see a JSON response with message: `"Welcome to HiveApi."`,
- `http://api.hive.local/v1` You should see a JSON response with message: `"Welcome to HiveApi (API V1)."`,

1. Make some HTTP calls to the API:

   Heads up!

   To make HTTP calls you can use Postman, HTTPIE or any other tool you prefer.

   here is a postman file available that provides most of the pre-defined routes of HiveApi.

CHAPTER 3

Overview

3.1 Quickstart

When a HTTP request is received, it first hits your predefined Endpoint (each endpoint has its own Route file).

3.1.1 Sample Route Endpoint

```php
<?php

$router->get('/hello', [
    'uses' => 'Controller@sayHello',
]);
```

After the user makes a request to the endpoint `[GET] api.hive.develop/v1/hello` it calls the function (`sayHello()`) in the respective `Controller` class.

3.1.2 Sample Controller Function

```php
<?php

class Controller extends ApiController
{
    public function sayHello(SayHelloRequest $request)
    {
            $helloMessage = Hive::call(SayHelloAction::class);

            $this->json([
                $helloMessage
            ]);
    }
}
```

This `sayHello()` function takes a `Request` class `SayHelloRequest` and automatically checks, if the user has the proper role (or permission) to call this endpoint. An `Exception` is immediately thrown, if the user does not have the proper access level. Otherwise, the actual function is executed.

In this context, the function calls an `Action` (`SayHelloAction`) to perform the actual business logic.

3.1.3 Sample Action

```php
<?php

class SayHelloAction extends Action
{
    public function run()
    {
        return 'Hello World!';
    }
}
```

An `Action` can do anything then (maybe) return a result. When the `Action` finishes its execution, the `Controller` function gets ready to build a `Response` and return this to the client that called the endpoint.

`Json` responses can be built using the helper function `json()` (`$this->json(['foo' => 'bar']);`).

3.1.4 Sample User Response

```
[
    "Hello World!"
]
```

CHAPTER 4

Architecture Pattern

The two most common architectures, used for building projects on top of HiveApi are:

- **Porto** (Route Request Controller Action Task Model Transformer).
- **MVC** (Model View Controller. The HiveApi MVC version is a little different than the standard MVC)

Porto is the HiveApi recommended architecture for building scalable APIs. However, it also support building APIs using the popular and well-known MVC architecture (with slight modifications).

> Heads up!
>
> HiveApi features are written using Porto, and can be used by any architecture.

Below you will see how you can both any of the architectures to build your project.

4.1 Porto

4.1.1 Introduction

Porto is an architecture pattern that consists of 2 layers, called **Containers** and **Ship** layer.

The **Container** layer holds your application business logic. This is similar to Modular, DDD and plugins architectures design. HiveApi, however, allows separating the business logic into multiple folders called **Containers**. The **Ship** layer, on the other hand, holds the infrastructure code (i.e., shared code between all **Containers**). This code is rarely modified at all.

HiveApi features themselves are developed using the Porto Software Architectural Pattern. This means, features provided by HiveApi live in their own Containers.

Spending 15 minutes, reading the Porto Document before getting started, is a great investment of time.

4.1.2 The Containers Layer

Read about the `Containers` layer here

Removing Containers

HiveApi comes with some default containers (e.g., for `Authentication` or `User` management). All containers are optional and can be easily re-written or extended.

Let's say you don't want to use the built in documentation generator feature of HiveApi. In order to get rid of that feature you can simply delete the `Documentation` container from your application.

To remove a container, simply delete the folder then run `composer update` to remove its dependencies.

Create new Container

In order to extend your application with new features, you can call follow various approaches.

Option 1) Using the Code Generator:

Call the command `php artisan hive:generate:container` from the command line. A wizard will guide you through the process of creating the most important aspects.

Refer to the Code Generator page for more details.

Option 2) manually:

1. Create a folder in the `app\Containers` folder.
2. Start creating components (i.e., `Actions`, `Tasks`) and wiring them all together.
3. The `Ship` layer will autoload and register everything for you.

For the autoloading to work flawlessly you **MUST** adhere to the component's naming conventions and directories. So you need to refer to the `documentation page` of the component when creating it.

Naming Conventions

- Containers names **SHOULD** start with Capital Letters. Use CamelCase to rename Containers.
- Namespace should be the same as the container name (i.e., if container name is `Printer`, the corresponding namespace should be `App\Containers\Printer`).
- Container MAY be named to anything however. A good practice, however, is to name it to its most important `Model` name.

  Example

  If the user story is "*A `User` can create a `Books` and `Books` can have `Comments`*" then you could have 3 Containers (i.e., `User`, `Book`, `Comments`).

4.1.3 The Ship Layer

Read about the `Ship` layer **here**

4.2 MVC

4.2.1 MVC Introduction

Due to the popularity of MVC, and the fact that many developers don't have enough time to learn about new architecture patterns, HiveApi also supports the MVC architecture. That is 97% compatible with the `Laravel MVC`.

Below you will learn how you can build your API on top of HiveApi, using your previous knowledge of the Laravel framework.

4.2.2 Difference between Standard MVC and HiveApi MVC

The Porto architecture, does not replace the MVC architecture, but rather extends it. So `Models`, `Views`, `Routes` and `Controllers` still exist, but in different places with a strict set of responsibilities for each component.

4.2.3 Setup an HiveApi MVC Project

1) First get a fresh version of HiveApi

2) Create the Application

If you open `app/Containers/` you will see a list of containers, whereas each container provide some features for you. However, you don't need to modify them, whether you are using the Porto or MVC architecture. So forget about all these folders for now.

All we need is to create a new folder (i.e., a new `Container`) called `Application` (which holds your MVC application). This is an alternative to the `app` folder on the root of the Laravel project. This folder will hold all your `Models`, `Views`, `Routes`, `Controllers` files, as you know it from a regular Laravel project.

3) Create route file

In Laravel 5.6, the `Route` files live in the `routes/` folder on the root of the project. But in HiveApi MVC, the routes files should live in:

- `app/Containers/Application/UI/API/Routes/` (for API Routes)
- `app/Containers/Application/UI/WEB/Routes/` (for WEB Routes)

Create `api.php` at `app/Containers/Application/UI/API/Routes/api.php` (i.e., Laravels `routes/api.php`) Create `web.php` at `app/Containers/Application/UI/API/Routes/web.php` (i.e., Laravels `routes/web.php`)

In both files create all your endpoints as you would in Laravel.

Heads up!

You must use `$router->` instead of the facade `Route::` in the route files.

Example:

```php
<?php

// Use this `$router` variable instead of Route::
$router->get('/', function () {
    return view('welcome');
});

// DO not use the `Route` facade
Route::get('/', function () {
    return view('welcome');
});
```

4) Create Controller

In Laravel 5.6, the `Controller` classes live in the `app/Http/Controllers/` folder. But in HiveApi MVC, the `Controller` classes should live in:

- `app/Containers/Application/UI/API/Controllers/Controller.php` (to handle API Routes) and **MUST** extend from `App\Ship\Parents\Controllers\ApiController`
- `app/Containers/Application/UI/WEB/Controllers/Controller.php` (to handle WEB Routes) and **MUST** extend from `App\Ship\Parents\Controllers\WebController`

5) Create Models

In Laravel 5.6, the `Model` classes live in the root of the `app/` folder. But in HiveApi MVC, the `Model` classes should live in `app/Containers/Application/Models`.

All model **MUST** `extend` from `App\Ship\Parents\Models\Model`.

> Note the `User` Model should remain in the `User` Container (`app/Containers/User/Models/User.php`), to keep all the features working without any modifications.

6) Create Views

In Laravel 5.6, the `View` files live in the `resources/views/` folder. In HiveApi MVC, the `View` files can live in that same directory or/and in this container folder `app/Containers/Application/UI/WEB/Views/`.

7) Create Transformers

In Laravel 5.6, the `Transformer` classes live in the `app/Transformers/` folder. But in HiveApi MVC, the `Transformer` classes should live in `app/Containers/Application/UI/API/Transformers/`.

Transformers, in turn, **MUST** extend from `App\Ship\Parents\Transformers\Transformer`.

8) Create Service Providers

In Laravel 5.6, the `Service Provider` classes live in the `app/Providers/` folder. But in Hiveapi MVC, the `Service Provider` classes can live in `app/Containers/Application/Providers/`. You can, however, put them anywhere else.

If you want the `Service Providers` to be automatically loaded (without having to register it in the `config/app.php` file), rename your file to `MainServiceProvider.php` (full path `app/Containers/Application/Providers/MainServiceProvider.php`). Otherwise you can create `Service Providers` anywhere and register them manually in Laravels `app.php` configuration file.

9) Create Migrations

In Laravel 5.6, the `Migration` classes live in the `database/migrations/` folder on the root of the project. In HiveApi MVC, the `Migration` classes can live in that same directory or/and in this container folder `app/Containers/Application/Data/Migrations/`.

10) Create Seeds

In Laravel 5.6, the `Database Seeder` files live in the `database/migrations/` folder on the root of the project. In HiveApi MVC, the `Database Seeder` files can live in that same directory or/and in this container folder `app/Containers/Application/Data/Seeders/`.

More Classes

All other class types work the same way, you can refer to the documentation for where to place them and what they should extend. For more details you can always get in touch with us on **Slack**.

4.2.4 How to use HiveApi features

HiveApi features are all provided as `Actions` & `Tasks` classes.

- Each `Action` class has single function `run` which does one feature only.
- Each `Task` class has single function `run` which does one job only (a tiny piece of the business logic).

You can use `Actions/Tasks` classes anyway you want:

- Using HiveApi Facade with HiveApi caller style `$user = \Hive::call('Car@GetDriversAction', [$request->id]);`
- Using HiveApi Facade with full class name `$user = \Hive::call(GetDriversAction::class, [$request->id]);`
- Using the helper `call()` function with full class name `$user = $this->call(GetDriversAction::class, [$request->id]);`
- Using the helper `call()` function with HiveApi caller style `$user = $this->call('Car@GetDriversAction', [$request->id]);`
- Without HiveApi involvement using plain PHP `$user = $action = new GetDriversAction::class; $action->run($request->id);`
- Without HiveApi involvement using plain Laravel IoC `$user = \App::make(GetDriversAction::class)->run($`

Be creative, at the end of the day it's a class with a function.

CHAPTER 5

Requests

5.1 Form Content Types (W3C)

By default HiveApi is configured to encode simple text/ASCII data as `application/json`. However, it does support other types as well.

5.1.1 JSON Payload

To tell the web server that you are sending JSON formatted payload (`{"name" : "John Doe", "age": 25}`), you need to add the `Content-Type : application/json` request header.

5.1.2 ASCII payload

To tell the server that you are sending simple text/ASCII payload (`name=John+Doe&age=25`), you need to add the `Content-Type : x-www-form-urlencoded` request header.

5.2 HTTP Request Headers

Heads Up!

Normally you should include the `accept : application/json` HTTP header when you call a JSON API. However, in HiveApi you can force your users to send `application/json` by setting `'force-accept-header' => true,` in `app/Ship/Configs/hive.php` or allow them to skip it completely by setting the `'force-accept-header' => false,`. By default this flag is set to true.

5.3 Calling Endpoints

5.3.1 Calling unprotected Endpoints (example):

```
curl -X POST -H "Accept: application/json" -H "Content-Type: application/x-www-form-
↪urlencoded; -F "email=admin@admin.com" -F "password=admin" -F "=" "http://api.hive.
↪local/v1/register"
```

5.3.2 Calling protected Endpoints by passing a Bearer Token (example):

```
curl -X GET -H "Accept: application/json" -H "Authorization: Bearer␣
↪eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..." -H "http://api.hive.local/v1/users"
```

CHAPTER 6

Responses

In HiveApi you can define your own response payload or use one of the supported serializers. Currently the supported serializers are (`ArraySerializer`, `DataArraySerializer` and `JsonApiSerializer`) provided by Fractal.

By default HiveApi uses `DataArraySerializer`. Below is an example of the response payload.

```
{
  "data": [
    {
      "id": 100,
      ...
      "relation 1": {
        "data": [ // multiple items
          {
            "id": 11,
              ...
          }
        ]
      },
      "relation 2": {
        "data": { // single item
          "id": 22,
          ...
          }
        }
      }
    },
    ...
  ],
  "meta": {
    "pagination": {
      "total": 999,
      "count": 999,
      "per_page": 999,
```

(continues on next page)

(continued from previous page)

```
      "current_page": 999,
      "total_pages": 999,
      "links": {
        "next": "http://api.hive.local/v1/accounts?page=999"
      }
    }
  },
  "include": [ // the other resources that may be included dynamically on request
    "xxx",
    "yyy"
  ],
  "custom": []
}
```

6.1 Paginated Response:

When the returned data is paginated the response payload will contain a `meta` description with information about the pagination.

```
{
  "meta": {
    "pagination": {
      "total": 999,
      "count": 999,
      "per_page": 999,
      "current_page": 999,
      "total_pages": 999,
      "links": {
        "next": "http://api.hive.local/v1/accounts?page=999"
      }
    }
  },
  "include": [ // what can be included
    "xxx",
    "yyy"
  ],
  "custom": []
}
```

6.2 Including Resources

The `include` field in the response informs the client about relationships that can be include on request. The client, in turn, may tell the server by adding additional query parameter, for example `/users?include=roles`

For more details read the `Relationships` section in the Query Parameters page.

6.3 Change the default Response payload:

By default, HiveApi returns the data by applying the `DataArray` Fractal Serializer (`League\Fractal\Serializer\DataArraySerializer`). To change this behaviour, you can adapt your `.env` file accordingly:

```
API_RESPONSE_SERIALIZER=League\Fractal\Serializer\DataArraySerializer
```

Currently, the supported Serializers are

- `ArraySerializer`
- `DataArraySerializer`
- `JsonApiSerializer`

More details can be found at the Fractal website and Laravel Fractal Wrapper.

In case of returning JSON data (`JsonApiSerializer`), you may also want to check some JSON response standards:

- JSON API (very popular and well documented)
- JSEND (very basic)
- HAL (useful in case of hypermedia)

6.4 Resource Keys

6.4.1 JsonApiSerializer

The selected serializer allows appending a `ResourceKey` to the transformed resource. You can set the `ResourceKey` in your response payload in 2 ways:

1. Manually set it via the respective parameter in the `$this->transform()` call. Note that this will only set the `top level` resource key and does not affect the resource keys from `included` resources!
2. Specify it on the respective `Model` by overriding the `$resourceKey` (`protected $resourceKey = 'FooBar';`). If no `$resourceKey` is defined at `Model` level, the lowercased, pluralized `ShortClassName` is used as key. For example, the `ShortClassName` of `App\Containers\User\Models\User::class` is simply `User`. The resulting `$resourceKey`, therefore, is `users`.

6.4.2 DataArraySerializer

By default the `object` keyword is used as a resource key for each response, and is manually defined in each transformer,

6.5 Error Responses Formats

Visit each feature, example the Authentication and there you will see how an unauthenticated response looks like, same for Authorization, Validation and so on.

6.6 Building a Responses from the Controller

Checkout the Controller Response Builder Helper functions for more information on this topic.

CHAPTER 7

Conventions and Principles

Below you find some naming conventions and basic principles to be used and applied in HiveApi.

7.1 HTTP Methods usage in RESTful APIs

- `GET` (SELECT) to retrieve a specific resource from the server, or a collection of resources.
- `POST` (CREATE) to create a new resource on the server.
- `PUT` (UPDATE) to update a resource on the server, providing the entire resource.
- `PATCH` (UPDATE) to update a resource on the server, providing only changed attributes.
- `DELETE` (DELETE) to remove a resource from the server.

7.2 Naming Conventions for Routes & Actions

- `GetAllResource` to fetch all resources. You can apply `?search` query parameter to filter data.
- `FindResourceById` to search for single resource by its unique identifier.
- `CreateResource` to create a new resource.
- `UpdateResource` to update/edit existing resource.
- `DeleteResource` to delete a resource.

7.3 General Guidelines and Principles for RESTful URLs

- A URL identifies a resource.
- URLs should include nouns, not verbs (verbs are "added" by HTTP methods)

- Use plural nouns only for consistency (no singular nouns).
- Use HTTP verbs (GET, POST, PUT, DELETE) to operate on the collections and elements.
- You should not need to go deeper than resource/identifier/resource.
- Put the version number at the base of your URL, for example `http://api.hive.local/v1/path/to/resource`.
- If an input data changes the logic of the endpoint, it should be passed in the URL. If not you should add it to the header (e.g., like the `Authentication Token`)
- Don't use query parameters to alter state.
- Don't use mixed-case paths if you can help it; lowercase is best.
- Don't use implementation-specific extensions in your URIs (.php, .py, .pl, etc.)
- Limit your URI space as much as possible. And keep path segments short.
- Don't put metadata in the body of a response that should be in a header

7.3.1 Good URL Examples

- Find a single Car by its unique identifier (ID):
  - `GET http://api.hive.local/v1/cars/123`
- Get all Cars:
  - `GET http://api.hive.local/v1/cars`
- Find/Search cars by one or more fields:
  - `GET http://api.hive.local/v1/cars?search=maker:mercedes`
  - `GET http://api.hive.local/v1/cars?search=maker:mercedes;color:white`
- Order and Sort query result:
  - `GET http://api.hive.local/v1/cars?orderBy=created_at&sortedBy=desc`
  - `GET http://api.hive.local/v1/cars?search=maker:mercedes&orderBy=created_at&sortedBy`
- Specify optional fields:
  - `GET http://api.hive.local/v1/cars?filter=id;name;status`
  - `GET http://api.hive.local/v1/cars/123?filter=id;name;status`
- Get all Drivers belonging to a Car:
  - `GET http://api.hive.local/v1/cars/123/drivers`
  - `GET http://api.hive.local/v1/cars/123/drivers/123/addresses`
- Include Drivers objects relationship with the car response:
  - `GET http://api.hive.local/v1/cars/123?include=drivers`
  - `GET http://api.hive.local/v1/cars/123?include=drivers,owner`
- Add new Car:
  - `POST http://api.hive.local/v1/cars`
- Add new Driver to a Car:
  - `POST http://api.hive.local/v1/cars/123/drivers`

7.3.2 General Principles for HTTP Methods

- Don't ever use `GET` to alter state (e.g., to prevent `Googlebot` from corrupting your data).
- Use `GET` as much as possible.
- Don't use `PUT` unless you are updating an entire resource. You should also provide a `GET` on the same URI.
- Don't use `POST` to retrieve information that is long-lived or that might be reasonable to cache.
- Don't perform an operation that is not idempotent with `PUT`.
- Use `GET` for things like calculations, unless your input is large, in which case use `POST`.
- Use `POST` in preference to `PUT` when in doubt.
- Use `POST` whenever you have to do something that feels `RPC`-like.
- Use `PUT` for classes of resources that are larger or hierarchical.
- Use `DELETE` in preference to `POST` to remove resources.

CHAPTER 8

Actions

Read the section in the Porto SAP Documentation (#Actions).

8.1 Rules

- All Actions **MUST** extend `App\Ship\Parents\Actions\Action`.

8.2 Folder Structure

```
app
    Containers
        {container-name}
            Actions
                CreateUserAction.php
                DeleteUserAction.php
                ...
```

8.3 Code Sample

Simplified Example from the `RegisterUserAction`

```php
<?php

namespace App\Containers\User\Actions;

class RegisterUserAction extends Action
{
    public function run(DataTransporter $data): User
```

(continues on next page)

(continued from previous page)

```
    {
        // create user record in the database and return it.
        $user = Hive::call(CreateUserByCredentialsTask::class, [
            $isClient = true,
            $data->email,
            $data->password,
            $data->name,
        ]);

        Mail::send(new UserRegisteredMail($user));

        return $user;
    }
}
```

Heads up!

Instead of passing these parameters `string $email, string $password, string $name, bool $isClient = false` from place to another over and over, consider using the Transporters classes (simple DTOs "Data Transfer Objects"). For more details read the Transporters page.

Injecting each Task in the constructor and then using it below through its property is really boring and the more Tasks you use the worse it gets. So instead you can use the function `call` to call whichever Task you want and pass any parameters to it.

The Action itself was also called using `Hive::call()` from the Controller, triggering `run()` function.

Refer to the Magical Call page for more info and examples on how to properly use the `call()` function.

8.4 Examples

```php
<?php

namespace App\Containers\User\Actions;

use App\Containers\User\Tasks\DeleteUserTask;
use App\Ship\Parents\Actions\Action;

class DeleteUserAction extends Action
{
    public function run($userId)
    {
        return Hive::call(DeleteUserTask::class, [$userId]);
    }
}
```

```php
<?php

namespace App\Containers\Email\Actions;

use App\Containers\Xxx\Tasks\Sample1Task;
use App\Containers\Xxx\Tasks\Sample2Task;
use App\Ship\Parents\Actions\Action;

class DemoAction extends Action
```

(continues on next page)

(continued from previous page)

```
{
    public function run($xxx, $yyy, $zzz)
    {
        $foo = Hive::call(Sample1Task::class, [$xxx, $yyy]);

        $bar = Hive::call(Sample2Task::class, [$zzz]);
    }
}
```

8.4.1 Calling an Action from a Controller

```
<?php

public function deleteUser(DeleteUserRequest $request)
{
    $user = Hive::call(DeleteUserAction::class, [$request->xxx, $request->yyy]);

    return $this->deleted($user);
}
```

The same Action **MAY** be called by multiple Controllers (`API`, `WEB` and `CLI`).

CHAPTER 9

Configuration Files

Configs are files that hold configurations for the specific container. For more details about them check the official Laravel documentation.

In each container, there are two types of config files:

- the container specific config file that contains the container specific configurations.
- the container third-party packages config files (i.e., a config file that belongs to a third-party package, required by the composer file of the container).

9.1 Principles

- Your custom config files and the third-party packages config files, should be placed in the container. If they are too generic then it can be placed on the Ship layer.
- Container can have as many config files as they need.

9.2 Rules

- When publishing a third-party package config file you **SHOULD** move it manually to its respective container or to the `Ship` Config folder.
- Framework config files (provided by Laravel) lives at the default config directory in the root of the project.
- You **SHOULD NOT** add any config file to the `config` directory.
- The container specific config file, **MUST** have the same name of the container in lower letters and post-fixed with `-container`, to prevent conflicts between third-party packages and container specific packages.

9.3 Folder Structure

```
app
    Containers
        {container-name}
            Configs
                {container-name}-container.php
                package-config-file1.php
                ...
    Ship
        Configs
            hashids.php
            hive.php
            ...
config
    app.php
    ...
```

9.4 Code Samples

```
<?php

return [

    /*
    |--------------------------------------------------------------------------
    | Basic Configuration
    |--------------------------------------------------------------------------
    */
    'custom-value' => 'foo',
    'enabled' => true,

    // some other config params here...
```

You can access the respective configuration key like this:

```
$value = Config::get('{container-name}-container.custom-value');     // returns 'foo'
$value = config('{container-name}-container.custom-value');          // same, but␣
↪using a function and not the Facade

$defaultValue = Config::get('{container-name}-container.unknown.key', 'defaultvalue');
↪    // returns 'defaultvalue' as the key is not set!
```

CHAPTER 10

Controllers

Read from the Porto SAP Documentation (#Controllers).

10.1 Rules

- All `APIController` **MUST** extend from `App\Ship\Parents\Controllers\ApiController`.
- All `WebController` **MUST** extend from `App\Ship\Parents\Controllers\WebController`.
- Controllers **SHOULD** use the `call()` function to call an `Action`. Do not manually inject the Action and invoke the `run()` method.
- Controllers **SHOULD** pass the `Transporter` object to the `Action` instead of passing data from the request. The `Transporter` object can be derived from the `Request`. `Transporters` are the best classes to store the state of the `Request` during its lifecycle.

10.2 Folder Structure

```
app
    Containers
        {container-name}
            UI
                API
                    Controllers
                        Controller.php
                WEB
                    Controllers
                        Controller.php
```

10.3 Code Sample

`Controller` for the `User` Container (`WEB`)

```php
<?php

class Controller extends WebController
{
    public function sayWelcome()
    {
        return view('welcome');
    }
}
```

`Controller` for the `User` Container (`API`)

```php
<?php

class Controller extends ApiController
{
    public function registerUser(RegisterUserRequest $request)
    {
        $user = Hive::call(RegisterUserAction::class, [$request->toTransporter()]);

        return $this->transform($user, UserTransformer::class);
    }

    public function deleteUser(DeleteUserRequest $request)
    {
        $user = Hive::call(DeleteUserAction::class, [$request->toTransporter()]);

        return $this->deleted($user);
    }
}
```

Note that `Actions` are called by using `Hive::call()`, which automatically invokes the `run()` function in the `Action` as well inform the action which UI called it, (`$this->getUI()`) in case you wanna handle the same `Action` differently based on the UI type.

The second parameter of the `call()` function is an array of the `Action` parameters in order. When you need to pass data to the Action, it is recommended to pass the `Transporter` object as it should be the place that holds the state of your current request.

Refer to the Magical Call page for more info and examples on how to use the `call()` function.

10.3.1 Calling a Controller function from a Route file

```php
<?php

$router->post('login', [
    'uses' => 'Controller@loginUser',
]);

$router->post('logout', [
    'uses'       => 'Controller@logoutUser',
    'middleware' => [
```

(continues on next page)

(continued from previous page)

```
        'api.auth',
    ],
]);
```

10.4 Controller Response Builder Helper Functions

Many helper function are there to help you build your `Response` faster, those helpers exist in the `vendor/hiveapi/core/src/Traits/ResponseTrait.php`.

10.4.1 Some functions

`transform()` is the most useful function, which you will be using in most cases.

- First required parameter accepts data as `object` or `Collection` of objects.
- Second required parameter is the `Transformer` class to be applied.
- Third (optional) parameter take the `includes` that should be returned by the response *($availableIncludes and $defaultIncludes in the `Transformer` class)*.
- Fourth (optional) parameter accepts meta data to be injected in the `Response`.

```
<?php
// $user is a User Object
return $this->transform($user, UserTransformer::class);

// $orders is a Collection of Order Objects
return $this->transform($orders, OrderTransformer::class, ['products', 'recipients',
↪'store', 'invoice'], ['foo' => 'bar']);
```

`withMeta` allows including meta data in the response.

```
<?php
$metaData = ['total_credits', 10000];

return $this->withMeta($metaData)->transform($receipt, ReceiptTransformer::class);
```

`json` allows passing data (as array) to be represented as json.

```
<?php
return $this->json([
    'foo' => 'bar'
]);
```

Other functions

- accepted
- deleted
- noContent

Some functions might not be documented here, so refer to the `vendor/hiveapi/core/src/Traits/ResponseTrait.php` and check the public functions.

CHAPTER 11

Migration Files

Migration files (short name for Database Migration Files are the *version control* of your database. They are very useful for generating and documenting the database tables.

11.1 Rules

- Migrations **SHOULD** be created inside the respective `Containers`.
- Migrations will be autoloaded by HiveApi
- There is no need to publish the Database Migrations yourself. Just run the `artisan migrate` command and Laravel will read the Migrations from your Containers.

11.2 Structure

```
app
    Containers
        {container-name}
            Data
                Migrations
                    2200_01_01_000001_create_users_table.php
                    2200_01_02_000001_add_fields_to_users_table.php
                    ...
```

11.3 Code Samples

Below is a code sample for the `2200_01_01_000001_create_users_table` migration file.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;

class CreateUsersTable extends Migration
{
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
            $table->softDeletes();
        });
    }

    public function down()
    {
        Schema::drop('users');
    }
}
```

For more information about the Database Migrations read the official Laravel Docs.

CHAPTER 12

Models

Read the official Porto SAP Documentation (#Models).

12.1 Rules

- All Models **MUST** extend from `App\Ship\Parents\Models\Model`.
- If the name of a model differs from the Container name you have to set the `$container` attribute in the repository. More details can be found here.

12.2 Folder Structure

```
app
    Containers
        {container-name}
            Models
                User.php
                Person.php
```

12.3 Code Sample

```php
<?php

namespace App\Containers\Demo\Models;

use App\Ship\Parents\Models\Model;

class Demo extends Model
```

(continues on next page)

(continued from previous page)

```
{
    protected $table = 'demos';

    protected $fillable = [
        'label',
        'user_id'
    ];

    protected $hidden = [
        'token',
    ];

    protected $casts = [
        'total_credits'     => 'float',
    ];

    protected $dates = [
        'created_at',
        'updated_at',
    ];

    public function user()
    {
        return $this->belongsTo(\App\Containes\User\Models\User::class);
    }
}
```

Notice the `Demo` Model has a relationship with the `User` Model, which lives in another Container.

12.4 Casts

The `$casts` attribute can be used to cast any of the model's attributes to a specific type when reading from and writing to the database. In the shown code sample the `total_credits` is casted to `float`.

More information about the available cast-types can be found in the Laravel eloquent-mutators documentation.

Date values can be defined within the `$dates` array to be automatically parsed to a `Carbon` class.

CHAPTER 13

Repositories

The Repository classes are an implementation of the `Repository Design Pattern`. Their major roles are separating the business logic from the data (or the data access Task). Repositories save and retrieve `Models` to/from the underlying storage mechanism (i.e., databases).

The Repository is used to separate the logic that retrieves the data and maps it to a `Model`, from the business logic that works on the `Model`.

13.1 Principles

- Every `Model` **SHOULD** have their own Repository.
- A `Model` **SHOULD** always get accessed through its Repository. You should never directly access the `Model`.

13.2 Rules

- All Repositories **MUST** extend from `App\Ship\Parents\Repositories\Repository`. Extending from this class will give access to functions like `find()`, `create()`, `update()` and much more.
- The name of the Repository **SHOULD** be same like its `Model` name (Model: `Foo` -> Repository: `FooRepository`).
- If a Repository belongs to a Model whose name is not equal to its Container name, then the Repository must set the `$container` property manually like this: `$container = 'ContainerName'`.

13.3 Folder Structure

```
app
    Containers
        {container-name}
            Data
                Repositories
                    UserRepository.php
```

13.4 Code Samples

Example for the `UserRepository`

```php
<?php

namespace App\Containers\User\Data\Repositories;

use App\Containers\User\Contracts\UserRepositoryInterface;
use App\Containers\User\Models\User;
use App\Ship\Parents\Repositories\Repository;

class UserRepository extends Repository
{
    protected $fieldSearchable = [
        'name'  => 'like',
        'email' => '=',
    ];
}
```

13.4.1 Using the Repository

```php
<?php

// paginate the data by 10
$users = $userRepository->paginate(10);

// search by 1 field
$cars = $carRepository->findByField('color', $color);

// searching multiple fields
$offer = $offerRepository->findWhere([
    'offer_id' => $offer_id,
    'user_id'  => $user_id,
])->first();
```

13.4.2 Manually "linking" a Model and its Repository

If the Repository belongs to Model with a name different than its Container name, the Repository class of that Model must manually set the property `$container` and define the Container name.

```php
<?php

namespace App\Containers\Authorization\Data\Repositories;
```

(continues on next page)

(continued from previous page)

```
use App\Ship\Parents\Repositories\Repository;

class RoleRepository extends Repository
{
    protected $container = 'Authorization'; // the container name. Must be set when␣
↪the model has different name than the container

    protected $fieldSearchable = [
    ];
}
```

13.4.3 Other Properties:

API Query Parameters Property

To enable query parameters (`?search=text,...`) in your API you need to set the property `$fieldSearchable` on the Repository class, to instruct the querying on your model.

```
<?php

    protected $fieldSearchable = [
      'name'  => 'like',
      'email' => '=',
    ];
```

All Other Properties

HiveApi uses the `andersao/l5-repository` package, to provide a lot of powerful features to the repository class. To learn more about all the properties you can use, visit the `andersao/l5-repository` package documentation.

CHAPTER 14

Routes

Read from the Porto SAP Documentation (#Routes).

14.1 Rules

- The API Routes files **MUST** be named according to their API version, exposure and functionality. Examples are `CreateOrder.v1.public.php`, `FulfillOrder.v2.public.php`, `CancelOrder.v1.private.php`...
- Web Routes files are pretty similar to API route files but they can be named anything.

14.2 Folder Structure

```
app
    Containers
        {container-name}
            UI
                API
                    Routes
                        CreateItem.v1.public.php
                        DeleteItem.v1.public.php
                        CreateItem.v2.public.php
                        DeleteItem.v1.private.php
                        ApproveItem.v1.private.php
                WEB
                    Routes
                        main.php
                        ...
```

14.2.1 API Routes

Example for the `User Login` API Endpoint

```php
<?php

$router->post('/login', [
    'uses' => 'Controller@loginUser',
]);
```

Example for a protected route to `List All Users` API Endpoint

```php
<?php

$router->get('users', [
    'uses'       => 'Controller@listAllUsers',
    'middleware' => [
        'api.auth', // use the authentication middleware to protect this endpoint!
    ]
]);
```

14.2.2 Difference between Public & Private routes files

HiveApi has two different types of endpoints:

- `Public` (External) endpoints mainly provided for third parties clients, and
- `Private` (Internal) endpoints for your own applications.

This will help generating separate documentations for each type and keep your internal API endpoints private.

14.3 Web Routes

Example Endpoint to display a `Hello View` in the browser

```php
<?php

$router->get('/hello', [
    'uses' => 'Controller@sayHello',
]);
```

In all the Web `Routes` files the `$router` variable is an instance of the default Laravel Router `Illuminate\Routing\Router`.

14.4 Protecting Endpoints:

Checkout the Authorization Page.

CHAPTER 15

SubActions

Read from the Porto SAP Documentation (#Sub-Actions).

15.1 Rules

- All SubActions **MUST** extend from `App\Ship\Parents\Actions\SubAction`.

15.2 Folder Structure

```
app
    Containers
        {container-name}
            Actions
                ValidateAddressSubAction.php
                BuildOrderSubAction.php
                ...
```

15.3 Code Sample

`ValidateAddressSubAction`

```
<?php

namespace App\Containers\Shipment\Actions;

use App\Containers\Recipient\Models\Recipient;
use App\Containers\Recipient\Tasks\UpdateRecipientTask;
use App\Containers\Shipment\Tasks\ValidateAddressWithEasyPostTask;
```

(continues on next page)

(continued from previous page)

```
use App\Containers\Shipment\Tasks\ValidateAddressWithMelissaDataTask;
use App\Ship\Parents\Actions\SubAction;

class ValidateAddressSubAction extends SubAction
{
    public function run(Recipient $recipient)
    {
        $hasValidAddress = true;

        $easyPostResponse = Hive::call(ValidateAddressWithEasyPostTask::class, [
↪$recipient]);

        // ...
    }
}
```

Heads up!

Every feature available for `Actions`, is also available for `SubActions`.

CHAPTER 16

Tasks

Read from the Porto SAP Documentation (#Tasks).

16.1 Rules

- All Tasks **MUST** extend from `App\Ship\Parents\Tasks\Task`.

16.2 Folder Structure

```
app
    Containers
        {container-name}
            Tasks
                ConfirmUserEmailTask.php
                GenerateEmailConfirmationUrlTask.php
                SendConfirmationEmailTask.php
                ValidateConfirmationCodeTask.php
                SetUserEmailTask.php
                ...
```

16.3 Code Sample

Simplified example for `FindUserByIdTask`:

```php
<?php

namespace App\Containers\User\Tasks;

```

(continues on next page)

(continued from previous page)

```php
use App\Containers\User\Data\Repositories\UserRepository;
use App\Ship\Exceptions\NotFoundException;
use App\Ship\Parents\Tasks\Task;
use Exception;

class FindUserByIdTask extends Task
{
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function run($id)
    {
        try {
            return $this->userRepository->find($id);
        } catch (Exception $e) {
            throw new NotFoundException();
        }
    }
}
```

16.3.1 Calling the Task from an Action

```php
<?php

namespace App\Containers\User\Actions;

use App\Containers\User\Models\User;
use App\Containers\User\Tasks\FindUserByIdTask;
use App\Ship\Exceptions\NotFoundException;
use App\Ship\Parents\Actions\Action;
use App\Ship\Transporters\DataTransporter;
use HiveApi\Core\Foundation\Facades\Hive;

class FindUserByIdAction extends Action
{
    public function run(DataTransporter $data): User
    {
        $user = Hive::call(FindUserByIdTask::class, [$data->id]);

        return $user;
    }
}
```

Note that the `Task` is called via the `Hive` Facade. All `Controllers`, `Actions`, `Tasks` and `SubActions`, however, also implement the `CallableTrait`, which allows you to directly `call()` another class via `$this->call(Classname::class)`.

CHAPTER 17

Transformers

Read from the Porto SAP Documentation (#Transformers).

17.1 Rules

- All API responses **MUST** be formatted via a `Transformer`.
- Every Transformer **SHOULD** extend from `App\Ship\Parents\Transformers\Transformer`.
- Each Transformer **MUST** have a `transform()` function.

17.2 Folder Structure

```
app
    Containers
        {container-name}
            UI
                API
                    Transformers
                        UserTransformer.php
```

17.3 Code Samples

```
<?php

namespace App\Containers\Item\UI\API\Transformers;

use App\Containers\Item\Models\Item;
```

(continues on next page)

(continued from previous page)

```php
use App\Ship\Parents\Transformers\Transformer;

class ItemTransformer extends Transformer
{

    protected $availableIncludes = [
        'images',
    ];

    protected $defaultIncludes = [
        'roles',
    ];

    public function transform(Item $item)
    {
        $response = [
            'object'      => 'Item',
            'id'          => $item->getHashedKey(),
            'name'        => $item->name,
            'description' => $item->description,
            'price'       => $item->price,
            'weight'      => $item->weight,
            'created_at'  => $item->created_at,
            'updated_at'  => $item->updated_at,
        ];

        return $response;
    }

    public function includeImages(Item $item)
    {
        return $this->collection($item->images, new ItemImageTransformer());
    }

    public function includeRoles(User $user)
    {
        return $this->collection($user->roles, new RoleTransformer());
    }
}
```

17.3.1 Using a Transformer to Return Data from a Controller

```php
<?php

    public function getAllClients(GetAllUsersRequest $request)
    {
        $users = Hive::call(GetAllClientsAction::class);

        return $this->transform($users, UserTransformer::class);
    }
```

You can even pass a `Transformer` object to the `transform()` method, like so:

```php
<?php
```

(continues on next page)

(continued from previous page)

```
    public function getAllClients(GetAllUsersRequest $request)
    {
        $users = Hive::call(GetAllClientsAction::class);

        $transformer = new MyCustomUserTransformer(true, 'foo', 4711);

        return $this->transform($users, $transformer);
    }
```

The parameters are passed to the `Transformer` via the `__construct()` constructor and can be used to parametrize the actual `transform()` method (e.g., based on specific flags).

17.4 Relationships (Includes)

Loading relationships with the Transformer (calling other Transformers) can be done in 2 ways:

1. The `Client` can specify the relationships to be included via Query Parameters.
2. The `Developer` can define relationships to be automatically included.

17.4.1 Apply Relationships via Query Parameters

The clients can request data with their relationships directly when calling the API by adding the `?include=x` query parameter. The Transformer, in turn, needs to have the `availableIncludes` defined with their functions like this:

```
<?php

namespace App\Containers\Account\UI\API\Transformers;

use App\Ship\Parents\Transformers\Transformer;
use App\Containers\Account\Models\Account;
use App\Containers\Tag\Transformers\TagTransformer;
use App\Containers\User\Transformers\UserTransformer;

class AccountTransformer extends Transformer
{
    protected $availableIncludes = [
        'tags',
        'user',
    ];

    public function transform(Account $account)
    {
        return [
            'id'       => $account->id,
            'url'      => $account->url,
            'username' => $account->username,
            'secret'   => $account->secret,
            'note'     => $account->note,
        ];
    }

    public function includeTags(Account $account)
    {
```

(continues on next page)

(continued from previous page)

```
        return $this->collection($account->tags, new TagTransformer());
    }

    public function includeUser(Account $account)
    {
        return $this->item($account->user, new UserTransformer());
    }
}
```

In order to get the `Tags` with the response when `Accounts` are requested, the clients needs to pass the `?include=tags` parameter with the `GET` request. To get Tags with User use the a comma separated list `?include=tags,user`.

17.4.2 Apply Relationships from Application Code

From the controller you can dynamically set the `DefaultInclude`:

```
<?php

    public function getAllClients(GetAllUsersRequest $request)
    {
        $users = Hive::call(GetAllClientsAction::class);

        return $this->transform($users, UserTransformer::class, ['tags', 'account']);
    }
```

You need to have `includeTags()` and `includeAccount()` functions defined on the transformer. If you want to include a relation with every response from this transformer you can define the relation directly in the transformer by adding it to the `$defaultIncludes`.

```
<?php

    protected $availableIncludes = [
        'users',
    ];

    protected $defaultIncludes = [
        'tags',
    ];

    // ..
```

17.5 Helper Functions for Transformers

- `user()` : returns the currently authenticated `User`.
- `ifAdmin($adminResponse, $clientResponse)` : merges normal client response with the admin extra or modified results, when current authenticated user is an `admin` user.

For more information about the Transformers read the official package documentation.

CHAPTER 18

Transporters

Transporters is a name chosen by HiveApi for DTO's (Data Transfer Objects). The latter are used to pass user data (coming from `Requests`, `Commands`, or other components) from one place to another (`Actions` to `Tasks` / `Controller` to `Action` / `Command` to `Action` / ...).

They are very useful for reducing the number of parameters in functions, which prevents the duplication of long parameters.

HiveApi relies on this third-party package as DTO. Refer to the dto package wiki for more details.

18.1 Rules

- All Transporters **MUST** extend from `App\Ship\Parents\Transporters\Transporter`.

18.2 Folder Structure

```
app
    Containers
        {container-name}
            Data
                Transporters
                    CreateUserTransporter.php
```

18.3 Code Sample

```
<?php

namespace App\Containers\Authentication\Transporters;
```

(continues on next page)

(continued from previous page)

```
use App\Ship\Parents\Transporters\Transporter;

class ProxyApiLoginTransporter extends Transporter
{

    /**
     * @var array
     */
    protected $schema = [
        'properties' => [
            'email',
            'password',
            'client_id',
            'client_password',
            'grant_type',
            'scope',
        ],
        'required'   => [
            'email',
            'password',
            'client_id',
            'client_password',
        ],
        'default'    => [
            'scope' => '',
        ]
    ];
}
```

18.3.1 Using a Transporter within a Controller

Normally you would use it like this

```
<?php
$dataTransporter = new DataTransporter($request);
$dataTransporter->bearerToken = $request->bearerToken();

Hive::call(ApiLogoutAction::class, [$dataTransporter]);
```

Since this example above has some required data, that data must be sent to the constructor:

```
<?php
$dataTransporter = new ProxyApiLoginTransporter(
    array_merge($request->all(), [
        'client_id'       => Config::get('authentication-container.clients.web.admin.
↪id'),
        'client_password' => Config::get('authentication-container.clients.web.admin.
↪secret')
    ])
);

$result = Hive::call(ProxyApiLoginAction::class, [$dataTransporter]);
```

18.3.2 Creating a Transporter for Tests

```php
<?php

$data = [
    'foo' => 'bar'
];

$transporter = new DataTransporter($data);
$action = App::make(RegisterUserAction::class);

$user = $action->run($transporter);
```

18.4 Automatically Transforming a Request to a Transporter

If you want to directly transform a `Request` to a `Transporter` you can simply call

```
$transporter = $request->toTransporter();
```

This method does take the `protected $transporter` of the `Request` class into account. If none is defined, a regular `DataTransporter` will be created.

Note, that `$transporter` will now have all fields from `$request` - so you can directly access them. In order to do so, you can call:

```php
<?php
// "simple" access via direct properties
$name = $transporter->name;

// complex access via method
$username = $transporter->getInputByKey('your.nested.username.field');
```

Of course, you can also "sanitize" the data, like you would have done in the `Request` classes by using `sanitizeData(array)`. Finally, if you need to access the original `Request` object, you can access it via

```
$originalRequest = $transporter->request;
```

18.5 Data Access

18.5.1 Set Data

You can set data of a Transporter in many ways

```
$dataTransporter = new DataTransporter($request);
$dataTransporter->bearerToken = $request->bearerToken();
```

If the data is defined as required like this on the Transporter:

```php
<?php
    protected $schema = [
        'type' => 'object',
        'properties' => [
```

(continues on next page)

(continued from previous page)

```
            'email',
            'password',
            'clientId',
            'clientPassword',
        ],
        'required'   => [
            'email',
            'password',
            'clientId',
            'clientPassword',
        ],
    ];
```

Then can set data on the Transporter like this:

```
$dataTransporter = new ProxyApiLoginTransporter(
    array_merge($request->all(), [
        'clientId'       => Config::get('authentication-container.clients.web.admin.id
↪'),
        'clientPassword' => Config::get('authentication-container.clients.web.admin.
↪secret')
    ])
);
```

18.5.2 Get Data

To get all data from the Transporter you can call `$data->toArray()` or `$data->toJson()`. There are many other functions on the class. To get specific data just call the data name, as you would when accessing data from a Request object `$data->username`.

18.6 Instance Access

18.6.1 Set Instances

Passing Objects does not work, because the third-party package cannot hydrate it. In order to pass an instances from one place to another within a Transporter object, you can do the following:

```
$transporter = new DataTransporter();
$transporter->setInstance("command_instance", $this);
```

Heads up!

Although you can set instances this way, they do not appear when calling `toArray()` or other similar functions, since they cannot be hydrated. See below how you can get the instance form the Transporter object.

18.6.2 Get Instances

```
$console = $data->command_instance;
```

CHAPTER 19

Views

Read from the Porto SAP Documentation (#Views).

19.1 Rules

- Views **SHOULD** be created inside the containers and, in turn, will be automatically available for use in the `WebControllers`.
- All Views are automatically namespaced with the lowercase name of the container.

19.2 Folder Structure

```
app
    Containers
        {container-name}
            UI
                WEB
                    Views
                        welcome.blade.php
                        profile.blade.php
```

19.3 Code Sample

Take a look at the `Welcome` page, that looks like this (simplified example!)

```
<!DOCTYPE html>
<html>
<head>
```

(continues on next page)

(continued from previous page)

```
    <title>Welcome</title>
</head>
<body>
<div class="container">
    <div class="content">
        <div class="title">Welcome</div>
    </div>
</div>
</body>
</html>
```

This view can be used within a `WebController` like this:

```php
<?php

namespace App\Containers\Welcome\UI\WEB\Controllers;

use App\Ship\Parents\Controllers\WebController;

class Controller extends WebController
{
    public function sayWelcome()
    {
        return view('welcome');
    }
}
```

19.4 Namespaces

By default all the views are namespaced to the lowercase name of their respective container. For example, if a Container is named `Store` and has a View `product-details`, you can access the view like this `view('store::product-details')`. If you try to access a view without the namespace (for example `view('just-welcome')`), it will not find your view.

CHAPTER 20

Commands

A command

- is a Laravel `artisan` command. Laravel has it's own default commands and you create your own application-specific commands as well.
- provides a way to interact with the Laravel application.
- can be scheduled by a Task scheduler, like CronJob or by the Laravel built in wrapper of the Cron Job "laravel scheduler".
- could be Closure based or Class based.
- "dispatch" is the term that is usually used to call a Command.

20.1 Principles

- Containers **MAY** or **MAY NOT** have one or more Commands.
- Every Command **SHOULD** call an `Action` to perform its job.
- Commands itself **SHOULD NOT** contain any business logic.
- The `Ship` layer **MAY** contain application wide commands.

20.2 Rules

- All Commands **MUST** extend from `App\Ship\Parents\Commands\ConsoleCommand`.

20.3 Folder Structure

```
app
    Containers
        {container-name}
            UI
                CLI
                    Commands
                        SayHelloCommand.php
                        ...
    Ship
        Commands
            GeneralCommand.php
            ...
```

20.4 Code Samples

```php
<?php

namespace App\Containers\Welcome\UI\CLI\Commands;

use App\Ship\Parents\Commands\ConsoleCommand;

class SayHelloCommand extends ConsoleCommand
{

    protected $signature = 'hive:welcome';

    protected $description = 'Just saying "Hi"';

    public function handle()
    {
        $this->info('Welcome to HiveApi'); // green color
        $this->line('Welcome to HiveApi'); // normal color
    }
}
```

20.4.1 Calling a Command from the Console

You can call your custom commands like any other artisan command:

```
php artisan hive:welcome
```

20.4.2 Calling a Command from your Application

You can call a command from your own application code like this:

```php
<?php
Artisan::call('hive:welcome');
```

20.4.3 Schedule Commands Execution

To schedule the execution of a Command checkout the Tasks Scheduling page.

20.5 Define Consoles Routes

To define Console route go to `app/Ship/Commands/Routes.php`.

CHAPTER 21

Criteria

Criteria are classes used to hold and apply query condition when retrieving data from the database through a `Repository`. Without using a Criteria class, you can add your query conditions to a Repository or to a Model as scope. However, by using Criteria, your query conditions can be shared across multiple Models and Repositories. It allows you to define the query condition once and use it anywhere in the App.

21.1 Principles

- Every Container **MAY** have its own Criteria. However, shared Criteria **SHOULD** be created in the `Ship` layer.
- A Criteria **MUST NOT** contain any extra code, if it needs data, the data **SHOULD** be passed to it from the `Action` or `Task`. It **SHOULD NOT** run (i.e., execute) any Task for data.

21.2 Rules

- All Criteria **MUST** extend from `App\Ship\Parents\Criterias\Criteria`.
- Every Criteria **SHOULD** have an `apply()` function.
- A simple query condition example `"where user_id = $id"`, this can be named `ThisUserCriteria`, and used with all Models who has relations with the `User` Model.

21.3 Folder Structure

```
app
    Containers
        {container-name}
            Data
                Criterias
```

(continues on next page)

(continued from previous page)

```
                    ColourRedCriteria.php
                    RaceCarsCriteria.php
                    ...
    Ship
        Criterias
            Eloquent
                CreatedTodayCriteria.php
                NotNullCriteria.php
                ...
```

21.4 Code Samples

Example for a shared Criteria (in the `Ship` layer)

```php
<?php

namespace App\Ship\Criterias\Eloquent;

use App\Ship\Parents\Criterias\Criteria;
use Prettus\Repository\Contracts\RepositoryInterface as PrettusRepositoryInterface;

class NotNullCriteria extends Criteria
{
    private $field;

    public function __construct($field)
    {
        $this->field = $field;
    }

    public function apply($model, PrettusRepositoryInterface $repository)
    {
        return $model->whereNotNull($this->field);
    }
}
```

21.4.1 Calling a Criteria from within a `Task`

```php
<?php

public function run()
{
    $this->userRepository->pushCriteria(new NotNullCriteria('email'));

    return $this->userRepository->paginate();
}
```

For more information about Criteria read the official package documentation.

CHAPTER 22

Events

Events

- provide a simple `Observer` implementation, allowing you to subscribe and listen for various events that occur in your application.
- are classes that can be fired from anywhere in your application.
- will usually be bound to one or many `Events Listeners` classes or has those Listeners registered to listen to it.
- "fire" is the term that is usually used to call an Event.

More details can be found at the official Laravel documentation.

22.1 Principles

- Events **CAN** be fired from `Actions` or `Tasks`. They are preferable to choose one place only (`Tasks` are recommended).
- Events **SHOULD** be created inside the Containers. However, generic Events **CAN** be created in the Ship layer.

22.1.1 Rules

- All Events **MUST** extend from `App\Ship\Parents\Events\Event`.

22.1.2 Folder Structure

```
app
    Containers
        {container-name}
            Events
```

(continues on next page)

(continued from previous page)

```
                SomethingHappenedEvent.php
            Listeners
                ListenToMusicListener.php
    Ship
        Events
            GlobalStateChanged.php
            SomethingBigHappenedEvent.php
```

22.2 Enabling Events

Before you can use events you need to add the `EventServiceProvider` to the `MainServiceProvider` of the Ship (if this has not been registered so far). See example below.

```
<?php

namespace App\Containers\Car\Providers;

class MainServiceProvider extends MainProvider
{

    /**
     * Container Service Providers.
     * @var array
     */
    public $serviceProviders = [
        EventServiceProvider::class,
    ];

    // ...
}
```

22.3 Usage

In Laravel you can create and register events in multiple way. Below is an example of an Event that handles itself.

```
<?php

namespace App\Containers\User\Events;

use App\Containers\User\Models\User;
use App\Ship\Parents\Events\Event;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Support\Facades\Log;

class UserRegisteredEvent extends Event implements ShouldQueue
{
    protected $user;

    public function __construct(User $user)
    {
```

(continues on next page)

(continued from previous page)

```
        $this->user = $user;
    }

    public function handle()
    {
        Log::info('New User registration. ID = ' . $this->user->getHashedKey() . ' |␣
↪Email = ' . $this->user->email . '.');
        // ...
    }

    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}
```

You will get more benefits creating `Events Listeners` for each Event. To do this you will need to extend this EventsProvider `HiveApi\Core\Abstracts\Providers\EventsProvider`.

Your custom `EventServiceProvider` needs to be registered in the containers `MainServiceProvider` as well.

```
<?php

namespace App\Containers\Car\Providers;

use App\Ship\Parents\Providers\MainProvider;

/**
 * Class MainServiceProvider.
 * The Main Service Provider of this container, it will be automatically registered␣
↪in the framework.
 */
class MainServiceProvider extends MainProvider
{

    /**
     * Container Service Providers.
     * @var array
     */
    public $serviceProviders = [
        EventServiceProvider::class,
    ];

    // ...
}
```

22.3.1 Dispatch Events

You can dispatch an Event from anywhere you want (ideally from `Actions` and `Tasks`). Consider the following example for dispatching the Event class from the example above.

```
<?php

// using helper function
```

(continues on next page)

(continued from previous page)

```
event(New UserEmailChangedEvent($user));

// manually
\App::make(\Illuminate\Contracts\Bus\Dispatcher\Dispatcher::class)->dispatch(New␣
↪UserEmailChangedEvent($user));
```

22.3.2 Queueing an Event

Events can implement `Illuminate\Contracts\Queue\ShouldQueue` to be queued.

22.3.3 Handling an Event

You can handle jobs on dispatching an event. To do so you need to implement one of the following interfaces:

- `HiveApi\Core\Abstracts\Events\Interfaces\ShouldHandleNow`
- `HiveApi\Core\Abstracts\Events\Interfaces\ShouldHandle`

This will force you to implement the `handle` method and will make HiveApi execute the method upon dispatching the event.

- The `ShouldHandleNow` Interface will make the event execute the handle method as soon as the event gets dispatched.
- The `ShouldHandle` Interface will create an event job and execute the handle method async (through Laravel jobs).

```
<?php

namespace App\Containers\Example\Events;

use HiveApi\Core\Abstracts\Events\Interfaces\ShouldHandle;
use App\Ship\Parents\Events\Event;

class ExampleEvent extends Event implements ShouldHandle
{
    /**
     * If ShouldHandle interface is implemented this variable
     * sets the time (in seconds or timestamp) to wait before a job is executed
     *
     * @var \DateTimeInterface|\DateInterval|int|null $jobDelay
     */
    public $jobDelay = 60;

    /**
     * If ShouldHandle interface is implemented this variable
     * sets the name of the queue to push the job on
     *
     * @var string $jobQueue
     */
    public $jobQueue = "example_queue";

    public function handle()
    {
        // Do some handling here
```

(continues on next page)

(continued from previous page)

```
    }
}
```

22.3.4 Broadcasting

To define Broadcasting route go to `app/Ship/Boardcasts/Routes.php`.

CHAPTER 23

Exceptions

Exceptions are classes the handles errors, and helps developers debug their code in a more efficient way.

23.1 Principles

- Exceptions **CAN** be thrown from anywhere in the application.
- Exceptions **SHOULD** be created inside the Containers. However, generic Exceptions **CAN** be created in the `Ship` layer.

23.2 Rules

- All Exceptions **MUST** extend `App\Ship\Parents\Exceptions\Exception`.
- Shared (generic) Exceptions between all Containers **SHOULD** be created in the `Ship` (i.e., `app/Ship/Exceptions/*`).
- Every Exception **SHOULD** have two properties `httpStatusCode` and `message`. Both properties will be displayed when an error occurs. You can override those values while throwing the error.

23.3 Folder Structure

```
app
    Containers
        {container-name}
            Exceptions
                AccountFailedException.php
    Ship
        Exceptions
```

(continues on next page)

(continued from previous page)

```
            IncorrectIdException.php
            InternalErrorException.php
```

23.4 Code Samples

```php
<?php

namespace App\Containers\User\Exceptions;

use App\Ship\Parents\Exceptions\Exception;
use Symfony\Component\HttpFoundation\Response;

class AccountFailedException extends Exception
{
    public $httpStatusCode = Response::HTTP_CONFLICT;

    public $message = 'Failed creating new User.';

    public $code = 4711;
}
```

```php
<?php

namespace App\Ship\Exceptions;

use App\Ship\Parents\Exceptions\Exception;
use Symfony\Component\HttpFoundation\Response as SymfonyResponse;

class InternalErrorException extends Exception
{
    public $httpStatusCode = SymfonyResponse::HTTP_INTERNAL_SERVER_ERROR;

    public $message = 'Something went wrong!';
}
```

You can also add custom data to your `Exception`.

```php
<?php

namespace App\Ship\Exceptions;

use App\Ship\Parents\Exceptions\Exception;
use Symfony\Component\HttpFoundation\Response as SymfonyResponse;

class AwesomeExceptionWithCustomData extends Exception
{
    public $httpStatusCode = SymfonyResponse::HTTP_INTERNAL_SERVER_ERROR;

    public $message = 'Something went wrong!';

    public $code = 1234;

    /*
```

(continues on next page)

(continued from previous page)

```
     * Everything you add here will be automatically added to the ExceptionFormatter␣
↪on the top level!
     * You can define any structure you want or maybe include translated messages
     */
    public function addCustomData() {
        return [
            'title' => 'nice',
            'description' => 'one fancy description here',
            'foo' => true,
            'meta' => [
                'bar' => 1234,
            ]
        ];
    }
}
```

23.4.1 Throwing an Exception in your Application

```
<?php

throw new AccountFailedException();
```

23.5 Usage

23.5.1 With Log for Debugging:

```
<?php

throw (new AccountFailedException())->debug($e); // debug() accepts string or␣
↪\Exception instance
```

23.5.2 Overriding the default `message`:

```
<?php

throw new AccountFailedException('I am the message to be displayed for the user');
```

23.5.3 Overwriting pre-set Custom Data

```
<?php

throw (new AwesomeExceptionWithCustomData())->overrideCustomData(['foo' => 'bar']);
```

23.6 Application Error Codes

HiveApi provides a convenient way to manage all `application error codes` in one place. Therefore, HiveApi provides, amongst others, the `\App\Ship\Exceptions\Codes\ApplicationErrorCodesTable` class, which already holds various information for multiple errors.

Thereby, one error look like this:

```php
<?php
const BASE_GENERAL_ERROR = [
    'code' => 1001,
    'title' => 'Unknown / Unspecified Error.',
    'description' => 'Something unexpected happened.',
];
```

Note that the `code` is used to be sent back to the client. The `title` and `description`, however, can be used to automatically generate a documentation regarding all defined error codes and their meaning. Please note that this feature is currently not implemented but will be added later on.

23.6.1 Linking Exceptions and Error Codes

In order to link an `error code` to an `Exception`, you simply need override the `useErrorCode()` method of the `Exception`.

Consider the following example:

```php
<?php
class InternalErrorException extends Exception
{
    public $httpStatusCode = SymfonyResponse::HTTP_INTERNAL_SERVER_ERROR;

    public $message = 'Something went wrong!';

    public $code = 4711; // this code will be overwritten by the useErrorCode()␣
↪method!

    public function useErrorCode()
    {
        return ApplicationErrorCodes::BASE_INTERNAL_ERROR;
    }
}
```

Please note that already defined `$code` values may be overwritten by the `useErrorCode()` method! Furthermore, this feature is completely optional - you may still use the known `public $code = 4711;` approach to manually set an error code.

23.6.2 Defining Own Error Code Tables

Of course, HiveApi allows you to define your own `CustomErrorCodesTable`. In fact, there already exists such a file where you can define your own error codes. Please note that the `ApplicationErrorCodesTable` may be adapted by HiveApi - the others will not.

If you like to split the errors in various files, you can easily create a `UserErrorCodesTable` in respective namespace and define the errors accordingly. However, you need to manually "register" this code table. This can be achieved in the `ErrorCodeManager::getCodeTables()` method.

Now you can easily use your `UserErrorCodesTable::USER_NOT_VERIFIED` error in your `Exception` class.

CHAPTER 24

Exception Formatters

In HiveApi you can format any `Exception` response the way you want, by using the `ExceptionFormatters`. They act similar as `Transformers` but work on `Exception` instead of "normal" objects.

HiveApi uses Heimdal, which allows you to format your API exceptions responses using Formatter classes. For more details visit the official package documentation.

By default, HiveApi provides some basic `ExceptionFormatters` for outputting `Exceptions` in an appropriate format. These Formatters, however, can by modified to your specific needs. For example, in case using the `JSON API` payloads, you may change the provided formatters to return JSON API Error response.

24.1 Rules

- All Formatters **MUST** extend from `HiveApi\Core\Exceptions\Formatters\ExceptionsFormatter`.

24.2 Folder Structure

```
app
    Ship
        Exceptions
            Formatters
                HttpExceptionFormatter.php
                ExceptionFormatter.php
                - ...
```

24.3 Code Sample

```
<?php

namespace App\Ship\Exceptions\Formatters;

use HiveApi\Core\Exceptions\Formatters\ExceptionsFormatter as CoreExceptionsFormatter;
use Exception;
use Illuminate\Http\JsonResponse;

class AuthorizationExceptionFormatter extends CoreExceptionsFormatter
{
    CONST STATUS_CODE = 403;

    public function responseData(Exception $exception, JsonResponse $response)
    {
        return [
            'code'        => $exception->getCode(),
            'message'     => $exception->getMessage(),
            'errors'      => 'You have no access to this resource!',
            'status_code' => self::STATUS_CODE,
        ];
    }

    function modifyResponse(Exception $exception, JsonResponse $response)
    {
        return $response;
    }

    public function statusCode()
    {
        return self::STATUS_CODE;
    }
}
```

- The `responseData()` is where you format the response. This is similar to the `transform()` function in `Transformers`.
- The `STATUS_CODE` is the status code which will be sent in header. (`status_code` could be the same as the header code).
- The `modifyResponse()` allows you to alter the response when needed. Example:

```
<?php

    public function modifyResponse(Exception $exception, JsonResponse $response)
    {
        // append exception headers to the response headers.
        if (count($headers = $exception->getHeaders())) {
            $response->headers->add($headers);
        }

        return $response;
    }
```

24.4 Creating Your Own Formatter

You can create and add your own Formatters (or override existing ones) at any time. All Formatters live in `App/Ship/Exceptions/Formatters`. By default, HiveApi provides formatters to format basic Exceptions (or HTTP Exceptions) as well as "common" Exceptions like `AuthenticationException` and so on.

24.4.1 Registering Your Formatters

In order to inform HiveApi to use your new `AwesomeExceptionFormatter` you need to `register` it. This can be done in the `App/Ship/Configs/optimus.heimdal.php` configuration file. Take a look at the `optimus.heimdal.formatters` key. This array defines a `key-value` list that declares a mapping between an `Exception` class and the corresponding `Formatter`.

Say, you want to `register` your newly created `AwesomeExceptionFormatter` for all `HttpExceptions` add a new line to the **top** of this array, like so:

```
'formatters' => [
    SymfonyException\HttpException::class =>␣
↪\Your\Custom\Namespace\AwesomeExceptionFormatter::class,

    // the already defined exception formatters from HiveApi
    // ...
]
```

Please note that the order of the formatters matter. When throwing an `Exception` with `throw new XException()` the first Formatter that matches respective criteria is used to format the `Exception`. In the respective example, your newly created `AwesomeExceptionFormatter` would be applied to format and output the Exception to the client.

CHAPTER 25

Factories

Factories (short name for Models Factories) are used to generate fake data with the help of `Faker` to be used for testing purposes. Factories are mainly used from Tests.

25.1 Rules

- Factories **SHOULD** be created in the containers.
- A Factory is just a plain PHP script. There are no classes or namespaces required.

25.2 Folder Structure

```
app
    Containers
        {container-name}
            Data
                Factories
                    UserFactory.php
                    ...
```

25.3 Code Samples

```php
<?php

// User
$factory->define(App\Containers\User\Models\User::class, function (Faker\Generator
↪$faker) {
    return [
```

(continues on next page)

(continued from previous page)

```
        'name'      => $faker->name,
        'email'     => $faker->email,
    ];
});
```

25.3.1 Calling the Factory from a Test Class

```
<?php

// creating 4 users
factory(User::class, 4)->create();
```

25.3.2 Example with Relationships

```
<?php

$countries = Country::all();

// creating 3 rewards and attaching country relation to them
$rewards = factory(Reward::class, 3)->make()->each(function ($reward) use (
↪$countries) {
    $reward->save();
    $reward->countries()->attach([$countries->random(1)->id, $countries->random(1)->
↪id]);
    $reward->save();
});
```

Use make instance of `create()` and pass any data you want, then `save()` after establishing the relationship.

25.3.3 Usage while overriding some values

```
<?php

// creating single Offer and setting a user id
$offer = factory(Offer::class)->make();
$offer->user_id = $user->id;
$offer->save();

// ANOTHER EXAMPLE:
// creating multiple Accounts
$users = factory(Account::class, 3)->make()->each(function ($account) use ($user) {
    $account->user_id = $user->id;
    $account->save();
});
```

For more information about the Model Factories read the official Laravel documentation.

CHAPTER 26

Jobs

A Job

- is a simple class that can execute one specific task.
- is a name given to a class that is usually created to be queued (its execution is usually deferred for later, after the execution of previous Jobs are completed).
- can be scheduled to be executed later by a queuing mechanism (queue system like beanstalkd).
- class is dispatched, it performs its specific job and dies.
- Laravel's queue worker will process every Job as it is pushed onto the queue.

More information can be found in the official Laravel documentation.

26.1 Rules

- All Jobs **MUST** extend from `App\Ship\Parents\Jobs\Job`.
- A Container **MAY** have more than one Job.

26.2 Folder Structure

```
app
    Containers
        {container-name}
            Jobs
                DoSomethingJob.php
                DoSomethingElseJob.php
```

26.3 Code Samples

`CreateAndValidateAddressJob`

```php
<?php

namespace App\Containers\Shipment\Jobs;

use App\Port\Job\Abstracts\Job;

class CreateAndValidateAddressJob extends Job
{
    private $recipients;

    public function __construct(array $recipients)
    {
        $this->recipients = $recipients;
    }

    public function handle()
    {
        foreach ($this->recipients as $recipient) {
            // do whatever you like
        }
    }
}
```

26.3.1 Calling a Job from an Action

```php
<?php

// using helper function
dispatch(new CreateAndValidateAddressJob($recipients));

// manually
App::make(\Illuminate\Contracts\Bus\Dispatcher\Dispatcher::class)->dispatch(new␣
↪CreateAndValidateAddressJob($recipients));
```

26.3.2 Execute Jobs

For running your Jobs checkout the Tasks Queuing page.

CHAPTER 27

Languages

Languages are not real components, but rather serve as additional resources to a specific container. More specifically, these files hold translations.

27.1 Rules

- Language files **CAN** be placed inside the containers. However, the default laravel `resources/lang` languages files are still loaded and can be used as well.
- All translations are automatically namespaced as the lowercase name of the container.

27.2 Folder Structure

```
app
    Containers
        {container-name}
            Resources
                Languages
                    en
                        messages.php
                        users.php
                    de
                        messages.php
                        users.php
```

27.3 Usage

To get a translation from a specific container, call it like this:

```php
<?php

trans('welcome::messages.headline.title');
```

where `welcome` is the name of the container to search for the localization file, `messages` is the actual file to search for, and `headline.title` is the localization key to be resolved within this file.

For more info about the localization checkout the Localization page.

CHAPTER 28

Mails

The Mail component allows you to define an email and send it whenever needed. For more details refer to the officiial Laravel documentation.

28.1 Principles

- Containers **MAY** or **MAY NOT** have one or more Mail.
- The Ship **MAY** contain general Mails.

28.2 Rules

- All Notifications **MUST** extend from `App\Ship\Parents\Mails\Mail`.
- Email Templates must be placed inside the `Mails/Templates` directory within the container (i.e., `app/Containers/{container}/Mails/Templates`).

28.3 Folder Structure

```
app
    Containers
        {container-name}
            Mails
                UserRegisteredMail.php
                Templates
                    user-registered.blade.php
    Ship
        Mails
            SomeMail.php
```

(continues on next page)

(continued from previous page)

```
            Templates
                some-template.blade.php
```

28.4 Code Samples

```php
<?php

namespace App\Containers\User\Mails;

use App\Containers\User\Models\User;
use Illuminate\Bus\Queueable;
use App\Ship\Parents\Mails\Mail;
use Illuminate\Contracts\Queue\ShouldQueue;

class UserRegisteredMail extends Mail implements ShouldQueue
{
    use Queueable;

    protected $user;

    public function __construct(User $user)
    {
        $this->user = $user;
    }

    public function build()
    {
        return $this->view('user::user-registered')
            ->to($this->user->email, $this->user->name)
            ->with([
                'name' => $this->user->name,
            ]);
    }
}
```

28.4.1 Sending the Mail from an Action

Notifications can be sent from `Actions` or `Tasks` using Laravels `Mail` Facade.

```
Mail::send(new UserRegisteredMail($user));
```

28.5 Email Templates

Templates should be placed inside the `Mails/Templates` folder. To access a Mail template (i.e., same as loading a view) you must call the container name then the view name.

In the example below we are using the `user-registered.blade.php` template in the `User` Container.

```
$this->view('user::user-registered')
```

28.6 Configure Emails

Open the `.env` file and set the `FROM_MAIL_ADDRESS` and `MAIL_FROM_NAME` keys. These keys, in turn, will be used globally whenever the `from` function is not called in the Mail.

```
MAIL_FROM_ADDRESS=support@example.com
MAIL_FROM_NAME="Support"
```

To use different email address in some classes, simply add `->to($this->email, $this->name)` to the `build` function in your Mail class.

By default HiveApi is configured to use Log Driver `MAIL_DRIVER=log`, you can change that from the `.env` file.

28.7 Queueing Notifications for Later Use

To queue a notification you should use `Illuminate\Bus\Queueable` and implement `Illuminate\Contracts\Queue\ShouldQueue`.

CHAPTER 29

Middlewares

Middleware provide a convenient mechanism for filtering and manipulating HTTP `Requests` entering your application or `Responses` sent back to the client. You can read more about middlewares in the official Laravel documentation.

29.1 Principles

- In HiveApi there are two types of middlewares: General (applied to all the endpoints by default) and Endpoint Middlewares (applied to specific endpoints).
- The Middlewares **CAN** be placed in `Ship` layer or the `Container` layer depend on their roles.

29.2 Rules

- If the Middleware is placed inside a Container it **MUST** be registered inside that Container.
- To register Middlewares in a Container the container needs to have a `MiddlewareServiceProvider`. And like all other Container Providers it **MUST** be registered in the `MainServiceProvider` of that Container.
- General Middlewares (like some default Laravel Middlewares) **SHOULD** live in the Ship layer `app/Ship/Middlewares/*` and are registered in the `Ship` main `ServiceProvider`.
- Third-Party packages Middleware **CAN** be registered in Containers or on the Ship layer (wherever they make more sense).

For example, The `jwt.auth` middleware "provided by the JWT package" is registered in the Authentication Container (`Containers/Authentication/Providers/MiddlewareServiceProvider.php`).

29.3 Folder Structure

```
app
    Containers
        {container-name}
            Middlewares
                WebAuthentication.php
    Ship
        Middleware
            Http
                EncryptCookies.php
                VerifyCsrfToken.php
```

29.4 Code Sample

```php
<?php

namespace App\Containers\Authentication\Middlewares;

use App\Ship\Engine\Butlers\Facades\ContainersButler;
use App\Ship\Parents\Middlewares\Middleware;
use Closure;
use Illuminate\Contracts\Auth\Guard;
use Illuminate\Http\Request;

class WebAuthentication extends Middleware
{
    protected $auth;

    public function __construct(Guard $auth)
    {
        $this->auth = $auth;
    }

    public function handle(Request $request, Closure $next)
    {
        if ($this->auth->guest()) {
            return response()->view(ContainersButler::getLoginWebPageName(), [
                'errorMessage' => 'Credentials Incorrect.'
            ]);
        }

        return $next($request);
    }
}
```

29.4.1 Registering a Middleware within a Container

```php
<?php

namespace App\Containers\Authentication\Providers;

```

(continues on next page)

(continued from previous page)

```
use App\Containers\Authentication\Middlewares\WebAuthentication;
use App\Ship\Parents\Providers\MiddlewareProvider;
use Tymon\JWTAuth\Middleware\GetUserFromToken;
use Tymon\JWTAuth\Middleware\RefreshToken;

class MiddlewareServiceProvider extends MiddlewareProvider
{
    protected $middleware = [
    ];

    protected $middlewareGroups = [
        'web' => [
        ],

        'api' => [
        ],
    ];

    protected $routeMiddleware = [
        'jwt.auth'          => GetUserFromToken::class,
        'jwt.refresh'       => RefreshToken::class,
        'auth:web'          => WebAuthentication::class,
    ];

    public function boot()
    {
        $this->loadContainersInternalMiddlewares();
    }
}
```

29.4.2 Registering a Middleware within the Ship

```
<?php

namespace App\Ship\Kernels;

use App\Ship\Middlewares\Http\ProcessETagHeadersMiddleware;
use App\Ship\Middlewares\Http\ProfilerMiddleware;
use App\Ship\Middlewares\Http\ValidateJsonContent;
use Illuminate\Foundation\Http\Kernel as LaravelHttpKernel;

class HttpKernel extends LaravelHttpKernel
{
    /**
     * The application's global HTTP middleware stack.
     * These middleware are run during every request to your application.
     *
     * @var array
     */
    protected $middleware = [
        // Laravel middleware's
        \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
        \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
        \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
        \App\Ship\Middlewares\Http\TrimStrings::class,
```

(continues on next page)

(continued from previous page)

```
        \App\Ship\Middlewares\Http\TrustProxies::class,

        // CORS package middleware
        \Barryvdh\Cors\HandleCors::class,
    ];

    /**
     * The application's route middleware groups.
     *
     * @var array
     */
    protected $middlewareGroups = [
        'web' => [
            \App\Ship\Middlewares\Http\EncryptCookies::class,
            \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
            \Illuminate\Session\Middleware\StartSession::class,
            \Illuminate\View\Middleware\ShareErrorsFromSession::class,
            \App\Ship\Middlewares\Http\VerifyCsrfToken::class,
            \Illuminate\Routing\Middleware\SubstituteBindings::class,
        ],

        'api' => [
            ValidateJsonContent::class,
            'bindings',
            ProcessETagHeadersMiddleware::class,
            ProfilerMiddleware::class,
            // The throttle Middleware is registered by the RoutesLoaderTrait in the␣
↪Core
        ],
    ];

    /**
     * The application's route middleware.
     * These middleware may be assigned to groups or used individually.
     *
     * @var array
     */
    protected $routeMiddleware = [
        'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
        'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
        'can'      => \Illuminate\Auth\Middleware\Authorize::class,
        'auth'     => \Illuminate\Auth\Middleware\Authenticate::class,
    ];
}
```

CHAPTER 30

Notifications

Notifications allows you to inform the client about a state changes in your application. Laravel supports sending such notifications across a variety of channels such as mail, SMS, Slack, Databases, . . .

When using the Database channel the notifications will be stored in a database to be displayed in your client interface.

For more details refer to the official Laravel documentation.

30.1 Principles

- Containers **MAY** or **MAY NOT** have one or more Notification.
- The Ship layer **MAY** contain Application general Notifications.

30.2 Rules

- All Notifications **MUST** extend from `App\Ship\Parents\Notifications\Notification`.

30.3 Folder Structure

```
app
    Containers
        {container-name}
            Notifications
                UserRegisteredNotification.php
                ...
    Ship
        Notifications
            SystemFailureNotification.php
            ...
```

30.4 Code Samples

Example for a simple Notification

```php
<?php

namespace App\Containers\User\Notifications;

use App\Containers\User\Models\User;
use App\Ship\Parents\Notifications\Notification;
use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;

class BirthdayReminderNotification extends Notification implements ShouldQueue
{
    use Queueable;

    protected $notificationMessage;

    public function __construct($notificationMessage)
    {
        $this->notificationMessage = $notificationMessage;
    }

    public function toArray($notifiable)
    {
        return [
            'content' => $this->notificationMessage,
        ];
    }

    public function toMail($notifiable)
    {
        // $notifiable is the object you want to notify "e.g. user"
        return (new MailMessage)
            ->subject("Happy Birthday")
            ->line("Hi, $notifiable->name")
            ->line($this->notificationMessage);
    }

    public function toSms($notifiable)
    {
        // ...
    }

    // ...
}
```

30.4.1 Using a Notification within an Action or Task

Notifications can be sent from `Actions` or `Tasks` using the `Notification` Facade.

```
\Notification::send($user, new BirthdayReminderNotification($notificationMessage));
```

Alternatively, you can use the `Illuminate\Notifications\Notifiable` trait on the notifiable object (e.g., the `User`) and then call it as follow:

```
// call notify, found on the Notifiable trait
$user->notify(new BirthdayReminderNotification($notificationMessage));
```

30.5 Select Channels

To select a notification channel, HiveApi provides the `app/Ship/Configs/notification.php` config file, where you can specify an array of supported channels (e.g., SMS, Email, WebPush, . . .), to be used for all your notifications.

If you wan to override the configuration for specific `Notification` classes, or if you prefer to defined the channels within each `Notification` class itself, you can override the `public function via($notifiable)` in respective class and define your channels.

Checkout the Laravel Notification Channels documentation for list of supported integrations.

30.6 Queueing a Notification

In order to queue a notification for later use, you should use `Illuminate\Bus\Queueable` and implement `Illuminate\Contracts\Queue\ShouldQueue`.

30.7 Use Database Channel

First you need to generate a notification migration file that holds your `Notifications` to be displayed for your clients. You can easily call respective Artisan command `php artisan notifications:table` or use `php artisan hive:generate:migration`. Then run `php artisan migrate` to migrate your database,

Please note that `HiveApi` already provides the `xxxxxx_create_notifications_table.php` in the default migrations files directory `app/Ship/Migrations/`. So you don't need to create / configure it manually. Nice, isn't it?

CHAPTER 31

Service Providers

Providers (short names for Service Providers) are the central place of configuring and bootstrapping a Container. They are the place where you register container bindings, event listeners, middlewares, routes, other providers, aliases, ... to the framework service container.

31.1 Principles

- There are two types of Providers in a Container, the main `Provider` and additional (job specific) `Providers` (e.g., `EventsProvider`, `BroadcastsProvider`, `AuthProvider`, `MiddlewareProvider`, `RoutesProvider`).
- A Container **MAY** have one or many Providers, or **MAY** have no Provider at all.
- A Container **CAN** have only one single main Provider.
- The main Provider is the place where all other job specific Providers are registered.
- Third-party package Providers **MUST** be registered inside the Containers main service provider. The same applies to their Aliases.
- Providers **CAN** be registered on the `Ship` main Provider, if they are general or are intended to be used across many containers. The same applies to their Aliases.

31.2 Rules

- The main Provider will be auto-registered by the `Ship` engine, so no need to register it manually anywhere.
- All Main Providers **MUST** extend from `App\Ship\Parents\Providers\MainProvider`.
- All other types of `Providers` (`EventsProvider`, `BroadcastsProvider`, `AuthProvider`, `MiddlewareProvider`, `RoutesProvider`) must extend from their parent providers `Ship/Parents/Providers/***`.
- The Main Provider **MUST** be named `MainServiceProvider` in every container.

- You **SHOULD NOT** register any `Provider` in the framework (`config/app.php`), only the `HiveApiProvider` should be registered there.

  Heads up!

  Laravel 5.5 introduces an `auto-discovery` feature that lets you automatically register `ServiceProviders`. Due to the nature and structure of HiveApi applications, this features **is turned off**, because it messes up the way how `config` files are loaded in HiveApi. This means, that you still need to **manually** register third-party `ServiceProviders` in the `ServiceProvider` of a `Container`.

31.3 Folder Structure

```
app
    Containers
        User
            Providers
                EventsServiceProvider.php
                MainServiceProvider.php
                ...
```

31.4 Code Samples

Main Service Provider Example:

```php
<?php

namespace App\Containers\Excel\Providers;

use App\Ship\Parents\Providers\MainProvider;

class MainServiceProvider extends MainProvider
{

    public $serviceProviders = [
        // ...
    ];

    public $aliases = [
        // ...
    ];

    public function register()
    {
        parent::register();

        $this->app->bind(UserRepositoryInterface::class, UserRepository::class);
    }
}
```

Heads up!

When defining the `register()` or `boot()` function in your Main provider “only”, you must call the parent functions (`parent::register()`, `parent::boot()`) from your extended function.

31.5 Register Service Providers

31.5.1 Containers Main Service Provider

There is no need to register the `MainServiceProvider` anywhere, as it will be automatically registered by HiveApi. The `MainServiceProvider`, in turn, is responsible for registering all the Containers additional (job specific) Providers.

31.5.2 Containers Additional Service Providers

You **MAY** add further `Service Providers` in a `Container`. However, in order to get them loaded in the framework you **MUST** register them all in the `MainServiceProvider` as follow:

```php
<?php

private $containerServiceProviders = [
    AuthServiceProvider::class,
    EventsServiceProvider::class,
    // ...
    ];
```

The same rule applies to `Aliases`.

31.5.3 Third party packages Service Providers

If a package requires registering its service provider in the `config/app.php`, you **SHOULD** register its service provider in the container where you rely on this package. However, if it is a “generic” package used by the entire framework and not one specific Container or feature, you can register that service provider in the `app/Ship/Providers/ShipProvider.php`. You should, however, **NEVER** register additional service providers in the `config/app.php` file.

31.6 Laravel 5.5 Auto Discovery feature.

This feature is disabled in HiveApi so far. More details can be found here.

31.7 Information about Laravel Service Providers

By default Laravel provides some service providers in its `app/providers` directory. In HiveApi those providers have been renamed and moved to the Ship Layer `app/Ship/Parents/Providers/*`:

- `AppServiceProvider`
- `RouteServiceProvider`
- `AuthServiceProvider`
- `BroadcastServiceProvider`
- `EventsServiceProvider`

> Heads up!
>
> You **SHOULD NOT** touch those providers, instead you have to extend them from a custom provider in order to modify them. For example: the `app/Containers/Authentication/Providers/AuthProvider.php` is extending the `AuthServiceProvider` in order to modify it.

Those providers are not auto-registered by default, thus writing any code will not be available, unless you extend them. Once extended, the child provider should be registered in the containers `MainServiceProvider`, which makes the parent available through inheritance.

This rule, however, does not apply to the `RouteServiceProvider` since it is required by HiveApi. This Provider is directly registered by the `HiveApiProvider`.

Check out how Service Providers are auto-loaded.

CHAPTER 32

Requests

Read from the **Porto SAP Documentation (#Requests)**.

32.1 Rules

- All Requests **MUST** extend from `App\Ship\Parents\Requests\Request`.
- A Request **MUST** have a `rules()` function, returning an array and an `authorize()` function to check for authorization (can return `true` when no authorization is required).

32.2 Folder Structure

```
app
    Containers
        {container-name}
            UI
                API
                    Requests
                        UpdateUserRequest.php
                        DeleteUserRequest.php
                        ...
                WEB
                    Requests
                        UpdateUserRequest.php
                        DeleteUserRequest.php
                        ...
```

32.3 Code Samples

See an example for the `UpdateUserRequest` class:

```php
<?php

namespace App\Containers\User\UI\API\Requests;

use App\Ship\Parents\Requests\Request;

class UpdateUserRequest extends Request
{

    protected $access = [
        'permission' => '',
        'roles'      => 'admin',
    ];

    protected $decode = [
    ];

    protected $urlParameters = [
    ];

    public function rules()
    {
        return [
            'email'    => 'email|unique:users,email',
            'password' => 'min:100|max:200',
            'name'     => 'min:300|max:400',
        ];
    }

    public function authorize()
    {
        return $this->check([
            'hasAccess|isOwner',
        ]);
    }
}
```

The properties of the `Request` class will be discussed in the further course of this section.

32.4 Using Requests in the Contorller

```php
<?php

public function updateUser(UpdateUserRequest $updateUserRequest)
{
    $data = $updateUserRequest->all();
    // or
    $name = $updateUserRequest->name;
    // or
    $name = $updateUserRequest['name'];
}
```

By just injecting the `Request` class you already applied the validation and authorization rules. When you need to pass data to the `Action`, you should pass the `Request` object as it is to the `Action` parameter. A more elaborate approach, however, would be to "morph" the `Request` to a `Transporter` class!

32.5 Request Properties

HiveApi adds some new properties to the existing `Request` class from Laravel. Each of these properties are very useful for some situations, and let you achieve your goals faster and cleaner. Below we will see a description for each property:

32.5.1 decode

The `$decode` property is used for decoding hashed IDs from any Request on the fly if you have enabled the HashID feature provided by HiveApi. Most probably you are passing or allowing your users to pass hashed (encoded) IDs into your application in order to "hide" your true IDs. Thus these IDs needs to be decoded somewhere. HiveApi has a property on its `Request` component to specify those hashed IDs in order to decode them before applying the validation rules.

Example:

```php
<?php

namespace App\Containers\Authorization\UI\API\Requests;

use App\Ship\Parents\Requests\Request;

class AssignUserToRoleRequest extends Request
{

    protected $decode = [
        'id',
    ];

    // ...
}
```

Heads up!

Validations rules that relies on your ID like (`exists:users,id`) will not work unless you decode your ID before passing it to the validation!

32.5.2 urlParameters

The `$urlParameters` property is used for applying validation rules on the URL parameters. Laravel, by default, does not allow validating URL parameters (i.e., the `id` in `/stores/{id}/items`). In order to be able to apply validation rules on URL parameters you can simply define your URL parameters in the `$urlParameters` property. This will also allow you to access those parameters directly from the Controller in the same way you access the data from the `Request`.

Example:

```
<?php

namespace App\Containers\Store\UI\API\Requests;

use App\Ship\Parents\Requests\Request;

class GetItemFromShopRequest extends Request
{

    /**
     * Defining the URL parameters (`/stores/{store_id}/items/{item_id}`) allows␣
↪applying
     * validation rules on them and allows accessing them like request data.
     *
     * @var  array
     */
    protected $urlParameters = [
        'store_id',
        'item_id',
    ];

    public function rules()
    {
        return [
            'store_id' => 'required|integer', // url parameter
            'item_id'  => 'required|min:35|max:45', // url parameter
        ];
    }

    // ...
}
```

32.5.3 access

The `$access` property, allows to define set of `Roles` and `Permissions` a client accessing the API **must** have in order to access this endpoint. The `$access` property is used by the `hasAccess` function defined below in the `authorize` function, to check if the user has the necessary `Roles` and `Permissions` to call this endpoint (i.e., access the controller function, where this `Request` object is injected).

Example:

```
<?php

namespace App\Containers\User\UI\API\Requests;

use App\Ship\Parents\Requests\Request;

class DeleteUserRequest extends Request
{
    /**
     * Define which Roles and/or Permissions has access to this request.
     *
```

(continues on next page)

(continued from previous page)

```
     * @var  array
     */
    protected $access = [
        'permission' => 'delete-users|another-permissions',
        'roles' => ['manager','admin']
    ];

    public function authorize()
    {
        return $this->check([
            'hasAccess|isOwner',
            'isKing',
        ]);
    }
}
```

If you do not like the `laravelish` style with `|` in order to separate the different `roles` or `permissions` (e.g., see the example above), you can also use the `array notation`.

32.6 How the Authorize Function Works

The `authorize` function calls a `check` function, which accepts an array of functions names, each returning a boolean. In the example above, three functions (i.e., `hasAccess`, `isOwner`, and `isKing`) are called.

The separator `|` between the functions indicates an `OR` operation, so if any of the functions `hasAccess` or `isOwner` returns `true`, the user will gain access and only when both return `false` the user will be prevented from accessing this endpoint.

Furthermore, if `isKing` (i.e., a custom function that could be implemented by you) returns `false`, no matter what all other functions returns, the user will be prevented from accessing this endpoint, because the default operation between all functions in the array is `AND`.

32.6.1 Add Custom Authorize Functions

The best way to add a custom authorize function is through a Trait, which can be added to your `Request` classes. In the example below we create a Trait named `isKingPermissionTrait` with a single method called `isKing`.

The `isKing()` method, in turn, calls a Task to verify that the current user is a king (e.g., if the user has the proper `Role` assigned).

```
<?php
namespace App\Containers\User\Traits;

use Apiato\Core\Foundation\Facades\Apiato;

trait isKingPermissionTrait
{
    public function isKing()
    {
        // The task needs to be implemented properly!
        return Apiato::call('User@CheckIfUserHasProperRoleTask', [$this->user(), [
↪'king']]);
    }
}
```

Now, add the newly created Trait to the Request to use the `isKing` function in the authorization check.

```php
<?php

namespace App\Containers\User\UI\API\Requests;

use App\Containers\User\Traits\isKingPermissionTrait;
use App\Ship\Parents\Requests\Request;

class FindUserByIdRequest extends Request
{
    use isKingPermissionTrait;

    // ...

    public function authorize()
    {
        return $this->check([
            'isKing',
        ]);
    }
}
```

Now, the Request uses the newly created `isKing` method to check the proper access rights.

32.7 Allow a Role to access every endpoint

You can allow some `Roles` to access every endpoint in the system without having to define that role in each `Request` object. This is useful you want to let users with `Admin` role access everything.

To do this define those roles in `app/Ship/Configs/hive.php` as follow:

```
'requests' => [
    'allow-roles-to-access-all-routes' => ['admin',],
],
```

This will append the `admin` role to all roles access in every `Request`.

32.8 Request Helper Functions

HiveApi also provides some helpful functions by default, so you can use them whenever you need them.

32.8.1 hasAccess

The `hasAccess` function, decides if the the user has access to this endpoint based on the `$access` property.

- If the user has any `Role` or `Permission` defined in the access' property, he will be given access.
- If you need more or less roles/permissions just add `|` between each permission.
- If you do not need to set a roles/permissions just set `'permission' => ''` or `'permission' => null`.

32.8.2 isOwner

The `hasAccess` function, checks if the passed URL ID is the same as the User ID of the request.

Example:

Let's say we have an endpoint `api.example.develop/v1/users/{ID}/delete` that deletes a specified user. And we only need users to delete their own user accounts.

With `isOwner`, the user of ID 1 can only call `/users/1/delete` and won't be able to call `/users/2/delete` or any other ID. This also works with hashed IDs!

32.8.3 getInputByKey

Use this method to get data from within the `$request` by entering the name of the field. This function behaves like `$request->input('key.here')`, however, it works on the `decoded` values instead of the original data.

Consider the following `Request` data in case you are passing `application/json` data instead of `x-www-form-urlencoded`:

```
{
  "data" : {
    "name"   : "foo",
    "description" : "bar"
  },
  "id" : "a2423nadabada0"
}
```

Calling `$request->input('id')` would return `"a2423nadabada0"`, however `$request->getInputByKey('id')` would return the decoded value (e.g., `4`).

Furthermore, one can define a `default` value to be returned, if the key is not present (or not set), like so: `$request->getInputByKey('data.name', 'undefined name')`

32.8.4 sanitizeData

Especially for `PATCH` requests, you like to submit only the fields, to be changed to the API in order to:

a) minimize the traffic b) partially update the respective resource

Checking for the presence (or absence) of specific keys in the request typically results in huge `if` blocks, like so:

```
<?php
// ...
if($request->has('data.name')) {
   $data['name'] = $request->input('data.name'); // or better use getInputByKey()
}
if($request->has('data.description')) {
   $data['description'] = $request->input('data.description'); // or better use␣
↪getInputByKey()
}
// ...
```

To avoid those `if` blocks, use `array_filter($data)` in order to remove `empty` fields from the request. However, in PHP `false` and `''` *(empty string)* are also considered as `empty` resulting in removing those fields from the request data (which is clearly not what you want).

You can read more about this problem here.

In order to simplify sanitizing your `Request Data` when using `application/json` instead of `x-www-form-urlencoded`, HiveApi offers a convenient `sanitizeInput(array $fields)` method.

Consider the following `Request` data:

```
{
    "data" : {
        "is_private" : false,
        "description" : "this is a rather long description text",
        "a" : null,
        "b" : 3453,
        "foo" : {
            "a" : "a",
            "b" : "b",
            "c" : 1234
        },
        "bar" : [
            "a", "b", "c"
        ]
    }
}
```

The method lets you specify a list of `$fields` to be accessed and extracted from the `$request`. This is done by using the `DOT notation`. Finally, call the `sanitizeInput()` method on the `$request`:

```
$fields = ['data.name', 'data.description', 'data.is_private', 'data.blabla', 'data.
↪foo.c'];
$data = $request->sanitizeInput($fields);
```

The extracted `$data` looks like this:

```
[
  "data" => [
    "is_private" => false
    "description" => "this is a rather long description text"
    "foo" => [
      "c" => 1234
    ]
  ]
]
```

Note that `data.blabla` is not within the `$data` array, as it was not present within the `$request`. Furthermore, all other fields from the `$request` are omitted as they are not specified. So basically, the method creates some kind of `filter` on the `$request`, only passing the defined values. Furthermore, the DOT notation allows you to easily specify the fields to would like to pass through. This makes partially updating an resource quite easy!

> Heads Up:
>
> Note that the `fillable fields` of an entity can be easily obtained with `$entity->getFillable()`!

32.8.5 mapInput

Sometimes you might want to map input from the request to other fields in order to automatically pass it to a `Action` or `Task`. Of course, you can manually map those fields, but you can also rely on the `mapInput(array $fields)` helper function.

This helper, in turn, allows to “redefine” keys in the request for subsequent processing. Consider the following example request:

```
{
    "data" : {
        "name" : "John Doe"
    }
}
```

Your `Task` to process this data, however, requests the field `data.name` as `data.username`. You can call the the helper like this:

```
$request->mapInput([
    'data.name' => 'data.username',
]);
```

The resulting structure would look like this:

```
{
    "data" : {
        "username" : "John Doe"
    }
}
```

32.9 Storing Data on the Request

During the `Request` lifecycle you may want to store some data on the request object and pass it to other `SubActions` (or maybe if you prefer to Tasks). To store (additional) data on the `Request` you may use:

```
$request->keep(['someKey' => $someValue]);
```

To retrieve the data back at any time during the request lifecycle use:

```
$someValue = $request->retrieve('someKey')
```

CHAPTER 33

Seeders

Seeders (short name for Database Seeders) are classes made to seed the database with real data. Tthis data usually should exist in the application after the installation, for example the default `Users`, their associated `Roles` and `Permissions`, or a list of available `Countries` for shippment.

33.1 Principles

- Seeders **SHOULD** be created in the Containers.
- If the container is using a third-party package that publishes a Seeder class, this class should be manually placed in the Container that make use of it. Do not rely on the package to place it on its right location.

33.2 Rules

- Seeders **SHOULD** be in the right directory inside the container to be loaded.
- To avoid any conflict between containers seeders classes, you **SHOULD** always prepend the Seeders of each container with the container name. For example use `UserPermissionsSeeder`, `ItemPermissionsSeeder`). If 2 seeders classes have the same name but live in different containers, one of them will not be loaded.
- If you wish to order the seeding of the classes, you can just append `_1`, `_2` to your classes.

33.3 Folder Structure

```
app
    Containers
        {container-name}
            Data
```

(continues on next page)

(continued from previous page)

```
                Seeders
                    ContainerNameRolesSeeder_1.php
                    ContainerNamePermissionsSeeder_2.php
                    ...
```

33.4 Code Samples

RoleSeeder

```php
<?php

namespace App\Containers\Order\Data\Seeders;

use App\Ship\Parents\Seeders\Seeder;
use HiveApi\Core\Foundation\Facades\Hive;

class OrderPermissionsSeeder_1 extends Seeder
{

    public function run()
    {
        Hive::call('Authorization@CreatePermissionTask', ['approve-reject-orders']);
        Hive::call('Authorization@CreatePermissionTask', ['find-orders']);
        Hive::call('Authorization@CreatePermissionTask', ['list-orders']);
        // ...
    }
}
```

Note that one `Seeder` class may seed multiple `Model` classes.

33.5 Running the Seeders

After registering the `Seeders` you can run this command:

```
php artisan db:seed
```

To run specific `Seeder` class you can specific its class in the parameter as follow:

```
php artisan db:seed --class="App\Containers\X\Data\Seeders\YourCustomSeeder"
```

33.5.1 Migrate & Seed at the same time

```
php artisan migrate --seed
```

For more information about the Database Seeders read the official Laravel documentation.

33.6 HiveApi Seeder Commands

33.6.1 Testing

It's useful sometimes to create a big set of testing data. HiveApi facilitates this task:

1. Open `app/Ship/Seeders/SeedTestingData.php` and write your testing data here.
2. Run this command any time you want this data available (example at staging servers):

```
php artisan hive:seed:test
```

33.6.2 Deployment

HiveApi also provides a `Seeder` for your production data. You can call the artisan command

```
php artisan hive:seed:deploy
```

CHAPTER 34

Values

Values are short names for the known `Value Objects` which are simple objects similar to Models in the concept of representing data, but they do not get stored in the DB, thus they don't have "official" Ids. They also do not hold functionality or change any state, they just hold data.

A Value Object is an immutable object that is defined by its encapsulated attributes. We create Value Object when we need it to represent/serve/manipulate some data that is attached as attributes. Usually such Values are destroyed later when they are not needed any more.

34.1 Rules

- All Values **MUST** extend from `App\Ship\Parents\Values\Value`.

34.2 Folder Structure

```
app
    Containers
        {container-name}
            Values
                Output.php
                Region.php
                ...
```

34.3 Code Sample

```
<?php

use App\Ship\Parents\Values\Value;
```

(continues on next page)

(continued from previous page)

```
class Location extends Value
{
    private $lat = null;
    private $long = null;

    protected $resourceKey = 'locations';

    public function __construct($lat, $long)
    {
        $this->lat = $lat;
        $this->long = $long;
    }

    public function getCoordinatesAsString()
    {
        return $this->lat . ' - ' . $this->long;
    }

    public function getCoordinatesAsArray()
    {
        return [$this->lat, $this->long];
    }
}
```

Note that these Value objects also need to have a `$resourceKey` if you plan to output them via `Serializers` (e.g., see the Response page for more details).

CHAPTER 35

Authentication

Middlewares are probably the best solution to apply a solid `Authentication` for your API. With HiveApi you can use two pre-defined `Authentication Middlewares`, to protect your endpoints:

- API Authentication: `auth:api`
- Web Authentication: `auth:web`

35.1 API Authentication (with OAuth 2.0)

To protect an API endpoints from being accessed by unauthenticated users, you may apply the `auth:api` middleware.

```php
<?php

$router->get('secret/info', [
    'uses'       => 'Controller@getSecretInfo',
    'middleware' => [
        'auth:api',
    ],
]);
```

All endpoints that are protected with the `auth:api` middleware are only accessible when sending them a valid `access token`. The `auth:api` middleware is provided by the official Laravel Passport package. So you can read its documentation for more details.

35.1.1 Overview

OAuth lets you authenticate using different methods, these methods are called `grants`. In order how to decide, which grant type you should use, please refer to this website and keep reading this documentation.

Definitions

- The client credentials are `client_id` & `client_secret`.

- The proxy is an endpoint, that you should call instead of calling the OAuth server endpoints directly. The proxy endpoint, in turn, will append the client credentials to your request and calls the OAuth server for you, then returns its response back to the client. Each `first-client` application should have its own proxy endpoints (at least one of `/login` and one of `/refresh-token`). By default, HiveApi provides an `Admin Web Client` endpoint.

  You can Login to the first party app with proxy or without proxy, while for the third party you only need to login without proxy. (same apply to refreshing token).

  For first party apps:

  - With Proxy << best and easiest way, (requires manually generating clients creating proxy endpoints for each client)
  - Without Proxy << if your frontend is not exposing the client credentials, you can call the Auth server endpoints directly without proxy.

  For third party apps:

  - Without Proxy << you don't need a proxy for the third party clients as they usually integrate with your API from the backend side which protects the client credentials.

35.1.2 A: First-Party Clients

First-party clients (i.e., your own frontend / mobile / web / ... application) usually consumes your private API. Those clients need to use the `Resource Owner Credentials Grant` (a.k.a `Password Grant Tokens`).

When this grant type is used, your server needs to authenticate the client application first (ensuring the request is sent from your trusted frontend application) and then needs to check if the user credentials are correct (ensuring the user is registered and has the proper access rights), before issuing an access token.

Note:

- On `register`, the API returns user data. You will need to log that user in (using the same credentials he passed) to get his `access token` and make other API calls.
- On `login`, the API returns the users `access token` and a `refresh token`. You will need to request the user data by making another call to the user endpoint, using his private token.

Example

1. Create a `password` client in your database to represent one of your applications (e.g., your mobile application). Call `php artisan passport:client --password` to generate a new password client.
2. After registration the user can enter his credentials (i.e., email & password) in your mobile application login screen.
3. Your mobile application should send a `POST` request to `http://api.example.develop/v1/oauth/token` containing the user credentials (`username` and `password`) and the client credentials (`client_id` and `client_secret`) in addition to the `scope` and `grant_type=password`:

Request

```
curl --request POST \
  --url http://api.example.develop/v1/oauth/token \
  --header 'accept: application/json' \
  --header 'content-type: application/x-www-form-urlencoded' \
  --data 'username=admin%40local.host&password=admin&client_id=2&client_
↪secret=SGUVv02b1ppQCgI7ZVeoTZDN6z8SSFLYiMOzzfiE&grant_type=password&scope='
```

Response

```
{
  "token_type": "Bearer",
  "expires_in": 31536000,
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUz...",
  "refresh_token": "TPSPA1S6H8Wydjkjl+xt+hPGWTagL..."
}
```

1. Your mobile application should save the `access token` and start requesting secure data, by sending the latter in the HTTP Header `Authorization = Bearer {Access-Token}`.

More information can be found at the official Laravel Passport documentation.

> WARNING:
>
> The Client ID and Secret should not be stored in JavaScript or browser cache, or made accessible in any way.

So in case of web applications (i.e., Angular, Vue, . . . applications) you need to hide your client credentials behind a proxy. By default, HiveApi provides a `Login Proxy` to use for all your trusted first party clients.

Login with Proxy for First-Party Clients

The overall idea is to create a designated endpoint for each trusted client, to be used for login.

HiveApi, by default, has one URL ready for your Web Admin Dashboard (i.e., `clients/web/admin/login`). You can add more as you need for each of your trusted first party clients applications (example: `clients/web/users/login`, `clients/mobile/users/login`).

Behind the scene, that endpoint appends the corresponding client ID and secret to your request and makes another call to your OAuth server with all the required data. This way, the client does not need to send the ID and secret with the request. Further, the client uses his own URL, which gives even more control to which client is accessing your API. Then, it returns the authentication response back to the client with the proper `access tokens` in it.

> Heads up!
>
> You have to manually extract the Client credentials from the database and put them in the `.env`, for each client.

When running `passport:install` it automatically creates one client for you with a new ID, so you can use that for your first app. Or you can use `php artisan passport:client --password` to generate them.

```
# Example ENV File
CLIENT_WEB_ADMIN_ID=2
CLIENT_WEB_ADMIN_SECRET=VkjYCUk5DUexJTE9yFAakytWCOqbShLgu9Ql67TI
```

Login without Proxy for First-Party Clients

Login from your application by sending a POST request to http://api.example.develop/v1/oauth/token with grant_type=password, the user credentials (username & password), client credentials (client_id & client_secret) and finally the scope for this token (can be empty).

35.1.3 B: For Third-Party Clients

Third-party clients (custom external applications, who wants to integrate with your API) always consumes your public API (external API) only.

For third-party clients you need to use the Client credentials grant (a.k.a Personal Access Tokens). This grant type is the simplest and is suitable for machine-to-machine authentication.

With this grant type your server needs to authenticate the client application only, before issuing an access token.

Example

1. User logs in to your clients application interface (an external application made for your users only), go to settings, create a new client (of type personal) and copy the ID and Secret. This step can be done via an API request as well, if you prefer. You may generate a personal client for testing purposes using php artisan passport:client --personal.
2. The user adds the client credentials to his “server side software” and sends a POST request to http://api.example.develop/v1/oauth/token containing the issued client credentials (client_id and client_secret) in addition to the scope and grant_type=client_credentials:

Request

```
curl --request POST \
  --url http://api.example.develop/v1/oauth/token \
  --header 'accept: application/json' \
  --header 'content-type: application/x-www-form-urlencoded' \
  --data 'client_id=1&client_secret=y1RbtnOvh9rpA91zPI2tiVKmFlepNy9dhHkzUKle&grant_
↪type=client_credentials&scope='
```

Response

```
{
  "token_type": "Bearer",
  "expires_in": 31536000,
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1Ni...",
  "refresh_token": "ZFDPA1S7H8Wydjkjl+xt+hPGWTagX..."
}
```

1. The Client will be granted an access token to be saved. Then the client can start requesting secure data, by sending the access token in the HTTP Header Authorization = Bearer {Access-Token}.

Note: When a new user is registered, will be issued a personal Access Token automatically. Check the User “Registration page”.

More information can be obtained via the official Laravel Passport documentation

Login without Proxy for Third-Party Clients

We usually do not need a proxy for third-party clients as they are most likely making calls form their servers, thus the Client ID and Secret should be secure and not exposed to the users.

Login by sending a `POST` request to `http://api.example.develop/v1/oauth/token` with `grant_type=client_credentials`, Client Credentials (`client_id` & `client_secret`) and finally the `scope` (can be empty).

Once issued, you can use that `access token` to make requests to protected endpoints. The `access token` should be sent in the `Authorization` header of type `Bearer` (example: `Authorization = Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUz...`)

Heads up!

There is no "session state" when using tokens for authentication

35.1.4 Login With Custom Attributes

By default, HiveApi allows `Users` to login with their `email` address. However, you may want to also allow `username` and `phone` to login your users.

Here is, how to configure and use this feature.

- You may need to adapt your database accordingly (e.g., add the respective field to the `users` table).
- You may need to adapt the `Task` that `create` a `User` object (e.g., the `CreateUserByCredentialsTask`) accordingly to support the new fields. This may also affect your `Register` logic.
- Check the `App\Containers\Authentication\Configs\authentication-container` Configuration file and check the `login` params in order to configure this feature.
- Adapt the `ProxyApiLoginTransporter` accordingly to support your new Login Fields. These fields need to be added to `properties`

35.1.5 Logout

Logout by sending a `DELETE` request to `http://api.example.develop/v1/logout/` containing the valid `access token` in the header.

```
{
  "message": "Token revoked successfully."
}
```

35.2 Web Authentication

To protect a `Web` endpoint from being accessible by unauthenticated users you can use the `auth:web` Middleware.

```
<?php

$router->get('private/page', [
    'uses'       => 'Controller@showPrivatePage',
    'middleware' => [
        'auth:web',
```

(continues on next page)

(continued from previous page)

```
    ],
]);
```

This middleware is provided by HiveApi and is different than the default Laravel Auth Middleware. If authentication failed, users will be redirected to a login page. To change the login page view go to the config file `app/Ship/Configs/hive.php`, and set the name of your login page there as follow:

```php
<?php

    /*
    |--------------------------------------------------------------------------
    | The Login Page URL
    |--------------------------------------------------------------------------
    */

    'login-page-url' => 'login',
```

This will be search for a `login.html`, `login.php`, or `login.blade.php` file.

35.3 Refresh Token

In case your server is issuing a short-living `access tokens`, the users will need to refresh their access tokens via the refresh token that was provided to them when the logging in.

35.3.1 Refresh Token with proxy for first-party clients

By default HiveApi provide this ready to use endpoint `http://api.example.develop/v1/clients/web/admin/refresh` for the Web Admin Dashboard Client to be used when you need to refresh tokens for that client. You can, of course, create as many other endpoints as you want for each client. See the code within `app/Containers/Authentication/UI/API/Routes/ProxyRefreshForAdminWebClient.v1.public.php` and create similar ones for each client. The most important change will be the `env('CLIENT_WEB_ADMIN_ID')` and `env('CLIENT_WEB_ADMIN_SECRET')`, passed to the `ProxyApiRefreshAction`.

Those proxy refresh endpoints work in 2 ways. Either by passing the `refresh_token` manually to the endpoint. Or by passing it with the HttpCookie. In both cases the code will work and the server will reply with a response similar to this:

```json
{
  "token_type": "Bearer",
  "expires_in": 31500,
  "access_token": "tnJ1eXAiOiJKV1QiLCJhbGciOiJSUzI1Zx...",
  "refresh_token": "ZFDPA1S7H8Wydjkjl+xt+hPGWTagX..."
}
```

Note that the response contains a new `access token` for login as well as a new `refresh token`.

35.3.2 Refresh Token without Proxy for First-Party or Third-Party Clients

The request to `http://api.example.develop/v1/oauth/token` should contain `grant_type=refresh_token`, the `client_id` & `client_secret`, in addition to the `refresh_token` and finally the `scope` which could be empty.

35.4 Force Email Confirmation

By default a user does not have to confirm his email address to be able to login. However, to enforce users to confirm their email (i.e., to prevent unconfirmed users from accessing the API), you can set `'require_email_confirmation' => true,` in `App\Containers\Authentication\Configs\authentication.php`.

When the email confirmation is enabled (i.e., value set to `true`), the API throws an exception, if the `User` is not yet `confirmed`.

CHAPTER 36

Authorization

HiveApi provides a Role-Based Access Control (RBAC) from its Authorization Container. Behind the scenes HiveApi uses Laravels Authorization functionality that was introduced in version 5.1.11 with the helper package laravel-permission. You can always refer to the correspond documentation for more information.

36.1 Usage

Authorization in HiveApi is very simple and easy.

1. First you need to make sure you have a seeded Super Admin, an `admin` role and optionally your custom permissions (usually permissions should be statically created in the code). HiveApi provides most of these features for you, you can find the code at any container `.../Data/Seeders/*` directory.
2. Second create some basic `Roles`, and attach some `Permissions` to these roles.
3. Now start creating `Users` (or use any existing user), and assign them to the newly created roles.
4. Finally, you need to protect your endpoints by Permissions (or/and Roles). The right place to do that is the Requests class.

36.2 Example

Protecting an endpoint with a specific permission

```php
<?php

namespace App\Containers\User\UI\API\Requests;

use App\Ship\Parents\Requests\Request;

class DeleteUserRequest extends Request
{
```

(continues on next page)

(continued from previous page)

```
    /**
     * Define which Roles and/or Permissions has access to this request.
     *
     * @var  array
     */
    protected $access = [
        'permissions' => 'delete-users', // Accepts Array and String ['delete-users',
↪'create-users'],
        'roles'       => '',
    ];

    /**
     * @return  bool
     */
    public function authorize()
    {
        return $this->check([
            'hasAccess|isOwner',
        ]);
    }
}
```

For a detailed explanation of this example, please visit the Requests page.

36.3 Responses

If the authorization fails, HiveApi throws an `Exception` that may look similar to this.

```
{
  "errors": "You have no access to this resource!",
  "status_code": 403,
  "message": "This action is unauthorized."
}
```

36.4 Seeding Users

By default, HiveApi ships with a `Super Admin` with Access to the Admin Dashboard. This user has the role `admin` assigned by default. The `admin` role, however, **has no permissions assigned**. This Super Admin Credentials are:

- email: admin@admin.com
- password: admin

This user is seeded by `app/Containers/Authorization/Data/Seeders/AuthorizationDefaultUsersSeeder_3.php`.

To give permissions to the `admin` role (or any other role), you can use the dedicated endpoints (from your custom Admin Interface) or use this command `php artisan apiato:permissions:toRole admin` to assign all permissions available in the system.

Checkout each container `Seeder` directories `app/Containers/{container-name}/Data/Seeders/`, to edit the default `Users`, `Roles` and `Permissions`.

36.5 Roles & Permissions guards

By default HiveApi uses a single guard called `web` for all it's roles and permissions, you can add/edit this behavior and support multiple guards at any time. Refer to the laravel-permission package for more details.

36.6 Permissions Inheriting with Levels

When you create a role you can set an optional parameter, called `level` (set to `0` by default). The default seeded `admin` role has it set to `999`.

Level allows inheriting permissions. Role with higher level is inheriting permission from roles with lower level.

Below is a short example of how it works:

Imagine, you have three roles: `User`, `Moderator` and `Admin`. `User` has a permission to read articles, `Moderator` can manage comments and `Admin` can create articles. `User` has a level 1, `Moderator` level 2 and `Admin` level 3. It means, `Moderator` and `Administrator` has also permission to read articles, but `Administrator` can manage comments as well.

```
if ($user->getRoleLevel() > 10) {
    //
}
```

If a `User` has multiple roles, the `getRoleLevel()` method returns the highest one. If you don't need the permissions inheriting feature, simply ignore the optional level parameter when creating roles.

CHAPTER 37

Caching

37.1 Enable / Disable Eloquent Query Caching

By default caching is disabled. To enable the latter, go to `app/Ship/Configs/repository.php` config file and set `cache.enabled => true`, or set it from the `.env` file using the `ELOQUENT_QUERY_CACHE` key.

Clients can skip the query caching and request new data by passing specific parameter to the endpoint. For more details, we refer to the Query Parameters page.

More details can be found at the offical package documentation.

37.2 Change Different Caching Settings

You can use different cache setting for each repository. To set cache settings on each repository, first the caching must be enabled. Second you need to set up some properties on the repository class to override the default values.

You don't need to use the `CacheableRepository` trait or implement the `CacheableInterface` since they already exist on the `AbstractRepository` class (`App\Ship\Parents\Repositories\Repository`).

CHAPTER 38

Default Endpoints

HiveApi comes shipped with many useful API endpoints to help speeding up the development process. To see all the endpoints in a beautiful documentation run the `php artisan apidocjs:generate` command, after installing the apidocjs tool.

See the API Docs Generator page for more details.

CHAPTER 39

ETags

HiveApi provides an ETag Middleware (located in `app/Ship/Middlewares/Http/ProcessETagHeadersMiddleware.php`) that implements the shallow technique. It can be used to reduce bandwidth on the client side (especially useful for mobile devices like smartphones or tablets).

By default the feature is disabled. To enable it go to `app/Ship/Configs/hive.php` configuration file and set `use-etag` to `true`. Of course your client should send the `If-None-Match` HTTP Header `(= etag)` in all requests for this feature to work properly.

CHAPTER 40

Generators

Code Generators are a nice and easy way to speed up development by creating boiler-plate code based on your inputs. You may already be familiar with the Laravel code generators (e.g., `php artisan make:controller`).

HiveApi code generator works the same way. However, they are far more powerful and can generate an entire Container with fully working CRUD operations, including routes, requests, controller, Actions, Repositories, Models, Migrations, documentation, … and and and.

40.1 Available Code Generator Commands

To see the list of code generators type `php artisan hive:generate`.

```
hive:generate:container          Create a Container for HiveApi from scratch
hive:generate:action             Create a Action file for a Container
hive:generate:configuration      Create a Configuration file for a Container
hive:generate:controller         Create a controller for a container
hive:generate:exception          Create a new Exception class
hive:generate:job                Create a new Job class
hive:generate:mail               Create a new Mail class
hive:generate:migration          Create an "empty" migration file for a Container
hive:generate:model              Create a new Model class
hive:generate:notification       Create a new Notification class
hive:generate:repository         Create a new Repository class
hive:generate:request            Create a new Request class
hive:generate:route              Create a new Route class
hive:generate:seeder             Create a new Seeder class
hive:generate:serviceprovider    Create a ServiceProvider for a Container
hive:generate:subaction          Create a new SubAction class
hive:generate:task               Create a Task file for a Container
hive:generate:transformer        Create a new Transformer class for a given Model
```

To get more info about each command, add `--help` to the command. Example: `php artisan hive:generate:route --help`. The help page shows all options, which can be directly passed to the command.

If you do not provide respective information via the command line options, a wizard will be displayed to guide you through the process of creating specific components.

For example, you can directly call `php artisan hive:generate:controller --file=UserController` to directly specify the class to be generated. The wizard, however, will ask you for the `--container` as well.

Note that **all** generators automatically inherit the options `--container` and `--file` (these are documented as well in the help page). Furthermore, a generator may have specific options as well (e.g., the `--ui` (user-interface) to generate something for).

40.2 Custom Code Stubs (aka. Customizing the Generator)

If you don't like the automatically generated code (or would like to adapt it to your specific needs) you can do this quite easily.

The existing `Generators` allow to read `custom stubs` from the `app/Ship/Generators/CustomStubs` folder. The name of file needs to be the same as in `vendor/hiveapi/core/src/Generator/Stubs`.

Say, if you like to change the `config.stub`, simply copy the file to `app/Ship/Generators/CustomStubs/config.stub` and start adapting it to your needs.

If you run the respective command (e.g., in this case `php artisan hive:generate:configuration`) this would read your specific `config.stub` file instead the pre-defined one!

CHAPTER 41

Hash IDs

Hashing your internal IDs, is very helpful feature for securing your application (i.e., to prevent some kind of attacks) and business reasons (to hide the real total records from your competitors).

41.1 Enable Hash ID

Set the `HASH_ID=true` in the `.env` file. Also with the feature make sure to always use the `getHashedKey()` on any model, whenever you need to return an ID (mainly within Transformers) whether hashed ID or not.

41.2 Example:

```
'id' => $user->getHashedKey(),
```

Note that if the feature is set to false (e.g., `HASH_ID=false`) the `getHashedKey()` will return the normal (unhashed) ID.

41.3 Usage

There are 2 ways an ID's can be passed to your system via the API:

- Via URL Segment: (e.g., `www.hive.local/items/abcdef`).
- Via URL Query Parameters (e.g., `GET www.hive.local/items?id=abcdef`.

In both cases, however, you will need to inform your API about specific parts of that need to be decoded..

Checkout the Requests page. After setting the `$decode` and `$urlParameters` properties on your `Request` class, the ID will be automatically decoded for you, to apply validation rules on it or/and use it from your controller (`$request->id` will now return the decoded ID).

41.4 Configuration

You can change the default length and characters used in the ID from the config file `app/Ship/Configs/hashids.php` or in the `.env` file by editing the `HASH_ID_LENGTH` value.

You can set the `HASH_ID_KEY` in the `.env` file to any random string. You can generate this from any of the online random string generators, or run `head /dev/urandom | tr -dc 'A-Za-z0-9!"#$%&'\''()*+,-./:;<=>?@[\]^_{|}~' | head -c 32 ; echo` on the linux commandline. HiveApi defaults to the `APP_KEY` should this not be set.

The `HASH_ID_KEY` acts as the salt during the process of hashing IDs. This should never be changed in production as it renders all previously generated IDs useless!

41.5 Testing

In your tests you must hash the ID's before making the calls, because if you tell your Request class to decode an ID for you, it will throw an exception when the ID is not encoded.

41.5.1 For Parameter ID's

Always use `getHashedKey()` on your models when you want to get the ID.

Example:

```
$data = [
    'roles_ids' => [
        $role1->getHashedKey(),
        $role2->getHashedKey(),
    ],
    'user_id'   => $randomUser->getHashedKey(),
];
$response = $this->makeCall($data);
```

*Or you can do this manually `Hashids::encode($id);`. *

41.5.2 For URL ID's

You can use this helper function `injectId($id, $skipEncoding = false, $replace = '{id}')`.

Example:

```
$response = $this->injectId($admin->id)->makeCall();
```

More details on the [Tests Helpers]({{ site.baseurl }}{% link _docs/miscellaneous/tests-helpers.md %}) page.

41.6 Availability

You can apply the `HiveApi\Core\Traits\HashIdTrait` to any `Model` or class, in order to have the `encode` and `decode` functions ready set up. By default you have access to these functions `$this->encode($id)` and `$this->decode($id)` from all your Tests class and Controllers.

CHAPTER 42

Localization

The Localization feature of HiveApi is provided by the Localization Container.

42.1 Select Request Language

A client can select the language of the response by adding the `Accept-Language` header to a request. By default the `Accept-Language` is set to the language defined in `config/app.php locale`.

Please note that `Accept-Language` only determines, that the client *would like* to get the information in this specific language. It is not given, that the API responds in this language (e.g., if respective language cannot be found). When the `Accept-Language` header is missing, the default locale will be applied.

> Heads up!
>
> Please be sure that your client does not send an `Accept-Language` header automatically. Some REST clients (e.g., `POSTMAN`) automatically add header-fields based on the operating systems settings. So your `Accept-Language` header may be set automatically without knowing!

The API will answer with the applied language in the `Content-Language` header of the response.

If the requested language cannot be resolved (e.g., it is not defined) the API throws an `UnsupportedLanguageException` to tell the client about this.

The overall workflow of the provided Middleware is as follows:

1. Extract the `Accept-Language` field from the request header. If none is set, use the default locale from the config file
2. Build a list of all supported localizations based on the configuration of the respective container. If a language (top level) contains regions (sub-level), order them like this: `['en-GB', 'en-US', 'en']` (the regions are ordered before languages, as regions are more specific)
3. Check, if the value from 1) is within the list from 2). If the value is within this list, set it as `application language`, if not throw an `Exception`.

42.2 Support New Languages

1. All languages to be supported are defined within the `app/Containers/Localization/Configs/localization.php` configuration file. For example, the configuration file may look like this:

```
<?php
    'supported_languages' => [
        'de',
        'en' => [
            'en-GB',
            'en-US',
        ],
        'es',
        'fr',
    ],
```

1. Create new language files:

Language file can be placed in any container, not only the Localization Container. Refer to the Localization page for more details. Example languages files are included in the `Welcome` container at `app/Containers/Welcome/Resources/Languages`.

42.3 Translating Strings

By default all the translation within a Container are namespaced to the Container name.

42.3.1 Example

If a Container named `Store` has `en` translation file called `notifications` that contains translation for `welcome` like "Welcome to our store". You can access this translation as follow `trans('store::notifications.welcome')`. If you remove the namespace (which is the lowercase of the container name) and try to access it like this `trans('notifications.welcome')` it will not find your translation and will print `notifications.welcome` only.

> Heads up!
>
> If you try to load a string for a language that is not available (e.g., there is no folder for this language), HiveApiwill stick to the default one that is defined in `app.locale` config file. This is also true, if the requested locale is present in the `supported_languages` array from the configuration file.

42.4 Disable the Localization Feature

You can remove the `LocalizationMiddleware`, by simply going to `app/Containers/Localization/Providers/MainServiceProvider.php` and removing the `MiddlewareServiceProvider` from the `$serviceProviders` property.

42.5 Get Available Localizations

HiveApi provides a convenient way to get all defined Localizations. These localizations can be retrieved via `GET /localizations` by default. Note that this route only outputs the "top level" locales, like `en` from the example above. However, if specific regions (e.g., `en-US`) are defined, these will show up in the result as well.

The `Transformer` for the localizations not only provide the `language` (e.g., `de`), but also outputs the name of the language in this specific language (e.g., `locale_name => Deutsch`). Furthermore, the language name is outputted in the applications default name (e.g., configured in `app.locale`). This would result in `default_name => German`.

The same applies to the regions that are defined (e.g., `de-DE`). Consequently, this results in `locale_name => Deutschland` and `default_name = Germany`.

42.6 Tests

To change the default language in your tests requests. You can set the `env` language in the configuration files.

CHAPTER 43

Pagination

For pagination HiveApi relies on the L5 Repository Package and the pagination gets applied whenever you use the `paginate` function on any model repository

43.1 Change the Default Pagination Limit

Open the `.env` file and set a number for `PAGINATION_LIMIT_DEFAULT`:

```
PAGINATION_LIMIT_DEFAULT=10
```

This is used in the `config/repository.php`, which is the config file of the `L5 Repository` package.

43.2 Limit

The `?limit=` query parameter can be applied by clients to define, how many results should be returned on one page (see also `Pagination`!).

43.3 Usage

```
api.domain.develop/endpoint?limit=100
```

This would return `100` items of a resource within one page of the response. Of course, the `limit` and `page` query parameter can be combined in order to get the next `100` resources:

```
api.domain.develop/endpoint?limit=100&page=2
```

In order to allow clients to request all data that matches their criteria (e.g., search-criteria) and disable pagination, you can manually override the `$allowDisablePagination` property in your specific `Repository` class. A client can then get all data (with no pagination applied) by requesting `api.domain.develop/endpoint?limit=0`. This will return all matching entities.

43.4 Skip the Pagination Limit

You can allow developers to skip the pagination limit as follow. First, you need to enable that feature from the server by setting `PAGINATION_SKIP = true`. Second, inform the developers (users) to pass `?limit=0` with the request they wish to get all it's data un-paginated.

CHAPTER 44

Profiler

Profiling is very important to optimize the performance of your developed application. Further, it helps you to better understand what happens when a request is received, as well as it can speed up the debugging process.

HiveAPi uses the third-party package laravel-debugbar, which uses the PHP Debug Bar, to collect the profiling data.

By default the `Laravel-DebugBar` package displays the profiling data in the web browser. However, HiveApi uses a custom middleware (`app/Ship/Middlewares/Http/ProfilerMiddleware.php`) to append the profiling data to the response.

Please note that this middleware **MAY** severely slow down your application and **SHOULD NOT** be used in production mode!

44.1 Sample Profiler Response

```
{
    // Actual Response Here...

    "_profiler": {
        "__meta": {
            "id": "X167f293230e3457f1bbd95d9c82aba4a",
            "datetime": "2017-09-22 18:45:27",
            "utime": 1506105927.799299,
            "method": "GET",
            "uri": "/",
            "ip": "172.20.0.1"
        },
        "messages": {
            "count": 0,
            "messages": []
        },
        "time": {
            "start": 1506105922.742068,
            "end": 1506105927.799333,
```

(continues on next page)

(continued from previous page)

```
            "duration": 5.057265043258667,
            "duration_str": "5.06s",
            "measures": [
                {
                    "label": "Booting",
                    "start": 1506105922.742068,
                    "relative_start": 0,
                    "end": 1506105923.524004,
                    "relative_end": 1506105923.524004,
                    "duration": 0.7819359302520752,
                    "duration_str": "781.94ms",
                    "params": [],
                    "collector": null
                },
                {
                    "label": "Application",
                    "start": 1506105923.535343,
                    "relative_start": 0.7932748794555664,
                    "end": 1506105927.799336,
                    "relative_end": 0.000002861022949121875,
                    "duration": 4.26399302482605,
                    "duration_str": "4.26s",
                    "params": [],
                    "collector": null
                }
            ]
        },
        "memory": {
            "peak_usage": 13234248,
            "peak_usage_str": "12.62MB"
        },
        "exceptions": {
            "count": 0,
            "exceptions": []
        },
        "route": {
            "uri": "GET /",
            "middleware": "api, throttle:30,1",
            "domain": "http://api.hive.local",
            "as": "apis_root_page",
            "controller":
↪"App\\Containers\\Welcome\\UI\\API\\Controllers\\Controller@apiRoot",
            "namespace": "App\\Containers\\Welcome\\UI\\API\\Controllers",
            "prefix": "/",
            "where": [],
            "file": "app/Containers/Welcome/UI/API/Controllers/Controller.php:20-25"
        },
        "queries": {
            "nb_statements": 0,
            "nb_failed_statements": 0,
            "accumulated_duration": 0,
            "accumulated_duration_str": "0μs",
            "statements": []
        },
        "swiftmailer_mails": {
            "count": 0,
            "mails": []
```

(continues on next page)

(continued from previous page)

```
        },
        "logs": {
            "count": 3,
            "messages": [
                {
                    "message": "...",
                    "message_html": null,
                    "is_string": false,
                    "label": "error",
                    "time": 1506105927.694807
                },
                {
                    "message": "...",
                    "message_html": null,
                    "is_string": false,
                    "label": "error",
                    "time": 1506105927.694811
                },
                {
                    "message": "[2017-09-18 17:38:15] testing.INFO: New User␣
↪registration. ID = 970ylqvaogmxnbdr | Email = user@mail.develop.      Thank you for␣
↪signing up.\n</div>\n</body>\n</html>\n  \n",
                    "message_html": null,
                    "is_string": false,
                    "label": "info",
                    "time": 1506105927.694812
                }
            ]
        },
        "auth": {
            "guards": {
                "web": "array:2 [\n  \"name\" => \"Guest\"\n  \"user\" => array:1 [\n␣
↪   \"guest\" => true\n  ]\n]",
                "api": "array:2 [\n  \"name\" => \"Guest\"\n  \"user\" => array:1 [\n␣
↪   \"guest\" => true\n  ]\n]"
            },
            "names": ""
        },
        "gate": {
            "count": 0,
            "messages": []
        }
    }
}
```

44.2 Configuration

By default the profiler feature is turned off. To turn it on edit the `.env` file and set `DEBUGBAR_ENABLED=true`. To control and modify the profiler response, you need to edit this config file `app/Ship/Configs/debugbar.php`.

CHAPTER 45

Query Parameters: Search

Below you see how to setup a Search Query Parameter, on a Model:

1. First, you need to define searchable fields on the Model Repository

```php
<?php

namespace App\Containers\User\Data\Repositories;

class UserRepository extends Repository
{
    protected $fieldSearchable = [
        'name'  => 'like',
        'id'    => '=',
        'email' => '=',
    ];
}
```

1. Next, you need to create a basic `List` and `Search Task`

```php
<?php

namespace App\Containers\User\Tasks;

class ListUsersTask extends Task
{
    private $userRepository;

    public function __construct(UserRepositoryInterface $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function run()
    {
        return $this->userRepository->paginate();
```

(continues on next page)

(continued from previous page)

```
    }
}
```

1. Create an `Action` to call this Task

```php
<?php

namespace App\Containers\User\Actions;

class ListAndSearchUsersAction extends Action
{
    public function run()
    {
        return Hive::call(ListUsersTask::class, [], ['applyRequestCriteria']);
    }
}
```

1. Use the Action from a Controller

```php
<?php

public function listAllUsers(Request $request)
{
    $users = Hive::call(ListAndSearchUsersAction::class);

    return $this->transform($users, UserTransformer::class);
}
```

1. Call it from anywhere as follow `GET http://api.hive.local/users?search=example@test.com`

CHAPTER 46

Rate Limiting (API Throttling)

HiveApi uses the default Laravel middleware for rate limiting (throttling). All API endpoints are throttled to prevent abuse and ensure stability. The exact number of calls that your client can make per day varies based on the type of request you are making.

The rate limit window is `1` minute per endpoint, with most individual calls allowing for `30` requests in each window. In other words, each client is allowed to make `30` calls per endpoint every `1` minute (for each unique access token).

You can change these values within the `app/Ship/Configs/hive.php` config file, or in your `ENV` file.

```
'throttle' => [
    'enabled' => env('API_RATE_LIMIT_ENABLED', true),
    'attempts' => env('API_RATE_LIMIT_ATTEMPTS', '30'),
    'expires' => env('API_RATE_LIMIT_EXPIRES', '1'),
]
```

```
API_RATE_LIMIT_ENABLED=true
API_RATE_LIMIT_ATTEMPTS=30
API_RATE_LIMIT_EXPIRES=1
```

HiveApi automatically returns how many hits you can preform on an endpoint in the response header:

```
X-RateLimit-Limit : 30
X-RateLimit-Remaining : 29
```

46.1 Enable/Disable Rate Limiting:

The API rate limiting middleware is enabled by default and applied to all endpoints by default. To disable it this feature set `API_RATE_LIMIT_ENABLED` to `false` in your `.env` file.

CHAPTER 47

Request Monitor

HiveApi provides a simple and easy way to monitor and log all the HTTP requests reaching your application. The request monitor can be very useful when testing and debugging your frontend applications who consumes your API. Especially when the frontend apps (mobile applications, JavaScript applications, ,..) are built by other developers.

The requests monitor is provided by the Debugger Container, more specifically by a dedicated `RequestsMonitorMiddleware` middleware.

47.1 Enable Requests Logging

Set the `REQUESTS_DEBUG` to true within your `.env` file. In order for this to start displaying the results you need to enable the debugging mode in Laravel by setting `APP_DEBUG` to true in the `.env` as well.

47.2 Usage

Simply tail the log file

```
tail -f storage/logs/debugger.log
```

47.3 Change the Default Log File

By default everything is logged in the `debugger.log` file, to change the default file go to `app/Containers/Debugger/Configs/debugger.php` config file and specify a file name.

```
<?php

/*
 |--------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
 | Log File
 |--------------------------------------------------------------------------
 | What to name the log file in the `storage/log` path.
 */

'log_file' => 'debugger.log',
```

This feature will not run in the `testing` environments, to enable it you need to manually edit the middleware.

CHAPTER 48

System Settings

At many cases you need to have some dynamic system settings, such as in a referral program, where you give *gifts* to anyone who refers new users. But those gifts can be changed in the future, so it's better not have them statically created in the code, instead read from the database where an Admin can manage them at any time.

The `app/Containers/Settings` Container helps storing and retrieving those key values settings. It also seed the database with the default configurations during the installation.

48.1 Seed Default Settings

Default Settings should be added to the `app/Containers/Settings/Database/Seeders/DefaultSystemSettingsSeeder.php`

48.2 Read Settings

You can use the pre-defined `Tasks` to read / write settings from the database

```php
<?php
$value = $this->findSettingsByKeyTask->run('whateverSettingsName');
```

You can search for settings by Key as in the example above, or create a class for each settings as follow:

```php
<?php
$value = $this->findWhateverSettingsTask->run();
```

CHAPTER 49

Validation

HiveApi internally uses the powerful Laravel validation system. However, HiveApi validations must be defined within the `Requests` components, since every request might have different rules.

The Validations rules are automatically applied, once injecting the Request in the Controller. Requests helps validating data sent to the API, accessibility, ownership and more. . .

49.1 Request

Consider the following example for a validation within a Request

```php
<?php

namespace App\Containers\User\UI\API\Requests;

use App\Ship\Parents\Requests\Request;

class RegisterUserRequest extends Request
{
    /**
     * @return  array
     */
    public function rules()
    {
        return [
            'email'    => 'required|email|max:200|unique:users,email',
            'password' => 'required|min:20|max:300',
            'name'     => 'required|min:2|max:400',
        ];
    }
}
```

49.1.1 Using a Request (with Validation) within a Controller

Usage from Controller Example:

```php
<?php
    public function registerUser(RegisterUserRequest $request)
    {
        // if this method is called, the Request was successfully validated!
        $user = Hive::call(RegisterUserAction::class, [$request->toTransporter()]);

        return $this->transform($user, UserTransformer::class);
    }
```

49.2 Responses

If the Request data cannot be validated, the framework responds with an `Exception` similar to this

```json
{
  "errors": {
    "email": [
      "The email field is required."
    ],
    "password": [
      "The password field is required."
    ]
  },
  "status_code": 422,
  "message": "The given data failed to pass validation."
}
```

CHAPTER 50

Versioning

Since Laravel does not support API versioning, HiveApi provide a very easy way to implement versioning for your API.

When creating a new API endpoint, specify the version number in the route file name following this naming format `{endpoint-name}.{version-number}.{documentation-name}.php`.

Example:

- `MakeOrder.v1.public.php`
- `MakeOrder.v2.public.php`
- `ListOrders.v1.private.php`

50.1 Usage

The endpoint inside that route file will be automatically accessible by adding the version number to the URL. Example:

- `http://api.hive.local/v1/register`
- `http://api.hive.local/v1/orders`
- `http://api.hive.local/v2/stores/123`

50.2 Version the API in Header instead of URL

First remove the URL version prefix:

1. Edit `app/Ship/Configs/hive.php` and set the prefix to `'enable_version_prefix' => 'false'`.
2. Implement the Header versioning anyway you prefer. This is not implemented in yet.

CHAPTER 51

Magical Call

This *magical* function allows you to `call()` any `Action` or `Task` `run()` function, from anywhere. Using the `Hive::call()` Facade.

The function `call()` is mainly used for calling HiveApi `Actions` from `Controllers`, and calling HiveApi `Tasks`from `Actions`.

Each `Action` knows which UI called it, using `$this->getUI()`. This may be useful for handling the same `Action` differently based on the `UI` type (`WEB` or `API`). This will work when calling the `Action` from `Controllers` and `Commands` using the magical `call()` function.

51.1 Usage

In the first argument you can pass the class full name, as follow `App\Containers\User\Tasks\CreateUserTask::class`, or you can pass the container name and class name, as follow `User@CreateUserTask`.

Using the "string based" style (i.e., `containerName@className`) helps removing direct dependencies between containers. The `call()` function, in turn, will verify the Container exist before calling the function and inform the user to install Container if not exist.

A huge downside of the "string-based" approach, however, is that you will lose auto-completion features from your IDE!

When a class is directly called using its full name, a warning will be logged informing you to use the "string based caller style". This message, however, can be disabled by changing the flag `hive.logging.log-wrong-hive-caller-style` in the `Ship/Configs/hive.php` file accordingly.

```
<?php

// Call "AssignUserToRoleTask" Task from the "Authorization" Container using the␣
↪hiveapi caller style
Hive::call('Authorization@AssignUserToRoleTask');

// Call "AssignUserToRoleTask" Task from the "Authorization" Container using class␣
↪full name.
```

(continues on next page)

(continued from previous page)

```
// This will cause to add an INFO entry to the log file!
Hive::call(\App\Containers\Authorization\Tasks\AssignUserToRoleTask::class);
```

51.1.1 Example Basic Usage

```
$foo = \HiveApi\Core\Foundation\Facades\Hive::call('Container@ActionOrTask');
```

- From `Controllers` and `Actions` you can use the `$this->call('Container@ActionOrTask')` instead of the Facade but it is not recommended.
- The magical `call()` function accepts the class full namespace (`\App\Containers\User\Tasks\GetAllUsersTask::` and the HiveApi caller style (`Containers@GetAllUsersTask`).
- There is also a `transactionalCall()` method available, that wraps everything in a `DB::Transaction` (see below).

51.1.2 Passing arguments to the `run()` function

```
$foo = Hive::call('Container@ActionOrTask', [$runArgument1, $runArgument2,
↪$runArgument3]);
```

51.1.3 Calling other functions before calling the `run()`

```
$foo = Hive::call('Container@ActionOrTask', [$runArgument], ['otherFunction1',
↪'otherFunction2']);
```

51.1.4 Calling other functions and pass them arguments before calling the `run()`

```
<?php
$foo = Hive::call('Container@ActionOrTask', [$runArgument], [
    [
       'function1' => ['function1-argument1', 'function1-argument2']
    ],
    [
       'function2' => ['function2-argument1']
    ],
]);

$foo = Hive::call('Container@ActionOrTask', [$runArgument], [
    'function-without-argument',
    [
      'function1' => ['function1-argument1', 'function1-argument2']
    ],
]);

$foo = Hive::call('Container@ActionOrTask', [], [
    'function-without-argument',
    [
      'function1' => ['function1-argument1', 'function1-argument2']
```

(continues on next page)

(continued from previous page)

```
    ],
    'another-function-without-argument',
    [
      'function2' => ['function2-argument1', 'function2-argument2', 'function2-
↪argument3']
    ],
]);
```

51.2 Transactional Magical Call

Sometimes, you want to wrap a call into one `Database Transaction` (see theofficial Laravel Documentation).

Consider the following example: You want to create a new `Team` and automatically assign yourself (i.e., your own `User`) to this newly created `Team`. Your `CreateTeamAction` may call a dedicated `CreateTeamTask` and a `AssignMemberToTeamTask` afterwards. However, if the `AssignMemberToTeamTask` fails, for unknown reasons, you may want to "rollback" (i.e., remove) the newly created `Team` from the database in order to keep the database in a valid state.

That's where `DB::transactions` comes into play!

HiveApi provides a `transactionalCall($class, $params, $extraMethods)` method with the similar parameters as already known from the `call()` method. Internally, this method calls this `call()` method anyways, but wraps it into a `DB::transaction`.

If any `Exception` occurs during the execution of the `$class` to be called, everything done in this context is automatically rolled-back from the database. However, respective operations on the file system (e.g., you may also have uploaded a profile picture for this `Team` already that needs to be removed in this case) need to be performed manually!

Typically, you may want to use the `transactionalCall()` on the `Controller` level!

51.3 Use Case Example

```php
<?php

return Hive::call('User@ListUsersTask', [], ['ordered']);
// can be called this way as well Hive::call(ListUsersTask::class, [], ['ordered']);

return Hive::call('User@ListUsersTask', [], ['ordered', 'clients']);

return Hive::call('User@ListUsersTask', [], ['admins']);

return Hive::call('User@ListUsersTask', [], ['admins', ['roles' => ['manager',
↪'employee']]]);
```

51.3.1 The ListUsersTask class

```php
<?php

namespace App\Containers\User\Tasks;
```

(continues on next page)

(continued from previous page)

```php
use App\Containers\User\Data\Criterias\AdminsCriteria;
use App\Containers\User\Data\Criterias\ClientsCriteria;
use App\Containers\User\Data\Criterias\RoleCriteria;
use App\Containers\User\Data\Repositories\UserRepository;
use App\Ship\Criterias\Eloquent\OrderByCreationDateDescendingCriteria;
use App\Ship\Parents\Tasks\Task;

class ListUsersTask extends Task
{
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        $this->userRepository = $userRepository;
    }

    public function run()
    {
        return $this->userRepository->paginate();
    }

    public function clients()
    {
        $this->userRepository->pushCriteria(new ClientsCriteria());
    }

    public function admins()
    {
        $this->userRepository->pushCriteria(new AdminsCriteria());
    }

    public function ordered()
    {
        $this->userRepository->pushCriteria(new␣
↪OrderByCreationDateDescendingCriteria());
    }

    public function withRole($roles)
    {
        $this->userRepository->pushCriteria(new RoleCriteria($roles));
    }
}
```

CHAPTER 52

Task Queuing

- Queues System, which executes `Jobs` one by one once he receives it or once it's scheduled (after being serialized and stored in a string somewhere).
- to be able to queue the Jobs you need a Queuing System such as Beanstalkd, Redis, Amazon SQS or simply the Database.
- Laravel has a "queue worker" that will process new Jobs as they are pushed onto the queue system, ("queue:work" and "queue:listen"). Its job is to push the jobs to the queue system in order to be executed later.
- to keep the "queue worker `php artisan queue:work` command, running permanently in the background, you should use a process monitor such as "Supervisor" to ensure that the queue worker does not stop running. It will simply make sure to execute the `php artisan queue:work` command.
- so its role is to schedule the execution of Artisan Command, Jobs, Event Listeners, and some other classes, at specific intervals or dates using the third party Queueing System.

More info can be found at the official Laravel documentation.

HiveApi detects by default, which queue you are planning to use (based on the configurations), and creates the required migration files (if driver `databaes` is used)

```
if (Config::get('queue.default') == 'database')
{
   // do something
}
```

52.1 Beanstalkd

In order to use Beanstalkd as your queue driver, you need to require the `"pda/pheanstalk":   "^3.1"` package first. You can include this in any `composer.json` file you want.

CHAPTER 53

Task Scheduling

- is a script executor program, such as `Cron Job`. A Cron Job is a time-based scripts scheduler in Unix-like computer operating systems.
- its role is to schedule the execution of `Artisan` Commands", periodically at fixed times, dates, or intervals.
- Laravel has a wrapper around the "Cron Job" called the Laravel scheduler. This allows the framework to schedule class like and Artisan Commands, Queued Jobs in addition to custom Shell Commands, to run later.

Below is a quick guide for how to schedule some scripts execution such as (custom Shell Commands, Laravel Commands, Laravel Jobs, and other classes), in order to run at specific intervals or dates.

53.1 Setup Server

First you need to setup your server by adding a single Cron entry as follow:

1. On the command line, type `crontab -e`
2. At the last line add the following: `* * * * * php /path/to/your/project/artisan schedule:run >> /dev/null 2>&1`

More details regarding the configuration can be found in the official Laravel documentation.

53.2 Setup and Configure your Application

First you need to create some commands, that needs to be scheduled. The can be created in the Containers `(app/Containers/{container-name}/UI/CLI/Commands)` or in the Ship (`app/Ship/Commands`). See the Commands Page for more details.

Once you have your command ready, go to `app/Ship/Kernels/ConsoleKernel.php` and start adding the commands you need to schedule inside the `schedule` function.

Note that you do not need to register the commands with the `$commands` property or point to them in the `commands()` function. HiveApi will do that automatically for you.

Example:

```php
<?php
    protected function schedule(Schedule $schedule)
    {
         $schedule->command('hive:welcome')->everyMinute();
         $schedule->job(new myJob)->hourly();
         $schedule->exec('touch me.txt')->dailyAt('12:00');
         // ...
    }
```

The official Laravel documentation provides more details.

CHAPTER 54

Useful Commands

HiveApi already ships with many useful commands to help you speed up the development process. You can see list of all commands, by typing `php artisan hive` and look for the `HiveApi` (`hive`) section.

54.1 Available Commands (Excerpt)

- `php artisan hive:list:actions` Show all Actions within the application
- `php artisan hive:list:tasks` Show all Tasks within the application
- `php artisan hive:seed:test` Seeds your custom testing data from `app/Ship/Seeders/SeedTestingData.php`.
- `php artisan hive:seed:deploy` Seeds your custom deployment data from `app/Ship/Seeders/SeedDeploymentData.php`.
- `php artisan hive:generate:xxx` Generate a specific component for the framework (e.g., `Action`, `Task`, . . .). For more details on the Code Generator.

54.2 List All Actions / Tasks Command

It's useful to be able to see all the implemented use cases in your application. To do so type `php artisan hive:list:actions`. You can also pass `--withfilename` flag to see all Actions with the files names.`hive:list:actions --withfilename`.

The same command works for Tasks as well.

54.3 List Container Dependencies Command

Sometimes it is required to show dependencies between containers (e.g., how they are *interlinked* amongst each others). HiveApi provides a command to list all dependencies for one specific container. The command does take the

`Hive::call()` and `$this->call()` (with `use X`) into account.

If you want to get the dependencies for one container, you can call

```
php artisan hive:list:dependencies app/Containers/X
```

where `X` is the name of the requested container (e.g., `User`).