Web Applications semestral work

Introduction

This project, *ShopList*, is a single page application built around a REST API. It implements all the features described in the recommended semestral work. The client is developed in ES6 with a MVVM pattern while the API is written in PHP as a framework for building REST APIs.

ShopList's files are separated in different directories. The **config** directory contains configuration files for the API entry point; we will get back to those later. **src** holds the Rest framework and the ShopList API application that uses the framework. Finally, **webroot** is the entry point of the entire application. It contains both the entry points of the API and of the client, along with the source files of the client. A web server for this application should be configured to point directly to this folder instead of the root: this prevents anyone of accessing the API's source files. Additionally, a request for a file different than one of the client's should be redirected to the API's entry point. This configuration is already achieved for Apache2 using htaccess files.

The client

The client's design is based on the pictures provided with the semestral work specification. It is composed of a table of four columns for the items in the shop list: the item's name, the amount, the shared actions and the specific actions. The specific actions are interactions that only concern the item they are on like edit or delete. The shared actions concern both the current item and the one underneath, like swapping their positions. The last component of the client is the list item creator that allows an item to be added to the list.

The programming part of the client is divided in five classes and an entry point. The API class acts as an abstraction layer for making request to the actual API using the fetch API. The classes use it to retrieve resources or send data. The *Item* and *Items* classes represents the collection of available items. The *Items* class is also responsible of updating the dataset element used by the list item creator. The *ListItem* and *ListItems* represents the collection of items added to the shop list. *ListItem* handles a row in the table described earlier, including updating its contents and calling the API when the user triggers actions. *ListItems* handles the ordering of the row in the table, the shared actions and the list item creator. The entry point of the client simply initializes instances of *API*, *Items* and *ListItems* to run the application.

The API

The API is a Composer project. It is split in two namespaces:

- **GECU\Rest**, the REST API framework mentioned earlier;
- **GECU\ShopList**, the actual files for the *ShopList* API.

The Rest framework

The main goal of this framework is to allow the developer to use pre-existing classes and describe them as resources, so that they can be directly served as an API. A resource has *routes*, paths of the API that point to the resource. For example, *GET /list* is a *ShopList* route that points to the collection of items added to the list (the list items). It is different than *POST /list*, which is the route for adding a list item to the collection. A route has five important properties:

- The name of the class of the resource;
- The HTTP method of the request;
- The path of the request, which can contain parameters, like the route *GET /item/{id}*, where *id* is the identifier of an item in the collection;
- An optional action name, a method of the resource class. If specified, the result of this method will be the response to the request instead of the resource itself;
- An optional request content factory. This property is a sort of function that transforms the body of the request into an object useable by the resource action.

The resource itself can have a factory, allowing it to be created with a custom behaviour. By default, the framework will simply use the constructor. The factories in this framework are extended versions of *callables*. They can be function names, invokable objects or an array similarly to the built-in PHP type, but the array accepts more possibilities. Normally, it should contain either a class name and a static method name or an instance and a method name. The Rest framework allows the combination of a class name and a non-static method name. In that case, the factory will first be instantiated using reflectivity and then the array will be a standard callable. It is also possible to specify the constructor as method, to simply consider the factory itself as the result.

The arguments necessary to invoke a factory or a resource action are automatically resolved. This allows the injection of service dependencies and context variables, like the request content generated by the request content factory. For example, the route *POST /list* expects the request content to be an instance of the *ListItem* class. The action as one argument of this type, so it is automatically injected. The action has no notions of requests nor responses, it is just a simple method. This is the sort of seamless programming experience the framework tries to reach. This feature is implemented using code from the **symfony/http-kernel** package.

The framework provides two solutions to describe a class as a resource: interfaces and annotations with the **doctrine/annotations** package. The *RoutableInterface* describes a method to retrieve the list of routes of the resource while the *ManufacturableInterface* describes a method for the resource factory. Both methods are static. Similarly, there are two annotations. The *ResourceFactory* annotation can be placed on the class of the resource with a value corresponding to the factory. It can also be place directly on a method of the class, without a value this time. The *Route* annotation can be placed either on the class or on a method of the resource. If placed on a method, the action will be this method.

Now that the resource classes are configured to be served as resources, they can be used by the central object of the framework: the *Api* class. This object requires the class names of the resources. Additionally, it can be provided with a container instance from the **symfony/dependency-injection** package. It will be used for the argument resolution mentioned earlier. Once configured, a single method of this class will create a request object from the global variables (using the **symfony/http-foundation** package), find the corresponding route if any and send back a response.

The ShopList resources

The complete ShopList API is described in the *openapi.yaml* file with the OpenAPI specification. *ShopList* has four resources: the item collection (*Items*), a specific item (*Item*), the list item collection (*ListItems*) and a specific list item (*ListItem*). *Item* and *ListItem* are Doctrine entities (**doctrine/orm** package). Most of the logic in the four classes only uses Doctrine to run. They cover the features required by the recommended semestral work (the database schema is actually the same). Everything else is handled by the Rest framework using annotations.

Configuring the framework

The configuration and execution of the framework is done in the entry point of the API (**webroot/index.php**). It requires the Composer autoloader and the configuration file from **config/configuration.php**. The configuration contains the database connection information, the service definitions for dependency injection and the class names of the resources. The entry point simply creates a container with the definitions from the configuration and an API instance. The API is then run.