Code Review

ENSE 477 2019-2020

Group ZAM
Muhammad Ahmed Zain Abedin Alwin Baby

Client Side Code Review

Introduction

Client side code of the application is made using Vue.js framework.The purpose of this code includes rendering dynamic and reactive web pages to enable end users perform CRUD operations on available resources. Interactions with the server are enabled using 3rd party library Axios.Other responsibilities include parsing and validating inputs to submit to the server as well as potential translation of outputs are handled by the client side code. Advance UI interface is enabled via Vuetify.js, a 3rd party interface library that provides fundamental components. The responsibility of these components is to display information or register events only.

Design

The code in general employs component based design where each UI item is a component in itself and conforms to composition patterns to make complex components that represent a complete page at the end of the hierarchy. Each custom component exists as a single file component that is imported by others for reuse. Each component has a single domain of responsibility that conforms to a single responsibility principle and enables a modular design.

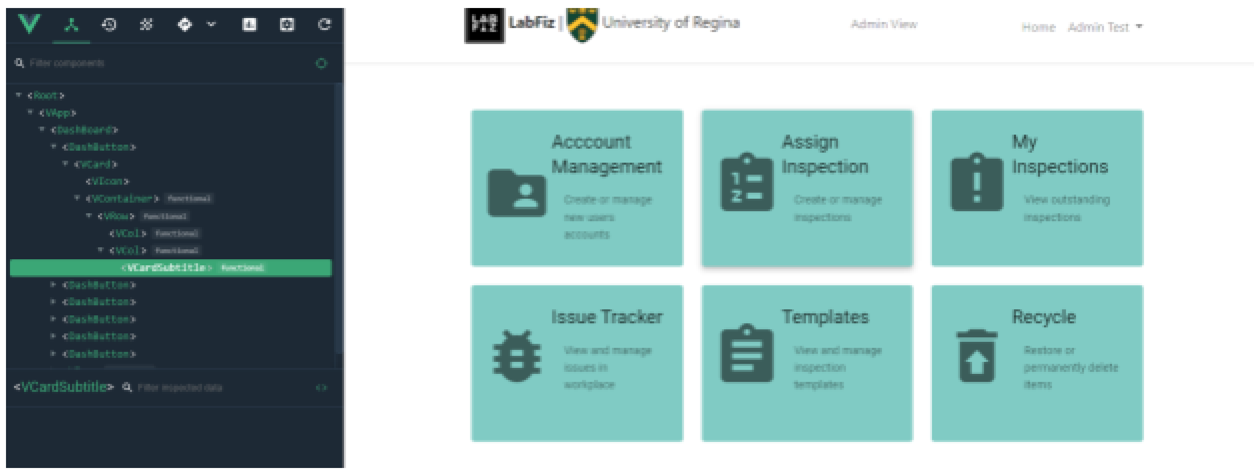

Fig 1- Component decomposition for Dashboard.vue component

General Review

The code successfully performs its duties as mentioned above. All applicable test cases prove the code passes all functional requirements.
The decomposition of components offer higher readability and logical separation of the code.This makes the code easy to maintain or update and has prevented redundancy. General

get and post functions however are implemented differently each time due to component specific functionality and do not conform to this principle despite of similar names in their respective component files.

The component structure scopes the use of variables as private unless explicitly passed on the event bus or as a prop hence the code does not employ any global variables.

The code is written following Vue.js style guide which covers framework specific good practices as well as other general items such as naming convention and code organization. It conforms to most of the rules listed in subsections; “Priority A: Essentials” and “Priority B: Strongly Recommended”. These rules have proven to prevent unexpected behaviour and have increased code readability as per community standards. The table in Appendix A shows a list of applicable rules to the code set.

Performance & Security

Currently the code hosted at either localhost or demo server via AWS Elastic Beanstalk performs efficiently and in an acceptable time. However there is room for improvement in first load and content delivery times.
The client side code is structured to be easily replaced by any other framework such as React or to upgrade the app to a single page application architecture. This design has resulted in inefficiencies in content delivery that can be improved by configuring Webpack configuration that bundles and compiles all Vue.js files to a single javascript file.
The performance can also be improved using Preload.js library to support nit picking of assets during load or delivering particulare compilation of vue file as per pages requirement in current multi page application architecture.This requires further research and experience from the team.

The application utilizes Webpack.js and Babel.js to generate end user usable javascript code. This has been configured to make the code cross browser compatible that does not support ES6 features. This also replaces low performing code with explicit code snippets that may not be readable to humans, hence preventing inner functioning of the code to the general public. Any abnormalities or system errors are caught and handled via applicable UI prompts and necessary input/output manipulation.

Documentation

Appropriate comments are available where code presents complexity however a general approach to write code that is self explanatory is followed. This includes function documentation as well. Data structures are documented in the API documentation.

Testing

The client side code only serves as the interface hence all major testing is done on the server side and on the API layer. All inputs and possible outputs are tested with appropriate headers for appropriate requests to server outside of API testing on the client side as well. Dynamic rendering and input and output availability are tested as mentioned in the End Point Testing document.

Backend Code Review

Brief Checklist

Am I able to understand the code easily?
`Yes`
Is the code written following the coding standards/guidelines?
`Yes`
Is the same code duplicated more than twice?
`No`
Can I unit test / debug the code easily to find the root cause?
`Yes. Run PHPUnit on the CLI.`
Are any functions or classes too big? If yes, is the function or class having too many responsibilities?
`No`

API Endpoint Naming Convention

Nouns are used for endpoints paths

Singular nouns are used for interacting with singular api resources:
`localhost/api/user/{id}`

Plural nouns are used for interacting with multiple api resources:
`localhost/api/users`

API Response Convention:

Response Fields	Description
Status	200 (Ok) or 422 (Unprocessable Entity) or 400 (Bad Request)
Message	Additional information about the response
Data	Json data is under the data key

Response Fields	Description
200	Request was processed successfully.
400	Request could not be processed, but input was valid. A common cause could not be found.
422	Request input data is invalid or malformed.

API Authentication

Api authentication is handled via the token driver module in laravel. This acts as a token based authentication guard. We went with the built in approach as the api is currently being used internally only. Other alternatives going forward could be jwt or passport.

Request header payloads need to declare an api token in order to access most of the endpoints. This is done through Bearer authentication.

Refresh tokens are also used to top up the access tokens as they are expirable by default with a limited lifetime. The access tokens can be topped up using the refresh route.

Bearer Authentication

HTTP requests to protected routes will require the Authorization field set in the HTTP header with a key value of Bearer + `''` + token.

Format:

Authorization: Bearer Token Example:

```
Authorization: Bearer 1shtRTbPCVs2xe7cviyaIAGWClT57y9YwjyVSFerKgXeFDh0LnvdpyM6CUvb
```

API URI Versioning:

Global uri versioning across all endpoints except for the login and register paths to prevent data breaches and potential security issues. The intended purpose of the versioning is allow for easier transitions between major/minor versions of api for clients.

Current client major version is 1.0.

API Design Patterns

Repository and service patterns are used across all api controllers. This allows us to decouple any business logic from higher level modules(like the controller) for api requests and delegate the responsibilities down to lower modules. This also allows us to utilize the single responsibility principle across multiple modules.

Sample Controller logic:

```
class LabController extends Controller
{
    protected $lab_service;

    public function __construct(RestServiceContract $service)
    {
        $this->lab_service = $service;
    }

    public function get($id)
    {
        $res = $this->lab_service->get($id);
        return response($res['response'], $res['status']);
    }

    public function get_all()
    {
        $res = $this->lab_service->get_all();
        return response($res['response'], $res['status']);
    }

    public function create(CreateRequest $request)
    {
        $res = $this->lab_service->create($request);
        return response($res['response'], $res['status']);
    }

    public function delete($id)
    {
        $res = $this->lab_service->delete($id);
        return response($res['response'], $res['status']);
    }
}
```

Exception handling is done through try and catch clauses for any throwing code calls. The errors will propagate back up to controller module and will be returned to the user:

```
class IssueService implements RestServiceContract
{
    protected $user_model, $issue_model;
    public function __construct(User $user, Issue $issue)
    {
        $this->user_model = new ModelRepository($user);
        $this->issue_model = new ModelRepository($issue);
    }

    public function get($id)
    {
        $result = ['status' => '400 (Bad Request)', 'message' => '', 'data' => ''];

        try {
            $result['data'] = $this->issue_model->getById($id);
            $assign_id =$result['data']['assigned_to'];
            $user_assigned = DB::table('users')->where('id',$assign_id)->first();
            $user_name = $user_assigned->first_name . ' '. $user_assigned->last_name;
            $result['data']['user_name'] =$user_name;
            $users =
DB::table('users')->select('id','first_name','last_name')->get();
            $result['data']['users'] =$users;

        } catch (Exception $ex) {
            $result['message'] = ' Could not find issue record.';
            return ['response' => $result, 'status' => 400];
        }

        $result['status'] = '200 (Ok)';
        $result['message'] = 'Issue retrieved successfully.';
        return ['response' => $result, 'status' => 200];
    }
    // ...
```

API Request Validation

Input requests are validated by extending from the FormRequest class in Laravel. This allows us to pre-evaluate code before entering any logic in the api controllers.

```
class CreateRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }
    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //'id' => array('required','int'),
            'title' => array('required', 'regex:/^[\s\w!-@#$^_:;,.]*$/', 'max:250'),
            'room' => array('required', 'string'),
            'assigned_to' => array('required', 'int'),
            'severity' => array('required', 'string'),
            'description' => array('required', 'regex:/^[\s\w!-@#$^_:;,.]*$/',
'max:250'),
            'due_date' =>array('required' ,'date'),

        ];
    }

    /**
     * @param Validator $validator
     */
    protected function failedValidation(Validator $validator)
    {
        throw new HttpResponseException(response()->json(
            [
                'status' => '422 (Unprocessable Entity)',
                'message' => $validator->errors(),
                'data' => ''
            ],
            422)
        );
    }
}
```

Middleware

Route Protection

Custom middleware is used for route-based authorization for specific routes:

```
Route::get('/users', 'Api\Auth\LoginController@get_all')->middleware(['admin_only']);
```

Appendix A: Vue.JS Style Checklist

Rules	Followed
Priority A Rules: Essential	
Multi-word component names	Yes
Component data	Yes
Prop definitions	Yes
Keyed v-for	Yes
Avoid v-if with v-for	Yes
Component style scoping	Yes
Private property names	n/a
Priority B Rules: Strongly Recommended	
Component files	Yes
Single-file component filename casing	Yes
Base component names	Yes
Single-instance component names	No
Tightly coupled component names	Yes
Order of words in component names	No
Self-closing components	Yes
Component name casing in templates	Yes
Component name casing in JS/JSX	n/a
Full-word component names	No
Prop name casing	Yes
Multi-attribute elements	Mostly
Simple expressions in templates	n/a
Simple computed properties	n/a
Quoted attribute values	n/a
Directive shorthands	Yes
Priority C Rules: Recommended	
Component/instance options order	Yes

Element attribute order	No
Empty lines in component/instance options	No
Single-file component top-level element order	Yes
Priority D Rules: Use with Caution	
v-if/v-else-if/v-else without key	No
Element selectors with scoped	Yes
Implicit parent-child communication	Yes
Non-flux state management	n/a