The plan

Before I start writing code for something like this I like to get the plan right in my head first.

For this I usually just write out

* The namespace of each class so I can get my head around the structure.
* The interfaces to ensure I write a consistent and well thought out public API.
* The abstract classes so I can see which methods can be abstracted and which methods need specific implementations.
* And finally the methods each class will need, usually without any of the code required to make them work.

1. Install Laravel =>composer create-project laravel/laravel email\_api
2. .env => connections to be mentioned for the db
3. composer require prettus/l5-repository
4. php artisan entity: email => creates controller, entity, serviceprovider, repository, transformer, presenter, validator, request, emails table class inside db/migrations
5. Run php artisan migrate:fresh –seed to create all tables to db
6. Now we need to register the services classes to the Repositoryserviceprovider
   1. We have three methods
      1. Bind the abstract and concrete class to repositoryservice provider register and add the repositoryserviceprovider to providers array of config/app.php
      2. Bind the abstract and concrete classes to Appserviceprovider as its already added yo config/app.php
      3. Repositoryserviceprovider=>boot() have the bindings and then register repositoryserviceprovider to appserviceprovider ( ***already mentioned in prettus/l5-repository***)
7. **Php artisan vendor:publish –provider providername**  - command used to copy few configuration file in your application from vendor package file.Like you are using a package where need to create a table on database and need to store few demo data.That time your packages will keep database schema and seeding and when you run vendor:publish those schema and seeding file will copy in your application.After that you just run php artisan migrate and php artisan db:seed
8. **Inside Migrations/emails table** 
   1. Add all the fields needed with its datatype
   2. $table->increments('id');
   3. $table->string('username');
   4. $table->string('subject');
   5. $table->text('content');
   6. $table->string('to');
   7. $table->string('from');
   8. $table->timestamps();
9. **Entity/Email.php** which is a model class and implements transformabletrait
   1. As it extends Model class which use GaurdsAttribute trait, we can mention the $fillable fields (Mass assignment refers to sending an array to the model to directly create a new record in Database)
10. **Validator** (talkimg about the most imp methods)
    1. ValidatorInterface => to have some consistent methods
       1. Two constants for assigning the validation rules => const RULE\_CREATE = 'create'; const RULE\_UPDATE = 'update';
       2. With(), passes(),errors()
    2. AbstractValidator => specific implementations for with(), errors(), passesorfail()
       1. With() -> takes in input request in array format and assigns the request data to $this->data returns the validator instance
       2. Errors => assigns errors to error property
       3. Passesorfail => calls the passes() and throws validationException if the validation fails
    3. LaravelValidator => defines the passes()
       1. Passes() -> takes in the rules (array format), calls the validator::make method from the illuminate validator factory to do the validation
       2. Checks validations->fails() true or not
    4. Emailvalidator inside validator
       1. Uses Validator interface and extends the Laravel validator
       2. It defines the rules array
          1. Mentions both create and update separately if needed or single rules array for both end points
       3. Can define messages => if needed to have custom messages
          1. Array
          2. [‘to.required’ => ‘To emailaddress is required’]
       4. Can defines $attributes array to have custom name for attributes
11. **Presenter and Transformer -** league fractallibrary provides with a presentation or transformation layer for complex data output. There are two methods to achieve
    1. Entity/ email which implements transformable,
       1. so it can define method: transform() and returns array of data (current model fields)
       2. In the Emailrepository define method presenter to return ModelFactalPresenter which itself is modeltransformer
       3. this model transformer has transform() which passes $model which is transformable object and calls $model->transform()
    2. EmailPresenter which is a fractalpresenter has getTransformer()
       1. It returns the Emailtransformer which is a transformerabstract instance
       2. Emailtransformer has a transform() which takes in the transformable model as argument can returns the array
       3. In the emailrepository -> presenter method returns EmailPresenter which is a ModalFractPresenter
       4. Instead of presenter() in repository , we can have it called inside the controller using $this->repository->setPresenter();
12. **Repository pattern**
    1. **It’s** a kind of container where data access logic is stored. It hides the details of data access logic from business logic. In other words, we allow business logic to access the data object without having knowledge of underlying data access architecture
       1. **It’s all about interface =?> we have repositoryInterface which has the method signatures. This helps you in making a very consistent source**
       2. **AbstractRepository which implements the repositoryInterface.**
          1. **$this->app = Application**
          2. **Model() is abstract**
          3. **makeModel() -> which get $this-> model which is an instance of Model and instantiate it using make()**
          4. **implements all the signatures mentioned in the interface eg: all() -> $this->model->get($coulmns)**
       3. **CriteriaInterface = > for applying criteria on the model**
          1. **Which has apply($model, respository instance of repositoryInterface)**
       4. **Criteriarepositoryinterface => which has method signatures like pop, push, getBy, skip, applycriteria**
       5. **Now the abstract repository has to implement criteriaRepositoryinterface too so that the criteria can be applied on the model**
          1. **Applyciteria => checks if the sjkipcriteria is false and checks if the criteria is an istance of criterisInterface and do apply($model, $this)**
          2. **pushCriteria => push criteria to the criteria property**
       6. **Define RequestCriteria implements CriteriaInterface and apply method defined for all the types of criteria added in the config/repository.php => repository.criteria.params []**
    2. **Controller class=> we can pushCriteria(app(\prettus\requestCriteria) ->all()**
13. **Caching**
    1. CacheInterface to be implemented and the cacherepository trait to be used inside the emailEloquentRepoisitory
    2. Config/repository.php => enable cache
14. **Routes**
    1. Route has prefix v1,
    2. middleware([])=> array of middlewares-> which is common to all the routes
    3. we put all the routes inside the group which accepts a callback function
    4. route has been given a common name ‘émails.’
    5. Individual route has been give -> name (‘índex’) so that the name of route would be emails.index
    6. Can also mention the namespace to point the controller used
    7. routeHelper class which helps in uterating through the pai/v1 folders and files to get the list of all routes defined
15. **Mail Configuration**
    1. .env => needs to mention all the available mail drivers configuration
    2. Config/mail.php
       1. Mail\_mailer env variable should be failover by default
       2. Mailers array -> mention all the mail configurations
       3. Failover =>
          1. mailer switching sequence for failover setup
          2. we need to mention how we order the transport on failover
16. **Controller**
    1. Index() => which returns the data
    2. Store() => create an entry into the db table email
       1. Validation checks are done by the create method itself
       2. Also an event RepositoryEntityCreating is being sent after the validation checks are done
       3. Two ways to approach the mail sending
          1. Event dispatching and event listener =>
             1. event created and dispatchedif email entry is created
             2. Event listener which extends Mailable and implements shouldqueue and handles the sending/queue of email using Mail façade
          2. sendEmail method which extends Mailable class which implements shoudlQueue
             1. Fill in the emailatrributes values
             2. Also fill in dynamicMailer markdown template with all the values
             3. And sending/queue of email using Mail façade

Example for transform having relationship value too in the array returning model attributes

<?php

use Acme\Model\Book;

use League\Fractal;

$books = Book::all();

$resource = new Fractal\Resource\Collection($books, function(Book $book) {

return [

'id' => (int) $book->id,

'title' => $book->title,

'year' => $book->yr,

'author' => [

'name' => $book->author\_name,

'email' => $book->author\_email,

],]}];

$avaiilableIncludes= [‘Author’];

Public function includeAuthor(Book $book){

$author = $book->author;

return $this->item($author, new AuthorTransformer);

} this is lazy loading

Fractal\Resource\Collection and Fractal\Resource\Item =? Both are resourceAbstract

Resourceabstract => construct => $data (array of object implementing arrayiterator) and transformer

Validator

Reference : <https://www.culttt.com/2014/01/13/advanced-validation-service-laravel-4/>

<https://github.com/andersao/laravel-validator>

<https://bosnadev.com/2015/03/07/using-repository-pattern-in-laravel-5/?utm_source=prettus-l5-repository&utm_medium=readme&utm_campaign=prettus-l5-repository>

Validation rules : <https://stackoverflow.com/questions/49057695/laravel-validate-unique-fields-on-update>