Advance Web App Code Snippet

Router methods

```
Route::any('/home', 'HomeController@index')

Route::match(['POST', 'PUT'], '/home', 'HomeController@index')

Route::post('/home', 'HomeController@store') //post/put/get/patch/delete

Route::redirect('/home', '/newHome', 301) //301=perma move code

Route::view('/home', 'home.index', ['extraData' => 'important to show in front end'])

Route::get('/user/{userId?}', function($userId = 'something'){
    //do something with the $userId
}) //we can also pass this to another controller

Route::resource('/home', 'HomeController') // use 'php artisan route:list' to check all the routes
```

CSRF form

include this line in the front-end form

```
{{ csrf_field() }}
//or
@csrf
```

Retrieve input

```
$request->query('studentId') // get the query param in the url

$request->input('name') // get the name in the form request

$request->name //if the data is passed through body
```

Basic CRUD API

```
public function index(){
    $books = Book::all();
    return response()->json([
        'books' => $books
```

```php
    ], 200)
}

public function show($request){
    $book = Book::find($request->bookId);
    return response()->json([
        'book' => $book
    ], 200)
}

public function store($request){
    $book = new Book();
    $book->fill($request->all())
    $book->save()
    return response()->json([
        'success' => true
    ], 201)
}

public function update($request){
    $book = Book::find($request->bookId)
    if(!$book){
        return response()->json([
            'errors' => 'Book not found'
        ], 404)
    }

    $book->update($request->all())
    return response()->json([
        'success' => true
    ], 204)
}

public function destroy($request){
    $book = Book::find($request->id)
    if(!$book){
        return response()->json([
            'errors' => 'Book not found'
        ], 404)
    }

    $book->delete()
    return response()->json([
        'success' => true
    ], 204)
}
```

Eloquent Basic

```
php artisan make:model Book --migration // this will create Book model and the migration table

// model class looks like this

class Book extends Model{
    user SoftDeletes; //Enable soft delete

    protected $dates = ['deleted_at']
    // if we were going to soft delete, go back to the migration file and add the following line
    $table->softDeletes()
    // ---

    protected $table = 'BookName'

    protected $fillable = [
        'name',
        'author'
    ] // mass-assignable for name and author

    protected $guarded = [
        'publisher'
    ] // mass-assignable for everything except publisher

    protected $primaryKey = 'bookId'
}

// some basic query methods using Eloquent
where('fieldName', 'condition', 'value')
get()
find() // find by primary key
all()
orderBy('fieldName', 'asc/desc')
first()
firstOrFail()
findOrFail()
pluck('fieldName') //to take out this field ONLY
count()
max()
latest()
oldest()
// limit the query count
take()
limit()

// skip the row
skip()
offset()

paginate(10) //with all the page information
simplePaginate(10) //only next and previous page
```

```
// update and insert
$photo = new Photo();
$photo->title = $request->title
$photo->fill($request->all)
$photo->save()
firstOrCreate() // will save immediately
firstOrNew() // will not save, you have to run save() explicitly

trashed() // check whether it is soft deleted
withTrashed() // show all the soft deleted rows as well
onlyTrashed() // only the rows that are soft deleted
restore() // restore soft deleted rows
forceDelete() // to ignore the soft delete
```

Eloquent relationships

```
// one-to-one
public function phone(){
    return $this->hasOne(Phone::class)
}

public function user(){
    return $this->belongsTo(User::class, 'customForeignKey', 'customParentPrimaryKey')
}

// one-to-many
public function photos(){
    return $this->hasMany(Photo::class);
}

public function album(){
    return $this->belongsTo(Album::class);
}

// many-to-many
public function Book(){
    return $this->belongsToMany(Book::class)
}

public function Author(){
    return $this->belongsToMany(Author::class)
}
//Pivot table is author_book, fields are author_id, book_id

//to do the relation query
$members = Member::with('groups')->get()
```

Eloquent Search

```
$members = Member::with('groups')->when($request->query('name'), function($query) use $request{
    return $query->where('name', 'like', '%'.$request->query(name).'%')
})
```

Resource

command to make a resource file

```
php artisan make:resource BookResource
```

command to make a collection file

```
php artisan make:resource BookCollection
//when the file name include 'Collection', the file will automatically extends from Collection
```

Using resource to pluck the fields

Within the resource file, look for toArray() function

```
public function toArray($request){
    return [
        'name' => $this->name
    ]
}
```

Use resource in a controller

```
public function show(){
    // logic here...
    return new BookResource($book);
}

public function index(){
    // logic here...
    return BookResource::collection($book);
}
```

Further usage for collection

Note that this cannot pluck the field in the collection

```
public function toArray(){
    return[
        'data' => $this->collection,
        'extraData' => 'Laravel is weak'
    ]
}
```

Pluck fields with collection resource

```
public function toArray(){
    return[
        'data' => BookResource::collection($this->collection),
        'extraData' => 'Laravel is weak'
    ]
}
```

Side note: Pagination will still work well with the Resource wrapping

Use when() and is_null() function to conditioning drop a field

```
public function toArray($request){
    return [
        'name' => $this->name
        'age' => $this->when(!is_null($this->age), $this->age)
    ]
}
```

Conditional Relationships query with resource;

with the help of whenLoaded() function

```
public function toArray($request){
    return [
        'id' => $this->id,
        'trips' => TripResource::collection($this->whenLoaded('trips')),
    ];
}
//to fullfill the whenLoaded(), do this in controller
public function index(){
    $vehicles = Vehicle::with('trips')->all()
    return new VehicleCollection($vehicles)
}
```

Validation ($request->validate)

Validation within a controller

```
public function store(Request $request){
    try{
        $request->validate([
            'name' => 'bail|required|regex:xxx|max:3|min:2|unique|email|sometimes|nullable'
        ])
        // use dot notation to access object
        // add bail to fail the validation right away
        // use sometimes when you want to validate the field only if it is there
        // logic here...
    }catch(ValidationException $exp){
        return response()->json([
            'errors' -> $exp->errors()
        ], 500);
    }
}
```

Validation with form request

use the following command the create the file

```
php artisan make:request GuguRequest
```

After the file creation, implement the rules() function

```
public function rules(){
    return [
        'name' => 'bail|required|regex:xxx|max:3|min:2|unique|email|sometimes|nullable'
    ]
}
```

To use this form request validation, just change the request type like so

```
// original
public function store(Request $reqeust){
    //
}
// change to this
public function store(GuguRequest $request){
    //
}
```

With this, you dont have to catch the exception as the form request will catch and handle it
Note that the default form request will redirect user to a webpage, we do not want that in API implementation. To change the behaviour, create the following file

```
abstract class ApiFormRequest extends FormRequest{
    // Override this function to implement your own exception handling
    protected function failedValidation(Validator $validator){
        throw new HttpResponseException(response()->json([
            'errors' => $validator->errors()
        ]), 500)
    }
}
```

To use this ApiFormRequest, change your request file to something look like this

```
class GuguRequest extends ApiFormRequest
```

Add after hook to form request

applicable to both FormRequest and ApiFormRequest

```
public function withValidator(Validator $validator){
   $validator->after(function($validator){
       $validator->errors()->add('messsageField', 'actual message')
   })
}
```

Customize error message in form request

applicable to both FormRequest and ApiFormRequest

```
public function messages(){
    return [
        'fieldName.condition' => 'your custom message',
        'name.required' => "Name field is required so that I don't have to call you 'OI'"
    ]
}
```

Creating validator within controller

```
public function store(Request $request){
    $customMsgs = [
        'name.required' => "Name field is required so that I don't have to call you 'OI'"
    ]
    $validator = Validator::make($request->all(), [
        'name' => 'required'
    ], $customMsgs) // third param is optional to create custom message

    // after hook
    $validator->after(function($validator){
        $validator->errors()->add('messageField', 'actual message')
    })

    if($validator->fails()){
        return response()->json([
            'errors' => $validator->errors()
        ], 500)
    }
    // other logics here
}
```

Deeper into Validator

We can validate the fields if certain conditions are met by using sometimes() method

```
$customMsgs = [
    'name.required' => "Name field is required so that I don't have to call you 'OI'"
]

$validator = Validator::make($request->all(), [
    'name' => 'required'
], $customMsgs) // third param is optional to create custom message

$validator->sometimes('the field you want to conditioning validate', 'validation rules here', funct
ion($input){
    return $input->name === 'Gugu'
})

//example
$validator->sometimes('reason', 'required|min:10', function($input){
    return $input->name === 'Gugu'
})
// A reason is required if you name is Gugu.
```

Using rules

create the rule file with the following command

```
php artisan make:rule HamGaFuGui
```

alter the passes() function and message() function accordingly

```
public function passes($attribute, $value){
    // return your logic(true/ false) here

    return strtoupper($value) === $value
}

public function message(){
    return 'Ni mei you mama HAHAHAH'
}
```

To use the rules, just write inside your validation rules like so

```
$request->validate([
    'name' => [new HamGaFuGui, 'required']
])
```

Cookie

```
// get cookie
$value = $request->cookie('name')
$value = Cookie::get('name')

// set cookie
return response()->cookie('fieldName', 'fieldValue')
```

Session

```
// get session
$value = $request->session()->get('keyName', 'fallbackValue') // second param is optional
$value = session('keyName', 'fallbackValue') // second param is optional
$value = $request->session()->all()

// session key checking
if($request->session()->has('keyName')) // will return false if value is null
if($request->session()->exists('keyName')) // will return true if value is null

// store session
$request->session()->put('key', 'value')
session(['key' => 'value'])

// delete session
$value = $request->session()->pull('key', 'default'); //pull will return the session value also
$request->session()->forget('key')
$request->session()->flush()
```

```
// To keep your session data for next request
$request->session()->flash('status', 'fail')
return redirect('login')->withInput(
    $request->flash()
)

//To get the session data from client side
<input type="text" name="username" value="{{ old('username') }}">
```

Authentication

command to create the auth

```
php artisan make:auth
```

To customize redirect destination after login, register, reset password go to the respective controller and add this line in the class

```
protected $redirectTo = '/'
// or if you want to do some logic before redirect, use a function
protected function redirectTo(){
    return '/'
}
```

Some authentication methods

```
Auth::user() // get user instance
Auth::id() // get user id
Auth::check() // to check if the user is authenticated
Auth::attempt()
Auth::logout()
```

Protecting routes

```
// we can do at router or controller
// Router
Router::post('form', 'SomeController@function')->middleware('auth')

// create a function in controller class
public function __construct(){
    $this->middleware('auth', ['except' => ['login']])
    // second param is optional
}
```

Authenticate user manually

```
$credentials = ['email'=>$request->email, 'password'=>$request->password]
if(Auth::attempt($credentials)){
    // passed
    return redirect()->intended('home')
}

// if we want to check if the user account is activated or not we can do like this
$credentials = ['email'=>$request->email, 'password'=>$request->password, 'active' => 1]

// if we want to remember user
if(Auth::attempt($credentials, true)){
    // passed
    return redirect()->intended('home')
}
// make sure the user table have 'remember_token' column
```

JWT

to create the secret key

```
php artisan jwt:secret
```

update user class to implements JWTSubject

```
class User extends Authenticatable implements JWTSubject{
    public function getJWTIdentifier(){
        return $this->getKey()
    }

    public function getJWTCustomClaims(){
        return []
    }
}
```

define the routes for the Auth API

```
Route::middleware('auth:api')->namespace('Auth')->prefix('auth')->group(function(){
    Route::post('login', 'AuthController@login'),
    //...
})
// url to login function is '/api/auth/login'
```

Implementing the logic.
Prepare a helper method to return the JWT token

```
protected function JWTResponse($token){
    return response()->json([
        'token' => $token,
        'expires_in' => 3600,
        'token_type' => 'bearer'
    ])
}
```

Login function

```
public function login($request){
    $credentials = ['email' => $request->email, 'password' => $reqeust->password]
    if(!$token = auth()->atttempt($credentials)){
        // fail login
        return response()->json([
            'errors': 'Invalid email or password'
        ], 500)
    }

    // use the helper method created before to return the token
    return $this->JWTResponse($token)
}
```

Gates

Defining gate in App\Providers\AuthServiceProvider in the **boot()** function

```
Gate::define('create-post', function($user, $post){
    return $user->id == $post->user_id
})
// instead of inline function, we can also route it to a controller
Gate::define('create-post', 'PostGate@create')
// Resource gate is also available, but i dont recommend it as the name constraint may confuse your
self
Gate::resource('post', 'PostGate')
```

Applying the gates

```
// in the controller
if (Gate::allows('create-post', $post)) {
    // The current user can update the post...
}
```

```
if (Gate::denies('create-post', $post)) {
    // The current user can't update the post...
    // straight away return here to prevent the code from continuing the function
}
```

Policies

command to create policy file

```
php artisan make:policy PostPolicy --model=Post
```

Keep in mind that the model is very important in policy as it uses the model class to identify which policy should be used.

Define policy

Navigate to the created policy file

```
// the function name can be whatever name you like
public function delete(User $user, Post $post){
    return $user->id == $post->user_id
}
```

Special case for admin, admin can access to everything, inside the policy, you can create a **before()** function to allow everything if it is a user

```
public function before($user, $ability){
    if($user->isAdmin){
        return true
    }
}
```

Register newly defined policy

Go to the same file where we define gate, App\Providers\AuthServiceProvider, within the class

```
protected $policies = [
    Post::class => PostPolicy::class,
]
```

Apply the policy

Apply via controller

```
// using the user model
$user = Auth::user()
if($user->can('delete', $post)){
    // Note that you dont have to pass in the user variable again
}
// in case the action does not require model, we can pass in the model class for identification
if($user->can('delete', Post::class)){
    //
}
```

Apply via route

```
Route::delete('/post/{post}', 'PostController@destroy')->middleware('can:delete, post')
// if it does not require model, again, we can pass in the model class
Route::delete('/post/{post}', 'PostController@destroy')->middleware('can:delete, App\Post')
```

Apply via controller helper

```
//with in any function of any controller
$this->authorize('delete', $post)
// if it does not require model
$this->authorize('delete', Post::class)
```

Front end "protection"

If the user cannot perform certain action, we might want to consider hiding the button, we can do so by using the policy as well

```
@can('update', $post)
    // show update button
@elsecan('create', $post)
    // show create button
@endcan

// if without model
@can('update', Post::class)
    // show update button
@elsecan('create', Post::class)
    // show create button
@endcan
```

Myth behind @can

@can is actually a shorthand notation for the following code

```
@if(Auth::user()->can('update', $post))
    //
@endif
```
