Project documentation

Dashboard Section

1. **Admin Controller**
   1. **all(Request $request) Method**

🔹 Overview  
This method retrieves a list of admin users with role ID 7. It supports optional pagination, based on the paginate parameter provided in the request.

🔹 **Key Functionality**

1-**Check for Pagination Requirement**

* Verifies if the request includes a paginate parameter.
* If paginate is 1: Fetch paginated admin data.
* If paginate is 0: Fetch all admin records without pagination.
* If paginate is not provided: Defaults to pagination with 20 items per page.

2-**Fetch Admin Records**

* Retrieves users from the users table where role\_id = 7.
* Sorts the results by latest (most recently created admins come first).
* Uses Laravel’s paginate() or get() method depending on the condition.

3-**Format Response with Resources**

* Wraps the result in EmployeeResource::collection() for consistent API output.
* If paginated, constructs a pagination array including:
  + total items
  + per\_page count
  + current\_page
  + total\_pages

4-**Return Standardized API Response**

* Uses a returnData() method to return the response with a message (e.g., \_\_('api.employee\_all')).
* If an exception occurs, catches it and returns a 422 error with a message using returnError().

🔹 **Error Handling**

* All logic is wrapped in a try-catch block.
* Any exceptions are caught and returned as a structured error response.
  1. **get($id) Method**

**Overview**  
This method retrieves a specific admin user by ID, only if they have the admin role (role\_id = 7).

🔹 **Key Functionality**

1-**Find Admin by ID**

* Uses Eloquent’s find() method to retrieve a user with the given ID and ensures they have role\_id = 7.
* If a matching user is found, proceeds to the next step.

2-**Return Admin Data**

* Wraps the admin record in EmployeeResource for consistent formatting.
* Returns the response using returnData() with a success message (\_\_('api.admin\_get')).

3-**Handle Not Found Case**

* If no admin is found, returns a 404 error using returnError() and a not-found message (\_\_('api.admin\_not\_found')).

🔹 **Error Handling**

* Encloses the logic in a try-catch block.
* If an exception occurs during the process, catches it and returns a 422 error with an error message (\_\_('api.error\_happened') + exception message).
  1. **add(StoreAdminRequest $request) Method**

🔹 **Overview**  
This method handles the creation of a new admin user along with a linked client profile for system integration. It processes uploaded images, saves user data, and ensures a consistent relationship between the admin and client models.

🔹 **Key Functionality**

1-**Handle Image Upload (Optional)**

* Checks if the request contains an uploaded image.
* If present, processes the image using handleFile() method.
* The processed path is stored and used when creating the user.

2-**Create Admin User**

* Creates a new user with the role of admin (role\_id = 7).
* Uses data from the validated StoreAdminRequest.
* Includes fields such as name, email, phone, password, status, city\_id, area\_id, and the uploaded image.

3-**Create Linked Client Profile**

* If the admin was created successfully, a corresponding Client record is created.
* The client is assigned a default role (type = 3), active status (status = 1), and default activation\_code.
* Links the client to the admin via admin\_id.
* Sets default values like gender and complete\_data.

4-**Link Admin to Client**

* After creating the client, sets the client\_id field on the admin record.
* Saves the updated admin instance.

5-**Return API Response**

* Returns the newly created admin data wrapped in EmployeeResource.
* Sends a success message using \_\_('api.admin\_add').

🔹 **Error Handling**

* All logic is wrapped in a try-catch block.
* On failure, a 422 error is returned with the caught exception message appended to a localized error string (\_\_('api.error\_happened')).
  1. **update($id, UpdateAdminRequest $request) Method**

🔹 **Overview**  
This method updates an existing admin user’s profile (with role\_id = 7) and their associated client profile. It handles optional updates to profile information, password, and image uploads, including deletion of the old image if replaced.

🔹 **Key Functionality**

1-**Find Admin by ID**

* Searches for the user with the given $id and verifies the role is admin (role\_id = 7).
* If not found, returns a 404 error response.

2-**Handle New Image Upload (If Present)**

* Checks if the request contains a new image file.
* If provided:
  + Processes and saves the new image using handleFile().
  + Deletes the old image from the server if it exists using File::delete() to avoid clutter.

3-**Update Admin Details**

* Updates the admin’s profile with values from the request:
  + Uses existing values as fallback if the request values are null.
  + Updates name, email, phone, role\_id, status, city\_id, area\_id, and image.

4-**Update Password (If Provided)**

* If the password is included in the request, updates it separately.

5-**Update Associated Client Record**

* Finds the Client associated with the admin’s client\_id.
* Updates client fields such as username, email, phone, status, and city\_id, using admin request data or defaulting to existing values.

6-**Return API Response**

* Returns the updated admin data using EmployeeResource.
* Sends a localized success message with \_\_('api.admin\_update').

🔹 **Error Handling**

* All logic is wrapped in a try-catch block.
* If any error occurs during the process, returns a 422 error with the exception message appended to \_\_('api.error\_happened').
  1. **destroy($admin\_id) Method**

🔹 **Overview**  
This method deletes an admin user (with role\_id = 7) from the system based on the provided ID. It also removes the associated profile image from the file system if it exists.

🔹 **Key Functionality**

1-**Find Admin by ID**

* Searches for the admin with the given $admin\_id and ensures the user has a role\_id of 7.
* If no such admin exists, a 404 error response is returned using \_\_('api.admin\_not\_found').

2-**Delete Admin Image (If Exists)**

* Checks if the admin has a profile image saved.
* If so:
  + Retrieves the full path of the image using public\_path().
  + Deletes the image file using File::delete() if it exists on the server.

3-**Delete Admin Record**

* Calls the delete() method on the admin model to remove the database record.

4-**Return API Response**

* Returns a success response with a localized message: \_\_('api.admin\_delete').

🔹 **Error Handling**

* Wrapped in a try-catch block to handle any unexpected errors.
* If an exception occurs, it returns a 422 error response with the exception message appended to \_\_('api.error\_happened').

1. **Area Controller**
   1. **all(Request $request) Method**

🔹 **Overview**  
This method retrieves a list of all areas in the system, optionally filtered by a specific city ID. It also loads related data for each area such as points and representatives.

🔹 **Key Functionality**

1-**Initialize Area Query**

* Starts a query on the Area model using Area::query() to allow conditional clauses.

2-**Optional Filtering by City**

* If the city\_id is present in the request, the query filters areas that belong to the specified city.

3-**Eager Load Relationships**

* Loads related points and representatives using Eloquent’s with() to reduce the number of queries.

4-**Fetch & Return Results**

* Retrieves the areas ordered by the latest entries using latest()->get().
* Wraps the results using AreaResource::collection() for consistent API formatting.
* Returns the formatted data with a localized success message (\_\_('api.area\_all')).

🔹 **Error Handling**

* The operation is enclosed in a try-catch block.
* If any exception occurs, it returns a 422 error response with a descriptive message:  
  \_\_('api.error\_happened') . $error->getMessage().
  1. **get($id) Method**

🔹 **Overview**  
This method retrieves a specific area by its ID, including its associated points and representatives.

🔹 **Key Functionality**

1-**Find Area by ID**

* Uses Area::find($id) with with(['points', 'representatives']) to eager load related data.
* Ensures efficient querying by fetching related points and representatives in a single call.

2-**Check if Area Exists**

* If no area is found, returns a 404 error with the message from \_\_('api.area\_not\_found').

3-**Return Area Data**

* If found, wraps the area data using AreaResource for consistent API formatting.
* Returns the result with a success message from \_\_('api.area\_get').

🔹 **Error Handling**

* Enclosed in a try-catch block to catch any unexpected errors.
* If an exception occurs, returns a 422 error with a descriptive message:  
  \_\_('api.error\_happened') . $error->getMessage().
  1. **add(storeAreaRequest $request) Method**

🔹 **Overview**  
Create a new area and associates it with points and representatives if provided.

🔹 **Key Functionality**

1-**Create New Area**

* Creates a new Area using the fields:
  + name
  + city\_id
  + status

2-**Create Points**

* Uses $area->points()->createMany($request->points) to associate multiple points with the area using Laravel’s createMany() method.
* Assumes each item in $request->points is an array with the correct structure for a Point model.

3-**Assign Representatives (Optional)**

* If representative\_id is passed, loops through it to build a bulk insert array for the RepresntiveRegions model.
* Bulk inserts representative-region associations using RepresntiveRegions::insert($data).

🔹 **Return Response**

* Returns the newly created area wrapped in an AreaResource.
* Response includes a localized success message from \_\_('api.area\_add').

🔹 **Error Handling**

* Wrapped in a try-catch block to catch and report any unexpected issues.
* Returns a 422 error with a custom message on failure.
  1. **update(updateAreaRequest $request) Method**

**🔹 Overview**

Updates an existing area by modifying its main details and replacing its related points and representatives.

**🔹 Key Functionality**

**1-Find Area**

* Retrieves the area record based on the id provided in the request.
* If the area is not found, returns a 404 error with a localized message.

**2-Update Area Fields**

* Updates the area's core attributes such as:
  + name
  + city\_id
  + status
* Retains the existing values for any fields not provided in the request.

**3-Update Points**

* Deletes all existing points related to the area.
* Replaces them with the new set of points provided in the request using the relationship.
* Assumes $request->points contains a valid array of points.

**4-Update Representatives (Optional)**

* If representative\_id is included in the request:
  + Deletes old representative associations for this area.
  + Loops through the new representative\_id values.
  + Prepares and performs a bulk insert into the RepresntiveRegions table to associate representatives with the area.

**🔹 Return Response**

* Returns the updated area wrapped in AreaResource.
* Includes a localized success message from \_\_('api.area\_update').

**🔹 Error Handling**

* Entire logic is wrapped in a try-catch block.
* If any exception occurs, it returns a 422 error response with a localized error message indicating something went wrong.
  1. **destroy($id) Method**

### 🔹 Overview

Deletes an existing area and all its associated points from the database.

### 🔹 Key Functionality

#### 1-Find Area

* Searches for the area using the provided id.
* If not found, returns a 404 error with a localized message.

#### 2-Delete Points

* Deletes all related points of the area using the relationship $area->points()->delete().

#### 3-Delete Area

* After points are deleted, removes the area itself from the database using $area->delete().

### 🔹 Return Response

* Returns a success message using \_\_('api.area\_delete').

### 🔹 Error Handling

* Entire operation is wrapped in a try-catch block.
* On any exception, returns a 422 error with a localized error message indicating the issue.
  1. **allPoints() Method**

### 🔹 Overview

Retrieves all areas with their associated points and returns only the latitude and longitude of each point.

### 🔹 Key Functionality

#### 1-Retrieve Areas with Points

* Uses Eloquent's with('points') to eager-load all areas along with their related points.

#### 2-Extract Coordinates

* Uses map() to iterate over each area and its points.
* For each point, extracts only the lat and lng fields and builds a simplified structure.

#### 3-Format as Array

* Converts the mapped collection into a plain array using toArray().

### 🔹 Return Response

* Returns the collected coordinates under the points key in a standardized success response.

### 🔹 Error Handling

* Entire method is wrapped in a try-catch block.
* Returns a 422 error with a localized message if an exception occurs.

1. **Auth Controller**
   1. **login(LoginRequest $request) Method**

### 🔹 Overview

Handles user login by verifying credentials and assigning an activation code if the login is successful.

### 🔹 Key Functionality

#### 1-Authentication Attempt

* Uses auth()->attempt() to verify that the provided email and password are valid.

#### 2-Assign Activation Code

* On successful authentication, fetches the user using their email.
* Sets a static activation code (1234) and saves it to the user record.  
  (Typically, this should be a randomly generated secure code.)

#### 3-Prepare and Send SMS

* Constructs an Arabic message with the activation code.
* SMS sending logic is present but currently commented out (sendSms() call).

### 🔹 Return Response

* Returns a success response with a localized message indicating that the activation code has been sent.
* On authentication failure, returns an error with a 403 status code and a message that the email or password is incorrect.

### 🔹 Error Handling

* Wrapped in a try-catch block to catch any unexpected exceptions.
* On error, returns a 403 response with the exception message.
  1. **verifyCode(Request $request) Method**

### 🔹 Overview

Verifies the activation code (OTP) sent to the user and issues an API token upon successful verification.

### 🔹 Key Functionality

#### 1-Retrieve User

* Fetches the user based on the provided email.

#### 2-Verify Activation Code

* Compares the user's stored activation code (activation\_code) with the one provided in the request (otp).

#### 3-Token Generation (On Success)

* If the OTP matches, a new API token is generated using Laravel Sanctum (createToken).
* The token is then prefixed with "Bearer" and added to the response data.

### 🔹 Return Response

* **On Success**:  
  Returns the authenticated user wrapped in AuthResource along with a success message (تم تسجيل الدخول بنجاح).
* **On Failure**:  
  Returns a 403 error with a message indicating that the activation code is incorrect.

### 🔹 Error Handling

* Enclosed in a try-catch block.
* If any exception occurs during execution, a 403 error is returned with the exception message.

1. **Brand Controller**
   1. **all(Request $request) Method**

### 🔹 Overview

Retrieves all brands from the database, with support for optional pagination based on the incoming request.

### 🔹 Key Functionality

#### 1-Pagination Check

* Checks the request for the paginate parameter:
  + If paginate = 1: returns paginated data (10 items per page).
  + If paginate = 0: returns all data without pagination.
  + If paginate is not provided: defaults to paginated data (10 items per page).

#### 2-Fetch Brands

* Uses Brand::latest() to order brands by the latest created.
* Depending on pagination logic, either:
  + Uses paginate(10) to get paginated results, or
  + Uses get() to retrieve all results.

#### 3-Format Pagination Data

* If pagination is used, constructs a custom pagination array:
  + total
  + per\_page
  + current\_page
  + total\_pages

### 🔹 Return Response

* Wraps the result in brandsResource::collection($brands) for consistent API formatting.
* Returns a localized message from \_\_('api.brand\_all').
* If pagination is used, also returns the pagination metadata.

### 🔹 Error Handling

* Entire logic is wrapped in a try-catch block.
* On failure, returns a 422 error with the exception message using \_\_('api.error\_happened').
  1. **get($brand\_id) Method**

### 🔹 Overview

Retrieves a single brand by its unique identifier (brand\_id).

### 🔹 Key Functionality

#### 1-Find Brand

* Uses Brand::find($brand\_id) to locate the brand by its ID.
* Checks if the brand exists:
  + If **not found**, returns a 404 error with a localized message \_\_('api.brand\_not\_found').

#### 2-Wrap Brand Resource

* If the brand exists, wraps it in a brandsResource to ensure consistent API response structure.

### 🔹 Return Response

* Returns the found brand in a key "brand" using brandsResource.
* Includes a localized success message from \_\_('api.brand\_get').

### 🔹 Error Handling

* Wrapped in a try-catch block to gracefully handle unexpected errors.
* On exception, returns a 422 error with the exception message and \_\_('api.error\_happened').
  1. **add(StoreBrandRequest $request) Method**

### 🔹 Overview

Creates a new brand and optionally uploads an image if provided in the request.

### 🔹 Key Functionality

#### 1-Handle Image Upload (Optional)

* Checks if the request includes an uploaded image using $request->file('image').
* If present, uses a helper method handleFile() to upload the image and retrieve its path.
* The uploaded image path is stored and later saved with the brand.

#### 2-Create Brand

* Creates a new Brand using:
  + name from the request.
  + image path if uploaded, otherwise null.
  + status from the request.

### 🔹 Return Response

* Returns the newly created brand wrapped in brandsResource.
* Includes a localized success message from \_\_('api.brand\_add').

### 🔹 Error Handling

* No explicit try-catch in this method, which assumes StoreBrandRequest handles validation and file upload is managed correctly.
* If needed, consider adding try-catch for improved robustness.
  1. **update(UpdateBrandRequest $request, $brand\_id) Method**

### 🔹 Overview

Updates an existing brand's information, including its name, image (if provided), and status.

### 🔹 Key Functionality

#### 1-Find Brand

* Retrieves the Brand model by its id ($brand\_id).
* If the brand is not found, returns a 404 error with the message 'api.brand\_not\_found'.

#### 2-Handle Image Upload (Optional)

* If a new image is uploaded in the request ($request->file('image')), it handles the upload via the handleFile() method and updates the image path.
* If no new image is provided, the existing image path is retained ($brand->image).

#### 3-Update Brand

* Updates the Brand using the provided fields in the request:
  + name (defaults to the current brand's name if not provided).
  + image (updates to the new image path if provided, otherwise retains the existing one).
  + status (updates to the new status from the request).

### 🔹 Return Response

* Returns the updated brand wrapped in brandsResource.
* Includes a localized success message from \_\_('api.brand\_update').

### 🔹 Error Handling

* The method is wrapped in a try-catch block to handle unexpected errors.
* If an error occurs, returns a 422 error with the custom message \_\_('api.error\_happened') along with the error message from the exception.
  1. **destroy($brand\_id) Method**

### 🔹 Overview

Deletes a brand along with its associated image if available.

### 🔹 Key Functionality

#### 1-Find Brand

* Retrieves the Brand model by its id ($brand\_id).
* If the brand is not found, returns a 404 error with the message 'api.brand\_not\_found'.

#### 2-Delete Image (If Exists)

* If the brand has an associated image ($brand->image), it deletes the image from the public disk using Storage::disk('public')->delete($brand->image).

#### 3-Delete Brand

* Deletes the brand from the database using $brand->delete().

### 🔹 Return Response

* If successful, returns a success message indicating that the brand was deleted, using the localized success message from \_\_('api.brand\_delete').

### 🔹 Error Handling

* The method is wrapped in a try-catch block to handle unexpected errors.
* If an error occurs, returns a 422 error with the custom message \_\_('api.error\_happened') along with the error message from the exception.

1. **CancelingReasons Controller**
   1. **all(Request $request) Method**

### 🔹 Overview

Retrieves a list of cancellation reasons, with optional pagination.

### 🔹 Key Functionality

#### 1-Check for Pagination

* The method checks if the request contains a paginate parameter and whether it's set to 1 (for pagination) or 0 (for no pagination).

#### 2-Pagination Logic

* **When Pagination is Enabled (paginate = 1):**
  + Retrieves a paginated list of cancellation reasons (CancelReason::latest()->paginate(10)), with a limit of 10 per page.
  + Constructs a pagination object containing:
    - Total number of records.
    - Records per page.
    - Current page.
    - Total pages.

#### 3-No Pagination Logic (paginate = 0 or Missing)

* **When Pagination is Disabled (paginate = 0):**
  + Retrieves all cancellation reasons without pagination (CancelReason::latest()->get()).
* **When Pagination is Not Specified:**
  + Defaults to paginated results similar to the first case.

#### 4-Format the Response

* The method uses CancelingReasonResource::collection() to format the cancellation reasons into a structured resource format.
* If pagination is enabled, both the reasons and the pagination information are included in the response.

### 🔹 Return Response

* **Success:**
  + If the process is successful, it returns the cancellation reasons along with the pagination information (if applicable), using the localized success message from \_\_('api.reason\_all').
* **Error Handling:**
  + The method is wrapped in a try-catch block to catch any unexpected errors.
  + If an error occurs, it returns a 422 error with the exception's message.
  1. **get($reason\_id) Method**

### 🔹 Overview

Retrieves a single cancellation reason based on the given reason\_id.

### 🔹 Key Functionality

#### 1-Find Cancellation Reason by ID

* The method attempts to find a cancellation reason using the provided reason\_id with CancelReason::find($reason\_id).
* If the cancellation reason exists, it proceeds to format and return the data.

#### 2-Check If Reason Exists

* If the cancellation reason is found, it uses the CancelingReasonResource to format the response into a structured resource.
* If the cancellation reason does not exist, it returns an error message indicating that the reason was not found.

### 🔹 Return Response

#### ****Success:****

* **When the reason is found**, it returns the data wrapped in a data key with the formatted cancellation reason using CancelingReasonResource.
* The response also includes a localized success message from \_\_('api.reason\_get').

#### ****Error Handling:****

* **When an error occurs**, either due to an exception or if the reason is not found, the method catches the exception and returns a 422 error along with the exception message.
* **When the reason is not found**, it specifically returns a 404 error with a localized message from \_\_('api.reason\_not\_found').

### 🔹 Return Data Structure

* **Success:**
  + A data key containing:
    - reason: The formatted cancellation reason object.
* **Error:**
  + A 404 error with a localized message (\_\_('api.reason\_not\_found')) if the reason is not found.
  1. **add(StoreCancellingReasonRequest $request) Method**

### 🔹 Overview

Adds a new cancellation reason using the provided request data.

### 🔹 Key Functionality

#### 1-Create New Cancellation Reason

* The method attempts to create a new cancellation reason by calling CancelReason::create($request->all()), which creates the reason using all the data from the request.
* The request is assumed to be validated and structured properly by the StoreCancellingReasonRequest.

### 🔹 Return Response

#### ****Success:****

* If the cancellation reason is created successfully, the method returns the newly created reason wrapped in a data key.
* The response also includes a success message, localized from \_\_('api.reason\_add').

#### ****Error Handling:****

* If an error occurs while creating the cancellation reason, the method catches the exception and returns a 422 error with the exception message.

### 🔹 Return Data Structure

* **Success:**
  + A data key containing:
    - reason: The newly created cancellation reason, formatted using the CancelingReasonResource.
* **Error:**
  + A 422 error with the exception message if something goes wrong during the creation process.
  1. **update($reason\_id, UpdateCancellingReasonRequest $request) Method**

### 🔹 Overview

Updates an existing cancellation reason based on the provided ID and updated data.

### 🔹 Key Functionality

#### 1-Find the Cancellation Reason

* The method attempts to find the cancellation reason by its ID using CancelReason::find($reason\_id).
* If no reason is found with the provided ID, it returns an error message.

#### 2-Update the Cancellation Reason

* If the cancellation reason is found, the method proceeds to update the record using $reason->update($request->all()).
* It updates the reason with the data provided in the request.

### 🔹 Return Response

#### ****Success:****

* If the update is successful, it returns the updated cancellation reason wrapped in a data key.
* The response includes a success message, localized from \_\_('api.reason\_update').

#### ****Error Handling:****

* If no cancellation reason is found with the provided ID, it returns a 404 error with the message 'api.reason\_not\_found'.
* If an exception occurs during the update process, it returns a 422 error with the exception message.

### 🔹 Return Data Structure

#### ****Success:****

* **data**: Contains the updated cancellation reason, formatted using the CancelingReasonResource.

#### ****Error:****

* A 404 error if the cancellation reason is not found.
* A 422 error if an exception occurs during the update process.
  1. **destroy($reason\_id) Method**

### 🔹 Overview

Deletes a cancellation reason by its ID.

### 🔹 Key Functionality

#### 1-Find the Cancellation Reason

* The method first attempts to find the cancellation reason using the provided ID via CancelReason::find($reason\_id).
* If the cancellation reason is not found, it returns an error message indicating the reason was not found.

#### 2-Delete the Cancellation Reason

* If the cancellation reason is found, the method proceeds to delete the record using $reason->delete().
* It then returns a success message.

### 🔹 Return Response

#### ****Success:****

* If the deletion is successful, it returns a success message, localized from \_\_('api.reason\_delete').

#### ****Error Handling:****

* If no cancellation reason is found with the provided ID, it returns a 404 error with the message 'api.reason\_not\_found'.
* If an exception occurs during the deletion process, it returns a 422 error with the exception message.

### 🔹 Return Data Structure

#### ****Success:****

* A success message indicating the cancellation reason was successfully deleted.

#### ****Error:****

* A 404 error if the cancellation reason is not found.
* A 422 error if an exception occurs during the deletion process.

1. **Car Controller** 
   1. **getCars(Request $request) Method**

🔹 **Overview**  
Fetches a list of cars, optionally filtered by a client’s ID.

🔹 **Key Functionality**  
1-**Initialize Query**  
The method begins by initializing a query on the Car model.

2-**Filter by Client (Optional)**  
If a client\_id is provided in the request, the method applies a filter to retrieve cars associated with that specific client by using $cars->where('user\_id', $request->client\_id).

3-**Retrieve Cars**  
It then fetches the cars, ordering them by the most recent entries (using latest()), and returns them as a resource collection, which structures the data properly for API responses.

🔹 **Return Response**

* **Success:**  
  If the cars are successfully retrieved, a JSON response is returned with the list of cars and a 200 status code.
* **Error Handling:**  
  If an exception occurs during the process, a 403 status code is returned along with the error message.

🔹 **Return Data Structure**

* **Success:**
  + A 200 status with a cars key, containing an array of car data (formatted via the CarResource).
* **Error:**
  + A 403 status with the exception message if any error occurs during the execution.
  1. **addCar(Request $request) Method**

🔹 **Overview**  
Adds a new car entry to the system with the provided details.

🔹 **Key Functionality**  
1-**Input Validation**  
The method starts by defining validation rules for the car data. The rules ensure that certain fields, such as brand\_id, car\_type\_id, and color\_id, are provided and valid. If validation fails, the method returns a 403 error with the first validation error message.

2-**Fetch Brand Image (Optional)**  
If the brand\_id is provided in the request, the method fetches the brand’s image from the Brand model to associate with the new car entry.

3-**Create Car Record**  
The method then proceeds to create a new car record in the database, populating the fields with the request data. The car is associated with the client (user) based on client\_id, and the brand\_id and car\_type\_id are linked with the respective models.

4-**Return Success**  
Upon successfully creating the car, it returns a success message, indicating the car was added successfully.

🔹 **Return Response**

* **Success:**  
  A success message is returned with a 200 status code, indicating the car was added successfully.
* **Error Handling:**  
  If validation fails, a 403 status code is returned along with the validation error message. If an exception occurs during the process, a 422 status code is returned along with the exception message.

🔹 **Return Data Structure**

* **Success:**  
  A 200 status with a success message indicating the car was successfully added.
* **Error:**
  + A 403 status with the validation error message if validation fails.
  + A 422 status with the exception message if an error occurs during car creation.
  1. **updateCar(Request $request) Method**

🔹 **Overview**  
Updates an existing car's details in the system based on the provided car\_id and other request data.

🔹 **Key Functionality**  
1-**Input Validation**  
The method defines validation rules for the request data. It ensures the car\_id exists in the cars table, and validates the brand\_id, car\_type\_id, and color\_id fields to ensure they exist in their respective tables. If validation fails, it returns a 403 error with the first validation error message.

2-**Find the Car to Update**  
Using the provided car\_id, the method attempts to find the car that needs to be updated using Car::find($request->car\_id).

3-**Fetch Brand Image (Optional)**  
If a new brand\_id is provided in the request, the method fetches the associated brand's image to update the car's image.

4-**Update Car Record**  
The method updates the car record with the new data from the request, including fields like name, car\_size\_id, car\_plate\_number, image, color\_id, brand\_id, and car\_type\_id.

5-**Return Success**  
If the car is successfully updated, a success message is returned, indicating the car was updated successfully.

🔹 **Return Response**

* **Success:**  
  If the update is successful, a 200 status is returned with a success message indicating the car was updated.
* **Error Handling:**  
  If validation fails, a 403 status is returned along with the first validation error message. If an exception occurs during the update, a 422 status is returned with the exception message.

🔹 **Return Data Structure**

* **Success:**  
  A 200 status with a success message indicating the car was successfully updated.
* **Error:**
  + A 403 status with the validation error message if validation fails.
  + A 422 status with the exception message if an error occurs during the update.
  1. **colors() Method**

🔹 **Overview**  
Retrieves all available colors from the colors table.

🔹 **Key Functionality**

1-**Fetch Color Data**  
The method uses the DB::table() method to retrieve all the records from the colors table, selecting the columns id, color\_name, and hex\_code.

2-**Return Color Data**  
After fetching the data, it returns a JSON response with the data under the colors key.

🔹 **Return Response**

**Success:**  
If the data retrieval is successful, the method returns the list of colors in the response, structured as follows:  
{ "colors": [...] }.

**Error Handling:**  
If an exception occurs during the data retrieval process, the method returns a 422 error with the exception message.

🔹 **Return Data Structure**

**Success:**

* A JSON response containing an array of color objects with id, color\_name, and hex\_code.

**Error:**

* A 422 error with the exception message in case of an error during the data retrieval process.
  1. **deleteCar() Method**

🔹 **Overview**  
Deletes a car record by its car\_id from the database.

🔹 **Key Functionality**

1-**Validate Request Data**  
The method defines validation rules to ensure that the car\_id is required and exists in the cars table.  
If the validation fails, it returns a 403 error with the first validation error message.

2-**Find the Car Record**  
After validation, the method attempts to find the car using the provided car\_id via Car::find($request->car\_id).

3-**Delete the Car Record**  
If the car is found, the method proceeds to delete the car using $car->delete().

4-**Return Success Response**  
It then returns a success message indicating that the car has been deleted successfully.

🔹 **Return Response**

**Success:**  
If the car is successfully deleted, the method returns a success message, localized from \_\_('api.deleteCar').

**Error Handling:**

* If the car is not found, it will return a 403 error with the validation error message.
* If any other exception occurs during the process, it returns a 422 error with the exception message.

🔹 **Return Data Structure**

**Success:**

* A success message indicating the car was successfully deleted.

**Error:**

* A 403 error if the car\_id is invalid or not found in the cars table.
* A 422 error if an exception occurs during the deletion process.

1. **CarPlate Controller**
   1. **all() Method**

🔹 **Overview**  
Retrieves all car plates, with optional filtering based on user role and pagination.

🔹 **Key Functionality**

1-**Check User Role**  
The method first checks the role of the authenticated user. If the user has a role ID of 6 (indicating an investor), it filters car plates based on the investor's email and phone number.

2-**Determine Pagination**  
The method checks if the request includes a paginate parameter:

* If paginate is set to 1, it paginates the results with 10 items per page.
* If paginate is set to 0, it retrieves all the car plates without pagination.
* If paginate is not specified, it defaults to pagination with 10 items per page.

3-**Retrieve Car Plates**

* The method retrieves car plates either filtered by the investor or all car plates depending on the user’s role.
* It uses CarPlate::latest() to fetch the car plates in the latest order.

4-**Return Data**

* If pagination is applied, the method includes pagination information in the response.
* If pagination is not applied, it returns all the car plates.

🔹 **Return Response**

**Success:**

* If the car plates are successfully retrieved, the method returns the car plates, along with pagination data if applicable, and a success message localized from \_\_('api.car\_plates\_all').

**Error Handling:**

* If an exception occurs during the process, it returns a 422 error with a localized message and the exception details.

🔹 **Return Data Structure**

**Success:**

* A list of car plates, along with pagination details if pagination is used.

**Error:**

* A 422 error if an exception occurs during the retrieval process, with the message 'api.error\_happened' followed by the exception details.
  1. **get() Method**

🔹 **Overview**  
Retrieves a specific car plate by its ID.

🔹 **Key Functionality**

1-**Find the Car Plate**  
The method first attempts to find the car plate using the provided car\_plate\_id via CarPlate::find($car\_palte\_id).

2-**Check if Car Plate Exists**  
If the car plate is not found, it returns a 404 error with the message 'api.car\_plate\_not\_found'.

3-**Return Car Plate Data**  
If the car plate is found, the method returns the car plate data wrapped in a CarPlateResource, along with a success message localized from \_\_('api.car\_plate\_get').

🔹 **Return Response**

**Success:**

* If the car plate is found, it returns the car plate data in the data key with a success message.

**Error Handling:**

* If no car plate is found with the given ID, it returns a 404 error with the message 'api.car\_plate\_not\_found'.
* If an exception occurs during the process, it returns a 422 error with a localized message 'api.error\_happened' followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* A data object containing the car plate details wrapped in a CarPlateResource.

**Error:**

* A 404 error if the car plate is not found.
* A 422 error if an exception occurs during the retrieval process, with the message 'api.error\_happened' followed by the exception details.
  1. **add() Method**

🔹 **Overview**  
Adds a new car plate to the system.

🔹 **Key Functionality**

1-**Create the Car Plate**  
The method attempts to create a new car plate using the provided data (code and investor\_id) via CarPlate::create([...]).

2-**Return Success**  
If the car plate is successfully created, the method returns the newly created car plate data wrapped in a CarPlateResource along with a success message localized from \_\_('api.car\_plate\_add').

🔹 **Return Response**

**Success:**

* If the car plate is successfully created, it returns the car plate data in the data key along with a success message.

**Error Handling:**

* If an exception occurs during the creation process, it returns a 422 error with a localized message 'api.error\_happened' followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* A data object containing the newly created car plate details wrapped in a CarPlateResource.

**Error:**

* A 422 error if an exception occurs during the creation process, with the message 'api.error\_happened' followed by the exception details.
  1. **update() Method**

🔹 **Overview**  
Updates an existing car plate by its ID.

🔹 **Key Functionality**

1-**Find the Car Plate**  
The method first attempts to find the car plate using the provided ID ($car\_plate\_id) via CarPlate::find($car\_plate\_id).

* If the car plate is not found, it returns an error message indicating that the car plate was not found.

2-**Update the Car Plate**  
If the car plate is found, the method proceeds to update the car plate fields. It uses the provided data from the request (code and investor\_id), and if any field is not provided, it retains the current value in the database.

3-**Return Updated Data**  
Once the car plate is updated, the method returns the updated car plate data wrapped in a CarPlateResource along with a success message.

🔹 **Return Response**

**Success:**

* If the car plate is successfully updated, it returns the updated car plate data in the data key along with a success message.

**Error Handling:**

* If the car plate is not found, it returns a 404 error with the message 'api.car\_plate\_not\_found'.
* If an exception occurs during the update process, it returns a 422 error with a localized message 'api.error\_happened' followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* A data object containing the updated car plate details wrapped in a CarPlateResource.

**Error:**

* A 404 error if the car plate is not found, with the message 'api.car\_plate\_not\_found'.
* A 422 error if an exception occurs during the update process, with the message 'api.error\_happened' followed by the exception details.
  1. **destroy() Method**

🔹 **Overview**  
Deletes a car plate by its ID.

🔹 **Key Functionality**

1-**Find the Car Plate**  
The method first attempts to find the car plate using the provided ID ($car\_plate\_id) via CarPlate::find($car\_plate\_id).

* If the car plate is not found, it returns an error message indicating that the car plate was not found.

2-**Delete the Car Plate**  
If the car plate is found, the method proceeds to delete the car plate from the database using $carPlate->delete().

* It then returns a success message indicating the deletion was successful.

🔹 **Return Response**

**Success:**

* If the car plate is successfully deleted, it returns a success message indicating the car plate has been deleted.

**Error Handling:**

* If the car plate is not found, it returns a 404 error with the message 'api.car\_plate\_not\_found'.
* If an exception occurs during the deletion process, it returns a 422 error with a localized message 'api.error\_happened' followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* A success message indicating the car plate was successfully deleted.

**Error:**

* A 404 error if the car plate is not found, with the message 'api.car\_plate\_not\_found'.
* A 422 error if an exception occurs during the deletion process, with the message 'api.error\_happened' followed by the exception details.

1. **CarType Controller**
   1. **all() Method**

🔹 **Overview**  
Retrieves a list of car types, with the option to filter by brand and paginate the results.

🔹 **Key Functionality**

1-**Build the Car Types Query**  
The method starts by initializing the car types query using CarType::query().

* If a brand\_id is provided in the request, the query is filtered by brand\_id.

2-**Handle Pagination**  
The method checks if pagination is requested via the paginate parameter in the request:

* **If paginate is 1**: The results are paginated with a page size of 10.
* **If paginate is 0**: The results are fetched without pagination.
* **Default**: If no pagination option is specified, the results are paginated by default.

3-**Return the Results**

* After fetching the car types, the method returns the data with the appropriate response structure:
  + If pagination is applied, the pagination details (total, per\_page, current\_page, total\_pages) are included.
  + The response contains the car types and pagination (if applicable).

🔹 **Return Response**

**Success:**

* The response includes the list of car types, along with pagination details if requested.

**Error Handling:**

* If an exception occurs during the data retrieval process, a 422 error is returned with a message 'api.error\_happened' followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* **With Pagination**: The response contains the car types and pagination data.
* **Without Pagination**: The response contains only the list of car types.

**Error:**

* A 422 error with the message 'api.error\_happened' followed by the exception details.
  1. **get() Method**

🔹 **Overview**  
Fetches a specific car type based on the provided car\_type\_id.

🔹 **Key Functionality**

1-**Find the Car Type**  
The method attempts to find the car type using CarType::find($car\_type\_id) based on the provided ID.

2-**Handle Missing Car Type**

* If no car type is found (i.e., the result is null), it returns a 404 error with the message 'api.car\_type\_not\_found'.

3-**Return Car Type Data**

* If the car type is found, the method wraps the result in the CarTypesResource and returns the car type data with the success message 'api.car\_type\_get'.

🔹 **Return Response**

**Success:**

* The response includes the car type data wrapped in the CarTypesResource and a success message.

**Error Handling:**

* If the car type is not found, a 404 error with the message 'api.car\_type\_not\_found' is returned.
* If an exception occurs during the data retrieval process, a 422 error is returned with the message 'api.error\_happened' followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* The response contains the requested car type data.

**Error:**

* A 404 error if the car type is not found.
* A 422 error if an exception occurs during the process.
  1. **add() Method**

🔹 **Overview**  
Adds a new car type to the database.

🔹 **Key Functionality**

1-**Create the Car Type**  
The method uses CarType::create() to insert a new record into the car\_types table. The car type is created with the name, brand\_id, and status values taken from the request data.

2-**Return the Created Car Type Data**  
After successfully creating the car type, the method returns the newly created car type wrapped in a CarTypesResource with the success message 'api.car\_type\_add'.

🔹 **Return Response**

**Success:**

* The response contains the newly created car type data, wrapped in the CarTypesResource and a success message indicating the car type was successfully added.

**Error Handling:**

* No explicit error handling is included in this method. If any exception occurs, the global error handling will capture it.

🔹 **Return Data Structure**

**Success:**

* The response contains the car type data and a success message.

**Error:**

* If an exception occurs during the creation process, it will be handled by the global error response (likely a 422 error with the exception message).
  1. **update() Method**

🔹 **Overview**  
Updates an existing car type in the database.

🔹 **Key Functionality**

1-**Find the Car Type**  
The method first attempts to find the car type using the provided car\_type\_id. If the car type does not exist, it returns a 404 error with the message 'api.car\_type\_not\_found'.

2-**Update Car Type Details**  
If the car type exists, the method updates the car type's details. It checks if the provided values for name, brand\_id, and status are present in the request. If a value is missing, it keeps the existing value from the database.

3-**Return Updated Car Type**  
After updating, the method returns the updated car type data wrapped in a CarTypesResource with a success message 'api.car\_type\_update'.

🔹 **Return Response**

**Success:**

* The response includes the updated car type data, wrapped in the CarTypesResource, and a success message indicating the car type was successfully updated.

**Error Handling:**

* If the car type is not found, it returns a 404 error with the message 'api.car\_type\_not\_found'.
* If any exception occurs during the update process, it returns a 422 error with the exception message.

🔹 **Return Data Structure**

**Success:**

* The response contains the updated car type data wrapped in the CarTypesResource and a success message.

**Error:**

* A 404 error if the car type is not found.
* A 422 error if an exception occurs during the update process.
  1. **destroy() Method**

🔹 **Overview**  
Deletes a car type by its ID.

🔹 **Key Functionality**

1-**Find the Car Type**  
The method first attempts to find the car type using the provided car\_type\_id. If the car type does not exist, it returns a 404 error with the message 'api.car\_type\_not\_found'.

2-**Delete the Car Type**  
If the car type exists, the method proceeds to delete the car type record from the database using the delete() method.

3-**Return Success Message**  
After successful deletion, the method returns a success message indicating the car type was successfully deleted.

🔹 **Return Response**

**Success:**

* The response includes a success message indicating that the car type was successfully deleted.

**Error Handling:**

* If the car type is not found, it returns a 404 error with the message 'api.car\_type\_not\_found'.
* If any exception occurs during the deletion process, it returns a 422 error with the exception message.

🔹 **Return Data Structure**

**Success:**

* A success message indicating the car type was successfully deleted.

**Error:**

* A 404 error if the car type is not found.
* A 422 error if an exception occurs during the deletion process.

1. City Controller
   1. **all() Method**

🔹 **Overview**  
Retrieves a list of cities, with optional support for pagination based on the request parameters.

🔹 **Key Functionality**

1-**Check for Pagination Request**  
The method checks whether the request includes the paginate parameter:

* If paginate is set to 1, results are paginated.
* If paginate is set to 0, all cities are retrieved without pagination.
* If paginate is not provided, it defaults to paginating the results.

2-**Fetch Cities**  
The cities are retrieved from the database, ordered by the latest entries.

* When paginated, only a fixed number of records (10 per page) are returned.
* When not paginated, all city records are retrieved.

3-**Build Pagination Metadata**  
If pagination is applied, a pagination array is prepared containing:

* Total number of records.
* Number of records per page.
* Current page number.
* Total number of pages.

4-**Format and Return Response**  
The city records are wrapped in a resource collection (citiesResource) and returned.

* If pagination is applied, the pagination metadata is also included.

🔹 **Return Response**

**Success:**

* A list of cities wrapped in a resource collection.
* Pagination metadata is included if pagination is enabled.
* A success message localized from \_\_('api.city\_all').

**Error Handling:**

* If any exception occurs during processing, a 422 error is returned with the message \_\_('api.error\_happened') followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* A data object containing:
  + cities: A list of formatted city records.
  + pagination: (optional) Pagination details if applicable.

**Error:**

* A 422 error if an exception occurs, with a localized message and the exception details.
  1. **add() Method**

🔹 **Overview**  
Adds a new city after checking for duplicates.

🔹 **Key Functionality**

1-**Duplicate Check**  
Checks if a city with the same name already exists in the cities table.  
If a match is found, a 409 Conflict response is returned with a localized message indicating the city already exists.

2-**Create New City**  
If the city doesn't already exist, a new city is created using the provided name from the request.

3-**Resource Wrapping**  
The newly created city is wrapped in a resource for consistent API formatting in the response.

🔹 **Return Response**

🔹 **Success**  
Returns the newly created city inside a data array.  
**Message:** localized success message indicating the city was added successfully.

🔹 **Duplicate Error**  
Returns a 409 Conflict status.  
**Message:** localized message indicating the city already exists.

🔹 **Exception/Error**  
Returns a 422 Unprocessable Entity status if any exception is thrown.  
**Message:** localized generic error message along with the exception message.

🔹 **Return Data Structure**

🔹 **Success**

* data contains the created city formatted using a resource.
* message contains a localized success message.

🔹 **Duplicate Error**

* error is set to 409.
* message contains a localized duplicate city message.

🔹 **Exception**

* error is set to 422.
* message contains a localized error message along with exception details.
  1. **get() Method**

🔹 **Overview**  
Fetches the details of a specific city using its ID.

🔹 **Key Functionality**

1-**City Lookup**  
Attempts to retrieve the city by its ID from the database.  
If the city is not found, it returns a 404 Not Found response with a localized error message.

2-**Resource Wrapping**  
If the city is found, it is wrapped in a resource for standardized API formatting.

🔹 **Return Response**

🔹 **Success**  
Returns the city inside a data object.  
**Message:** localized message indicating successful retrieval.

🔹 **City Not Found**  
Returns a 404 Not Found status if no city is found with the given ID.  
**Message:** localized message indicating the city was not found.

🔹 **Exception/Error**  
Returns a 422 Unprocessable Entity status if an exception occurs.  
**Message:** localized error message along with the exception details.

🔹 **Return Data Structure**

🔹 **Success**

* data contains the city formatted using a resource.
* message contains a localized success message.

🔹 **City Not Found**

* error is set to 404.
* message contains a localized message indicating the city was not found.

🔹 **Exception**

* error is set to 422.
* message contains a localized error message with exception details.
  1. **update() Method**

🔹 **Overview**  
Updates the details of an existing city using the provided city ID.

🔹 **Key Functionality**

1-**Find City by ID**  
The method searches for the city using the given ID:

* If the city is **not found**, it returns a 404 error with a localized message from \_\_('api.city\_not\_found').

2-**Update City Information**  
If the city is found, its name is updated using the value provided in the request:

* If the name is **not provided**, the current name is retained.

3-**Format and Return Response**  
After the update, the updated city record is wrapped in a citiesResource for consistent API formatting and returned in the response.

🔹 **Return Response**

**Success:**

* Returns the updated city wrapped in a citiesResource.
* Includes a localized success message from \_\_('api.city\_update').

**City Not Found:**

* Returns a 404 error with a localized message from \_\_('api.city\_not\_found').

**Error Handling:**

* If an exception occurs, returns a 422 error with the message \_\_('api.error\_happened') followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* A data object containing:
  + city: The updated city record formatted using citiesResource.
* message: A localized success message.

**City Not Found:**

* error: 404
* message: A localized not-found message.

**Exception:**

* error: 422
* message: A localized error message with the exception details.
  1. **destroy() Method**

🔹 **Overview**  
Deletes a city based on its ID after verifying that no related neighborhoods exist.

🔹 **Key Functionality**

1-**Find City by ID**  
The method looks up the city using the provided ID:

* If the city is **not found**, a 404 error is returned with a localized message from \_\_('api.city\_not\_found').

2-**Check for Related Neighborhoods**  
Before deleting, the method checks whether the city is linked to any neighborhoods:

* If there **are related neighborhoods**, a 409 conflict error is returned with a custom message indicating that the city cannot be deleted until its neighborhoods are removed.

3-**Delete City**  
If no related neighborhoods are found, the city is soft-deleted from the database.

🔹 **Return Response**

**Success:**

* Returns a localized success message from \_\_('api.city\_delete') after successful deletion.

**City Not Found:**

* Returns a 404 error with a localized message from \_\_('api.city\_not\_found').

**Conflict (Related Neighborhoods Exist):**

* Returns a 409 error with a custom message indicating that the city is linked to neighborhoods and cannot be deleted.

**Error Handling:**

* If any exception occurs, returns a 422 error with the message \_\_('api.error\_happened') followed by the exception details.

🔹 **Return Data Structure**

**Success:**

* message: A localized success message confirming deletion.

**City Not Found:**

* error: 404
* message: A localized not-found message.

**Conflict (Neighborhoods Exist):**

* error: 409
* message: A message explaining the deletion restriction due to related neighborhoods.

**Exception:**

* error: 422
* message: A localized error message with the exception details.

1. **Client Controller**

**10.1 all(Request $request) Method**

🔹 **Overview**  
Retrieves a list of clients with flexible filtering, searching, sorting, and pagination options. It also supports detailed analytics when is\_notify is enabled.

🔹 **Key Functionality**

1-**Initialize Query**  
Starts building the client query using Client::query() and eager-loads the count of completed reservations (status = 6) using withCount().

2-**Search Filtering**  
If the search parameter is present, it filters clients of type 0 by:

* username
* phone
* email

Each field is searched using a partial match (LIKE %value%).

3-**Sorting Options**  
If the sort parameter is present:

* most-orderd: Sorts by highest number of completed reservations.
* least-orderd: Sorts by lowest number of completed reservations.  
  If not present, defaults to sorting by id in descending order.

4-**Status Filtering**

* status=zero-orders:  
  Clients without reservations or only those whose reservations have status = 6.
* status=zero-packages:  
  Clients without any packageSubscriptions.
* status=one-packages:  
  Clients who **have** packageSubscriptions.

All of these filters are scoped to clients of type 0.

5-**City-Based Filtering**

If city\_id is provided, filters clients whose **last reservation** is in the specified city.

6-**Pagination Handling**

Based on the paginate flag:

🔸 **paginate=1 and is\_notify=1:**

* Returns paginated client results (perPage = 500).
* Builds an array with categorized client stats:
  + All clients
  + Clients without reservations
  + Clients with/without packages
  + Clients without reservations for the past:
    - Week
    - Month
    - 24, 48, and 72 hours
* Each group includes:
  + id
  + username (label)
  + count (number of clients in that group)

🔸 **paginate=1 without is\_notify:**

* Regular paginated results (perPage = 10)
* Returns transformed clients via ClientResource with pagination meta.

🔸 **paginate=0:**

* Returns **all clients** as a flat list without pagination.
* Transformed using ClientResource.

🔸 **If paginate not provided:**

* Defaults to paginated results (perPage = 500).

🔹 **Return Response**

Always returns data using the helper returnData():

**Structure:**

{

"data": {

"clients": [...],

"pagination": {

"total": X,

"per\_page": X,

"current\_page": X,

"total\_pages": X

}

},

"message": "Localized message (\_\_('api.client\_all'))"

}

For is\_notify=1, the clients array includes stats for analytical grouping as well.

🔹 **Return Data Structure (if notify mode)**

Each item in the clients array represents:

* A specific category of clients.
* username: a label (e.g., "كل العملاء", "عملاء لم يحجزوا منذ اسبوع")
* count: total number of clients in that group.

**10.2 get() Method**

### 🔹 Overview

Fetches the details of a specific client by their unique ID and returns the information in a structured resource format.

**🔹 Key Functionality**

**1-Find Client by ID**

The method attempts to locate a client using the provided ID:

* If the client does not exist, it returns a 404 error with a localized "client not found" message.

**2-Return Client Data**

If the client is found, the method wraps the client's information using a resource transformer and returns it in the response.

**🔹 Return Response**

**🔹 Success:**

Returns a successful response containing the client's data, along with a localized success message.

**🔹 Client Not Found:**

Returns a 404 error response with a localized message indicating that the client was not found.

**🔹 Exception:**

In case of an unexpected error, returns a 422 error response including a localized error message and the exception details.

**🔹 Return Data Structure**

**Success:**

* Status: Success
* Data: Contains the client information
* Message: Localized message confirming successful retrieval

**Client Not Found:**

* Error Code: 404
* Message: Localized message indicating the client could not be found

**Exception:**

* Error Code: 422
* Message: Localized message describing the error that occurred

**10.3 add() Method**

### 🔹 Overview

Creates and stores a new client in the system after validating the input and handling an optional image upload.

**🔹 Key Functionality**

**1-Receive Validated Data**

Receives input from a validated request (StoreClientRequest), which ensures all required fields meet the defined validation rules before proceeding.

**2-Initialize and Fill Client**

A new client instance is created and populated with the input data, excluding the image field, which is handled separately.

**3-Handle Image Upload (Optional)**

If an image is provided:

* It is stored in the uploads/clients directory within the public storage disk.
* The stored image path is assigned to the client's image attribute.

**4-Save Client**

The client record, including the uploaded image if provided, is saved to the database.

**🔹 Return Response**

**🔹 Success:**

Returns a successful response containing the newly created client's data, wrapped in a resource, along with a localized success message.

**🔹 Exception:**

In the event of any unexpected error during the process, a 422 error is returned with a localized message and details about the exception.

**🔹 Return Data Structure**

**Success:**

* Status: Success
* Data: Contains the created client information
* Message: Localized message confirming successful addition

**Exception:**

* Error Code: 422
* Message: Localized error message with exception details

**10.4 update() Method**

### 🔹 Overview

Updates the information of an existing client using validated input, with optional handling for password and image changes.

**🔹 Key Functionality**

**1-Find Client by ID**

* Searches for the client using the provided client\_id.
* If no matching client is found, a 404 error is returned with a localized "not found" message.

**2-Update Client Fields**

* Populates the client model with validated request data, excluding the image.
* If a new password is provided, it is updated separately.

**3-Handle Image Replacement (Optional)**

If a new image is provided:

* Deletes the old image from storage, if it exists.
* Uploads the new image to the uploads/clients directory within the public storage disk.
* Updates the client’s image path with the new file.

**4-Save Changes**

* The updated client record is saved to the database, including the new image and/or password if applicable.

**🔹 Return Response**

**🔹 Success:**

* Returns a success response containing the updated client's data wrapped in a resource.
* Includes a localized message confirming the update.

**🔹 Client Not Found:**

* Returns a 404 error if the client with the given ID does not exist.
* Includes a localized "client not found" message.

**🔹 Exception:**

* If an error occurs during the update process, a 422 error is returned.
* Includes a localized error message and the exception details.

**🔹 Return Data Structure**

**Success:**

* Status: Success
* Data: Contains the updated client information
* Message: Localized message confirming successful update

**Client Not Found:**

* Error Code: 404
* Message: Localized message indicating the client was not found

**Exception:**

* Error Code: 422
* Message: Localized error message with exception details

**10.5 destroy() Method**

### 🔹 Overview

Soft-deletes a client record after performing necessary cleanup such as modifying the phone number and removing the profile image from storage.

**🔹 Key Functionality**

**1-Find Client by ID**

* Looks up the client using the provided client\_id.
* If the client does not exist, returns a 404 error with a localized "client not found" message.

**2-Modify Phone Number Before Deletion**

* Updates the client’s phone number to include a \_deleted suffix along with the current timestamp.
* This ensures phone number uniqueness and avoids future conflicts.

**3-Delete Profile Image from Storage**

* If the client has a stored image, it is removed from the storage disk before deletion.

**4-Soft Delete Client**

* Performs a soft delete on the client record, preserving the record in the database but marking it as deleted.

**🔹 Return Response**

**🔹 Success:**

* Returns a localized success message confirming the client was successfully deleted.

**🔹 Client Not Found:**

* Returns a 404 error if the client ID does not exist.
* Includes a localized message indicating the client was not found.

**🔹 Exception:**

* If any exception occurs during the process, returns a 422 error.
* Includes a localized error message with the exception details.

**🔹 Return Data Structure**

**Success:**

* Status: Success
* Message: Localized success message confirming deletion

**Client Not Found:**

* Error Code: 404
* Message: Localized "client not found" message

**Exception:**

* Error Code: 422
* Message: Localized error message with exception details

**10.6 exportNoOrdersNotifications() Method**

### 🔹 Overview

Sends notifications to clients who have not placed any orders (i.e., reservations) or have only unconfirmed ones. The method also supports sending notifications to a custom list of client IDs passed in the request.

**🔹 Key Functionality**

**1-Retrieve Client IDs from Request**

* Accepts a list of client\_ids from the request.
* If the list is sent as a JSON string, it is decoded into an array.

**2-Identify Clients with No Orders**

* Performs a left join between the clients and reservations tables.
* Filters to include only clients of type 0 (e.g., regular clients).
* Groups results by client ID.
* Applies a havingRaw condition to detect clients who either:
  + Have no reservations, or
  + Only have unconfirmed reservations (status = 0).

**3-Filter for “No Order” Clients If Needed**

* If the request contains 0 in the client IDs list, it is interpreted as a request to notify all clients who have no orders.
* In that case, the client ID list is replaced with the list of "no order" clients identified in the previous step.

**4-Create and Send Notifications**

* Iterates through the final list of client IDs.
* For each client:
  + A new notification record is created and saved.
  + A helper method (sendNotification) is invoked to actually send the notification.

**🔹 Return Response**

**🔹 Success:**

* Returns a localized success message confirming that notifications were sent successfully.

**🔹 Return Data Structure**

**Success:**

* Status: Success
* Message: Localized message confirming that notifications were successfully sent to targeted clients.

**10.7 sendNotification() Method**

### 🔹 Overview

This private helper method is responsible for sending a push notification to a specific client using their device token.

**🔹 Key Functionality**

**1-Retrieve Client by ID**

* Looks up the client based on the provided client\_id.
* Ensures the client exists before proceeding.

**2-Send Notification**

* If the client is found and valid, a push notification is sent using the send\_notification() helper function.
* The notification payload includes:
  + A title from the request.
  + A body message from the request.

**🔹 Return Response**

**🔹 Always returns true:**

* This indicates that the notification logic completed, regardless of whether the client was found or not.

**🔹 Notes**

* This method assumes that the send\_notification() helper handles actual communication with a push notification service (e.g., Firebase).
* There’s no error thrown if the client is not found; the method simply skips sending.

**10.8 getFrequentlyRequestingClients() Method**

### 🔹 Overview

This method retrieves a list of the most frequently requesting clients based on their reservation history within a specified date range. It also supports pagination for the results.

**🔹 Key Functionality**

**1-Date Filtering**

* The method accepts two optional query parameters: from and to dates.
* It dynamically builds the query based on the provided dates to filter the reservations:
  + If both dates are provided, it filters the reservations within the date range.
  + If only the from date is provided, it filters reservations from that date onward.
  + If only the to date is provided, it filters reservations up to that date.
  + If no dates are provided, it defaults to fetching all reservations with a status of 3.

**2-Fetch Clients with Reservation Count**

* Clients are retrieved with a count of their related reservations (filtered by the status 3).
* The result is sorted by the reservation count in descending order, showing the most frequent clients first.

**3-Pagination (Optional)**

* If the paginate query parameter is set to 1, pagination is applied:
  + The results are sliced based on the current page and a fixed number of items per page (200).
  + Pagination metadata such as total records, current page, and total pages is returned along with the client data.

**🔹 Return Response**

**🔹 Success Response**

* If pagination is requested, the response includes:
  + clients: A collection of clients with their reservation counts.
  + pagination: Metadata including total, per\_page, current\_page, and total\_pages.
* If pagination is not requested, the response only includes the clients data.

**🔹 Error Handling**

* If an error occurs, the method returns a 422 error with the message details.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: "العملاء الاكثر طلبا" (Localized message for "Most Requested Clients").
* **Data**:
  + **clients**: A collection of ClientResource data (client information along with reservation counts).
  + **pagination** (if applicable):
    - total: Total number of clients matching the criteria.
    - per\_page: Number of items per page (fixed at 200).
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

**🔹 Error**

* **Error**: 422
* **Message**: The error message detailing the issue.

**10.9 clientReservations() Method**

### 🔹 Overview

This method retrieves a list of reservations made by a specific client, paginated to display a set number of records per page. It sorts the reservations by the latest and returns the paginated results along with the necessary pagination metadata.

**🔹 Key Functionality**

**1-Retrieve Reservations**

* The method fetches reservations associated with a given client\_id.
* It retrieves the reservations in descending order based on the most recent reservation (using the latest() method).
* A fixed number of records per page (10) is applied using pagination.

**2-Pagination**

* The method calculates the pagination metadata, including:
  + total: Total number of reservations.
  + per\_page: Number of reservations displayed per page (set to 10).
  + current\_page: The current page of results.
  + total\_pages: The total number of pages based on the number of records and per-page limit.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **reservations**: A collection of reservations in ReservationResource format.
  + **pagination**: Metadata including total, per\_page, current\_page, and total\_pages.
* **Message**: Returns a localized success message from \_\_('api.reservation\_all') indicating the retrieval of all client reservations.

**🔹 Error Handling**

* If an error occurs, the method returns a 422 error with the exception message details.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: A localized message confirming the successful retrieval of client reservations.
* **Data**:
  + **reservations**: A collection of reservation data.
  + **pagination**:
    - total: Total number of reservations.
    - per\_page: Number of reservations per page (set to 10).
    - current\_page: Current page of the results.
    - total\_pages: Total number of pages.

**🔹 Error**

* **Error**: 422
* **Message**: The error message detailing the issue.

**10.10 getFrequentlyRequestingClientsInMorning() Method**

### 🔹 Overview

This method retrieves the list of clients who made reservations between 6:00 AM and 11:59 AM, focusing on reservations that have a specific status. It can paginate the results and return metadata along with the client data.

**🔹 Key Functionality**

**1-Time Filtering**

* The method defines the morning time range between 6:00 AM and 11:59 AM.
* It filters reservations by checking whether the reservation's from time falls within this range.
* It uses a raw SQL query to handle the time format conversion, adjusting the from time based on whether it's in Arabic AM/PM format or standard English format.

**2-Status Filter**

* Only reservations with a status of 3 are included in the result set, filtering out other statuses.

**3-Client Query**

* The method uses withCount to count the number of matching reservations for each client.
* The clients are then ordered by the number of reservations, starting with those who have the most.

**4-Pagination**

* The method supports pagination if the paginate parameter is passed in the request:
  + It paginates the clients, showing a maximum of 200 clients per page.
  + Metadata is included in the response to indicate the total number of clients, the current page, and the total pages.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **clients**: A collection of clients, represented by ClientResource.
  + **pagination** (if paginated):
    - total: Total number of clients.
    - per\_page: Number of clients displayed per page (set to 200).
    - current\_page: Current page number.
    - total\_pages: Total number of pages.
* **Message**: Returns a localized success message "الطلبات الاكثر طلبا في الصباح" (Most requested orders in the morning) in Arabic.

**🔹 Error Handling**

* If any error occurs, the method returns a 422 error with the exception message.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: A localized success message confirming the retrieval of clients who made the most requests in the morning.
* **Data**:
  + **clients**: A collection of client data.
  + **pagination** (if paginated):
    - total: Total number of clients.
    - per\_page: Number of clients per page (set to 200).
    - current\_page: Current page of the result.
    - total\_pages: Total number of pages.

**🔹 Error**

* **Error**: 422
* **Message**: The error message detailing the exception.

**10.11 getClientMorningReservations() Method**

### 🔹 Overview

This method retrieves the list of morning reservations (from 6:00 AM to 11:59 AM) for a specific client. It also supports pagination and returns reservation details with metadata.

**🔹 Key Functionality**

**1-Client Existence Check**

* The method first checks if the client exists by querying the Client model with the provided clientId.
* If the client is not found, it returns a 404 error with the message "العميل غير موجود" (Client not found).

**2-Time Filtering**

* The method defines the morning time range between 6:00 AM and 11:59 AM.
* It uses a raw SQL query to convert the reservation's from time to a consistent format and filters those within the morning time range.
* The time format conversion adjusts for Arabic AM/PM formatting (ص for AM and م for PM).

**3-Status Filter**

* It filters the reservations to include only those with a status of 3 (likely representing completed or confirmed reservations).

**4-Pagination**

* The method paginates the results to show up to 10 reservations per page.
* Metadata is included in the response to provide the total number of reservations, the current page, and the total pages.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **reservations**: A collection of morning reservations for the specified client, represented by ReservationResource.
  + **pagination**: Metadata providing details about pagination.
* **Message**: Returns a localized success message "الحجوزات الصباحية للعميل" (Morning reservations for the client) in Arabic.

**🔹 Error Handling**

* If the client does not exist, the method returns a 404 error.
* If any other error occurs during the query, the method returns a 422 error with the exception message.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: A localized success message confirming the retrieval of the client's morning reservations.
* **Data**:
  + **reservations**: A collection of reservation data.
  + **pagination** (if paginated):
    - total: Total number of morning reservations for the client.
    - per\_page: Number of reservations displayed per page (set to 10).
    - current\_page: Current page of the result.
    - total\_pages: Total number of pages.

**🔹 Error**

* **Error**: 404 if the client is not found, or 422 for other errors.
* **Message**: The error message detailing the exception.

**10.12 getFrequentlyRequestingClientsInEvening() Method**

### 🔹 Overview

This method retrieves the list of clients who have made the most evening reservations (from 12:00 PM to 11:59 PM). It supports dynamic date filtering and pagination, returning clients sorted by the number of reservations they made.

**🔹 Key Functionality**

**1-Evening Time Range**

* The method defines the evening time range between 12:00 PM and 11:59 PM.
* It ensures that the from time of the reservation falls within this range.

**2-Date Threshold Filtering**

* The method filters the reservations to include only those that occurred after a specified threshold date (2024-02-20).

**3-Status Filter**

* The reservations are filtered to include only those with a status of 3 (likely representing confirmed or completed reservations).

**4-Pagination**

* The method supports pagination. If the paginate query parameter is set to 1, it slices the results into pages, returning up to 200 clients per page.
* Metadata is included in the response to provide the total number of clients, the current page, and the total number of pages.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **clients**: A collection of clients with the most evening reservations, represented by ClientResource.
  + **pagination**: Pagination metadata (if requested).
* **Message**: Returns a localized success message "الطلبات الاكثر طلبا في المساء" (Most requested clients in the evening) in Arabic.

**🔹 Error Handling**

* If an error occurs during the query execution, the method returns a 422 error with the exception message.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: A localized success message confirming the retrieval of the top clients with evening reservations.
* **Data**:
  + **clients**: A collection of client data.
  + **pagination** (if paginated):
    - total: Total number of top clients.
    - per\_page: Number of clients displayed per page (set to 200).
    - current\_page: Current page of the result.
    - total\_pages: Total number of pages.

**🔹 Error**

* **Error**: 422 error if there is a problem with the query.
* **Message**: The error message detailing the exception.

**10.13 getClientReservationsInEvening() Method**

### 🔹 Overview

This method retrieves the list of reservations for a specific client made during the evening (from 12:00 PM to 11:59 PM). It supports pagination, returning up to 10 reservations per page. If the client doesn't exist, an error response is provided.

**🔹 Key Functionality**

**1-Evening Time Range**

* The method defines the evening time range between 12:00 PM and 11:59 PM.
* It filters the reservations based on the from time being within this range.

**2-Status Filter**

* The method filters the reservations to include only those with a status of 3 (likely representing confirmed or completed reservations).

**3-Pagination**

* The method supports pagination. It returns up to 10 reservations per page.
* Pagination metadata is included, such as the total number of reservations, current page, and total pages.

**4-Client Existence Check**

* The method checks if the client exists in the database before attempting to retrieve their reservations.
* If the client is not found, a 404 error is returned with the message 'العميل غير موجود' (Client not found).

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **reservations**: A collection of reservations made by the client during the evening, represented by ReservationResource.
  + **pagination**: Pagination metadata, including total reservations, current page, and total pages.
* **Message**: Returns a localized success message "حجوزات العميل في المساء" (Client evening reservations) in Arabic.

**🔹 Error Handling**

* If an error occurs during the query execution or if the client doesn't exist, the method returns a 422 error with the exception message, or a 404 if the client is not found.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: A localized success message confirming the retrieval of the client's evening reservations.
* **Data**:
  + **reservations**: A collection of reservation data for the client.
  + **pagination**: Metadata for pagination:
    - total: Total number of reservations.
    - per\_page: Number of items per page (set to 10).
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

**🔹 Error**

* **Error**:
  + 404 error if the client is not found.
  + 422 error if there is an issue during the query execution.
* **Message**: An error message detailing the exception or the client not being found.

**10.14 getMostRatedClients() Method**

### 🔹 Overview

This method retrieves the top clients based on the number of reviews (ratings) they have made for their reservations. It supports pagination and returns clients in descending order of their review count. The method also includes a fallback to return all clients if pagination is not requested.

**🔹 Key Functionality**

**1-Join Operations**

* **Reservations**: The method joins the clients table with the reservations table to retrieve reservations related to each client.
* **Reviews**: The method further joins with the reviews table to count the number of reviews (ratings) each client has made for their reservations.

**2-Count of Reviews**

* The selectRaw method is used to count the number of reviews for each client, giving us the number of rated orders (reservations\_count).

**3-Grouping Data**

* The method groups the data by the client’s id, username, and phone to prevent SQL errors from multiple aggregation results.

**4-Sorting**

* Clients are ordered in descending order by the reservations\_count, meaning the clients with the most reviews appear first.

**5-Pagination**

* If the paginate parameter is provided in the request with a value of 1, pagination is applied.
  + The number of clients per page is set to 200.
  + Pagination metadata is returned, including total records, current page, and total pages.
* If pagination is not required, all clients are returned in one response.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **clients**: A collection of clients, each with their id, username, phone, and reservations\_count (the count of reviews they received).
  + **pagination**: Pagination metadata (if requested), including:
    - total: Total number of clients.
    - per\_page: Number of items per page (set to 200).
    - current\_page: Current page number.
    - total\_pages: Total number of pages.
* **Message**: A localized success message: "العملاء الاكثر تقييما للطلبات" (The most rated clients).

**🔹 Error Handling**

* The method returns the appropriate success response without explicit error handling for database issues, as it's assumed the query will run without issues under normal circumstances.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: A localized success message confirming the retrieval of the top-rated clients.
* **Data**:
  + **clients**: A collection of client data, each client having:
    - id: Client's unique identifier.
    - username: Client's username.
    - phone: Client's phone number.
    - reservations\_count: The total number of reviews the client has received.
  + **pagination** (if requested):
    - total: Total number of clients.
    - per\_page: Number of items per page.
    - current\_page: Current page number.
    - total\_pages: Total pages based on the number of clients and items per page.

**🔹 Error**

* **Error**:
  + Not explicitly defined in the code, as it assumes successful execution if no issues occur during the query.

**10.15 getReviewedReservations() Method**

### 🔹 Overview

This method retrieves a list of reservations for a specific client that have been reviewed. It supports pagination and returns the reservations in the latest order.

**🔹 Key Functionality**

**1-Filter by Client ID**

* The method filters the reservations based on the provided clientId to ensure that only the specific client's reservations are considered.

**2-Ensure Reviewed Reservations**

* The method uses the whereHas() method to ensure that the reservations being retrieved have an associated review.

**3-Sorting**

* The reservations are ordered by the most recent first, using the latest() method.

**4-Pagination**

* The method retrieves a paginated list of reviewed reservations, with 10 items per page.
  + Pagination metadata is returned, including:
    - total: The total number of reservations that match the criteria.
    - per\_page: The number of reservations per page (set to 10).
    - current\_page: The current page number.
    - total\_pages: The total number of pages based on the number of reservations and items per page.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **reservations**: A collection of reservations, each having detailed information about the reservation.
  + **pagination**: Pagination metadata, which includes:
    - total: The total number of reservations.
    - per\_page: The number of reservations displayed per page.
    - current\_page: The current page of results.
    - total\_pages: The total number of pages.
* **Message**: A localized success message: "تم استرجاع الطلبات بنجاح" (Reservations retrieved successfully).

**🔹 Error Handling**

* If there is an error during the execution of the method (e.g., database issues), it returns an error message with status code 422 and the exception's message.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: A localized success message confirming the retrieval of reviewed reservations.
* **Data**:
  + **reservations**: A collection of reservations, each reservation containing the relevant details (based on the ReservationResource).
  + **pagination** (optional if paginated):
    - total: The total number of reservations.
    - per\_page: The number of reservations per page (set to 10).
    - current\_page: The current page of results.
    - total\_pages: The total number of pages based on the number of reservations and items per page.

**🔹 Error**

* **Error**:
  + If an error occurs, the response will contain the error code 422 and the exception message ($error->getMessage()).

**10.16 getHighestRatingClients() Method**

### 🔹 Overview

This method retrieves a list of clients with the highest average ratings based on their associated reviews. It supports pagination and returns the clients ordered by the highest average rating.

**🔹 Key Functionality**

**1-Join with Reservations and Reviews**

* The method performs a leftJoin with the reservations and reviews tables to get the total ratings and average rating score for each client.

**2-Aggregate Rating Data**

* The method calculates:
  + total\_rating: The sum of all review scores for the client's reservations.
  + avg\_rating: The average rating score for the client's reservations.

**3-Group by Client**

* The method groups the data by client (clients.id, clients.username, clients.phone) to ensure each client’s ratings are aggregated correctly.

**4-Sorting**

* The method sorts the clients by their avg\_rating in descending order, ensuring that clients with the highest average rating come first.

**5-Pagination**

* The method supports pagination:
  + It retrieves the top clients with their aggregated ratings.
  + Pagination metadata is returned, including:
    - total: The total number of clients matching the criteria.
    - per\_page: The number of items displayed per page (set to 200).
    - current\_page: The current page number.
    - total\_pages: The total number of pages based on the number of clients and items per page.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **clients**: A collection of clients, each containing:
    - The client’s ID, username, phone, total rating, and average rating.
  + **pagination** (optional if paginated):
    - total: The total number of clients matching the query.
    - per\_page: The number of clients per page (200 in this case).
    - current\_page: The current page number.
    - total\_pages: The total number of pages based on the number of clients and items per page.
* **Message**: A localized success message: "العملاء الاعلي تقييما للطلبات" (The highest-rated clients for orders).

**🔹 Error Handling**

* If any error occurs, the method will return an error response with:
  + **Error Code**: 422.
  + **Error Message**: The exception’s message will be included.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: The success message confirming the retrieval of the highest-rated clients.
* **Data**:
  + **clients**: A collection of clients with their details (id, username, phone, total\_rating, avg\_rating).
  + **pagination** (if paginated):
    - total: The total number of clients in the result set.
    - per\_page: Number of clients per page.
    - current\_page: The current page.
    - total\_pages: The total pages of results.

**🔹 Error**

* **Error**: If an error occurs, an error code 422 will be returned along with the error message.

**10.17 getHighestRatedReservationsForClient() Method**

### 🔹 Overview

This method retrieves a list of the highest-rated reservations for a specific client. It supports pagination and sorts the reservations by the review score in descending order, showing the highest-rated reservations first.

**🔹 Key Functionality**

**1-Eager Load Review Data**

* The method eagerly loads the review relationship for each reservation to ensure the review details are available when filtering by score.

**2-Filter by Client ID**

* The method filters the reservations by the specific client\_id provided as a parameter to ensure only the reservations related to the given client are considered.

**3-Join with Reviews**

* It performs a leftJoin with the reviews table to pull in the review data for each reservation.

**4-Ensure Reviews Have a Score**

* The method uses whereNotNull('reviews.score') to ensure only reservations with a valid review score are included in the results.

**5-Sorting**

* The reservations are sorted by the review score in descending order (orderByDesc('reviews.score')), ensuring the highest-rated reservations come first.
* Additionally, the reservations are ordered by their creation date in descending order (latest('reservations.created\_at')), so the most recent reservations appear first if multiple reservations have the same score.

**6-Pagination**

* The method supports pagination:
  + It retrieves the reservations with their associated reviews, ordered by the highest review scores.
  + Pagination metadata is returned, including:
    - total: The total number of reservations matching the criteria.
    - per\_page: The number of items displayed per page (set to 10).
    - current\_page: The current page number.
    - total\_pages: The total number of pages based on the number of reservations and items per page.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **reservations**: A collection of the highest-rated reservations for the client, each including the reservation details and review score.
  + **pagination**: Pagination metadata:
    - total: The total number of reservations matching the query.
    - per\_page: The number of reservations per page (10 in this case).
    - current\_page: The current page number.
    - total\_pages: The total number of pages based on the number of reservations and items per page.
* **Message**: A localized success message: "تم استرجاع الطلبات بنجاح" (Reservations have been successfully retrieved).

**🔹 Error Handling**

* If any error occurs, the method will return an error response with:
  + **Error Code**: 422.
  + **Error Message**: The exception’s message will be included.

**🔹 Return Data Structure**

**🔹 Success**

* **Message**: The success message confirming the retrieval of the highest-rated reservations for the client.
* **Data**:
  + **reservations**: A collection of reservations, including details like id, from, to, and the associated review score.
  + **pagination** (if paginated):
    - total: The total number of reservations in the result set.
    - per\_page: Number of reservations per page.
    - current\_page: The current page.
    - total\_pages: The total pages of results.

**🔹 Error**

* **Error**: If an error occurs, an error code 422 will be returned along with the error message.

**10.18 getMostPopularHours() Method**

### 🔹 Overview

This method retrieves statistics for the most popular hours for reservations based on the provided year and month (or just the year) as input. It supports both specific month filtering and year-based filtering with detailed hour-wise reservation data.

**🔹 Key Functionality**

**1-Filter by Year and Month (if provided)**

* **If both year and month are provided**:
  + The method will filter reservations by the specified year and month (whereYear('date', $request->year) and whereMonth('date', $request->month)).
* **If only year is provided**:
  + The method will handle different cases based on the year:
    - If the year is **2024**, the date range will be from **March 1st, 2024** to **December 31st, 2024**.
    - For other years, the method will select reservations between **January 1st** and **December 31st** of that year.

**2-Time Conversion**

* The method uses REGEXP\_REPLACE to adjust the from time field, converting the Arabic AM/PM indicators (ص, م) into English (AM, PM).
* It uses STR\_TO\_DATE and TIME\_FORMAT to parse the time into a standardized format, ensuring it works correctly with both 12-hour and 24-hour time formats.

**3-Group by Hour**

* The reservations are grouped by hour (using groupByRaw("hour")), and the total number of reservations for each hour is counted (COUNT(\*) as total\_reservations).

**4-Filtering and Sorting**

* **Non-null hours**: Only hours that have reservations are included in the result (havingRaw("hour IS NOT NULL")).
* The results are ordered by the total number of reservations in descending order, ensuring the most popular hours appear first.

**5-Data Preparation**

* The results are mapped to a simpler structure with the hour and total reservation count:
  + hour: The specific hour of the day (in the format hh:mm AM/PM).
  + total\_reservations: The total number of reservations for that hour.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **statistics**: A list of hours with the corresponding total reservation count for each hour.
* **Message**: A localized success message: "Hourly reservation statistics."

**🔹 Error Handling**

* If any error occurs, an error response with a message will be returned, though the current implementation does not have explicit error handling for this method.

**10.19 clients\_wallets() Method**

### 🔹 Overview

This method retrieves a list of clients with their wallet operations, filtered based on user role and with pagination options. It allows users with certain roles to access client wallet details.

**🔹 Key Functionality**

**1-User Role Check**

* **Role Validation**: The method checks if the authenticated user ($user) has a role ID of either **8** or **1**. These roles are authorized to access the clients' wallets data.
  + If the user has a valid role, the method proceeds to fetch the data.
  + If the user does not have the required role, an error response with status code 403 (Forbidden) is returned.

**2-Client Filtering**

* **Client Type Filtering**: The method filters clients by type = 0, which represents a specific type of client.
* **Wallet Operations**: The method also eager loads the wallet\_operations relationship for each client, which is assumed to contain data about the client's wallet transactions.

**3-Pagination**

* The method checks the paginate parameter passed in the request:
  + **If paginate is set to 1**: The method retrieves a paginated list of clients (paginate(10)), with pagination metadata included in the response.
  + **If paginate is set to 0**: The method retrieves all clients without pagination (get()).
  + **Default**: If the paginate parameter is not provided, the method defaults to paginating the client list (paginate(10)).

**4-Pagination Metadata**

* **Total**: Total number of clients.
* **Per Page**: Number of clients per page (10).
* **Current Page**: The current page number.
* **Total Pages**: The total number of pages.

**5-Response Structure**

* **Success**: The response includes:
  + wallets: A collection of clients with their wallet operations.
  + pagination: Pagination details (only when paginate is set to 1).
* **Error Handling**: If any exception occurs, the method catches it and returns a 422 status code with the error message.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **wallets**: A collection of client data with their associated wallet operations.
  + **pagination**: Pagination details (when paginate is set to 1).
* **Message**: A localized success message: "Client wallets".

**🔹 Error Response**

* **Forbidden Access**: If the user’s role is not authorized, a 403 error is returned with a message "Forbidden".
* **General Error**: If an exception is thrown, a 422 error is returned with the exception message.

**10.20 representatives\_wallets() Method**

### 🔹 Overview

This method retrieves a list of representatives (clients with type = 1 and status = 1) along with their wallet operations, filtered based on user role and pagination options. Only users with roles 1 or 8 are authorized to access this data.

**🔹 Key Functionality**

**1-User Role Check**

* **Role Validation**: The method checks if the authenticated user ($user) has a role ID of either **8** or **1**. These roles are authorized to access the representatives' wallets data.
  + If the user has a valid role, the method proceeds to fetch the data.
  + If the user does not have the required role, an error response with status code 403 (Forbidden) is returned.

**2-Client Filtering**

* **Client Type**: The method filters clients by type = 1, which represents representatives.
* **Client Status**: It also filters by status = 1, ensuring only active representatives are retrieved.
* **Wallet Operations**: The method eager loads the wallet\_operations relationship for each representative.

**3-Pagination**

* The method checks the paginate parameter passed in the request:
  + **If paginate is set to 1**: The method retrieves a paginated list of representatives (paginate(10)), with pagination metadata included in the response.
  + **If paginate is set to 0**: The method retrieves all representatives without pagination (get()).
  + **Default**: If the paginate parameter is not provided, the method defaults to paginating the list (paginate(10)).

**4-Pagination Metadata**

* **Total**: Total number of representatives.
* **Per Page**: Number of representatives per page (10).
* **Current Page**: The current page number.
* **Total Pages**: The total number of pages.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **wallets**: A collection of representatives' data with their associated wallet operations.
  + **pagination**: Pagination details (only when paginate is set to 1).
* **Message**: A localized success message: "Client wallets".

**🔹 Error Response**

* **Forbidden Access**: If the user’s role is not authorized, a 403 error is returned with a message "Forbidden".
* **General Error**: If an exception is thrown, a 422 error is returned with the exception message.

**10.21 walletOperationsOfClient() Method**

### 🔹 Overview

This method retrieves a list of wallet operations for a specific client, filtered by the client ID and status. It also calculates the total amount for the client’s wallet operations and supports pagination.

Only users with roles 1 or 8 are authorized to access this method. The method also returns pagination metadata and the total sum of wallet operations.

**🔹 Key Functionality**

**1-User Role Check**

* **Role Validation**: The method checks if the authenticated user ($user) has a role ID of either **8** or **1**. These roles are authorized to access the wallet operations data of clients.
  + If the user has a valid role, the method proceeds to fetch the data.
  + If the user does not have the required role, an error response with status code 403 (Forbidden) is returned.

**2-Wallet Operations Filtering**

* **Status**: The method filters wallet operations by status = 1, ensuring only active operations are retrieved.
* **Client ID**: It filters wallet operations based on the provided client\_id, ensuring that only the operations related to the specified client are fetched.

**3-Pagination**

* The method retrieves a paginated list of wallet operations for the given client using paginate(10), which returns 10 items per page.
* **Pagination Metadata**: The response includes metadata about the pagination:
  + **Total**: Total number of wallet operations.
  + **Per Page**: Number of wallet operations per page (10).
  + **Current Page**: The current page number.
  + **Total Pages**: The total number of pages.

**4-Total Amount Calculation**

* The method calculates the total sum of the wallet operations for the specified client (client\_id) by using sum('amount'). This gives the total value of all the client's wallet operations with status = 1.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **operations**: A collection of wallet operations for the client. Each operation includes attributes such as:
    - id: The operation ID.
    - type: The type of operation (deposit, withdrawal, etc.).
    - amount: The amount of the operation.
    - created\_at: The timestamp when the operation was created.
  + **total\_amounts**: The total sum of wallet operations for the client.
  + **pagination**: Pagination metadata for the response.
* **Message**: A localized success message: "Client operations".

**10.22 walletChargingOperationsOfClient() Method**

### 🔹 Overview

This method retrieves wallet charging operations of clients (type 0), with an optional status filter. It supports both pagination and fetching all records, depending on the request. The method also handles filtering by operation type and status.

Only authorized users can access this method. It returns the wallet charging operations along with pagination metadata if needed.

**🔹 Key Functionality**

**1-Filtering by Status**

* If the status is provided in the request, the method filters the wallet operations by that specific status.

**2-Client Type Filtering**

* The method retrieves wallet charging operations only for clients with type = 0.

**3-Operation Type**

* The method specifically filters for wallet operations of type 1, which represent charging operations.

**4-Pagination**

* **With Pagination**: If the request includes paginate=1, the method fetches the charging operations with pagination (10 items per page).
  + Pagination metadata is included in the response:
    - **Total**: Total number of wallet charging operations.
    - **Per Page**: Number of wallet charging operations per page (10).
    - **Current Page**: The current page number.
    - **Total Pages**: Total number of pages.
* **Without Pagination**: If pagination is not requested, all wallet charging operations are fetched without pagination.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **operations**: A collection of wallet charging operations for clients. Each operation includes attributes such as:
    - id: The operation ID.
    - type: The type of operation (charging).
    - amount: The amount of the operation.
    - created\_at: The timestamp when the operation was created.
  + **pagination**: Metadata related to pagination (only if paginate=1).
* **Message**: A localized success message: "Client operations".

**🔹 Error Response**

* If an exception is thrown, a 422 error is returned with the exception message.
* If the user does not have the required role, a 403 response is returned with a message: "Forbidden".

**10.23 client\_cars() Method**

### 🔹 Overview

This method retrieves all cars associated with a client. The method supports filtering by client\_id and pagination, making it easy to fetch car data for a specific client or all clients with the option of paginated results.

The method returns the car data, including pagination metadata if requested.

**🔹 Key Functionality**

**1-Filtering by client\_id**

* If the client\_id is provided in the request, it filters the cars to only include those owned by the specific client (using user\_id).

**2-Pagination**

* **With Pagination**: If the request includes paginate and it is truthy, the method fetches cars with pagination (10 cars per page).
  + Pagination metadata is included in the response:
    - **Total**: Total number of cars.
    - **Per Page**: Number of cars per page (10).
    - **Current Page**: The current page number.
    - **Total Pages**: Total number of pages.
* **Without Pagination**: If pagination is not requested, all cars are retrieved without pagination.

**3-Response Structure**

* **Success**: The response includes:
  + **cars**: A collection of cars.
  + **pagination** (only if pagination is requested): Metadata for pagination.
* **Error Handling**: In case of an exception, the method returns a 422 error with the exception message.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **cars**: A collection of car data. Each car includes attributes such as:
    - id: The car's ID.
    - make: The car's make (e.g., Toyota).
    - model: The car's model.
    - year: The car's year of manufacture.
    - user\_id: The ID of the client who owns the car.
    - created\_at: The timestamp when the car was created.
  + **pagination**: Pagination metadata (only if paginate is set to true in the request).
* **Message**: A localized success message: "All cars".

**Error Response**

* If an exception is thrown, a 422 error is returned with the exception message.

**10.24 getMonthlyClientsStats() Method**

### 🔹 Overview

This method calculates and returns the client statistics for a specific month. It provides the following for each day in the specified month:

* **Total Clients**: Number of clients who registered on that specific day.
* **Total Reservations**: Number of reservations made by those clients on the same day they registered.
* **Reservation Ratio**: The ratio of reservations to the total number of clients for each day.

The statistics are grouped by day, and the response contains data for every day of the specified month, even if no clients or reservations were made on certain days.

**🔹 Key Functionality**

**1-Fetch Client and Reservation Data**

* A LEFT JOIN is used between the clients and reservations tables to fetch the statistics for each day.
  + The clients.created\_at date is compared with the reservations.created\_at date to group reservations by the day clients registered.
  + Only reservations with a status that is not equal to 6 are considered.

**2-Filtering by Year and Month**

* The query is filtered by clients.created\_at for both the year and month provided in the request.

**3-Initialize Data for Each Day of the Month**

* An array is created to store the statistics for each day of the month.
* The array is initialized with default values:
  + **Total Clients**: Set to 0.
  + **Total Reservations**: Set to 0.
  + **Reservation Ratio**: Set to 0.

**4-Merge Client Stats into Data Array**

* The statistics for each day are merged into the data array.
* The **reservation ratio** is calculated as (total\_reservations / total\_clients) \* 100, rounded to two decimal places.

**5-Return Data**

* The data array is returned in a sequential format, containing the statistics for each day of the month.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **statistics**: A list of statistics for each day in the specified month.
    - For each day, the following fields are present:
      * day: The day of the month.
      * total\_clients: The number of clients registered on that day.
      * total\_reservations: The number of reservations made by those clients on that day.
      * reservation\_ratio: The ratio of reservations to clients on that day, represented as a percentage.
* **Message**: A localized success message: "Clients statistics for the month."

**🔹 Error Response**

* If an error occurs, the method returns a 422 error with the exception message.

**10.25 getClientsRegisteredInDay() Method**

### 🔹 Overview

This method retrieves the clients who were registered on a specific day, along with the count of reservations made by each client on that same day. The method provides pagination and handles errors gracefully.

**🔹 Key Functionality**

**1-Fetch Clients Registered on a Specific Day**

* The query fetches clients that were registered on the day provided in the request. This is done by using the whereDate method, which compares the created\_at field with the provided year, month, and day.

**2-Count Reservations for Each Client on the Same Day**

* Using the withCount method, the query counts the number of reservations for each client on the same day they were registered. This is done by applying the whereDate filter to the reservations.created\_at field.

**3-Pagination**

* The result is paginated with 10 items per page by default.
* The pagination data includes:
  + total: Total number of clients.
  + per\_page: Number of clients per page.
  + current\_page: Current page number.
  + total\_pages: Total number of pages.

**4-Return Data**

* The method returns the list of clients along with the count of reservations for each client, paginated. The ClientResource is used to format the client data.
* The response includes the pagination information for easy navigation of pages.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **clients**: A paginated list of clients who were registered on the specified day.
    - Each client will include a reservations\_count\_of\_some\_day field, which contains the count of reservations they made on the same day they registered.
  + **pagination**: The pagination data for the results.
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: The total number of pages.
* **Message**: A localized success message: "All clients registered on the specified day."

**🔹 Error Response**

* If an error occurs, the method returns a 422 error with the exception message.

**10.26 number\_of\_groups() Method**

### 🔹 Overview

This method calculates the number of groups needed based on the total number of clients, where each group contains up to 200 clients. It also provides a custom label for each group, including the date of the last send action for the group if available.

**🔹 Key Functionality**

**1-Fetch Total Number of Clients**

The method doesn't fetch the total number of clients directly but instead calculates the number of groups by determining the highest client ID (max('id')). The total number of clients is indirectly inferred from this.

**2-Group Calculation**

* The total number of clients is divided by a predefined groupSize of 200.
* The number of groups is calculated using ceil() to ensure that any remainder clients still form a new group.

**3-Fetching Last Send Date**

* The method fetches the last send date for each group from the type\_actions table, where the type column represents the group ID.
* The send\_date is formatted into a readable format (Y-m-d).

**4-Group Creation**

* A set of groups is created with the following properties:
  + **id**: Group number.
  + **value**: The label for the group, including the range of client IDs in that group (e.g., "Group 1 (from 1 to 200)") and the last send date if available.

**5-Returning Data**

The method returns the group data in a structured format, along with any associated send date if present. This information is returned via a returnData response, which will include:

* A list of groups.
* Each group will have an ID, a label with the range of clients in that group, and the send date (if available).

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **groups**: An array of groups, each containing:
    - id: Group ID.
    - value: A string description of the group, e.g., "Group 1 (from 1 to 200)", and an additional message about the last send date if available.

**🔹 Error Response**

* If an error occurs, the method returns a 422 error with the exception message.

**10.27 getClientsWithoutReservationsReport() Method**

### 🔹 Overview

This method fetches clients who do not have any reservations and returns the list either with pagination or as a full list, depending on the request. It also filters clients by their type and status before returning the data.

**🔹 Key Functionality**

**1-Check for Pagination**

* The method first checks if the paginate parameter is present in the request and if it's set to 1. If it is, pagination will be applied; otherwise, the entire list of clients will be returned.

**2-Fetching Clients Without Reservations**

* **Using whereDoesntHave('reservations')**: This condition ensures that only clients who do **not** have any associated reservations are retrieved.
* **Additional Filters**:
  + **Client type**: Only clients with a type of 0 are considered.
  + **Client status**: Only clients with a status of 1 are considered.

**3-Handling Pagination**

* If pagination is required:
  + **latest()**: The clients are sorted by their latest creation date.
  + **paginate(200)**: Limits the number of results per page to 200 clients.
  + A pagination object is created with information about the total, current page, items per page, and total pages.
* **If no pagination is required**: All matching clients are fetched with .get().

**4-Returning Data**

* **If paginated**: The response includes the list of clients and pagination details.
* **If not paginated**: The response includes just the list of clients.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **clients**: A collection of client data that matches the conditions (i.e., clients without reservations, of type 0, and with status 1).
  + **pagination**: If pagination is applied, it includes the following information:
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: The total number of pages.

**🔹 Error Response**

* If an error occurs, the method returns a 422 error with the exception message.

**10.28 getClientsWithPackagesReport() Method**

### 🔹 Overview

This method fetches clients who are subscribed to active packages (i.e., packages with a status of 1). It allows for both paginated and non-paginated results based on the request, and filters clients by their type and status before returning the data.

**🔹 Key Functionality**

**1-Check for Pagination**

* The method first checks if the paginate parameter is provided in the request and if it is set to 1. If true, pagination will be applied; otherwise, all clients will be returned as a single result set.

**2-Fetching Clients with Active Package Subscriptions**

* **Using whereHas('packageSubscriptions')**: This condition ensures that only clients who are associated with at least one active package subscription are fetched.
  + **where('status', 1)**: The active status of the package subscription is 1, ensuring only clients with active subscriptions are retrieved.
* **Additional Filters**:
  + **Client type**: Only clients with a type of 0 are included.
  + **Client status**: Only clients with a status of 1 (active clients) are considered.

**3-Handling Pagination**

* If pagination is required:
  + **latest()**: The clients are sorted by their latest creation date.
  + **paginate(200)**: The results are paginated, with a limit of 200 clients per page.
  + A pagination object is created with:
    - total: Total number of clients matching the query.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: The total number of pages.
* **If no pagination is required**: All matching clients are fetched using .get().

**4-Returning Data**

* **If paginated**: The response includes both the list of clients and pagination details.
* **If not paginated**: Only the list of clients is returned.

**🔹 Return Response**

**🔹 Success Response**

* **Data**:
  + **clients**: A collection of clients who have an active package subscription, are of type 0, and have a status of 1.
  + **pagination**: If pagination is applied, it includes the following information:
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: The total number of pages.

**🔹 Error Response**

* If an error occurs (e.g., a database error), the method returns an error response with a status code 422 and the exception message.

**10.29 getClientsWithoutPackagesReport() Method**

### 🔹 Overview

This method is designed to fetch clients who either:

1. Do not have any package subscriptions.
2. Have package subscriptions, but the subscriptions are inactive (i.e., their status is 0).

It supports both paginated and non-paginated results, depending on the request. It also filters the clients based on their type and status.

### 🔹 Key Functionality

#### 1-****Check for Pagination****

* The method first checks if the paginate parameter is included in the request and if it is set to 1. If true, pagination will be applied. If not, all clients will be returned without pagination.

#### 2-****Fetching Clients without Active Package Subscriptions****

* **Using whereDoesntHave('packageSubscriptions')**: This condition filters clients who do not have any package subscriptions.
* **Using orWhereHas('packageSubscriptions', ...)**: This condition filters clients who have a package subscription, but the status of the subscription is 0 (inactive). This is combined with the whereDoesntHave() condition using the orWhere clause.
* **Additional Filters**:
  + **Client type**: Only clients with a type of 0 are included.
  + **Client status**: Only clients with a status of 1 (active clients) are considered.

#### 3-****Handling Pagination****

* If pagination is required:
  + **latest()**: Clients are ordered by their creation date (latest first).
  + **paginate(200)**: The results are paginated with a maximum of 200 clients per page.
  + A pagination object is created with:
    - total: Total number of clients matching the query.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: The total number of pages.
* **If pagination is not required**: All matching clients are fetched using .get().

#### 4-****Returning Data****

* **If paginated**: The response includes both the list of clients and pagination details.
* **If not paginated**: Only the list of clients is returned.

### 🔹 Return Response

#### 🔹 ****Success Response****

* **Data**:
  + **clients**: A collection of clients who either do not have package subscriptions or have inactive subscriptions, and meet the filtering criteria (i.e., type 0 and status 1).
  + **pagination**: If pagination is applied, it includes the following information:
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: The total number of pages.

#### 🔹 ****Error Response****

* If an error occurs (e.g., a database error), the method returns an error response with a status code 422 and the exception message.

**10.30 getClientsWithoutReservationsForWeekReport() Method**

### 🔹 Overview

This method is designed to fetch clients who:

1. **Do not have any reservations** in the past week. This is determined by checking if the reservations relationship exists and filtering by the reservation's date.

It supports both **pagination** and **no pagination** (fetching all clients) based on the request. Additionally, it filters clients by type (0) and status (1).

### 🔹 Key Functionality

#### 1-****Check for Pagination****

* The method checks if the paginate parameter is present in the request and if it's set to 1. If true, pagination is applied to the results. If not, all clients are returned.

#### 2-****Fetching Clients without Reservations for the Past Week****

* **Using whereDoesntHave('reservations')**: This condition filters clients who do not have any reservations.
* **Additional Condition on Reservations**:
  + The where clause within the whereDoesntHave ensures that the reservations date is checked. If no reservations are made in the past week, the client is included in the result.
  + **Date Condition**: The query checks if the reservation's date is greater than or equal to the date of 7 days ago (using Carbon::now()->subWeek()->toDateString()).
* **Additional Filters**:
  + **Client type**: Filters clients with a type of 0.
  + **Client status**: Filters clients with a status of 1 (active clients).

#### 3-****Handling Pagination****

* If pagination is required:
  + **latest()**: Orders the clients by their creation date, with the most recent ones appearing first.
  + **paginate(200)**: Limits the results to 200 clients per page.
  + Pagination details are calculated, including:
    - total: The total number of clients.
    - per\_page: The number of clients per page.
    - current\_page: The current page number.
    - total\_pages: The total number of pages.
* **If pagination is not required**: All matching clients are fetched using .get().

#### 4-****Returning Data****

* **If paginated**: Returns the clients along with pagination details.
* **If not paginated**: Returns just the list of clients.

### 🔹 Return Response

#### 🔹 ****Success Response****

* **Data**:
  + **clients**: A collection of clients who have not made any reservations in the past week and meet the filtering criteria (i.e., type 0 and status 1).
  + **pagination**: If pagination is applied, the response includes the pagination details:
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: Total number of pages.

#### 🔹 ****Error Response****

* If any error occurs (such as a database issue or invalid data), the method will catch the error and return:
  + **Error message**: The exception message from the catch block.

**10.31 getClientsWithoutReservationsForMonthReport() Method**

🔹 **Overview**

* + The getClientsWithoutReservationsForMonthReport() method is similar to the previous one, with the difference being that it checks for clients who have not made any reservations in the past **month** instead of the past week.

### 🔹 Key Functionality

1. **Pagination Check**:
   * Similar to previous methods, it checks if the paginate parameter is present in the request and if it's set to 1. If true, pagination is applied. Otherwise, all results are fetched without pagination.
2. **Fetching Clients without Reservations in the Past Month**:
   * **Using whereDoesntHave('reservations')**: This filters clients who do not have any reservations.
   * **Date Condition**:
     + The query checks if the reservation date is **greater than or equal** to the date one month ago (Carbon::now()->subMonth()->toDateString()).
     + This ensures the clients who have made reservations within the last month are excluded.
3. **Client Filters**:
   * **Client type**: Filters clients with a type of 0 (you can adjust based on your specific needs).
   * **Client status**: Filters active clients (status = 1).
4. **Pagination**:
   * **With Pagination**: If pagination is enabled, the results are limited to 200 clients per page. The latest() method orders the results by the most recent clients.
   * **Without Pagination**: If no pagination is needed, all the matching clients are fetched with the .get() method.
5. **Returning the Data**:
   * If **pagination** is used, it returns the clients along with the pagination details (total, per\_page, current\_page, total\_pages).
   * If **pagination** is not used, it returns only the clients.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: Returns a success message, such as "Clients without reservations for the past month fetched successfully."
* **Data**:
  + **clients**: A collection of clients who have not made any reservations in the last month.
  + **pagination** (only if pagination is applied):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: Total number of pages.

**Error Response**:

* If any error occurs (e.g., database issues), the method will return an error message in the response.

**10.32 getClientsWithoutReservationsForThreeMonthsReport() Method**

🔹 **Overview**

* The getClientsWithoutReservationsForThreeMonthsReport() method is almost identical to the previous methods, but it checks for clients who have not made any reservations in the last **three months** instead of a week or a month.

### 🔹 Key Functionality

1. **Pagination Check**:
   * Checks if the paginate parameter is set in the request and if it's equal to 1. If true, pagination will be applied. Otherwise, all the results are fetched without pagination.
2. **Fetching Clients without Reservations in the Past Three Months**:
   * **Using whereDoesntHave('reservations')**: This filters clients who do not have any reservations.
   * **Date Condition**:
     + The query checks if the reservation date is **greater than or equal** to the date three months ago (Carbon::now()->subMonth(3)->toDateString()).
     + This ensures the clients who have made reservations in the last three months are excluded.
3. **Client Filters**:
   * **Client type**: Filters clients with a type of 0.
   * **Client status**: Filters only active clients (status = 1).
4. **Pagination**:
   * **With Pagination**: If pagination is enabled, the results will be limited to 200 clients per page. The latest() method orders the results by the most recent clients.
   * **Without Pagination**: If pagination is not needed, it fetches all the matching clients with the .get() method.
5. **Returning the Data**:
   * If **pagination** is used, it returns the clients along with pagination details like total, per\_page, current\_page, and total\_pages.
   * If **pagination** is not used, it returns just the clients.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that the clients without reservations for the past three months are fetched successfully.
* **Data**:
  + **clients**: A collection of clients who haven't made any reservations in the last three months.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: Total number of pages.

#### ****Error Response****:

* If any error occurs (such as a database issue), the method will return an error response with a code and the error message.

**12.33 getClientsWithoutReservationsForSixMonthsReport() Method**

🔹 **Overview**

The getClientsWithoutReservationsForSixMonthsReport() method is very similar to the previous ones, with the key difference being that this version targets clients who have **not** made any reservations in the **last six months**.

### 🔹 Key Functionality

1. **Pagination Check**:
   * The method checks if the paginate parameter is set in the request and if it's equal to 1. If true, pagination will be applied. Otherwise, the results are fetched without pagination.
2. **Fetching Clients without Reservations in the Past Six Months**:
   * **Using whereDoesntHave('reservations')**: Filters clients who do not have any reservations.
   * **Date Condition**:
     + The query checks if the reservation date is **greater than or equal** to six months ago (Carbon::now()->subMonth(6)->toDateString()).
     + This ensures that clients who have made reservations in the last six months are excluded.
3. **Client Filters**:
   * **Client type**: Filters clients with a type of 0.
   * **Client status**: Filters only active clients (status = 1).
4. **Pagination**:
   * **With Pagination**: If pagination is enabled, the results will be limited to 200 clients per page. The latest() method orders the results by the most recent clients.
   * **Without Pagination**: If pagination is not requested, the query will fetch all matching clients.
5. **Returning the Data**:
   * If **pagination** is enabled, it returns the clients along with pagination details such as total, per\_page, current\_page, and total\_pages.
   * If **pagination** is not enabled, it simply returns the list of clients.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that the clients without reservations for the past six months are fetched successfully.
* **Data**:
  + **clients**: A collection of clients who haven't made any reservations in the last six months.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: The current page number.
    - total\_pages: Total number of pages.

#### ****Error Response****:

* If an error occurs (e.g., database issues), the method will return an error response with a code and the error message.

**10.34 getClientsWithoutReservationsForNineMonthsReport() Method**

🔹 **Overview**

* The getClientsWithoutReservationsForNineMonthsReport() method is designed to retrieve clients who have not made any reservations in the past **nine months**. It is similar to the other methods, but specifically targets a **nine-month** period.

### 🔹 Key Functionality

1. **Pagination Check**:
   * The method first checks if the paginate parameter is provided in the request and is set to 1. If pagination is enabled, it will return a paginated response; otherwise, it will fetch all the matching clients without pagination.
2. **Fetching Clients without Reservations in the Last Nine Months**:
   * **Using whereDoesntHave('reservations')**: Filters clients who do **not** have any reservations.
   * **Date Condition**:
     + The query checks if the reservation date is **greater than or equal** to **nine months ago** using Carbon::now()->subMonth(9)->toDateString().
3. **Client Filters**:
   * **Client Type**: Filters clients whose type is 0.
   * **Client Status**: Filters clients that are **active** (status = 1).
4. **Pagination**:
   * **With Pagination**: If pagination is enabled, the method limits the results to 200 clients per page.
   * **Without Pagination**: If pagination is not requested, it fetches all the matching clients.
5. **Returning the Data**:
   * If pagination is enabled, it returns the clients along with pagination details such as total, per\_page, current\_page, and total\_pages.
   * If pagination is not enabled, it simply returns the full list of clients.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that clients without reservations in the last nine months have been fetched.
* **Data**:
  + **clients**: A collection of clients who haven't made any reservations in the last nine months.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

#### ****Error Response****:

* If an error occurs (e.g., a database issue), the method returns an error response with an error code and message.

**10.35 getClientsWithoutReservationsForYearReport() Method**

🔹 **Overview**

* The getClientsWithoutReservationsForYearReport() method is designed to retrieve clients who haven't made any reservations in the last **year**. It follows the same structure as the other methods, with a specific focus on a **12-month** period.

### 🔹 Key Functionality

1. **Pagination Check**:
   * The method first checks if the paginate parameter is present in the request and is set to 1. If pagination is enabled, it will return the data in pages; otherwise, it will fetch all clients without pagination.
2. **Fetching Clients without Reservations in the Last Year**:
   * **Using whereDoesntHave('reservations')**: Filters clients who do **not** have any reservations.
   * **Date Condition**:
     + The query filters clients where the reservation date is **greater than or equal** to **one year ago** using Carbon::now()->subMonth(12)->toDateString().
3. **Client Filters**:
   * **Client Type**: Filters clients with type equal to 0.
   * **Client Status**: Filters only active clients (status = 1).
4. **Pagination**:
   * **With Pagination**: If pagination is enabled, the method limits the results to 200 clients per page.
   * **Without Pagination**: If pagination is not requested, it fetches all matching clients.
5. **Returning the Data**:
   * If pagination is enabled, it returns the clients along with pagination details such as total, per\_page, current\_page, and total\_pages.
   * If pagination is not enabled, it returns the list of clients without pagination.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that clients without reservations in the last year have been fetched.
* **Data**:
  + **clients**: A collection of clients who have not made any reservations in the last year.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

#### ****Error Response****:

* If an error occurs (e.g., a database issue), the method returns an error response with an error code and message.

**10.36 getClientsWithoutReservationsAtBuraidah() Method**

🔹 **Overview**

* The getClientsWithoutReservationsAtBuraidah() method is designed to fetch clients who have not made any reservations and are located in **Buraidah** (a city in Saudi Arabia). This method supports both paginated and non-paginated responses and filters clients based on their location and reservation status.

### 🔹 Key Functionality

1. **Pagination Check**:
   * The method first checks if the paginate parameter is present and set to 1. If pagination is enabled, it will return the data in pages; otherwise, it fetches all clients without pagination.
2. **Fetching Clients without Reservations in Buraidah**:
   * **Using whereHas('locations')**: Filters clients who have a location with a name like "بريدة" (the Arabic name for Buraidah).
   * **Using whereDoesntHave('reservations')**: Filters clients who have **no reservations**.
   * **Client Type and Status**: The method filters clients where:
     + type = 0 (assuming this is the type of client you're interested in).
     + status = 1 (active clients).
3. **Pagination**:
   * **With Pagination**: If pagination is enabled, it limits the results to 200 clients per page.
   * **Without Pagination**: If pagination is not requested, it fetches all matching clients.
4. **Returning the Data**:
   * If pagination is enabled, the method returns the clients along with pagination details, such as total, per\_page, current\_page, and total\_pages.
   * If pagination is not enabled, it returns the list of clients without pagination.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that clients without reservations in Buraidah have been fetched.
* **Data**:
  + **clients**: A collection of clients located in Buraidah who have not made any reservations.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: Current page number.

#### ****Error Response****:

* If an error occurs (e.g., a database issue), the method returns an error response with an error code and message.

**10.37 getClientsSendGift() Method**

🔹 **Overview**

* The getClientsSendGift() method is designed to fetch clients who have sent gifts. It allows for both paginated and non-paginated responses and filters clients based on their type, status, and whether they have associated gift-sending information.

### 🔹 Key Functionality

1. **Pagination Check**:
   * The method first checks if the paginate parameter is set to 1. If so, it will return a paginated response with up to 200 clients per page. If pagination is not requested, it fetches all matching clients without pagination.
2. **Fetching Clients Who Have Sent Gifts**:
   * **Using whereHas('giftsSender')**: Filters clients who have an associated gift-sender relationship. This means the clients must have sent at least one gift.
   * **Client Type and Status**: Filters clients where:
     + type = 0 (presumably active clients).
     + status = 1 (active clients).
3. **Pagination**:
   * **With Pagination**: When pagination is requested, the query limits the number of clients to 200 per page.
   * **Without Pagination**: If pagination is not enabled, all matching clients are retrieved.
4. **Returning the Data**:
   * If pagination is enabled, the method returns the clients along with pagination details like total, per\_page, current\_page, and total\_pages.
   * If pagination is not enabled, it returns the list of clients without pagination.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that clients who sent gifts have been fetched.
* **Data**:
  + **clients**: A collection of clients who have sent gifts.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

#### ****Error Response****:

* If an error occurs (e.g., a database issue), the method returns an error response with an error code and message.

**10.38 getClientsReceiveGift() Method**

🔹 **Overview**

* The getClientsReceiveGift() method is designed to fetch clients who have received gifts. It supports both paginated and non-paginated responses, and filters clients based on specific conditions such as type, status, and whether they have received gifts.

### 🔹 Key Functionality

1. **Pagination Check**:
   * The method first checks if the paginate parameter is set to 1. If so, it will return a paginated response with a maximum of 200 clients per page. If pagination is not requested, it fetches all matching clients without pagination.
2. **Fetching Clients Who Have Received Gifts**:
   * **Using whereHas('giftsReceive')**: Filters clients who have an associated giftsReceive relationship, meaning these clients have received at least one gift.
   * **Client Type and Status**: Filters clients where:
     + type = 0 (active clients).
     + status = 1 (active clients).
3. **Pagination**:
   * **With Pagination**: When pagination is requested, the query limits the number of clients to 200 per page.
   * **Without Pagination**: If pagination is not enabled, all matching clients are retrieved.
4. **Returning the Data**:
   * If pagination is enabled, the method returns the clients along with pagination details like total, per\_page, current\_page, and total\_pages.
   * If pagination is not enabled, it returns the list of clients without pagination.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that clients who have received gifts have been fetched.
* **Data**:
  + **clients**: A collection of clients who have received gifts.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

#### ****Error Response****:

* If an error occurs (e.g., a database issue), the method returns an error response with an error code and message.

**10.39 getClientsOrderedOnce() Method**

🔹 **Overview**

* The getClientsOrderedOnce() method is designed to retrieve clients who have placed exactly one reservation. This function allows for both paginated and non-paginated results depending on the request parameters.

### 🔹 Key Functionality

1. **Pagination Check**:
   * The method first checks if the paginate parameter is set to 1. If pagination is requested, it will return a paginated response with up to 200 clients per page.
   * If pagination is not requested, it retrieves all matching clients without pagination.
2. **Fetching Clients with One Reservation**:
   * **Using whereHas('reservations')**: This filters clients who have associated reservations.
   * **Query inside whereHas**: The query groups reservations by client\_id and filters for clients who have exactly **one reservation**:
     + groupBy('client\_id'): Groups reservations by the client.
     + havingRaw('COUNT(id) = 1'): Ensures only clients with exactly one reservation are returned.
   * **Client Type and Status**: Filters clients based on:
     + type = 0 (active clients).
     + status = 1 (active clients).
3. **Pagination**:
   * **With Pagination**: If the paginate parameter is present and set to 1, it limits the results to 200 clients per page.
   * **Without Pagination**: If pagination is not enabled, it returns all matching clients in a single response.
4. **Returning the Data**:
   * If pagination is enabled, the response will include details of the pagination (total, per\_page, current\_page, total\_pages).
   * If pagination is not enabled, the response will simply return the list of clients.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that the clients who have ordered once have been fetched.
* **Data**:
  + **clients**: A collection of clients who have placed exactly one reservation.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: Current page number.

#### ****Error Response****:

* If an error occurs (e.g., a database issue), the method will return an error response with an error code and message.

**10.40 getClientsOrderedFive() Method**

🔹 **Overview**

* The getClientsOrderedFive() method is designed to retrieve clients who have placed **at least five reservations**. Like the previous methods, this function supports both paginated and non-paginated results depending on the request parameters.

### 🔹 Key Functionality

1. **Pagination Check**:
   * The method first checks if the paginate parameter is set to 1. If pagination is requested, it returns a paginated response with up to 200 clients per page.
   * If pagination is not requested, it retrieves all matching clients without pagination.
2. **Fetching Clients with at Least Five Reservations**:
   * **Using whereHas('reservations')**: Filters clients who have associated reservations.
   * **Query inside whereHas**: The query groups reservations by client\_id and filters for clients who have **at least five reservations**:
     + groupBy('client\_id'): Groups reservations by the client.
     + havingRaw('COUNT(id) >= 5'): Ensures only clients with five or more reservations are returned.
   * **Client Type and Status**: Filters clients based on:
     + type = 0 (active clients).
     + status = 1 (active clients).
3. **Pagination**:
   * **With Pagination**: If the paginate parameter is present and set to 1, it limits the results to 200 clients per page.
   * **Without Pagination**: If pagination is not enabled, it returns all matching clients in a single response.
4. **Returning the Data**:
   * If pagination is enabled, the response will include details of the pagination (total, per\_page, current\_page, total\_pages).
   * If pagination is not enabled, the response will simply return the list of clients.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating that the clients who have ordered at least five times have been fetched.
* **Data**:
  + **clients**: A collection of clients who have placed five or more reservations.
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

#### ****Error Response****:

* If an error occurs (e.g., a database issue), the method will return an error response with an error code and message.

**10.41 targeting\_report() Method**

🔹 **Overview**

* The targeting\_report() method is designed to generate a detailed report of clients based on various criteria, such as completed reservations, pending reservations, subscription count, gift sending/receiving, wallet balance, and registration date. It supports both paginated and non-paginated results.

### 🔹 Key Functionality

1. **Eager Loading Counts**:
   * The method uses the withCount() method to load the counts of related records:
     + completed\_reservations\_count: Counts reservations with status 3 (completed).
     + completed\_reservations\_with\_coupon: Counts completed reservations that have a coupon.
     + pending\_reservations\_count: Counts reservations that are not completed (status != 3).
     + subscription\_count: Counts active package subscriptions (status = 1).
     + sent\_gifts\_count: Counts the number of gifts sent by the client.
     + received\_gifts\_count: Counts the number of gifts received by the client.
2. **Filtering Based on Request Parameters**:
   * **Completed Reservations Count**: Filters clients based on the count of completed reservations.
   * **Completed Reservations with Coupon**: Filters clients based on the count of completed reservations that have a coupon.
   * **Pending Reservations Count**: Filters clients based on the count of pending reservations.
   * **Subscription Count**: Filters clients based on the count of active subscriptions.
   * **Wallet Balance**: Filters clients based on wallet balance range (wallet\_balance\_from and wallet\_balance\_to).
   * **Sent and Received Gifts Count**: Filters clients based on the count of gifts they have sent or received.
   * **Date Range**: Filters clients based on the registration date range (start\_date and end\_date).
3. **Pagination**:
   * The method checks if pagination is enabled (paginate = 1). If so, it returns a paginated list of clients with a maximum of 200 clients per page.
   * It includes pagination details (total, per\_page, current\_page, total\_pages) in the response.
   * If pagination is not enabled, it fetches all the matching clients without pagination.
4. **Returning the Data**:
   * If pagination is enabled, the response includes the pagination data along with the client targeting report.
   * If pagination is not enabled, the response includes just the client targeting report.

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating the targeting report has been fetched.
* **Data**:
  + **targeting\_report**: A collection of clients with the calculated counts for each category (e.g., completed reservations, sent gifts, etc.).
  + **pagination** (only if pagination is enabled):
    - total: Total number of clients.
    - per\_page: Number of clients per page.
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

#### ****Error Response****:

* If an error occurs (e.g., database issue, invalid parameters), the method will return an error response with an error code and message.

**10.42 allReservayionsDataReport() Method**

🔹 **Overview**

* The allReservayionsDataReport() method is designed to generate a report of **reservations** based on specific date ranges, and it can be customized to filter by a date range specified in the request or default to the current month's start and end.

### 🔹 Key Functionality

1. **Date Range Handling**:
   * The method defaults to fetching reservations for the **current month** by determining the **start** (now()->startOfMonth()) and **end** (now()->endOfMonth()) of the current month.
   * If the user provides custom date ranges in the request (from and to), the method adjusts the start and end dates accordingly using **Carbon** to parse and format the date strings.
2. **Filtering Reservations**:
   * The method fetches **reservations** with the status 3 (likely indicating completed reservations).
   * It then filters the results using the **whereBetween()** method to select records where the date field falls between the computed start and end dates (either the current month or the ones provided by the user).
3. **Returning the Data**:
   * The method returns the **reservations** in a collection, wrapped in a resource format (ReservationResource::collection($reservations)).
   * The response includes the data along with a success message in the language of the API (using the \_\_('api.allReservayionsDataReport') translation).

### 🔹 Return Structure

#### ****Success Response****:

* **Message**: A success message indicating the report has been successfully fetched.
* **Data**:
  + **reservations**: A collection of reservations filtered by the date range and status.

#### ****Error Response****:

* If there is an issue (e.g., invalid dates, database query failure), the method will return an error response with an error code and message.

1. **Company Controller**

**11.1 all(Request $request) Method**

### 🔹 Overview

Retrieves all companies from the database, with optional support for pagination. The response is returned as a structured collection using the CompanyResource.

### 🔹 Key Functionality

#### 1-Check for Pagination Request

The method checks if the request includes a paginate parameter:

* If paginate = 1, it retrieves paginated results (10 per page).
* If paginate = 0, it retrieves **all companies** without pagination.
* If paginate is not provided, it **defaults** to pagination (10 per page).

#### 2-Query Companies

The companies are fetched using Company::latest(), ordering them by created\_at in **descending** order (most recent first).

#### 3-Format the Response

The companies are wrapped in a CompanyResource collection for consistent API structure.  
When paginated, a pagination object is added containing:

* total: total number of companies
* per\_page: results per page
* current\_page: current page number
* total\_pages: total number of pages

### 🔹 Return Response

🔹 **Success:**  
Returns a structured response containing the list of companies and, if paginated, pagination metadata.  
The data is wrapped using CompanyResource::collection($companies).  
The response also includes a localized message from \_\_('api.comapny\_all').

🔹 **Error Handling:**  
If an exception occurs (e.g. database error), it is caught by the catch block.  
Returns a 422 error response using returnError() along with the exception message.

**11.2 get($company\_id) Method**

### 🔹 Overview

Retrieves a single company record based on the provided $company\_id.

### 🔹 Key Functionality

#### 1-Find Company by ID

Uses Company::find($company\_id) to attempt to retrieve the company record from the database.

#### 2-Check If Company Exists

* If the company **exists**, it is returned wrapped in a CompanyResource.
* If the company **does not exist**, it returns a 404 error with a localized message from \_\_('api.company\_not\_found').

### 🔹 Return Response

🔹 **Success:**  
If the company is found:

* Returns the data wrapped in a data key.
* Uses CompanyResource to format the returned company.
* Includes a localized success message from \_\_('api.company\_found').

🔹 **Error Handling:**

* If the company is **not found**, returns a 404 error with a relevant message.
* If any **exception** is thrown during execution (e.g. database connection issue), it catches the error and returns a 422 error with the exception message.

**11.3 add(storeCompanyRequest $request) Method**

### 🔹 Overview

Adds a new company record to the database using the data provided in the validated storeCompanyRequest.

### 🔹 Key Functionality

#### 1-Create New Company

* Uses Company::create([...]) to insert a new company into the database.
* Fields included: name, email, phone, lat, and long.
* All inputs are expected to be validated by the storeCompanyRequest.

#### 2-Wrap in Resource

* Once created, the new company is formatted using CompanyResource for a structured API response.

### 🔹 Return Response

🔹 **Success:**

* Returns the created company wrapped inside a data key.
* Uses CompanyResource to structure the response.
* Includes a localized success message from \_\_('api.company\_added').

🔹 **Error Handling:**

* If any exception occurs (e.g. database issue, unexpected failure), it is caught and returned with:
  + HTTP status code 422
  + Exception message as the error content.

**11.4 update(updateCompanyRequest $request) Method**

### 🔹 Overview

Updates an existing company's details based on the provided request data.

### 🔹 Key Functionality

#### 1-Find Company by ID

* The method attempts to locate the company using the id provided in the request.
* If the company does not exist, a 404 error is returned with a localized message from \_\_('api.company\_not\_found').

#### 2-Update Company Data

* If the company exists, it updates the following fields:
  + name
  + email
  + phone
  + lat
  + long
* The data is updated using Laravel’s update() method.

#### 3-Resource Wrapping

* The updated company data is returned using CompanyResource for a structured API response format.

### 🔹 Return Response

🔹 **Success:**

* Returns the updated company wrapped in a data key using CompanyResource.
* Includes a localized success message from \_\_('api.company\_updated').

🔹 **Error Handling:**

* If the company is not found, returns a 404 error with a localized message.
* If an exception occurs during the update process, returns a 422 error with the exception message.

**11.5 destroy($company\_id) Method**

### 🔹 Overview

Deletes a specific company based on the provided company\_id.

### 🔹 Key Functionality

#### 1-Find Company by ID

* The method looks up the company using Company::find($company\_id).
* If the company does not exist, it returns a 404 error with a localized message from \_\_('api.company\_not\_found').

#### 2-Delete Company

* If the company is found, it is deleted using Laravel’s delete() method.

### 🔹 Return Response

🔹 **Success:**

* Returns a success message using returnSuccess() with the localized message from \_\_('api.company\_deleted').

🔹 **Error Handling:**

* If the company is not found, a 404 error response is returned.
* If any exception occurs during the deletion process, it is caught and a 422 error is returned with the exception message.

1. **Complaint Controller**
   1. **getComplaints(Request $request) Method**

### 🔹 Overview

Retrieves complaints based on the role of the authenticated user and whether pagination is requested or not.

### 🔹 Key Functionality

1-**Filter by Admin Role**  
If the authenticated user has a role ID of 7, the method filters the complaints to only include those associated with that specific admin.

2-**Handle Pagination**

* If pagination is requested (paginate = 1), the method retrieves the complaints in a paginated format, including pagination metadata such as total items, items per page, current page, and total pages.
* If pagination is not requested (paginate = 0), the method retrieves all complaints without pagination.

### 🔹 Return Response

🔹 **Success:**

* Returns the complaints wrapped inside a data key using a structured resource collection.
* If paginated, it includes pagination details in the response.
* Includes a success message indicating that the data was retrieved successfully.

🔹 **Error Handling:**

* If any error occurs, the method catches the exception and returns a JSON response with a 403 status code and the error message.
  1. **changeStatus(Request $request) Method**

### 🔹 Overview

Updates the status of a specific complaint to indicate it has been resolved.

### 🔹 Key Functionality

1-**Find Complaint by ID**  
The method attempts to locate a complaint using the provided complaint\_id. If it does not exist, it returns an error message indicating that the complaint was not found.

2-**Update Complaint Status**  
If the complaint is found, the method sets its status to 1, representing that the complaint has been marked as solved.

### 🔹 Return Response

🔹 **Success:**

* Returns a success response using a localized message to indicate that the complaint status has been successfully changed.

🔹 **Error Handling:**

* If the complaint is not found, returns a 404 error with a relevant localized message.
* If any other error occurs, it returns a 422 error with a message describing what went wrong.
  1. **destroy(Request $request) Method**

### 🔹 Overview

Deletes a specific complaint from the system based on the provided complaint\_id.

### 🔹 Key Functionality

1-**Find Complaint by ID**  
The method checks for the existence of the complaint using the complaint\_id. If it does not exist, an error response is returned indicating that the complaint was not found.

2-**Delete Complaint**  
If the complaint exists, the method proceeds to delete it from the database.

### 🔹 Return Response

🔹 **Success:**

* Returns a success response using a localized message to confirm the complaint was successfully deleted.

🔹 **Error Handling:**

* Returns a 404 error if the complaint is not found.
* If any exception occurs during the process, a 422 error is returned with a descriptive message of the error.

1. **Contact Controller**
   1. **all(Request $request) Method**

### 🔹 Overview

Retrieves a list of contact messages, with the option to filter by status and paginate the results.

### 🔹 Key Functionality

1-**Filter by Status**  
The method checks if a status filter is provided in the request and applies it to filter the contacts by their type (status).

2-**Pagination**

* If the paginate parameter is set to 1, the results are paginated, returning a paginated list of contacts with metadata (total, per page, current page, and total pages).
* If paginate is set to 0, all the contact messages are returned without pagination.
* If no paginate parameter is provided, the default behavior is pagination with 10 items per page.

### 🔹 Return Response

🔹 **Success:**

* The success response includes the list of contacts formatted using ContactResource, along with pagination information (if applicable), and a localized success message.

🔹 **Error Handling:**

* If an exception occurs during the process, a 422 error is returned with a descriptive error message.

**13.2 get($id) Method**

### 🔹 Overview

Retrieves a specific contact message by its unique identifier (id).

**🔹 Key Functionality**

1-**Find Contact by ID**  
The method attempts to find a contact message using the provided id with ContactMessage::find($id).

2-**Check if Contact Exists**  
If the contact is found, it is formatted using ContactResource. If the contact does not exist, it returns a 404 error indicating that the contact was not found.

**🔹 Return Response**

**🔹 Success:**  
When the contact is found, it returns the contact data wrapped in a data key, formatted using the ContactResource, and a localized success message.

**🔹** **Error Handling:**  
If an error occurs or the contact is not found, it returns a 422 error with the exception message or a 404 error when the contact is not found.

**13.3 changeStatus($contact\_id, Request $request) Method**

### 🔹 Overview

Changes the status (or type) of a specific contact message based on the provided contact\_id and the new status from the request.

**🔹 Key Functionality**

1-**Find Contact by ID**  
The method attempts to find the contact message using the provided contact\_id with ContactMessage::find($contact\_id).

2-**Check if Contact Exists**  
If the contact is found, it updates the type field with the new status value from the request. If the contact does not exist, it returns a 404 error indicating that the contact was not found.

3-**Update Contact Status**  
If the contact is found and successfully updated, the new status is saved, and a success message is returned.

**🔹 Return Response**

**🔹 Success:**  
When the contact status is successfully updated, it returns a success response with a localized message indicating the status change.

**🔹** **Error Handling:**  
If an error occurs or the contact is not found, it returns a 422 error with the exception message or a 404 error when the contact is not found.

**13.4 destroy($id) Method**

### 🔹 Overview

Deletes a specific contact message based on the provided id.

**🔹 Key Functionality**

1-**Find Contact by ID**  
The method attempts to find the contact message using the provided id with ContactMessage::find($id).

2-**Check if Contact Exists**  
If the contact is found, it proceeds to delete the contact from the database. If the contact does not exist, it returns a 404 error indicating that the contact was not found.

3-**Delete Contact**  
If the contact is found and successfully deleted, a success message is returned indicating that the contact has been deleted.

**🔹 Return Response**

**🔹 Success:**  
When the contact is successfully deleted, it returns a success response with a localized message confirming the deletion.

**🔹** **Error Handling:**  
If an error occurs or the contact is not found, it returns a 422 error with the exception message or a 404 error when the contact is not found.

1. **Coupon Controller**

**14.1 all(Request $request) Method**

**🔹 Overview**

Retrieves a list of coupons based on optional filters such as type, code, and status, with support for both paginated and non-paginated responses.

**🔹 Key Functionalities**

1-**Initialize Coupon Query**  
Begins with a base query on the Coupon model.

2-**Filter by Type (Optional)**  
If the type is present in the request, filters coupons by that type.

3-**Filter by Code (Optional)**  
If a code is provided, performs a LIKE search to filter coupons that match the code pattern.

4-**Filter by Status (Optional)**  
If statusCoupon is provided:

* **active**: Coupons that are currently active (active\_from is in the future **or** active\_to is in the past) **and** have remaining usage times.
* **Not active**: Coupons that have expired (active\_to is in the past) **and** have no usage times left.

5-**Handle Pagination**

* If paginate is 1: returns a paginated result set of 10 items per page with pagination metadata.
* If paginate is 0: returns all matching coupons without pagination.
* If paginate is not set: defaults to paginated response.

**🔹 Return Response**

**🔹 Success:**  
Returns a success response containing a list of coupons wrapped in a resource along with pagination info if applicable.

**🔹 Error Handling:**  
If any error occurs, returns a 422 error with a localized message and error details.

**14.2 check\_date\_of\_coupon($coupon, $date) Method**

### 🔹 Overview

Determines if a given coupon is valid on a specific date based on its active date range (active\_from and active\_to).

**🔹 Key Functionalities**

1-**No Date Restrictions**  
If both active\_from and active\_to are null, the coupon is always considered valid.

2-**Valid Within Date Range**  
If both active\_from and active\_to are set, the coupon is valid if the provided date falls within this range.

3-**Only active\_to Set**  
If only the end date (active\_to) is set, the coupon is valid if the provided date is **before or on** that end date.

4-**Only active\_from Set**  
If only the start date (active\_from) is set, the coupon is valid if the provided date is **on or after** that start date.

5-**Invalid Date Range**  
If none of the above conditions are met, the coupon is not valid on the given date.

**🔹 Return Response**

* ✅ **Returns true** if the coupon is valid on the given date.
* ❌ **Returns false** if the coupon is not valid on the given date.

**14.3 get($coupon\_id) Method**

### 🔹 Overview

Fetches and returns the details of a specific coupon based on its ID.

**🔹 Key Functionalities**

1- **Coupon Retrieval**  
Searches the database for a coupon matching the provided coupon\_id.

2-**Not Found Handling**  
If no coupon is found, it returns a 404 error with a "coupon not found" message.

3-**Successful Fetch**  
If the coupon exists, its details are returned using a resource wrapper, formatted appropriately for the API response.

**🔹 Return Response**

* ✅ **Success**: Returns a structured response containing the coupon's details.
* ❌ **Failure**: Returns a 404 Not Found or 422 Unprocessable Entity with an appropriate error message.

**14.4 add(StoreCouponRequest $request) Method**

### 🔹 Overview

Creates and stores a new coupon in the system using validated request data.

**🔹 Key Functionalities**

1-**Coupon Creation**  
Initializes a new coupon and fills its properties using the validated request data (excluding car size associations).

2-**Car Size Association**  
If the coupon is successfully saved, it attaches related car size IDs via a many-to-many relationship.

3-**Logging**  
Logs relevant request data for debugging purposes.

**🔹 Return Response**

* ✅ **Success**: Returns the created coupon data wrapped in a resource along with a success message.
* ❌ **Failure**: Returns a 422 Unprocessable Entity error with the specific exception message.

**14.5 update(UpdateCouponRequest $request, $coupon\_id) Method**

### 🔹 Overview

Updates an existing coupon with the provided request data, including reattaching associated car sizes.

**🔹 Key Functionalities**

1-**Find Coupon**  
Searches for the coupon using the given ID. Returns an error response if not found.

2-**Update Attributes**  
Updates the coupon’s attributes with validated request data (excluding car size relationships).

3-**Sync Car Sizes**  
If the update succeeds, synchronizes the coupon's associated car sizes to reflect the current selection.

**🔹 Return Response**

* ✅ **Success**: Returns the updated coupon data wrapped in a resource and a success message.
* ❌ **Failure**: Returns a 422 Unprocessable Entity error with the specific exception message.

**14.6 destroy($coupon\_id) Method**

### 🔹 Overview

Deletes a coupon from the system by its ID after verifying its existence.

**🔹 Key Functionalities**

1-**Find Coupon**  
Searches for the coupon using the given ID.  
Returns an error response if the coupon is not found.

2-**Delete Coupon**  
If the coupon exists, it is deleted from the database.

**🔹 Return Response**

✅ **Success**:  
Returns a success message confirming that the coupon has been deleted.  
Message: \_\_('api.coupon\_delete')

❌ **Failure**:

* If the coupon is not found, returns a 404 error with the message: \_\_('api.coupon\_not\_found')
* If an exception occurs, returns a 422 error with the message: \_\_('api.error\_happened') . $error->getMessage()

1. **Employee Controller**

**15.1 all(Request $request) Method**

### 🔹 Overview

Retrieves a list of employees (users without the "admin" role) with optional pagination based on the request input.

**🔹 Key Functionalities**

1-**Filter Non-Admin Users**  
Uses whereDoesntHave to exclude users who have the "admin" role, ensuring only employees are fetched.

2-**Handle Pagination Logic**  
Checks the request for the paginate flag to determine whether to return paginated data or the full list:

* If paginate = 1, returns paginated data (10 per page).
* If paginate = 0, returns all employees without pagination.
* If not specified, returns paginated data (20 per page) by default.

3-**Prepare Pagination Metadata**  
When pagination is used, constructs a pagination array including:

* total records
* records per page
* current page
* total pages

**🔹 Return Response**

✅ **Success**:  
Returns a list of employees (wrapped in EmployeeResource) with or without pagination metadata based on the request.  
Message: \_\_('api.employee\_all')

❌ **Failure**:  
Returns a 422 Unprocessable Entity error with the message:  
\_\_('api.error\_happened') . $error->getMessage()

**15.2 get($id) Method**

### 🔹 Overview

Fetches the details of a specific employee (user) by ID, excluding users with the "admin" role (role\_id = 1).

**🔹 Key Functionalities**

1-**Filter by Role**  
Uses a query to find a user whose role\_id is not equal to 1 (i.e., not an admin).

2-**Find by ID**  
Attempts to retrieve the employee using the provided $id.

3-**Conditional Response**

* If the employee exists, wraps the result using EmployeeResource.
* If not found, returns a 404 error.

**🔹 Return Response**

✅ **Success**:  
Returns the employee's data wrapped in EmployeeResource.  
Message: \_\_('api.employee\_get')

❌ **Not Found**:  
Returns a 404 error if the employee does not exist or has role\_id = 1.  
Message: \_\_('api.employee\_not\_found')

❌ **Failure**:  
Returns a 422 Unprocessable Entity error with the message:  
\_\_('api.error\_happened') . $error->getMessage()

**15.3 add(StoreEmployeeRequest $request) Method**

### 🔹 Overview

Creates a new employee (user) by accepting validated request data, including handling image upload if provided.

**🔹 Key Functionalities**

1-**Image Upload (Optional)**  
Checks if an image file is uploaded. If present, it processes the image using a helper method (handleFile).

2-**Create Employee**  
Uses the User::create() method to store the employee's information, including:

* Name, email, phone, password, role, and status.
* Optionally stores the employee's image path if an image is uploaded.

3-**Return Response**  
Once the employee is created, it returns the employee’s data wrapped in EmployeeResource.

**🔹 Return Response**

✅ **Success**:  
Returns the newly created employee’s data wrapped in EmployeeResource.  
Message: \_\_('api.employee\_add')

❌ **Failure**:  
Returns a 422 Unprocessable Entity error with the exception message:  
\_\_('api.error\_happened') . $error->getMessage()

**15.4 update($id, Request $request) Method**

### 🔹 Overview

Updates the details of an existing employee, including handling optional image updates and password changes.

**🔹 Key Functionalities**

1-**Find Employee**  
Searches for the employee by their ID, excluding admins (role ID != 1). If the employee is not found, it returns a 404 error.

2-**Handle Image Upload (Optional)**  
If a new image is uploaded, it processes the image using a helper method (handleFile). If the employee already has an image, the old image file is deleted from the storage.

3-**Update Employee Data**  
Updates the employee’s attributes, including:

* Name, email, phone, role ID, image path, and status.
* If a password is provided, it updates the password as well.

4-**Return Response**  
Once the employee data is updated, it returns the updated employee’s data wrapped in EmployeeResource.

**🔹 Return Response**

✅ **Success**:  
Returns the updated employee’s data wrapped in EmployeeResource.  
Message: \_\_('api.employee\_add')

❌ **Failure**:  
Returns a 404 error if the employee is not found or a 422 Unprocessable Entity error with the exception message:  
\_\_('api.error\_happened') . $error->getMessage()

**15.5 destroy($employee\_id) Method**

### 🔹 Overview

Deletes an employee's record and their associated image, if available.

### 🔹 Key Functionalities

1-**Find Employee**  
Searches for the employee by their ID, excluding admins (role ID != 1). If the employee is not found, it returns a 404 error.

2- **Delete Image (Optional)**  
If the employee has an associated image, it deletes the image file from the storage.

3-**Delete Employee Record**  
Deletes the employee’s record from the database once the image (if any) is deleted.

### 🔹 Return Response

✅ **Success**:  
Returns a success message indicating that the employee was successfully deleted.  
Message: \_\_('api.employee\_delete')

❌ **Failure**:  
Returns a 404 error if the employee is not found or a 422 Unprocessable Entity error with the exception message:  
\_\_('api.error\_happened') . $error->getMessage()

1. **FinancialAdmin Controller**
   1. **all(Request $request) Method**

### 🔹 Overview

Retrieves all financial admins, with optional pagination based on the request.

**🔹 Key Functionalities**

1-**Check for Pagination**  
Determines whether the request asks for paginated data by checking the paginate parameter.

2-**Fetch Financial Admins with Pagination**

* If paginate is set to 1, it fetches the financial admins with pagination, showing a paginated result and returning pagination details.
* If paginate is set to 0, it fetches all financial admins without pagination.
* If paginate is not provided, it defaults to paginating with a page size of 20.

3-**Pagination Details**

* Includes the total count of records, items per page, the current page, and total number of pages in the response.

**🔹 Return Response**

✅ **Success**:  
Returns a list of financial admins in the response, along with pagination details if applicable.  
Message: \_\_('api.financial\_admin\_all')

❌ **Failure**:  
Returns a 422 Unprocessable Entity error with the exception message:  
\_\_('api.error\_happened') . $error->getMessage()

* 1. **get($id) Method**

**🔹 Overview**

Retrieves a specific financial admin by their ID.

**🔹 Key Functionalities**

1-**Find Financial Admin by ID**

* Searches for a financial admin using the given id and role ID 8.
* If the admin is found, it returns their details as part of the response.

2-**Error Handling**

* If no financial admin is found with the provided ID, an error response with a 404 status is returned.

**🔹 Return Response**

✅ **Success**:  
Returns the data for the financial admin in the response using the EmployeeResource.  
Message: \_\_('api.financial\_admin\_get')

❌ **Failure**:  
Returns a 404 error if the financial admin is not found with the specified ID.  
Message: \_\_('api.financial\_admin\_not\_found')  
OR  
Returns a 422 Unprocessable Entity error with the exception message:  
\_\_('api.error\_happened') . $error->getMessage()

* 1. **add(StoreAdminRequest $request) Method**

**🔹 Overview**

Creates a new financial admin (user with role\_id of 8) with the provided request data.

**🔹 Key Functionalities**

1-**Handle Image Upload**

* If an image file is provided, it processes the image using the handleFile method.

2-**Create New Financial Admin**

* Creates a new financial admin (role\_id = 8) with the provided name, email, phone, password, image, and status.

**🔹 Return Response**

✅ **Success**:  
Returns the newly created financial admin data wrapped in a resource.  
Message: \_\_('api.financial\_admin\_add')

❌ **Failure**:  
Returns a 422 Unprocessable Entity error with the exception message:  
\_\_('api.error\_happened') . $error->getMessage()

* 1. **update($id, UpdateAdminRequest $request)** **Method**

**🔹 Overview**

Updates the information of an existing financial admin (user with role\_id of 8) with the provided request data, including updating the admin's image.

**🔹 Key Functionalities**

1-**Find Admin**

* Searches for an existing financial admin by id and role\_id (should be 8). Returns an error if not found.

2-**Handle Image Upload**

* If a new image is provided, it replaces the old image after deleting it from storage.

3-**Update Admin Details**

* Updates the admin’s name, email, phone, role, image, and status with the provided values.
* If a new password is provided, it updates the password as well.

**🔹 Return Response**

✅ **Success**:  
Returns the updated financial admin data wrapped in a resource.  
Message: \_\_('api.financial\_admin\_update')

❌ **Failure**:

* Returns a 404 error if the admin is not found: \_\_('api.financial\_admin\_not\_found').
* Returns a 422 Unprocessable Entity error if an exception occurs: \_\_('api.error\_happened') . $error->getMessage()
  1. **destroy($admin\_id) Method**

**🔹 Overview**

Deletes a financial admin (user with role\_id of 8) by their id. If the admin has an associated image, it is deleted first.

**🔹 Key Functionalities**

1-**Find Admin**

* Searches for a financial admin by id and role\_id (should be 8). If not found, returns an error.

2-**Delete Image**

* If the admin has an image associated, it deletes the image from the server storage before deleting the admin record.

3-**Delete Admin**

* Deletes the admin from the database once the image has been handled.

**🔹 Return Response**

✅ **Success**:  
Returns a success message upon successful deletion of the admin:  
\_\_('api.financial\_admin\_delete')

❌ **Failure**:

* Returns a 404 error if the admin is not found: \_\_('api.financial\_admin\_not\_found').
* Returns a 422 Unprocessable Entity error if an exception occurs: \_\_('api.error\_happened') . $error->getMessage()

1. **Gift Controller**
   1. **all(Request $request) Method**

**🔹 Overview**

This method retrieves a list of "gifts" based on the request parameters. It handles optional filtering, pagination, and city-based access restrictions based on the user's role.

**🔹 Key Functionalities**

1-**Filter by Payment Status**

* If a status parameter is provided in the request, the query filters the gifts by the payment\_status.

2-**City-Based Filter (Role-Based)**

* For users with role\_id 7 or 17, the query further filters gifts based on the city\_id of the user. This ensures that the user only sees gifts related to their assigned city.

3-**Pagination**

* If the paginate parameter is present in the request:
  + paginate = 1: The response will include paginated results with 10 gifts per page.
  + paginate = 0: All gifts are returned without pagination.
  + Default behavior is paginated results with 10 gifts per page.

**🔹 Return Response**

* **Success**:  
  Returns a collection of gifts (paginated or not) in the response, along with pagination details if requested.
  + For paginated results, it includes the total number of gifts, the number of gifts per page, the current page, and the total number of pages.
  + For non-paginated results, it returns the list of gifts directly.
* **Failure**:
  + Returns a 422 Unprocessable Entity error if an exception occurs:  
    \_\_('api.error\_happened') . $error->getMessage()
  1. **search(Request $request) Method**

🔹 **Overview**  
Searches for gifts with a paid status (payment\_status = 1) based on a phone number associated with either the sender or receiver. Returns a paginated list of results.

🔹 **Key Functionalities**  
1-**Initial Gift Query**  
Filters gifts that have a payment\_status of 1, meaning only gifts marked as paid are considered in the search.

2-**Search by Phone Number**  
If the request contains a phone\_sender or phone\_receiver value, it searches for gifts where:

* The sender’s phone number matches the search query.
* OR the receiver’s phone number matches the search query.

3-**Pagination**  
The results are paginated, returning 10 gifts per page.

🔹 **Return Response**  
✅ **Success**:  
Returns a list of gifts matching the search criteria, along with pagination details.

❌ **Failure**:  
Returns a 422 Unprocessable Entity error if an exception occurs, with the message: \_\_('api.error\_happened') . $error->getMessage().

* 1. **use($gift\_id) Method**

🔹 **Overview**  
Toggles the status of a gift between "used" and "unused" based on the current status. If the gift is found, it updates the status accordingly.

🔹 **Key Functionalities**  
1-**Find Gift**  
Searches for a gift by its ID ($gift\_id). If not found, it returns an error response with a 404 status: \_\_('api.gift\_not\_found').

2-**Toggle Gift Status**

* If the gift's status is 0 (unused), it is updated to 1 (used).
* If the gift's status is already 1 (used), it is reverted back to 0 (unused).

3-**Return Response**  
✅ **Success**:  
Returns a success message indicating that the gift has been successfully used or toggled: \_\_('api.gift\_use').

❌ **Failure**:  
Returns a 404 error if the gift is not found: \_\_('api.gift\_not\_found').  
Returns a 422 Unprocessable Entity error if an exception occurs: \_\_('api.error\_happened') . $error->getMessage().

* 1. **store(Request $request) Method**

🔹 **Overview**  
This method processes the creation of a gift from one client to another. It validates the input, generates the necessary gift record, updates the recipient's balance, and sends an SMS notification to the recipient.

🔹 **Key Functionalities**  
1-**Input Validation**  
The method validates the request inputs for required fields like phone, total\_price, client\_id, reason, etc. It ensures the phone number is at least 10 digits and the client\_id exists.

2-**Phone Number Formatting**  
The phone number is cleaned and formatted to a valid 10-digit format if it's longer than 10 digits.

3-**City Retrieval**  
The method checks if a city\_id is provided. If not, it defaults to the city of the currently authenticated user.

4-**Check Client Existence**  
It checks if the recipient client exists in the database based on the phone number and client\_type. If the client is found, it processes the gift creation for that client, adding the total price to the recipient's balance.

5-**Gift Creation**  
A new gift record is created in the Gift table, with details about the sender, recipient, city, reason, and more. The status and payment\_status are set to 1 (indicating the gift is active and paid).

6-**Wallet Operation**  
If the application's setting requires wallet operations to be verified, a WalletOperation record is created for the sender, indicating the gift transaction.

7-**SMS Notification**  
An SMS message is constructed and sent to the recipient, notifying them of the gift. The message contains a link to the Hamimm app and details about the gift.

8-**Gift Code and Saving**  
If the recipient client is not found, the system generates a random code for the gift, saves it, and sends a similar SMS message to the provided phone number.

🔹 **Return Response**  
✅ **Success**:  
Returns a success message when the gift is successfully created and sent: \_\_('api.your\_gift\_sent\_successfully').

❌ **Failure**:  
If validation fails, returns a 403 error with validation errors. If an exception occurs, returns a 422 Unprocessable Entity error with the exception message: \_\_('api.error\_happened') . $error->getMessage().

1. **Home Controller**
   1. **index(Request $request) Method**

🔹 **Overview**  
This method is responsible for fetching various statistics and reservation details depending on the user's role. If the logged-in user is an investor or admin, the method gathers relevant data related to reservations, income, and other key metrics. The method handles both investor-specific and general statistics for admins.

🔹 **Key Functionalities**

1-**Check User Role**  
It first checks if the logged-in user is an investor (role\_id == 6). If the user is an investor, the method proceeds to fetch investor-specific data.

2-**Investor Data Retrieval**  
For an investor:

* It retrieves the investor's associated data from the Investors model based on the user's email and phone.
* It calculates various statistics related to reservations for that investor:
  + **Total Reservations**: The total number of reservations associated with the investor.
  + **Cancelled Reservations**: The number of cancelled reservations (status = 6).
  + **Completed Reservations**: The number of completed reservations (status = 9).
  + **Total Income**: The total income from completed reservations.
  + **Maintenance Costs**: The total maintenance costs associated with the investor.
  + **Obligations**: The total obligations of the investor.
  + **Net Income**: The total income after subtracting obligations from maintenance costs.

3-**General Admin Statistics**  
For admins or other users:

* It calculates general statistics like:
  + Number of admins, representatives, and clients.
  + Number of active clients.
  + Total reservations and finished reservations.
  + Lists of the latest admins, representatives, and clients.
  + A list of today's reservations with specific status codes (2 and 7).

4-**Return Data**  
Depending on the user's role:

* **Investor**: Returns statistics specific to the investor's reservations, income, and obligations.
* **Admin**: Returns general data about reservations, clients, admins, and other statistics.

5-**Error Handling**  
If any error occurs during data retrieval, it catches the exception and returns a 422 error with the exception message.

🔹 **Return Response**  
✅ **Success**:  
Returns a success message with the relevant statistics data in the data key.

❌ **Failure**:  
If an error occurs during the process, returns a 422 Unprocessable Entity error with the exception message.

* 1. **external\_dashboard(Request $request) Method**

🔹 **Overview**  
The external\_dashboard method is designed to provide a summary of various statistics, including the number of clients, representatives, reservations, and more. It's intended to gather and return information that can be used in an external dashboard. The data fetched in this method is primarily focused on active reservations and key user roles.

🔹 **Key Functionalities**

1-**Date-based Reservations Filter**

* The method begins by setting the current date using Carbon::now()->format('Y-m-d') and queries reservations where:
  + The reservation status is either 2 or 7 (which likely indicate active or pending statuses).
  + The reservation date is today's date.
* The query is ordered by the most recent reservation.

2-**Dashboard Data Retrieval**  
The method retrieves various statistics for the dashboard:

* **Number of Representatives**: The total number of Client records where type is 1 (representatives).
* **Number of Clients**: The total number of Client records where type is 0 (clients).
* **Total Reservations (excluding cancelled ones)**: The count of all reservations that do not have a status of 6 (likely cancelled reservations).
* **Latest Admins**: A list of the latest 5 users with the role 1 (admin), formatted using the EmployeeResource.
* **Latest Representatives**: A list of the latest 5 representatives (clients with type 1), formatted using the RepresentiveResource.
* **Latest Clients**: A list of the latest 5 clients (clients with type 0), formatted using the ClientResource.
* **Today's Reservations**: A list of reservations for today with specific statuses, formatted using the ReservationResource.

3-**Return Data**  
The method constructs an array $data containing:

* Key statistics related to clients, representatives, and reservations.
* A list of the latest 5 admins, representatives, and clients.
* Today's reservations.

4-**Error Handling**  
If any error occurs during data retrieval or processing, it catches the exception and returns a 422 error with the exception message.

5-**Return Response**  
✅ **Success**:  
Returns a JSON response containing the key statistics and reservation details.

❌ **Failure**:  
If an error occurs, returns a 422 Unprocessable Entity error with the exception message.

1. **Location Controller**
   1. **get(Request $request) Method**

🔹 **Overview**  
The get method is designed to retrieve a list of Locations, applying filters based on the request data and the user's role. It also supports pagination if specified, returning a collection of locations with or without pagination depending on the request parameters.

🔹 **Key Functionalities**

1-**Location Query Initialization**

* The method begins by initializing a query for the Locations model.
* It ensures that only locations where the outside field is 0 are included.

2-**Filtering by Client ID**

* If a client\_id is provided in the request, the query is filtered by user\_id to only include locations associated with that specific client.

3-**Role-based Filtering**

* If the logged-in user has a role ID of 7 or 17, the query is further filtered by area\_id to only include locations that match the user's area.

4-**Pagination Logic**

* If the request contains a paginate parameter set to 1, pagination is applied:
  + The query results are paginated with a limit of 10 locations per page.
  + The response includes pagination details such as:
    - total: Total number of locations.
    - per\_page: Number of locations per page.
    - current\_page: Current page number.
    - total\_pages: Total number of pages.

5-**Collection Formatting**

* Regardless of pagination, the results are wrapped in a LocationResource collection, which transforms the raw location data into a resource format (likely formatted for the API).

6-**Return Data**  
The method returns a JSON response with the following structure:

* **With Pagination**: If pagination is applied, it returns the list of locations along with pagination details.
* **Without Pagination**: If pagination is not requested, it returns all locations matching the query.

7-**Error Handling**

* Any exception thrown during the process is caught, and a 403 error is returned with the exception's message.
  1. **add(Request $request) Method**

🔹 **Overview**  
The add method is responsible for adding a new location to the system. It handles validation, checks if the location exists in a specific area, and then stores the location either as an external location or within the app's areas based on the check.

🔹 **Key Functionalities**

1-**Validation**

* The method starts by defining the validation rules for the incoming request:
  + type (required)
  + name (required, string, max 255 characters)
  + lat (required, numeric, minimum value of 0)
  + long (required, numeric, minimum value of 0)
  + address (optional, string, max 255 characters)
* A validator checks the incoming request data, and if any rule fails, it returns a 403 response with the first error message.

2-**Location Existence Check**

* The method then checks if the location exists in the areas of the app or not using a custom method checkLocationExist. This method likely checks if the coordinates (latitude and longitude) exist within the predefined area of the app.

3-**Location Area Check**

* Another custom method, checkLocation, is called to determine the area in which the location should be categorized. This probably returns the appropriate area for valid locations.

4-**Handling External Locations**

* If the location is not found in the predefined app areas ($result == false), the method creates a new location marked as "outside" by setting outside to 1 and stores it with the provided information.
  + The system returns an error message saying that the location is outside the app's service area: "العنوان غير موجود ضمن مناطق التطبيق" (The address is not within the app's areas).

5-**Storing Valid Locations in App Areas**

* If the location is valid (i.e., found within the areas), it creates a new location record without the outside flag, linking it to an area (area\_id) based on the result of the checkLocation method.

6-**Return Success**

* If the location is successfully added, the method returns a success response with a message indicating that the location was added.

7-**Error Handling**

* Any exceptions encountered during the process are caught and returned as a 500 error with the exception's message.
  1. **update(Request $request) Method**

🔹 **Overview**  
The update method is responsible for updating an existing location in the system. It performs validation, checks if the location is valid within the app's areas, and updates the location accordingly. If the location is outside the app's areas, it marks it as an external location.

🔹 **Key Functionalities**

1-**Validation**

* The method starts by defining validation rules for the incoming request:
  + type, name, lat, long, and address are required.
  + location\_id is required and must correspond to an existing record in the locations table.
* A validator checks the incoming request data, and if any rule fails, it returns a 403 response with the first error message.

2-**Find the Location**

* The location to be updated is retrieved using Locations::find($request->location\_id). This finds the location by its id, which was provided in the request.

3-**Location Existence Check**

* The method then checks if the new location (based on latitude and longitude) is within the predefined app areas using the checkLocationExist method.
* It also checks the area using the checkLocation method, which likely returns the area based on the coordinates.

4-**Handling External Locations**

* If the location is not found within the app's areas ($result == false), it updates the existing location to set outside to 1, marking it as an external location. It also updates the other fields with the provided data.
  + The system returns an error message saying that the location is outside the app's service area: "العنوان غير موجود ضمن مناطق التطبيق" (The address is not within the app's areas).

5-**Updating Valid Locations in App Areas**

* If the location is valid (i.e., found within the app's areas), the method updates the location with the provided data, including the area ID (area\_id).

6-**Return Success**

* If the location is successfully updated, the method returns a success response with a message indicating that the location was updated.

7-**Error Handling**

* Any exceptions encountered during the process are caught and returned as a 500 error with the exception's message.
  1. **delete(Request $request) Method**

🔹 **Overview**  
The delete method is responsible for deleting a location from the system. It validates the request, retrieves the location by its ID, and deletes it. If any errors occur, it catches them and returns an error message.

🔹 **Key Functionalities**

1-**Validation**

* The method starts by defining validation rules for the incoming request:
  + location\_id is required and must exist in the locations table.
* A validator checks the incoming request data. If any rule fails (i.e., the location\_id does not exist or is missing), it returns a 403 response with the first validation error message.

2-**Find and Delete Location**

* If validation passes, the method retrieves the location by its id using Locations::find($request->location\_id). If the location exists, the method proceeds to delete it.
* The delete() method is called on the retrieved location to remove it from the database.

3-**Return Success**

* If the location is successfully deleted, the method returns a success response with a message indicating that the location was deleted (\_\_('api.deleteLocation')).

4-**Error Handling**

* If an exception is thrown during the process (such as an issue with database interaction), the method catches the error and returns a 500 error with the exception's message.
  1. **addressTypes() Method**

🔹 **Overview**  
The addressTypes() method is responsible for returning a predefined list of address types in a JSON format. It includes categories such as "home", "work", and "other". It also handles any exceptions that might occur during the process.

🔹 **Key Functionalities**

1-**Define Address Types**

* The method defines an array $addressTypes, which holds a list of predefined address types. Each address type is represented by an associative array with an id and name.
  + id is the unique identifier for each address type.
  + name is a translatable string (using the \_\_('api.home'), \_\_('api.work'), and \_\_('api.other') functions) representing the name of the address type.

2-**Return Address Types**

* After defining the address types, the method returns the data in a JSON format. The addressTypes key is used to encapsulate the list of address types in the response.
* A 200 status code is returned, indicating a successful request.

3-**Error Handling**

* If an error occurs (e.g., a failure during data handling), the method catches the exception and returns a 403 response with the exception message.
  1. **checkLocationExist() Method**

🔹 **Overview**  
The checkLocationExist() method is responsible for checking if a given latitude and longitude (coordinates) are inside any defined geographic area in your system. This is done by looping through areas, creating polygons from the points of each area, and checking if the given coordinates lie inside any of those polygons.

🔹 **Key Functionalities**

1-**Initialize Variables**

* The method starts with a flag $exist set to false, which will be used to track if the coordinates are found inside any area.

2-**Retrieve Areas**

* The method retrieves all Area records from the database using Area::all(). Each area presumably contains points representing its boundaries.

3-**Loop Through Areas**

* For each area, the points attribute (likely stored in a JSON format) is decoded using json\_decode(). These points represent the boundary of the area.

4-**Create Polygon**

* A new Polygon object is created for each area.
* The method then loops through the decoded points and adds each point as a vertex of the polygon using addPoint() with a new Coordinate object for each point. Each Coordinate holds a latitude and longitude value.

5-**Check if Coordinates Are Inside Polygon**

* The method checks if the given coordinates ($lat, $lng) lie inside the current polygon using the contains() method. This method likely checks if the point lies inside the area defined by the polygon's vertices.

6-**Set $exist Flag**

* If the given coordinates are found inside any of the polygons, the $exist flag is set to true, and the loop breaks (since we don't need to check further once a match is found).

7-**Return Result**

* The method returns the $exist flag (either true or false) based on whether the given coordinates were found inside any area.

8-**Error Handling**

* If any error occurs during the process (e.g., issues with decoding the points, creating polygons, or checking containment), the method catches the exception and returns a 403 error with the exception message.
  1. **outside\_locations() Method**

🔹 **Overview**  
The outside\_locations() method is responsible for fetching locations that are marked as "outside" (i.e., where the outside field is set to 1). It retrieves the latest location entry per user and can optionally paginate the results based on the paginate query parameter.

🔹 **Key Functionalities**

1-**Query Setup**

* The method starts by defining a query that:
  + Filters locations where the outside field is set to 1 (indicating that these locations are outside the defined areas).
  + Groups the locations by user\_id, ensuring that only one location per user is selected.
  + Selects the maximum (MAX) id to ensure that only the latest location for each user is fetched.
  + Orders the results by id in descending order (orderByDesc('id')), ensuring that the latest records come first.

2-**Fetch Locations**

* The query is executed using get() to fetch the grouped locations, containing the latest location id for each user.

3-**Handle Pagination**

* If the paginate query parameter is provided and equals 1, the method:
  + Uses the pluck('id') method to get the list of ids from the previously fetched results.
  + Fetches the full location details for these ids using whereIn('id', $locations->pluck('id')) and orders them by the latest entries (latest()).
  + The locations are then paginated with paginate(10), returning 10 results per page.
  + A pagination array is created to return pagination metadata (total results, current page, total pages, etc.).

4-**Return Paginated Locations**

* If pagination is enabled, the method returns the paginated locations wrapped inside a LocationResource collection along with the pagination metadata (total, per\_page, current\_page, total\_pages).

5-**Return All Locations (No Pagination)**

* If pagination is not requested:
  + The method fetches all the full location details for the selected ids using whereIn('id', $locations->pluck('id')) and orders them by the latest entries (latest()).
  + It returns these locations wrapped inside a LocationResource collection.

6-**Error Handling**

* If any error occurs (e.g., during the database query or data processing), the method catches the exception and returns a 422 error with the exception message.

1. **Maintenance Controller**
   1. **all() Method**

🔹 **Overview**  
The all() method retrieves maintenance records, with different behavior based on the role of the authenticated user (either investor or admin). The method also supports pagination, allowing the user to specify whether to receive a paginated or full list of results.

🔹 **Key Functionalities**

1-**User Role Handling**

* The method first checks the role of the authenticated user ($user->role\_id):
  + **Investor** (role\_id == 6):
    - It fetches an Investor record based on the authenticated user's email and phone number, ensuring the investor's status is active (status == 1).
    - It then retrieves the Maintenance records related to that investor by querying the plate relationship (filtering by investor\_id).
  + **Admin** or Other Roles:
    - It fetches all Maintenance records, regardless of the investor\_id.

2-**Maintenance Query Setup**

* Whether the user is an investor or an admin, the Maintenance query is configured to include the plate relationship and orders the results by the latest (latest()).

3-**Pagination Handling**

* The method checks for the paginate query parameter:
  + If paginate == 1: It paginates the results (10 records per page) and includes pagination metadata (total results, current page, total pages, etc.).
  + If paginate == 0: It retrieves all Maintenance records without pagination.
  + If the paginate parameter is not set, it defaults to paginating the results (10 records per page).

4-**Returning Data**

* The method formats the retrieved Maintenance records into a collection using MaintenanceResource::collection($maintenances).
* Depending on the pagination setting, it either returns the paginated results along with pagination metadata or the full list of records without pagination.
* The data is returned as part of a response in the format { "data": { "maintenances": ..., "pagination": ... }, "message": "maintenance\_all" }.

5-**Error Handling**

* Any errors encountered during the process (e.g., database query issues) are caught and a 422 error is returned with the exception message.
  1. **get($id) Method**

🔹 **Overview**  
The get($id) method retrieves a specific Maintenance record by its id and returns the details as part of a structured response.

🔹 **Key Functionalities**

1-**Retrieve Specific Maintenance Record**

* The method uses the find() method to fetch the Maintenance record with the specified id.
  + If a record with the given id exists, it retrieves the associated Maintenance record.

2-**Return Data**

* The retrieved Maintenance record is transformed into a resource using the MaintenanceResource class.
* The transformed Maintenance resource is included in the response under the "maintenance" key.

3-**Response Format**

* The method responds with a success message and includes the maintenance details in the data field of the JSON response.
* \_\_('api.maintenance\_get') is likely a translation key for the success message (could be something like "Maintenance details retrieved successfully").

4-**Error Handling**

* If any error occurs during the execution (e.g., if the maintenance record is not found, or an unexpected error happens), the exception is caught and a 422 error is returned with the exception message.
  1. **add() Method**

🔹 **Overview**  
The add() method is responsible for creating a new maintenance record, handling image uploads, and saving all relevant data. It ensures that all required fields are collected from the request, and handles image storage if provided.

🔹 **Key Functionalities**

1-**Image Upload Handling**

* The method first checks if the request includes a file under the key image using $request->file('image').
* If the image is present, it is stored in the uploads/maintenances directory under the public disk. The file path is saved for later reference in the database.

2-**Creating the Maintenance Record**

* After handling the image upload (if applicable), a new maintenance record is created with the following attributes:
  + **plate\_id**: ID of the plate associated with the maintenance (from the request).
  + **total**: The total cost or value associated with the maintenance.
  + **address**: The address where the maintenance is being carried out.
  + **user\_id**: The ID of the authenticated user creating the maintenance record (via $request->user()->id).
  + **image**: The stored path of the image (if uploaded).
* The Maintenance::create() method is used to insert the new record into the database.

3-**Returning Data**

* After successfully creating the maintenance record, the method uses a MaintenanceResource to format the newly created maintenance record.
* A success response is returned containing the formatted maintenance data, with a message indicating the successful creation of the maintenance record (maintenance\_add).

4-**Error Handling**

* Any errors that occur during the process (such as file upload issues or database errors) are caught in the catch block.
* If an error occurs, a 422 error response is returned with the exception's message, indicating what went wrong during the operation.
  1. **update() Method**

🔹 **Overview**  
The update() method is responsible for updating an existing maintenance record. It handles updating the attributes of the maintenance record, including handling an image upload if a new image is provided. It ensures that the existing image is deleted when replaced.

🔹 **Key Functionalities**

1-**Finding the Maintenance Record**

* The method first attempts to find the maintenance record by ID (Maintenance::find($id)).
* If the maintenance record is not found, it returns a 404 error with the message 'maintenance\_not\_found'.

2-**Image Handling**

* The method checks if a new image is provided in the request ($request->file('image')).
* If a new image is provided:
  + It stores the new image in the uploads/maintenances directory under the public disk.
  + If the maintenance record already has an existing image, the old image is deleted using Storage::disk('public')->delete($maintenance->image) to free up storage space.
* If no new image is provided, it retains the existing image path ($maintenance->image).

3-**Updating the Maintenance Record**

* The method then updates the maintenance record with the new data:
  + **plate\_id**: The plate ID is updated if provided in the request, or it retains the current value.
  + **total**: The total value is updated if provided in the request, or it retains the current value.
  + **address**: The address is updated if provided in the request, or it retains the current value.
  + **image**: The image path is updated with the new image if uploaded, or it retains the current image path.
* The update() method is called to persist the changes in the database.

4-**Returning Data**

* After the maintenance record is successfully updated, the method returns a success response containing the updated maintenance record, formatted using MaintenanceResource::collection($maintenance).
* The success message maintenance\_update is included in the response.

5-**Error Handling**

* Any errors that occur during the process (such as record not found or file upload issues) are caught in the catch block.
* If an error occurs, a 422 error response is returned with the exception's message.
  1. **destroy() Method**

🔹 **Overview**  
The destroy() method is responsible for deleting a maintenance record along with its associated image from the storage, if it exists. The method ensures that any related media files are cleaned up before the record is permanently deleted.

🔹 **Key Functionalities**

1-**Finding the Maintenance Record**

* The method begins by attempting to find the maintenance record using its ID (Maintenance::find($id)).
* If the record is not found, the method proceeds with the delete operation or handles any issues (though it’s assumed that it exists due to the try-catch structure).

2-**Image Deletion**

* If the maintenance record has an associated image ($maintenance->image), the method deletes it from the storage using:
  + Storage::disk('public')->delete($maintenance->image).
* This step ensures that the image file is removed from the disk and doesn't occupy unnecessary space.

3-**Deleting the Maintenance Record**

* The method then deletes the maintenance record itself using the delete() method on the $maintenance instance.

4-**Returning Data**

* After successfully deleting the maintenance record, a success response is returned with the message maintenance\_destroy.
* This indicates that the record was deleted successfully.

5-**Error Handling**

* Any errors that occur during the process (such as the maintenance record not being found or an issue with file deletion) are caught in the catch block.
* If an error occurs, a 422 error response is returned, along with the exception message.

1. **Manager Controller**
   1. **all() Method**

🔹 **Overview**  
The all() method retrieves a list of managers (employees), with support for pagination. The method checks the request for a paginate parameter, determining whether to return paginated data or all records. It returns the data along with pagination information if needed.

🔹 **Key Functionalities**

1-**Pagination Handling**

* The method first checks if the paginate parameter is present in the request:
  + If paginate == 1: It retrieves a paginated list of managers, with 10 records per page.
  + If paginate == 0: It retrieves all managers without pagination.
  + If paginate is not specified, it defaults to paginating the results with 20 records per page.

2-**Query Setup**

* The method uses Manager::latest() to fetch the managers in descending order based on the creation time.
* Depending on the value of paginate, it either fetches the records with pagination (paginate(10) or paginate(20)) or retrieves all the records (get()).

3-**Pagination Metadata**

* If pagination is applied, the method calculates the pagination metadata:
  + total: The total number of records.
  + per\_page: The number of records per page.
  + current\_page: The current page number.
  + total\_pages: The total number of pages.

4-**Returning Data**

* After fetching the manager records, the method formats them using ManagerResource::collection($employees) to ensure that the data is returned in the correct structure.
* The response includes the managers data along with the pagination metadata (if pagination is applied), in the format { "data": { "managers": ..., "pagination": ... }, "message": "manager\_all" }.

5-**Error Handling**

* Any errors during the process (e.g., database issues) are caught by the catch block.
* If an error occurs, a 422 error response is returned, with a message indicating the error that occurred.
  1. **get() Method**

🔹 **Overview**  
The get() method retrieves a specific manager by their ID. If the manager exists, the method returns their data in the desired format. If the manager is not found, it returns a 404 error response.

🔹 **Key Functionalities**

1-**Finding the Manager**

* The method attempts to find the manager using Manager::find($id) based on the provided ID.
* If the manager exists, it returns the manager's data; otherwise, it returns a 404 Not Found error.

2-**Manager Data Retrieval**

* If the manager is found, the method formats the manager data using the ManagerResource resource class to ensure the response is structured correctly.
* The data is then returned as part of the response in the format { "data": { "manager": ... }, "message": "manager\_get" }.

3-**Error Handling**

* If the manager is not found, a 404 error is returned with the message 'manager\_not\_found'.
* If any other error occurs (e.g., database issues), the error is caught, and a 422 error is returned with a generic message and the error details.
  1. **add() Method**

🔹 **Overview**  
The add() method is used to create a new manager record in the system. It accepts input data, validates it using the StoreManagerRequest, and creates the manager record in the database. If successful, it returns the created manager's details; otherwise, it returns an error.

🔹 **Key Functionalities**

1-**Manager Creation**

* The method creates a new Manager record using the data passed from the request (name and phone).
* It uses the Manager::create() method to insert the new manager record into the database.

2-**Return Success Data**

* If the creation is successful, the newly created manager's details are returned in a structured format using the ManagerResource resource class.
* The data is returned as part of the response in the format: { "data": { "manager": ... }, "message": "manager\_add" }.

3-**Error Handling**

* If any errors occur during the process (e.g., validation issues or database problems), they are caught by the catch block.
* The error is returned with a 422 status code and a message stating 'api.error\_happened', along with the exception message to provide more details about the issue.
  1. **update() Method**

🔹 **Overview**  
The update() method allows the update of an existing manager's data. It accepts the id of the manager to be updated and the new data in the request. If the manager exists, it updates the provided fields; otherwise, it returns an error message.

🔹 **Key Functionalities**

1-**Find Manager**

* The method first attempts to find the manager by its id using Manager::find($id).
* If the manager exists, the update process proceeds; otherwise, an error message is returned indicating that the manager was not found.

2-**Manager Data Update**

* The manager's data is updated using the update() method. The updated fields (name and phone) are provided from the request.
* If the request doesn't provide a value for a specific field (e.g., name or phone), the existing value is retained using the null coalescing operator (??).

3-**Return Updated Data**

* If the update is successful, the method returns the updated manager's data, formatted using the ManagerResource resource class.
* The data is returned as part of the response in the format: { "data": { "manager": ... }, "message": "manager\_add" }.

4-**Error Handling**

* If the manager cannot be found or any error occurs during the update process, an error message is returned.
* If the manager is not found, a 404 error is returned with the message 'api.manager\_not\_found'.
* If any other error occurs during the update process, a 422 error is returned, and the exception message is included in the response.
  1. **destroy() Method**

🔹 **Overview**  
The destroy() method is used to delete a manager record from the database. It checks if the manager exists by their manager\_id, and if found, deletes the manager. If not found, it returns an error message. The method also handles any potential exceptions that may occur during the deletion process.

🔹 **Key Functionalities**

1-**Find Manager**

* The method first attempts to find the manager by its manager\_id using Manager::find($manager\_id).
* If the manager is found, it proceeds to delete the record. Otherwise, it returns an error indicating the manager wasn't found.

2-**Delete Manager**

* If the manager is found, the delete() method is called on the manager instance to remove the record from the database.

3-**Return Success Response**

* After successfully deleting the manager, the method returns a success response with a message indicating that the manager has been deleted. The message is retrieved from the \_\_('api.manager\_delete') translation string.

4-**Error Handling**

* If the manager does not exist, a 404 error is returned with the message 'api.manager\_not\_found'.
* If any error occurs during the deletion process, a 422 error is returned with the exception message.

1. **Material Controller**
   1. **all() Method**

🔹 **Overview**  
The all() method retrieves all material records from the database. It supports pagination based on the paginate query parameter, allowing the user to specify whether to retrieve the results in a paginated format or as a full list. The materials are ordered by count in descending order.

🔹 **Key Functionalities**

1-**Material Query Setup**

* The method begins by initializing a query for the Materials model using Materials::query().

2-**Pagination Handling**

* The method checks the paginate query parameter:
  + If paginate == 1: It retrieves a paginated list of materials, ordering them by count in descending order. The results are limited to 10 per page. Additionally, pagination metadata (total, per page, current page, and total pages) is included.
  + If paginate == 0: It retrieves all materials without pagination but still orders them by count in descending order.
  + If the paginate parameter is not provided, it defaults to returning a paginated list of materials.

3-**Returning Data**

* The MaterialResource::collection($materials) is used to format the material records into a resource collection before returning them in the response.
* Depending on whether pagination is applied, the response contains either the full list of materials or the paginated results along with the pagination metadata.

4-**Error Handling**

* If any error occurs during the process (e.g., database query issues), a 422 error is returned with the exception message.
  1. **additionServices() Method**

🔹 **Overview**  
The additionServices() method retrieves a list of services that have a type of 1 and a status of 1. The services are returned in a resource format using the ServiceResource. This method is designed to fetch active services of a particular type.

🔹 **Key Functionalities**

1-**Service Query Setup**

* The method initiates a query to retrieve services from the Service model where:
  + type is equal to 1
  + status is equal to 1
* This filters out services that are not of the required type or are inactive.

2-**Returning Data**

* The services are returned as a collection of ServiceResource instances using ServiceResource::collection($services). This resource format helps to structure the data before sending it back to the client.
* The response includes the services under the data key, with the message indicating successful data retrieval.

3-**Error Handling**

* Any errors encountered during the query execution (e.g., database query issues) are caught, and a 422 error is returned with the exception message.
  1. **add() Method**

🔹 **Overview**  
The add() method handles the creation of a new material. It performs validation on the incoming request data and ensures that all required fields meet the specified criteria. Upon successful validation, it creates a new record in the Materials table. In case of any errors, an appropriate error message is returned.

🔹 **Key Functionalities**

1-**Validation**

* The method defines validation rules for the incoming request:
  + count must be a required numeric integer.
  + service\_id must be required and should exist in the services table.
  + weight\_id must be required and should exist in the weights table.
* The Validator::make() method is used to validate the request data against these rules.
* If validation fails, the method returns a 422 status with the first validation error message.

2-**Creating Material**

* If validation passes, the method proceeds to create a new material entry in the Materials table using the validated request data ($request->all()).
* The newly created material is wrapped in a MaterialResource for proper data formatting before being returned.

3-**Returning Data**

* The method returns the created material as part of the data key, formatted through the MaterialResource class.
* The response message indicates the successful addition of the material.

4-**Error Handling**

* If any exceptions occur during the process (e.g., database errors), the method catches them and returns a 422 error with the exception message.
  1. **update() Method**

🔹 **Overview**  
The update() method handles the process of updating an existing material. It first validates the incoming request data to ensure all required fields are provided and valid. If validation passes, it retrieves the existing material record by its ID and updates it with the provided data. Any errors encountered during the process are handled appropriately.

🔹 **Key Functionalities**

1-**Validation**

* The method defines validation rules for the incoming request:
  + material\_id must be required and must exist in the materials table.
  + count must be a required numeric integer.
  + service\_id must be required and should exist in the services table.
  + weight\_id must be required and should exist in the weights table.
* The Validator::make() method validates the request data against these rules.
* If validation fails, the method returns a 422 status with the first validation error message.

2-**Finding and Updating Material**

* If validation passes, the method retrieves the existing material using Materials::find($request->material\_id).
* The material is then updated using $material->update($request->all()), applying the new data from the request.

3-**Returning Data**

* After successfully updating the material, the method returns the updated material wrapped in a MaterialResource.
* A success message ('تم تحديث الماده بنجاح') is returned along with the updated data.

4-**Error Handling**

* If any exceptions occur during the process (e.g., database errors), the method catches them and returns a 422 error with the exception message.
  1. **destroy() Method**

🔹 **Overview**  
The destroy() method is responsible for deleting a material from the database. It validates the incoming request to ensure the correct material ID is provided. After validation, it deletes the material if it exists in the database. The method handles errors gracefully, returning appropriate success or failure responses.

🔹 **Key Functionalities**

1-**Validation**

* The method defines validation rules for the incoming request:
  + material\_id is required and must exist in the materials table.
* The Validator::make() method is used to validate the request data against these rules.
* If validation fails, the method returns a 422 status with the first validation error message.

2-**Deleting Material**

* If validation passes, the method retrieves the material using Materials::find($request->material\_id).
* The material is then deleted using $material->delete().

3-**Returning Success**

* After successfully deleting the material, the method returns a success message ('تم حذف الماده بنجاح') with a 200 status.

4-**Error Handling**

* If any exceptions occur during the process (e.g., if the material cannot be found or deleted), the method catches them and returns a 422 error with the exception message.
  1. **materialRequests() Method**

🔹 **Overview**  
The materialRequests() method retrieves a list of material requests, optionally paginated, based on the authenticated user's role. If the user is an admin (or any other role), all requests are shown. If the user is a regular user (role ID 7), only the requests made by that user are returned. The method also handles pagination based on the request.

🔹 **Key Functionalities**

1-**Role-Based Data Retrieval**

* The method checks the role of the authenticated user ($request->user()->role\_id):
  + If the user's role is 7 (likely a regular user), it filters the requests to return only those made by the authenticated user ($request->user()->id).
  + If the user is an admin (or any role other than 7), it retrieves all material requests.

2-**Pagination Handling**

* The method checks if the paginate query parameter is set:
  + If paginate == 1, the results are paginated, showing 10 records per page. Pagination metadata (total, per\_page, current\_page, total\_pages) is included in the response.
  + If paginate == 0, it retrieves all the material requests without pagination.

3-**Returning Data**

* The method returns the data in a consistent format:
  + If paginated, it includes pagination metadata and the paginated list of material requests (MaterialRequestResource::collection($materialRequests)).
  + If no pagination is requested, it returns the full list of material requests.
* The data is returned with the message 'تم استرجاع الداتا' (Data retrieval successful).

4-**Error Handling**

* If any exceptions occur (e.g., during the database query), the method catches them and returns a 403 error with the exception message.
  1. **accept() Method**

🔹 **Overview**  
The accept() method is designed to accept a material request by updating its status in the database. It validates the incoming request, updates the material request's status, and dispatches a notification to inform the relevant party about the acceptance.

🔹 **Key Functionalities**

1-**Request Validation**

* The method validates the incoming request using the Validator:
  + It checks that the material\_request\_id exists in the material\_requests table.
  + If validation fails, it returns a 422 response with the error message from the validation.

2-**Update Material Request Status**

* The method retrieves the MaterialRequest instance using the material\_request\_id passed in the request.
* It updates the status field of the corresponding material request to 1, indicating that the request has been accepted.

3-**Notification Dispatching**

* The method uses SendNotification::dispatch() to send a notification about the acceptance of the material request. This notification is delayed by 2 seconds.
* The notification includes the message 'تم قبول طلب المواد بنجاح' (Material request has been successfully accepted), and is sent to the representative associated with the material request ($material->represetive\_id).

4-**Response**

* If the operation is successful, it returns a 200 status code with the message 'تم قبول الطلب بنجاح' (Request accepted successfully).

5-**Error Handling**

* If any exception occurs during the process, it catches the exception and returns a 403 error with the exception message.
  1. **reject() Method**

🔹 **Overview**  
The reject() method handles rejecting a material request by updating its status in the database and restoring the material count. It validates the request, updates the material request's status, adjusts the material count, and sends a notification about the rejection.

🔹 **Key Functionalities**

1-**Request Validation**

* The method starts by validating the incoming request using the Validator:
  + It checks if the material\_request\_id exists in the material\_requests table.
  + If validation fails, it returns a 422 response with the first error message.

2-**Update Material Request Status**

* The method retrieves the MaterialRequest instance using the material\_request\_id from the request.
* It then updates the status field of the material request to 2, indicating that the request has been rejected.

3-**Adjust Material Count**

* The method retrieves the associated material using the material\_id from the MaterialRequest instance.
* It then updates the count of the material by adding the quantity from the rejected request back to the material's count.

4-**Notification Dispatching**

* After the rejection, the method dispatches a notification via SendNotification::dispatch() to notify the representative about the rejection of the material request.
* The notification includes the message 'تم رفض طلب المواد بنجاح' (Material request has been rejected successfully) and is sent to the representative associated with the material request ($material->represetive\_id).

5-**Response**

* If the operation is successful, the method returns a success response (200) with the message 'تم رفض الطلب بنجاح' (Request rejected successfully).

6-**Error Handling**

* If any exception occurs during the process, it is caught and returns a 403 error with the exception message.

1. **Neighborhood Controller**
   1. **all() Method**

🔹 **Overview**  
The all() method retrieves a list of neighborhoods and can return either paginated data or all records depending on the request parameters. It also handles error cases gracefully and returns structured responses.

🔹 **Key Functionalities**

1-**Check Pagination Requirement**  
The method first checks if the paginate parameter is passed in the request. It checks:

* If paginate == 1: It fetches the neighborhoods with pagination.
* If paginate == 0: It fetches all neighborhoods without pagination.
* If paginate is not provided: It defaults to paginating the neighborhoods.

2-**Pagination Handling**

* When pagination is needed, the Neighborhood::latest()->paginate(10) method is used to fetch 10 neighborhoods per page in descending order of creation.
* It constructs a pagination structure ($pagination) containing:
  + total: Total number of neighborhoods.
  + per\_page: Number of neighborhoods per page.
  + current\_page: Current page number.
  + total\_pages: Total number of pages.

3-**Return Paginated Data**  
If pagination is requested, the method:

* Returns the paginated neighborhoods along with the pagination details.

4-**Return All Data**  
If pagination is not requested (paginate == 0), the method:

* Returns all neighborhoods in the latest order without pagination.

5-**Default Pagination**  
If the paginate parameter is not passed, the method defaults to paginated data with 10 neighborhoods per page.

6-**Error Handling**

* If an error occurs during the process (such as a database issue), the catch block catches the exception and returns a 422 response with the error message.

7-**Response Format**

* If the operation succeeds, it returns a success response with the neighborhoods data and a pagination structure if paginated.
* The response is structured using the NeighborhoodResource::collection() to transform the Neighborhood models into a more structured format.
  1. **get() Method**

🔹 **Overview**  
The get() method is designed to retrieve a single neighborhood based on the provided neighborhood\_id. It also handles error cases such as when the neighborhood is not found and returns a structured response.

🔹 **Key Functionalities**

1-**Retrieve Neighborhood by ID**  
The method uses Neighborhood::find($neighborhood\_id) to fetch a neighborhood record by its unique neighborhood\_id. This method will return null if no neighborhood is found with that ID.

2-**Check if Neighborhood Exists**  
After attempting to retrieve the neighborhood, it checks:

* If the neighborhood is found ($neighborhood is not null), it proceeds to return the neighborhood data.
* If the neighborhood is not found, it returns an error response with a 404 status and a message indicating that the neighborhood wasn't found.

3-**Return Data**  
If the neighborhood exists, the method returns:

* A success response with the neighborhood data.
* The data is wrapped in a NeighborhoodResource for structured transformation.
* The response message is localized, using the key 'api.neighborhood\_get' for the success message.

4-**Error Handling**  
If an exception occurs during the process (e.g., database issues, or any unforeseen errors), the method:

* Catches the exception (\Throwable $error) and returns a 422 error with the exception message.

5-**Response Format**

* On success: Returns a structured response with the neighborhood data.
* On failure (e.g., if the neighborhood is not found): Returns an error message indicating that the neighborhood was not found.
* The responses are structured in a uniform format with keys like "data" and error codes.
  1. add() Method

🔹 **Overview**  
This add() method is responsible for creating a new neighborhood. It uses a custom request class StoreNeighborhoodRequest to validate incoming data before inserting it into the database.

🔹 **Key Functionalities**

1-**Validation with Form Request**  
The method leverages StoreNeighborhoodRequest, which is a custom Laravel form request. This class should contain validation rules such as:

public function rules()

{

return [

'name' => 'required|string|unique:neighborhoods,name',

'city\_id' => 'required|exists:cities,id',

// any other required fields

];

}

Using a form request helps keep the controller clean and the validation logic reusable and testable.

2-**Create New Neighborhood**  
Neighborhood::create($request->all());

* Inserts a new neighborhood using mass assignment.
* Assumes the model has $fillable or $guarded properties correctly defined to allow this.

3-**Success Response**  
If creation is successful:

* Returns a success response containing the newly created neighborhood.
* The data is wrapped in a NeighborhoodResource for formatting.
* Uses \_\_('api.neighborhood\_add') for the response message, supporting localization.

4-**Error Handling**  
If any error occurs during the creation (e.g., DB constraint violation, etc.):

* The error is caught and a 422 error response is returned with the exception message.
  1. **update() Method**

**🔹 Overview**

The update() method is responsible for updating the details of a specific neighborhood based on its unique ID. It validates the input, checks for the existence of the neighborhood, and returns appropriate structured responses for success or failure.

**🔹 Key Functionalities**

**1-Retrieve Neighborhood by ID**  
The method begins by attempting to fetch the neighborhood using the provided ID. If the neighborhood exists, the update process continues. If it doesn't exist, the method returns a 404 error response.

**2-Check if Neighborhood Exists**  
After retrieving the neighborhood, the method checks whether a record was found:

* If it exists, the method proceeds to the update step.
* If not, it responds with a 404 status and a localized error message indicating that the neighborhood could not be found.

**3-Update the Neighborhood**  
If the neighborhood exists, it is updated with the data provided in the request. The request data has already been validated through a form request class, ensuring clean and safe data is used in the update process.

**4-Return Data**  
Upon successful update, the method returns a structured response:

* The updated neighborhood data is returned in a standardized format using a resource class.
* A localized success message is included to confirm the update was successful.

**5-Error Handling**  
If an exception occurs during any step of the process, the method captures it and returns a 422 error response:

* This ensures that the system doesn't crash unexpectedly.
* The error message from the exception is included in the response to help with debugging or user feedback.
  1. **destroy() Method**

**🔹 Overview**

The destroy() method is used to delete a specific neighborhood by its unique ID. It ensures the neighborhood exists before attempting deletion and handles both success and error responses in a structured and localized manner.

**🔹 Key Functionalities**

**1-Retrieve Neighborhood by ID**  
The method starts by attempting to find the neighborhood using the provided ID. This ensures that only existing records are considered for deletion.

**2-Check if Neighborhood Exists**  
Once the neighborhood is retrieved:

* If the record is found, the method proceeds to delete it.
* If not, it immediately returns a 404 error response indicating the neighborhood does not exist.

**3-Delete the Neighborhood**  
If the neighborhood exists, it is deleted from the database. This could be a soft delete or a hard delete depending on the model configuration.

**4-Return Success Message**  
Upon successful deletion, the method returns a structured success response:

* It includes a localized success message confirming the deletion.
* No neighborhood data is returned, only a confirmation message.

**5-Error Handling**  
If an exception is thrown during the process (e.g., database issues or unexpected errors):

* The method catches the error and returns a 422 response.
* The response includes the error message for further insights or debugging.
  1. **mostFrequent() Method**

**🔹 Overview**

The mostFrequent() method is responsible for identifying and returning the neighborhoods that have the highest number of reservations. It supports both paginated and non-paginated responses and formats the results using a dedicated resource class.

**🔹 Key Functionalities**

**1-Determine Pagination Preference**  
The method checks whether the request includes the paginate parameter:

* If set to 1, it applies pagination to the results.
* If set to 0 or not provided, it returns the full list.

**2-Query Reservations by Neighborhood**  
It processes the Reservation data to:

* Filter out records where the neighborhood is null.
* Group reservations by neighborhood name.
* Count the total number of reservations per neighborhood.
* Sort the results in descending order based on reservation count.

This logic helps identify the most frequently reserved neighborhoods.

**3-Handle Pagination (If Enabled)**  
When pagination is active:

* The method retrieves a paginated set of top neighborhoods.
* It constructs a pagination metadata array (total, per\_page, current\_page, total\_pages).
* The response includes both the data and pagination info.

**4-Return Results with Resource Formatting**  
In both cases (paginated or not), the method wraps the output using the TopNeighborhoodResource to ensure consistent formatting. It also attaches a localized success message.

**5-Error Handling**  
If any exception occurs during execution (e.g., a database query fails or unexpected input is encountered):

* The method catches the exception.
* Returns a 422 error response along with a localized error message and the exception details for debugging.

1. **Notification Controller**
   1. **all() Method**

**🔹 Overview**

The all() method is responsible for retrieving a list of notifications. It supports both paginated and non-paginated responses, depending on the request parameters, and returns the results in a structured, localized format.

**🔹 Key Functionalities**

**1-Check for Pagination Request**  
The method first determines whether the response should be paginated:

* If the request contains the paginate parameter and it is set to 1, it retrieves paginated results.
* If the paginate parameter exists and is set to 0, it retrieves all notifications without pagination.
* If the parameter is not provided, it defaults to pagination.

**2-Fetch Notifications**  
Based on the pagination preference:

* Notifications are fetched using the latest records first.
* If pagination is applied, it uses a limit (in this case, 10 notifications per page).

**3-Construct Pagination Metadata**  
When pagination is used:

* It constructs a pagination metadata array containing:
  + Total number of notifications.
  + Number of notifications per page.
  + Current page number.
  + Total number of pages.

**4-Format the Response**  
All notification data is wrapped in a NotificationResource for structured output. The response includes:

* Either a paginated or full list of notifications.
* A success message using the localized key api.notification\_all.

**5-Error Handling**  
If any error or exception occurs during the process:

* The method catches the exception.
* Returns a 422 error response with the exception message.
  1. **get() Method**

**🔹 Overview**

The get() method is designed to retrieve a specific notification based on the provided ID. It ensures proper error handling in cases where the notification is not found and delivers a structured response using localization.

**🔹 Key Functionalities**

**1-Retrieve Notification by ID**  
The method attempts to find the notification using its unique ID. If a notification exists with that ID, it proceeds; otherwise, it handles the "not found" scenario.

**2-Check if Notification Exists**

* If the notification is found, the method proceeds to return the data.
* If no matching notification is found, it returns an error response with a 404 status and a localized message using the key api.notification\_not\_found.

**3-Return Notification Data**

* When the notification is successfully found:
  + The data is transformed using the NotificationResource for consistent output.
  + The response includes a success message, localized using the key api.notification\_get.

**4-Error Handling**

* If any unexpected error or exception occurs (such as database issues or invalid data), the method:
  + Catches the exception using a try-catch block.
  + Returns a standardized error response with a 422 status and the exception message.
  1. add() method

🔹 **Overview**  
The add() method is designed to create and distribute notifications to clients through multiple channels (WhatsApp, SMS, or in-app notifications). It handles both test and production scenarios with proper segmentation of client groups and delivers structured responses with localization support.

🔹 **Key Functionalities**  
1-**Process Notification Request**  
The method first handles any attached image file by saving it through the handleFile() helper. It then processes the notification based on the specified type (1=WhatsApp, 2=SMS, 3=In-App).

2-**WhatsApp Notification Handling**  
For WhatsApp type (1), the method:

* Processes test numbers when group\_number=0
* Handles production batches in groups of 200 clients
* Creates notification records with type=6
* Queues WhatsApp messages with 1-second delay between jobs

3-**Client Segmentation Logic**  
When client\_ids are provided, the method:

* Handles 11 predefined client segments (0-10)
* Processes dynamic client IDs for custom targeting
* Creates appropriate notification records for each segment type
* Triggers the correct delivery method per notification type

4-**Error Handling**  
The method includes comprehensive exception handling that:

* Catches all Throwable exceptions
* Returns standardized error responses
* Provides exception details for debugging
  1. **getClientIdsByGroup() Method**

🔹 **Overview**  
The getClientIdsByGroup() method retrieves an array of client IDs belonging to a specified group segment. It's designed to support batch processing of clients by dividing them into equal-sized groups.

🔹 **Key Functionalities**  
1-**Group Calculation**

* Uses fixed group size of 500 clients
* Calculates database offset using formula:  
  ($groupNumber - 1) \* 500

2-**Database Query**

* Filters for active clients (type = 0)
* Orders results by client ID
* Applies calculated pagination (skip/take)
* Returns only ID field as array

3-**Return Value**

* Always returns an array of IDs
* Empty array when no clients in group

🔹 **Parameters**

* $groupNumber (integer):
  + 1-based index of the group to retrieve
  + Minimum value: 1

🔹 **Response Format**  
**Success**: Returns sequential array of client IDs

**Empty Result**: Returns empty array when group contains no clients

* 1. **destroy() Method**

🔹 **Overview**  
The destroy() method is designed to permanently delete a notification from the system based on the provided ID. It follows a strict error-handling pattern and provides localized response messages.

🔹 **Key Functionalities**  
1-**Notification Retrieval**

* Attempts to find the notification using the provided ID
* Uses Laravel's find() method which returns null if not found

2-**Existence Verification**

* Returns 404 error if notification doesn't exist
* Uses localized message key 'api.notification\_not\_found'

3-**Deletion Process**

* Permanently removes the notification record
* Returns success message upon completion
* Uses localized message key 'api.notification\_delete'

4-**Error Handling**

* Catches all Throwable exceptions
* Returns 422 status for any unexpected errors
* Includes original error message in response

🔹 **Parameters**

* $id (integer/string):
  + The primary key of the notification to delete
  + Must correspond to existing notification for successful operation
    - **Response Format**  
      **Success**:
      * {
      * "success": true,
      * "message": "Notification deleted successfully"
      * }
      * **Notification Not Found (404)**:
      * {
      * "error": "Notification not found",
      * "code": 404
      * }
      * **Processing Error (422)**:
      * {
      * "error": "Database connection failed",
      * "code": 422
      * }
  1. **processNotifications() Method**

🔹 **Overview**  
The processNotifications() method handles the creation of notification details and triggers the appropriate delivery method for each client in a collection. This private method supports three delivery channels (WhatsApp, SMS, and in-app) with queued execution.

🔹 **Key Functionalities**  
1-**Notification Detail Creation**

* Creates a NotificationDetail record for each client
* Preserves all request data except 'client\_ids'
* Links to both client and parent notification

2-**Multi-Channel Delivery**

* WhatsApp (type=1): Queues SendWhatsApp job with 1-second delay
* SMS (type=2): Directly calls sendSms() method
* In-App (type=3): Queues SendNotification job with 1-second delay

3-**Logging**

* Logs the notification type being processed
* Marks channel-specific processing (visible in logs as "whatsapp", "sms", or "app")

🔹 **Parameters**

* $clients (Collection):
  + Eloquent collection of Client models
  + Each must have id and phone fields
* $request (Request):
  + Contains notification data (title, body, type)
* $notification\_id (integer):
  + Parent notification's ID
* $imagePath (string - optional):
  + Path to attached image (for WhatsApp)

🔹 **Processing Flow**

1. Creates notification detail record
2. Identifies delivery type from request
3. Triggers appropriate delivery method
4. Repeats for all clients in collection

🔹 **Error Handling**

* No explicit try-catch (handled by calling method)
* Depends on queue system for job failures
* Assumes valid client data and existing notification
  1. **sendSms() Method**

🔹 **Overview**  
The sendSms() method handles the sending of SMS messages to clients through a third-party SMS gateway API. This protected method formats phone numbers, constructs API requests, and processes delivery responses.

🔹 **Key Functionalities**  
1-**API Configuration**

* Uses predefined credentials (API key, username, sender ID)
* Selects appropriate API endpoint
* Applies URL encoding for message content

2-**Phone Number Processing**

* Formats recipient numbers using internal formatPhoneNumber() method
* Ensures proper number structure for API compatibility

3-**Secure API Communication**

* Establishes HTTPS connection with SSL verification disabled
* Uses GET request method with custom headers
* Implements proper resource cleanup

4- **Response Handling**

* Captures and logs raw API response
* Returns boolean true regardless of delivery outcome

🔹 **Parameters**

* $phone (string):
  + Recipient phone number (raw format)
  + Will be processed by formatPhoneNumber()
* $message (string):
  + Content to be sent (160 character limit recommended)
  1. **getNotificationDetails() Method**

🔹 **Overview**  
The getNotificationDetails() method retrieves paginated details for a specific notification. It provides structured data including both notification content and pagination metadata, with comprehensive error handling.

🔹 **Key Functionalities**  
1-**Data Retrieval**

* Fetches notification details by notification\_id
* Orders results by latest first
* Paginates results with 10 items per page

2-**Pagination Handling**

* Calculates pagination metadata including:
  + Total records
  + Items per page
  + Current page
  + Total pages

3-**Response Formatting**

* Wraps data in NotificationResource collection
* Includes both notification data and pagination info
* Uses localized message ('api.notification\_all')

4-**Error Handling**

* Catches all Throwable exceptions
* Returns 422 status for errors
* Includes original error message

🔹 **Parameters**

* $notification\_id (integer):
  + The ID of the parent notification
  + Must correspond to existing notification

🔹 **Response Format**  
**Success**:

{

"data": {

"notification": [...], // Array of notification details

"pagination": {

"total": 50,

"per\_page": 10,

"current\_page": 1,

"total\_pages": 5

}

},

"message": "Notification details retrieved successfully"

}

**Error (422)**:

{

"error": "Database connection failed",

"code": 422

}

1. **Obligation Controller** 
   1. **all() method**

🔹 **Overview**  
The all() method retrieves obligations based on the user's role (either investor or admin) and provides an option to paginate the data or return all data in a single response. It handles user-specific filtering, pagination, and exception management, offering flexibility in how data is returned.

🔹 **Key Functionalities**  
1-**User Role Check**  
The method first checks the user's role by retrieving the role\_id and comparing it to the value 6 (indicating the "investor" role). Based on this role, it determines whether to fetch obligations specific to the investor or general obligations for admins.

* If the user is an investor (role\_id == 6), it retrieves obligations associated with the investor's client by checking if the investor\_id matches, and the client has an active and valid status.
* For other roles (admin or otherwise), it fetches all obligations without user-specific filtering.

2-**Obligations Retrieval**  
After determining the user’s role, the method fetches obligations using the Obligations model. It applies the latest() method to ensure the most recent obligations are retrieved.

* If the user is an investor, obligations are filtered by the investor’s ID and client’s type and status.
* If the user is an admin, all obligations are fetched without additional filtering.

3-**Pagination Logic**  
The method handles pagination based on the request parameter paginate. If paginate is set to 1, it returns a paginated response (default: 10 items per page). If paginate is set to 0, it returns all the obligations without pagination. If the paginate parameter is not provided, it defaults to paginated results.

* Pagination metadata is included in the response if pagination is enabled, providing details such as total items, items per page, and the current page.

4-**API Response**  
The method formats the API response based on whether pagination is used or not:

* If pagination is enabled, the response includes the obligations data along with pagination information.
* If pagination is disabled, the response only includes the obligations data.

5-**Error Handling**  
The method is wrapped in a try-catch block to ensure any exceptions are caught and handled gracefully. If an error occurs, a standardized error response with status code 422 is returned, along with the exception message.

* 1. **get() method**

🔹 **Overview**  
The get() method retrieves a specific obligation by its ID and returns it in a structured API response. It handles errors such as when the obligation is not found and provides a clean response with error handling.

🔹 **Key Functionalities**  
1-**Obligation Retrieval**  
The method first attempts to find the obligation using the Obligations::find($obligation\_id) query. If the obligation with the provided ID exists, it proceeds to return the data.

* If the obligation is not found, a 404 Not Found error response is returned with a relevant error message: 'api.obligation\_not\_found'.

2-**Obligation Resource Transformation**  
Once the obligation is found, it is wrapped in an ObligationResource. This is used to transform the data into a format suitable for the API response, ensuring consistency with the application's resource structure.

3-**API Response**

* On success, the method returns a structured response with the obligation data under the "data" key, along with a success message: 'api.obligation\_get'.
* If the obligation is not found, it returns an error response with a 404 status code.

4-**Error Handling**  
The method is enclosed in a try-catch block to handle any potential exceptions. If an exception occurs during the database query or transformation, it returns an error response with a status code of 422 and the error message.

🔹 **Response Structure**

* On success, the response will include:
  + obligation: The specific obligation wrapped in the ObligationResource format.
* If the obligation is not found, the response will include:
  + An error message with status code 404, indicating that the obligation was not found.
* If an error occurs during processing, the response will include:
  + A standardized error response with a status code of 422 and the exception message.

🔹 **Error Handling**  
The method ensures proper exception handling by catching all Throwable exceptions. In case of an error, it returns a clear and structured error response, ensuring that any issues encountered during execution are communicated effectively to the client.

* 1. **add() method**

🔹 **Overview**  
The add() method is responsible for creating a new obligation entry in the system. It first checks if the client exists, then proceeds to store any uploaded image, and finally creates the obligation record. It returns a structured response with the newly created obligation data or an error message if something goes wrong.

🔹 **Key Functionalities**  
1-**Client Existence Check**  
The method starts by checking if the provided client\_id exists in the database. If the client is not found, it returns a 404 Not Found error with the message 'api.client\_not\_found'.

2-**Image Upload Handling**  
If an image is provided in the request, the method attempts to store it in the uploads/obligations folder under the public disk. The path of the stored image is then saved in the obligation record.

3-**Obligation Creation**  
After validating the client and handling the image upload (if applicable), the method creates a new obligation record using the data provided in the request:

* total: The total amount for the obligation.
* client\_id: The ID of the associated client.
* type: The type of the obligation.
* image: The path of the uploaded image (if any).

The newly created obligation is then returned in the response.

4-**API Response**

* On success, the method returns the newly created obligation in a structured response under the "data" key, wrapped in an ObligationResource, along with a success message: 'api.obligation\_add'.
* If an error occurs (e.g., client not found or image upload fails), an error response with a 422 status code and the exception message is returned.
  1. **update() method**

🔹 **Overview**  
The update() method is designed to update an existing obligation record. It checks if the client exists, retrieves the obligation by ID, handles optional image replacement, and updates the obligation with new data from the request. The method then returns the updated obligation data in a structured API response or an error message if any issues arise.

🔹 **Key Functionalities**  
1-**Client Existence Check**  
If the request includes a client\_id, the method first checks if the client exists in the database. If the client is not found, it returns a 404 Not Found error with the message 'api.client\_not\_found'.

2-**Obligation Retrieval**  
The method attempts to find the obligation using the provided obligation\_id. If the obligation is not found, it returns a 404 Not Found error with the message 'api.obligation\_not\_found'.

3-**Image Replacement Handling**

* If the request contains a new image, the method stores the image and updates the image path for the obligation.
* If a new image is uploaded, the method deletes the old image from the storage to prevent orphaned files.
* If no new image is provided, the existing image path remains unchanged.

4-**Obligation Update**  
The method then updates the obligation with the new data provided in the request, using the following fields:

* total: The total amount for the obligation (if provided).
* client\_id: The client ID associated with the obligation (if provided).
* type: The type of the obligation (if provided).
* image: The path to the updated image (if provided).

It uses the update() method to save the changes.

5-**API Response**

* On success, the method returns the updated obligation in a structured response under the "data" key, wrapped in an ObligationResource, along with a success message: 'api.obligation\_update'.
* If the obligation or client is not found, it returns an error response with a 404 status code.

6-**Error Handling**  
The method is enclosed in a try-catch block to handle any exceptions during the update process. If an exception occurs (e.g., a client not found, an image upload issue, or a general error), it returns a 422 error response with the exception message.

* 1. **destroy() method**

🔹 **Overview**  
The destroy() method is designed to delete an existing obligation record from the system. It first checks if the obligation exists, deletes the associated image (if any), and then removes the obligation from the database. The method returns a success message if the deletion is successful or an error message if any issues arise during the process.

🔹 **Key Functionalities**  
1-**Obligation Retrieval**  
The method first attempts to find the obligation using the provided obligation\_id. If the obligation is not found, it returns a 404 Not Found error with the message 'api.obligation\_not\_found'.

2-**Image Deletion**  
If the obligation has an associated image, the method deletes the image file from the public storage using the Storage::disk('public')->delete() method. This ensures that the image file does not remain orphaned in the storage.

3-**Obligation Deletion**  
After ensuring that the image is deleted (if applicable), the method deletes the obligation record from the database using the delete() method.

4-**API Response**

* On success, the method returns a success message with the text 'api.obligation\_delete'.
* If the obligation is not found, it returns an error response with a 404 status code, indicating that the obligation does not exist.

5-**Error Handling**  
The method is wrapped in a try-catch block to handle any exceptions during the deletion process. If any exception occurs (e.g., database errors, image deletion failures), it returns a 422 error response with the exception message.

* 1. **obligationTypes() method**

🔹 **Overview**  
The obligationTypes() method is designed to return a list of predefined obligation types. It sends an array of obligation types along with their respective names and types, supporting localization for the type names. This method is helpful when the frontend or client application needs a list of available obligation types to display to the user.

🔹 **Key Functionalities**  
1-**Predefined Obligation Types**  
The method defines an array called $types containing four types of obligations. Each entry in the array includes:

* name: The localized name of the obligation type, retrieved using the \_\_() function for localization.
* type: A unique identifier for each obligation type.

2-**API Response**

* The method returns the $types array inside a structured response with a key "types". The response is wrapped under a "data" key.
* A success message 'api.obligation\_types' is included in the response to indicate that the obligation types have been successfully retrieved.

3-**Error Handling**  
The method is wrapped in a try-catch block to handle any exceptions that may occur during the execution. If any exception arises (e.g., an issue with fetching or formatting the types), the method returns a 422 error response with the exception message.

**26. Payment Controller**

**26.1 all() method**

🔹 **Overview**  
The all() method retrieves all payment types from the database and returns them in a structured API response. It uses a resource collection to format the payment types before sending them to the client, providing a clear structure for displaying payment types.

🔹 **Key Functionalities**  
1-**Retrieving Payment Types**  
The method fetches all records from the PaymentTypes model using the get() method. This retrieves all available payment types stored in the database.

2-**Resource Formatting**  
After fetching the payment types, the method formats the retrieved data using a PaymentTypeResource collection. This ensures the payment types are returned in a structured and standardized format, suitable for API responses.

3-**API Response**

* The formatted payment types are returned inside a "paymentTypes" key, wrapped under a "data" key.
* A success message, 'api.paymentType\_all', is included in the response to indicate that all payment types have been successfully retrieved.

4-**Error Handling**  
The method is enclosed in a try-catch block to handle any exceptions that might occur during the retrieval or formatting process. If an exception is caught (e.g., a database connection issue or formatting error), the method returns a 422 error response with the exception message.

**26.2 active() method**

🔹 **Overview**  
The active() method is designed to retrieve all payment types from the database, format them using a resource collection, and return them in a structured API response. This method is similar to the all() method but is intended for cases where you may need to provide a list of active payment types in your application.

🔹 **Key Functionalities**  
1-**Retrieving Payment Types**  
The method retrieves all payment types from the PaymentTypes model using the get() method, fetching all records available in the database.

2-**Resource Formatting**  
After fetching the payment types, the method formats them using the PaymentTypeResource collection. This ensures that the data is returned in a clean, standardized format suitable for API responses.

3-**API Response**

* The payment types are returned inside the "paymentTypes" key, wrapped under a "data" key.
* A success message ('api.paymentType\_all') is included in the response to indicate that all payment types have been successfully retrieved.

4-**Error Handling**  
The method is wrapped in a try-catch block to handle any exceptions that may arise during the data retrieval or formatting process. If an exception is thrown (e.g., a database error or issue with resource formatting), the method returns a 422 error response along with the exception message.

**26.3 payment\_types\_for\_filter() method**

🔹 **Overview**  
The payment\_types\_for\_filter() method is designed to return a list of predefined payment types that can be used for filtering purposes in the dashboard. It returns these types with corresponding labels in the selected language, allowing for easy integration into a filter interface where users can select payment types.

🔹 **Key Functionalities**  
1-**Defining Payment Types**  
The method defines a list of payment types in an array. Each payment type consists of:

* value: A numeric identifier for the payment type.
* label: A localized string representing the payment type, fetched using the \_\_() function for translation (ensuring support for multiple languages).

2-**Returning Data**  
The list of payment types is returned within the "types" key, which is wrapped inside the "data" key. This provides a clean and consistent response structure for the API.

3-**API Response**  
The method includes a success message ('dashboard.paymentType\_all'), indicating that the payment types have been successfully retrieved.

4-**Localization Support**  
The labels for each payment type are retrieved using the \_\_() helper function, which allows for localization based on the selected language. This ensures that the payment types are displayed in the appropriate language for the user.

**26.4 online\_payment\_methods() method**

🔹 **Overview**  
The online\_payment\_methods() method is designed to return a list of online payment methods available for users. These methods are represented with numeric values and localized labels, making it easy to integrate into a user interface for selecting a payment method. The list can be used in scenarios such as online transactions or checkout processes.

🔹 **Key Functionalities**  
1-**Defining Online Payment Methods**  
The method defines a static list of online payment methods, each with:

* value: A numeric identifier for the payment method.
* label: A localized name for the payment method, ensuring that the correct language is displayed to the user.

2-**Returning Data**  
The list of payment methods is returned inside the "types" key, which is wrapped in a "data" key to ensure a clean API response format.

3-**Localization Support**  
Each payment method's label is fetched using the \_\_() function, which supports translation. This allows the labels to be displayed in the appropriate language based on the user's settings.

4-**API Response**  
The method includes a success message ('dashboard.online\_payment\_methods'), indicating that the online payment methods were successfully retrieved.

**27.Package Controller**

**27.1 all() method**

🔹 **Overview**  
The all() method is responsible for fetching a list of packages, with an option to filter based on city and paginate the results. The method handles various scenarios depending on whether the client requests paginated data or all available packages. It also includes error handling to ensure smooth operation.

🔹 **Key Functionalities**  
1-**Package Query Setup**  
The method initializes a query for the Package model. It then checks for the presence of a city\_id in the request:

* If city\_id is provided, it filters the packages by matching the city\_name field to the city name associated with the given city ID.

2-**Pagination Handling**  
The method checks if pagination is requested:

* If paginate is set to 1, the results are paginated with a default page size of 10. The response includes pagination metadata such as total count, per page, current page, and total pages.
* If paginate is set to 0, it retrieves all available packages without pagination.
* If the paginate parameter is not provided, it defaults to paginating the results with the same page size of 10.

3-**Response Data**  
The method returns the package data in a standardized format using PackageResource::collection(). It also includes the pagination metadata (if pagination is applied).

4-**Error Handling**  
A try-catch block ensures that any errors that occur during the execution are caught and returned with a status code of 422, along with an error message.

**27.2 get($package\_id) method**

🔹 **Overview**  
The get($package\_id) method is responsible for retrieving the details of a single package identified by its package\_id. It handles errors and ensures that if the package is not found, an appropriate response is returned.

🔹 **Key Functionalities**  
1️⃣ **Package Lookup**  
The method first attempts to find a Package model record by its package\_id. If no record is found, it returns a 404 error with a message indicating that the package could not be found.

2️⃣ **Successful Response**  
If the package is found, it returns the data in a structured format using a PackageResource for better presentation. The response also includes a localized success message, such as 'api.package\_get'.

3️⃣ **Error Handling**  
The method is wrapped in a try-catch block, ensuring that if any error occurs during the process, a 422 status code is returned, along with a message describing the error.

🔹 **Response Structure**

* On success:
  + data: Contains the package details under the key "package", formatted with the PackageResource.
* On failure:
  + Error message: If the package is not found, it returns a 404 error. If any exception occurs, it returns a 422 error with a description of the issue.

🔹 **Edge Case Considerations**

* If the package\_id does not correspond to any existing package, the system will return a "Package not found" message with a 404 status.
* Error handling ensures that if any issues occur during the process (such as database access issues), the system gracefully handles it and returns an error message.

**27.3 add() method**

🔹 **Overview**  
The add(StorePackageRequest $request) method is designed to create a new package in the system. It validates the incoming request, processes data like tax calculations and image handling, attaches related services, and stores city-specific pricing information. After the package is created, it returns a structured response with the package details.

🔹 **Key Functionalities**  
1-**Tax Calculation**  
The method calculates the cost before tax (cost\_without\_tax) based on the total cost provided and the tax rate from the first Application instance in the database. This ensures the price is calculated accurately by removing the tax portion.

2-**Image Upload**  
If an image is provided in the request, the method handles file storage using a helper function handleFile(), which stores the image and returns its path. If no image is provided, the image field is set to null.

3-**Package Creation**  
The package is created using the provided request data, such as title, description, slug, terms, period, and status. The calculated cost\_without\_tax is stored, along with the image path.

4-**Service Attachments**  
The method attaches main and additional services to the package using a services() relationship. For the main service, the quantity is attached as main\_service\_count. For additional services, the count for each service is also stored.

5-**City-Specific Pricing**  
If city-specific pricing data is provided, the method processes the pricing data for different cities and saves the information in the CityPackagesPrices table, associating the city ID and cost with the package.

6-**Response**  
After successfully creating the package and attaching related services and city pricing, a response is returned with the package details formatted using the PackageResource. The response also includes a localized success message.

🔹 **Response Structure**

* **Success Response**:
  + data: Contains the newly created package, formatted using PackageResource.
  + Localized success message (e.g., 'api.package\_add').
* **Error Handling**:
  + If any validation or process issues occur, the system will handle them through automatic request validation and exception handling.

🔹 **Edge Case Considerations**

* **Missing Services**: If additional services are not provided, the method only attaches the main service, ensuring it doesn't fail if additional services are optional.
* **City Prices**: If no city pricing data is provided, it skips the city price handling, ensuring it doesn't break if this part is absent.

**27.4 update(UpdatePackageRequest $request, $package\_id) method**

🔹 **Overview**  
The update(UpdatePackageRequest $request, $package\_id) method updates an existing package in the system. It accepts a request containing updated data for the package, processes it (including tax calculations, image updates, and service re-attachments), and returns the updated package data as a response.

🔹 **Key Functionalities**  
1-**Package Retrieval**  
The method first attempts to find the package using the provided $package\_id. If the package is not found, a 404 error response is returned with a message indicating the package is missing.

2-**Tax Calculation**  
Similar to the add method, the tax rate is fetched from the Application model, and the cost before tax (cost\_without\_tax) is calculated using the formula:  
cost\_without\_tax = cost / (1 + (tax / 100)).

3-**Image Update**  
If a new image is provided, it is processed and saved, replacing the old image. If no new image is uploaded, the existing image path remains unchanged.

4-**Package Update**  
The package's attributes (e.g., title, description, slug, terms, etc.) are updated based on the request data. If any field is not provided, the current value of the package is retained.

5-**Service Management**

* **Detaching Old Services**: The services() relationship is detached to remove old services associated with the package.
* **Main and Additional Services**: New services are attached to the package:
  + The main service is attached with the provided main\_service\_count.
  + Additional services are processed and attached with the given counts if provided.

6-**City-Specific Pricing Update**  
If city-specific pricing data is provided, the method:

* Deletes the existing city prices for the package.
* Reprocesses the new city pricing data, creating new entries in the CityPackagesPrices table for each city and its respective price.

7-**Response**  
The updated package is returned using the PackageResource to ensure proper data formatting. A localized success message ('api.package\_update') is included in the response.

🔹 **Response Structure**

* **Success Response**:
  + data: Contains the updated package details, formatted using PackageResource.
  + Localized success message (e.g., 'api.package\_update').
* **Error Handling**:
  + If any validation or process issues occur, the system will handle them with appropriate error messages, returning a 422 status code and the error message.

**27.5 destroy($package\_id) method**

🔹 **Overview**  
The destroy($package\_id) method is responsible for deleting a package from the system. It ensures that the package is deletable by checking if there are any subscriptions associated with it. If the package is deletable, it also handles the removal of the package image from storage and performs the deletion in the database.

🔹 **Key Functionalities**  
1-**Package Retrieval**  
The method first tries to find the package by its ID. If the package does not exist, a 404 error response is returned with a message indicating that the package was not found.

2-**Check for Subscriptions**  
Before proceeding with the deletion, the method checks whether the package has any associated subscriptions (package\_subscriptions). If there are active subscriptions, the package cannot be deleted, and a 400 error response is returned with a message indicating that the package cannot be deleted because it has subscriptions.

3-**Image Deletion**  
If the package has an associated image, the image is deleted from storage using Laravel's Storage facade. This ensures that unused files do not remain in the system.

4-**Package Deletion**  
If there are no subscriptions and the image is successfully deleted, the package is removed from the database.

5-**Response**  
After successfully deleting the package, a localized success message ('api.package\_delete') is returned. In case of any errors, a 422 error response is returned with an appropriate error message.

🔹 **Response Structure**

* **Success Response**:
  + The response includes a success message, localized with 'api.package\_delete', indicating that the package has been successfully deleted.
* **Error Handling**:
  + If the package does not exist, a 404 error is returned with a localized error message.
  + If there are active subscriptions, a 400 error is returned, indicating the package cannot be deleted due to subscriptions.
  + If an error occurs during the deletion process, a 422 error is returned with the error message.

**27.6 packageSubscriptions() method**

🔹 **Overview**  
The packageSubscriptions method is designed to fetch and filter package subscriptions based on various criteria provided in the request, such as city, client name, phone, status, and pagination preferences. It ensures flexibility in how the subscriptions are retrieved, either in a paginated format or as a simple collection.

🔹 **Key Functionalities**  
1-**Query Building**  
The method starts by constructing a query on the PackageSubscription model, utilizing the with('client') relationship to eagerly load associated client data. The query is then filtered based on several optional request parameters:

* **City**: If a city\_id is provided, it filters the subscriptions based on the specified city.
* **Client Name**: If a name (client username) is provided, the query filters subscriptions where the associated client's username matches the provided name.
* **Phone**: If a phone number is provided, the query filters subscriptions where the associated client's phone matches the provided number.

2-**Role-Based Filtering**  
The method checks the user's role (role\_id) to apply role-specific filters:

* **City-Based Filtering**: If the user's role is either 17 or 7, it further restricts the query to only show subscriptions for the user's city.

3-**Status Filtering**  
The method checks if a status is provided in the request and filters the subscriptions based on this status.

4-**Ordering**  
The query orders the subscriptions by the latest entries using the latest() method to ensure the most recent subscriptions are fetched first.

5-**Pagination**

* If the request has a paginate parameter set to 1, the method applies pagination with a limit of 20 subscriptions per page. The pagination details (total, per page, current page, total pages) are included in the response.
* If pagination is not requested, the method retrieves all matching subscriptions using get().

6-**Response Structure**

* **Paginated Response**: If pagination is enabled, the method returns the paginated list of subscriptions along with pagination details.
* **Non-Paginated Response**: If pagination is not enabled, the method returns the full collection of subscriptions.

In both cases, the subscriptions are formatted using SubscriptionResource::collection() for consistent response formatting.

🔹 **Error Handling**

* The method uses a try-catch block to handle any potential errors. If an error occurs, a 422 error response is returned along with the error message.

**27.7 checkReservation() method**

🔹 **Overview**  
The checkReservation method is designed to check whether a reservation already exists for a given representative on a specific date and time, ensuring no overlapping reservations are made for the same representative at the same time. This method is crucial for preventing double-booking conflicts.

🔹 **Key Functionalities**

1-**Query Building**  
The method initiates a query on the Reservation model, checking for an existing reservation based on the following parameters:

* **representative\_id**: It ensures the reservation is for the specific representative.
* **date**: It checks that the reservation matches the specified date.
* **from**: It ensures that the reservation matches the specified "from" time.
* **status != 6**: The reservation's status must not be 6, likely indicating a canceled or invalid reservation.

2-**Reservation Existence Check**  
The method queries the Reservation model for a record matching the given parameters. If such a reservation exists and the status is not 6, the method returns true, indicating that the reservation already exists. If no such reservation is found, it returns false.

3-**Return Logic**

* **True**: If a reservation exists with the same representative, date, and time, and its status is not 6, the function returns true, indicating that the reservation cannot be made due to a conflict.
* **False**: If no such reservation exists, it returns false, indicating that the representative is available at the specified time.

🔹 **Error Handling**  
The method is straightforward and doesn't contain explicit error handling within the function itself, but any failure in database query or issues related to missing parameters will be handled by the caller, which can catch any errors and handle them appropriately.

**27.8 makeReservationUsingPackage() method**

🔹 **Overview**  
The makeReservationUsingPackage method is responsible for handling the reservation creation process when a client uses a specific package for their service booking. The method validates the incoming request, checks the availability of the representative and services, verifies the client's subscription to the package, and ultimately creates the reservation while managing the usage of services associated with the package.

🔹 **Key Functionalities**

1-**Request Validation**  
The method starts by validating the incoming request using Validator::make(), ensuring that all required fields such as car\_id, date, time, main\_service\_id, package\_id, and location\_id are provided and valid. If the validation fails, the method returns a 403 response with the validation errors.

2-**Reservation Check**  
Before proceeding with the reservation creation, the method checks whether a reservation already exists using the checkReservation() method. If a reservation exists with the same date and time for the specified representative, the method returns a 200 response indicating that the time slot is already reserved.

3-**Service and Package Validation**  
The method verifies whether the client is subscribed to the requested package and whether they have access to the requested main service. If the client is not subscribed to the package or does not have the required service, a 404 error response is returned. Additionally, if the client has additional services, their availability is also checked and deducted from the subscription's count.

4-**Shift and Available Slot Check**  
The method calculates the available time slots for the requested reservation based on the representative's shift times. It breaks down the shift into intervals (e.g., every 90 minutes) and checks whether the requested time falls within an available slot. If no representative is available, the method returns a 403 response indicating that no representative is available for the reservation.

5-**Reservation Creation**  
Once the validation and checks are successful, the method creates a new reservation record. It assigns values like client\_id, location\_id, car\_id, and service details to the reservation and also calculates the from and to times based on the requested reservation time and the duration of the service. The reservation is then saved to the database.

6-**Reservation Services**  
If additional services are provided in the request, they are saved in the reservation\_services table to track the specific services included in the reservation.

7-**Representative Assignment**  
If a representative is found available, the reservation is updated with the representative\_id, and a notification is sent to the representative and the client regarding the new reservation.

8-**Response Structure**

* **Success Response**: If the reservation is successfully created, a success response with a status code of 200 is returned, along with a message indicating that the reservation was added successfully.
* **Error Response**: If an error occurs at any point, a 403 error response is returned with an appropriate message (e.g., package not found, no representative available, time slot already taken).

**27.9 saveReservationServices() method**

🔹 **Overview**  
The saveReservationServices method is responsible for associating additional services with a reservation. It attaches the selected services to the reservation and stores their associated cost.

🔹 **Key Functionalities**

1-**Service IDs Check**

* The method first checks if the $services array has any items. If the array contains service IDs, the method proceeds to attach them to the reservation.

2-A**ttach Services to Reservation**

* For each service\_id in the $services array, the method attempts to find the corresponding service using Service::find($service\_id).
* If the service exists, it attaches the service to the reservation using the services() relationship method. Along with attaching the service, it also stores the associated cost of that service in the pivot table (the table that links the services and reservations tables).

3-**Pivot Table**

* The pivot table (reservation\_service) will hold the relationship between the reservation and the services, as well as the cost of each service. This is useful for tracking which services are included in the reservation and their respective costs.

4-**Return Value**

* After attaching all services to the reservation, the method returns true to indicate that the operation was successful.

**27.10 representativeReservationNotification() method**

🔹 **Overview**  
The representativeReservationNotification method is responsible for sending notifications to both the client and the representative regarding a reservation. The method uses Firebase Cloud Messaging (FCM) to send push notifications and also saves notifications in the system for both users.

🔹 **Key Functionalities**

1-**Firebase Notification to Client**

* The method uses the FireBasePushNotification class to send a push notification to the client.
* The notification is sent using the client’s device token ($reservation->client->device\_token), with the body and title fetched from the language file (trans('notifications.client\_representative\_reservation\_body') and trans('notifications.client\_representative\_reservation')).

2-**Saving Notification for Client**

* After sending the notification, the method saves a record of the notification in the system by calling saveNotification().
* The saved notification includes the title and body of the notification, and is associated with the client's ID ($reservation->client->id).

3-**Firebase Notification to Representative**

* A similar push notification is sent to the representative of the reservation using the representative's device token ($reservation->representative->device\_token).
* The body and title of the notification for the representative are also fetched from the language file.

4-**Saving Notification for Representative**

* The method saves the notification for the representative in the system, just like it did for the client, using saveNotification().
* The representative's ID ($reservation->representative->id) is used to associate the notification with the correct representative.

🔹 **Supporting Functions and Considerations**

1. **FirebasePushNotification Class**
   * The FireBasePushNotification class is assumed to handle the logic for sending push notifications via Firebase Cloud Messaging. The to() method in this class is responsible for sending the push notifications to the provided device tokens with the appropriate message.
2. **saveNotification Method**
   * The saveNotification() method likely stores the notification in the database, allowing it to be viewed later in the system (e.g., in a notifications dashboard). It saves a record with the title and body of the notification for the given user (client or representative).
3. **Localization**
   * The trans() function is used to fetch notification titles and bodies from language files, ensuring that notifications can be displayed in the user's preferred language (for example, in Arabic or English).
4. **Error Handling**
   * The method does not have explicit error handling for cases where the device token might be invalid or missing for either the client or representative. It assumes that both users (client and representative) exist and have valid device tokens.

**27.11 newReservationNotification() method**

🔹 **Overview**

The newReservationNotification method sends a notification to the client when a new reservation is created. Here's a detailed breakdown of how it works:

**🔹 Key Functionality**

1-**Firebase Notification to Client**

* The method uses the FireBasePushNotification class to send a push notification to the client.
* The notification is sent using the client’s device token ($reservation->client->device\_token).
* The body and title of the notification are retrieved from the language files using trans():
  + The body: trans('notifications.new\_reservation\_body')
  + The title: trans('notifications.complete\_reservation')

2-**Purpose of the Notification**

* This notification informs the client about the creation of a new reservation. The body and title are likely predefined in the language files and may be displayed in the client’s preferred language (e.g., Arabic or English).

**🔹 Supporting Functions and Considerations**

1. **FirebasePushNotification Class**
   * The FireBasePushNotification class is assumed to handle the logic for sending push notifications via Firebase Cloud Messaging (FCM). The to() method is used to send the notification to the specified device token.
2. **Localization**
   * The trans() function retrieves the notification's title and body from language files, ensuring that the message is correctly translated for different languages. This is useful for multi-language support.

**27.12 saveNotification() method**

🔹 **Overview**

The saveNotification method is responsible for saving a notification record to the

database. Here's a detailed breakdown of how it works:

### 🔹 ****Key Functionality****

1-**Create a New Notification**

* A new Notification object is created using new Notification().
* The client\_id, title, and body are then set from the provided arguments ($client\_id and $message).

2-**Saving the Notification**

* After setting the attributes, the save() method is called to store the notification in the database.

### 🔹 ****Supporting Considerations****

1. **Model Assumptions**
   * The Notification model should have the necessary attributes (like client\_id, title, body) to match the database schema. This should include the client\_id as a foreign key, and the title and body should be strings to store the notification details.
2. **Database Schema**
   * Ensure that the database has a notifications table with at least the following fields:
     + client\_id: This would be a foreign key referencing the clients table.
     + title: A string field to store the notification title.
     + body: A string field to store the notification body.
     + Timestamps (created\_at, updated\_at) are often automatically managed if using Eloquent.

### 🔹 ****Edge Cases to Consider****

1. **Database Constraints**
   * Ensure that the client\_id exists in the clients table before saving the notification. If the client does not exist or is deleted, the save operation might fail due to foreign key constraints.
2. **Notification Duplication**
   * If the same notification is sent multiple times to the same client, this method will create multiple identical records in the notifications table. Depending on your application requirements, you might want to check for duplicate notifications before saving them.
3. **Error Handling**
   * There's no error handling here if the save fails (e.g., if there is a database error). It might be useful to wrap the save operation in a try-catch block to catch any potential issues.

**28. Investors Controller**

**28.1 all() method**

🔹 **Overview**

The all method you provided is designed to retrieve investment applications and

return them, either paginated or as a full list, based on the request parameters. Here's a

breakdown of its functionality:

### 🔹 ****Key Functionality:****

1. **Filtering by Status**:
   * If a status is provided in the request, it filters the applications by that status. If no status is provided, it defaults to filtering by status = 0.
2. **Pagination**:
   * It checks if pagination is required by the client:
     + If paginate = 1, it paginates the results and returns pagination data (total, per\_page, current\_page, total\_pages).
     + If paginate = 0, it fetches all the records without pagination.
     + If no paginate parameter is passed, it defaults to paginating the results with 10 records per page.
3. **Error Handling**:
   * It wraps the logic in a try-catch block to catch any potential exceptions and return a proper error message if something goes wrong.
4. **Return Data**:
   * The method uses the returnData method to return the results along with a success message.
   * It uses the InvestmentApplicationsResource to format the response data.

**28.2 get($application\_id) method**

**🔹** Overview

The get method you provided is designed to fetch a specific investment application based on the provided application\_id. If the application is found, it returns the data in the InvestmentApplicationsResource format. If the application is not found, it returns a 404 error. The method is wrapped in a try-catch block to handle any potential exceptions.

Here's a breakdown of the logic:

### 🔹 ****Key Functionality:****

1. **Fetch Application by ID**:
   * The method uses Investors::find($application\_id) to retrieve the application by its ID.
2. **Error Handling**:
   * If the application is not found, a 404 error response is returned with a relevant message: 'investment\_application\_not\_found'.
3. **Resource Formatting**:
   * If the application is found, it's wrapped in a InvestmentApplicationsResource before being returned.
4. **Catch Errors**:
   * Any other errors are caught by the catch block, and an error message with the exception message is returned.

**28.3 accept($application\_id) method**

**🔹 Overview**

The accept method you provided handles accepting an investment application by creating a new user based on the application data and updating the application's status.

### 🔹 ****Functionality Overview:****

1. **Find Application**:
   * The method first searches for the application\_id in the Investors model.
2. **Create User**:
   * If the application exists, it creates a new User using the details from the application, including fields like name, email, phone, etc., and assigns a role\_id of 6 (presumably for the user role).
3. **Update Application Status**:
   * The application's status is then updated to 1, likely indicating the "accepted" state.
4. **Return Response**:
   * The method returns the updated application data wrapped in a resource, along with a success message.

**28.4 reject($application\_id) method**

**🔹 Overview**

The reject method handles rejecting an investment application by updating its status to 2, presumably indicating a rejected state. Here's a breakdown and improvement of the function:

### 🔹 ****Functionality Overview:****

1. **Find Application**:
   * The method searches for the application\_id in the Investors model.
2. **Update Application Status**:
   * If the application is found, its status is set to 2, which presumably marks it as rejected.
3. **Return Response**:
   * A success message is returned indicating the application has been rejected.
4. **Error Handling**:
   * If any errors occur, an error message is returned with a 422 status code.

**28.5 destroy($application\_id) method**

**🔹 Overview**

The destroy method deletes an investment application by its ID, and the process is wrapped in error handling. Here’s a breakdown of what the function does:

### 🔹 ****Functionality Overview:****

1. **Find Application**:
   * The method searches for the application\_id in the Investors model.
2. **Delete Application**:
   * If the application is found, it is deleted from the database.
3. **Return Response**:
   * A success message is returned, indicating the application was successfully deleted.
4. **Error Handling**:
   * If any error occurs during the process, an error message is returned with a 422 status code.

**28.6 investors() method**

**🔹 Overview**

The investors() method retrieves all users with the role ID of 6 (presumably representing investors), and it returns this data in the API response. It’s wrapped in a try-catch block to handle any potential errors during the process.

### 🔹 ****Functionality Overview:****

1. **Fetch Investors**:
   * The method uses User::where('role\_id', 6)->get() to fetch all users with the role\_id of 6 (investors).
2. **Return Response**:
   * The retrieved investors are returned in a structured response using the returnData method.
3. **Error Handling**:
   * Any errors that occur during the process are caught and returned with a 422 error code and the error message.

**29. Permission Controller**

**29.1 all() Method**

🔹 **Overview**  
The all() method is responsible for retrieving a list of all available permissions defined in the application. These permissions are loaded from a configuration file and returned in a structured format, complete with error handling and a localized success message.

🔹 **Key Functionalities**

1-**Fetch Permissions from Configuration**  
The method loads a set of predefined permissions from a configuration file (likely trans\_ar.premissions). These are typically stored as key-value pairs where the key is the internal identifier and the value is the human-readable or translated name.

2-**Structure the Permissions List**  
Each permission is assigned a unique incremental ID and organized into an array of items containing the ID, key, and value. This format ensures consistency and clarity when consumed by frontends or other services.

3-**Return Data with Localization**  
After formatting, the permissions list is returned with a structured response and a localized success message: تم الحصول على جميع الصلاحيات.

4-**Error Handling**  
If any unexpected error occurs during execution (e.g., the config file is missing or misconfigured), the method catches the exception and returns an error response. This response includes a 422 status code and the exception message to aid in debugging.

**30. Representative Controller**

**30.1 all() Method**

🔹 **Overview**  
The all() method is responsible for retrieving a filtered and optionally paginated list of representatives (Client entries with type = 1). It incorporates role-based access control, multiple filtering options (like city, area, and search), and localization for structured responses and messages.

🔹 **Key Functionalities**

1-**Identify Authenticated User**  
The method begins by retrieving the currently authenticated user from the request context. This user's role determines the scope of data they can access.

2-**Initialize Base Query**  
A base query is created to fetch clients where the type equals 1 (representatives). This query will be modified based on filters and user roles.

3-**Apply City Filter**  
If a city\_id is provided in the request, the method filters representatives to only those who belong to the specified city.

4-**Apply Area Filter**  
If an area\_id is provided, the method retrieves the IDs of all admins linked to that area, then filters representatives based on those admin IDs.

5-**Role-Based Filtering**

* For users with role ID 6 (Investors), the method verifies the investor's identity using their email and phone number, and ensures they are approved (status = 1). If valid, the data is filtered by the investor’s ID. Otherwise, an unauthorized error is returned.
* For users with role ID 17 (Admins), the method ensures that only representatives linked to that specific admin are shown. If a search keyword is provided, it filters by matching the username or phone.

**30.2 returnPaginatedData() Method**

🔹 **Overview**  
The returnPaginatedData() method is a private utility responsible for structuring a paginated response for a collection of representative records. It enhances the API response by including detailed pagination metadata and transforming the data using a resource class for consistency.

🔹 **Key Functionalities**

1-**Extract Pagination Metadata**  
The method retrieves essential pagination details from the paginated $data object, including:

* Total number of items.
* Number of items per page.
* Current page number.
* Total number of pages.

These details are organized into a pagination array to be returned with the data.

2-**Transform Data Using Resource Class**  
The paginated representative data is transformed using the RepresentiveResource collection. This ensures the data returned is standardized and formatted as per API design requirements.

3-**Return Structured Response**  
The method returns a formatted JSON response containing:

* The transformed representative data.
* The pagination metadata.
* A localized success message (api.representative\_all).

**30.3 get() Method**

🔹 **Overview**  
The get() method is responsible for retrieving a specific representative by their ID. It includes role-based access control, ensuring that certain users can only access data they’re authorized to see. The method returns a structured response with appropriate localization and error handling.

🔹 **Key Functionalities**

1-**Role-Based Access Control (Admin Restriction)**  
If the authenticated user has a specific role (e.g., Admin with role\_id == 7), the method restricts the representative query to only those associated with that admin. If no matching representative is found under this condition, a 404 error response is returned with a localized message (api.representative\_not\_found).

2-**Retrieve Representative by ID**  
Regardless of user role, the method attempts to find the representative based on the provided representative\_id and ensures the type is 1. If the representative is not found, it returns a 404 error with the same localized message (api.representative\_not\_found).

3-**Return Transformed Representative Data**  
If a valid representative is retrieved, it is transformed using the RepresentiveResource to ensure consistent API formatting. The data is then returned along with a localized success message (api.representative\_get).

4-**Error Handling**  
Any unexpected error or exception encountered during execution is caught using a try-catch block. If such an error occurs, a 422 error response is returned along with the exception message and a localized fallback message (api.error\_happened).

30.4 add() Method

🔹 **Overview**  
The add() method is responsible for creating a new representative (type 1 client). It validates the request, handles optional image uploads, ensures the provided admin (if any) has the correct role, and saves the representative to the database. A structured and localized response is returned, with full error handling included.

🔹 **Key Functionalities**

1-**Image Handling (Optional)**  
If an image file is provided in the request, it is processed using a dedicated file-handling method. The resulting image path is stored with the representative's data.

2-**Admin Role Validation (if admin\_id provided)**  
When an admin\_id is present in the request, the method checks if the associated user exists and verifies that they hold the correct role (role ID 7). If the role does not match, a 403 error response is returned with a relevant message indicating that the user is not a valid admin.

3-**Representative Creation**  
Based on the presence or absence of an admin\_id, the method creates a new representative with all the necessary details:

* Personal information (username, phone, email)
* Security credentials (hashed password)
* Additional data (image, investor, plate ID, status)

The new record is saved to the database accordingly.

4-**Success Response**  
After successfully creating the representative, the method returns the structured data wrapped using the RepresentiveResource along with a localized success message (api.representative\_add).

5-**Error Handling**  
If an exception or unexpected error occurs at any step, the method captures it using a try-catch block. A 422 error response is returned containing the exception message and a localized fallback message (api.error\_happened).

**30.5 update() Method**

🔹 **Overview**  
The update() method updates the data of a specific representative (type 1 client) by ID. It supports optional image uploads, validates the admin role if provided, and safely updates all fields with fallback defaults if not present. Proper error handling ensures robustness and clarity in response.

🔹 **Key Functionalities**

1-**Find and Validate Representative**  
The method retrieves the representative by ID and confirms that the record belongs to a client of type 1. If not found, it returns a 404 error with a localized message (api.representative\_not\_found).

2-**Image Upload and Replacement (Optional)**  
If a new image file is provided:

* It’s processed using the handleFile() method.
* If the representative already has an image, the old image is deleted from the storage (using the File facade and base\_path()).

This ensures storage cleanliness and avoids orphaned files.

3-**Admin Role Validation (Optional)**  
If an admin\_id is passed in the request:

* The method checks that the corresponding user exists and has the correct role (role ID 7).
* If the admin is invalid, a 403 error is returned with a custom message.

4-**Field-by-Field Update with Defaults**  
The method updates the representative’s data using the provided request fields. If a field is not present in the request, it retains the current value from the database. This applies to:

* Basic info: username, phone, email
* Secure field: password (always rehashed if given)
* Optional info: image, investor\_id, plate\_id, status
* Admin ID: Only updated if provided, otherwise preserved

5-**Success Response**  
Upon successful update, a structured JSON response is returned containing:

* The updated representative, wrapped in RepresentiveResource
* A localized success message (api.representative\_update)

6-**Error Handling**  
All exceptions are caught using a try-catch block. If any unexpected issue occurs (e.g., invalid image file, DB error), a 422 error is returned with the exception message appended to a fallback localization string (api.error\_happened).

**30.6 destroy() Method**

**Overview**

This method handles the deletion of a representative (client with a specific type). It performs several checks before actually deleting the representative, ensuring only authorized users can perform the action.

### Key Steps:

1-**Finding the Representative:**  
The method first tries to find the representative using the provided ID. If no representative is found, an error is returned indicating that the representative does not exist.

2-**Authorization Check:**  
If the user is an admin (role ID 7), the method checks if the representative belongs to that admin. If the admin does not own the representative, an error is returned to prevent unauthorized deletion.

3-**Image Deletion (If Exists):**  
If the representative has an associated image, the method checks if the image exists in the file system and deletes it.

4-**Delete the Representative:**  
If all checks pass, the representative is deleted from the database.

5-**Success Response:**  
If everything goes well, a success message is returned indicating the representative has been deleted.

6-**Error Handling:**  
If any issues arise during the process, the method catches the errors and returns an error message with the exception details.

**30.7 reservations() Method**

**Overview**

This method retrieves a list of reservations for a specific representative. It allows pagination if requested, providing the option to retrieve all reservations or just a specific set of results based on pagination.

### Key Steps:

1-**Finding the Representative:**  
The method first attempts to find the representative by their ID and ensures that the representative is of the correct type. If no representative is found, an error is returned indicating that the representative does not exist.

2-**Pagination Check:**  
If the request specifies pagination (paginate = 1), the method retrieves the reservations with pagination. The pagination details such as total records, records per page, current page, and total pages are included in the response.

3-**Fetching Reservations:**  
The method retrieves the reservations associated with the representative. If pagination is not requested, all reservations are retrieved in a simple list format.

4-**Success Response:**  
The reservations are then formatted using the ReservationResource collection, and the method returns the data, along with a success message indicating that the reservations were successfully retrieved.

5- **Error Handling:**  
If any errors occur during the process (such as failing to find the representative or encountering issues with the database), the method catches the exceptions and returns an error response with the details of the exception.

**30.8 highest\_reviews() Method**

**Overview**

This method retrieves the representatives with the highest average review scores, ensuring that only representatives who have reviews with a score are included. The results are sorted by the highest average review score and support pagination.

### Key Steps:

1-**Query for Representatives with Reviews:**  
The method starts by querying the Client model, specifically looking for representatives who have associated reviews. The query filters for reviews that have a non-null score.

2-**Calculating Average Review Scores:**  
The method uses the withAvg() function to calculate the average score of the reviews for each representative. This helps to determine which representatives have the highest ratings.

3-**Sorting by Average Score:**  
The representatives are then ordered by their average review score in descending order, so that the top-rated representatives are shown first.

4-**Pagination of Results:**  
The results are paginated, with 10 representatives per page. This helps manage large datasets and ensures efficient retrieval of data.

5-**Success Response:**  
The top-rated representatives are returned, formatted using the RepresentiveResource collection, along with a success message indicating that the highest-reviewed representatives have been successfully retrieved.

6-**Error Handling:**  
If any error occurs during the process (such as a failure in the database query or an issue with calculating averages), the method catches the exception and returns an error response with the exception's details.

**30.9** **lowest\_reviews() Method**

Overview

This method retrieves representatives with the lowest average review scores, ensuring that only those with reviews containing a score are included. The results are sorted by the lowest average review score and support pagination.

### Key Steps:

1-**Query for Representatives with Reviews:**  
The method begins by querying the Client model to fetch representatives who have associated reviews, specifically looking for reviews with non-null scores.

2-**Calculating Average Review Scores:**  
The method uses the withAvg() function to calculate the average review score for each representative. This allows the method to determine the lowest-rated representatives based on their review scores.

3-**Sorting by Average Score (Ascending):**  
The representatives are ordered by their average review score in ascending order, meaning the representatives with the lowest average scores will be listed first.

4-**Pagination of Results:**  
The results are paginated, with 10 representatives per page, ensuring efficient data retrieval and managing large datasets.

5-**Success Response:**  
The lowest-rated representatives are returned, formatted using the RepresentiveResource collection, along with a success message indicating the representatives with the lowest reviews have been successfully retrieved.

6-**Error Handling:**  
In case of any error (e.g., failure during the database query or error in calculating averages), the method catches the exception and returns an error response with the exception's message.

**30.10 topRepresentativesBasedOnReservations() Method**

**Overview**

This method retrieves the top representatives based on the number of reservations they have, sorting them in descending order of their reservation count. It supports pagination to ensure efficient handling of large datasets.

### Key Steps:

1-**Query for Representatives with Reservations:**  
The method starts by querying the Client model, filtering representatives who have at least one reservation using whereHas('representative\_reservations'). This ensures that only representatives with associated reservations are considered.

2-**Counting Reservations:**  
The method uses the withCount('representative\_reservations') function to count how many reservations each representative has. This count is used to determine the ranking of representatives based on their reservation volume.

3-**Sorting by Reservation Count (Descending):**  
Representatives are sorted in descending order based on the reservation count (representative\_reservations\_count), meaning the representatives with the most reservations will appear first.

4-**Pagination of Results:**  
The results are paginated, showing 10 representatives per page to optimize data retrieval and display, especially when dealing with a large number of representatives.

5-**Success Response:**  
The top representatives based on reservations are returned, formatted using the RepresentiveResource collection, along with a success message indicating the retrieval was successful.

6-**Error Handling:**  
If an error occurs during the database query or any other part of the method (e.g., an issue with counting reservations), an exception is caught, and an error response with the exception's details is returned.

**30.11 getTopRepresentativesMonthly() Method**

**Overview**

This method retrieves the top-performing representatives based on their completed reservations for a specific month and year. It includes filtering by the reservation status and supports pagination for efficient handling of large datasets.

**Key Steps:**

1-**Querying for Representatives' Reservations:** The method uses the Client model and withCount to count two specific metrics related to reservations for each representative:

* **Finished Reservations (finished\_reservations\_count)**: This counts the number of reservations that have a status of 3 (finished) within the specified month and year.
* **Total Reservations (reservations\_count)**: This counts the total number of reservations (regardless of status) within the specified month and year.

2-**Filtering by Year and Month:** The whereYear and whereMonth conditions are used to filter reservations based on the provided year and month parameters from the request.

3-**Excluding Representatives with No Finished Reservations:** The having clause is used to filter out representatives who have zero finished reservations, ensuring only representatives with completed reservations are included.

4-**Sorting Representatives by Finished Reservations:** The representatives are sorted in descending order based on the count of finished reservations (finished\_reservations\_count), so the representatives with the most completed reservations appear at the top.

5-**Pagination of Results:** The results are paginated, showing 10 representatives per page. This ensures efficient data handling when dealing with a large number of representatives.

6-**Returning the Results:** The method returns a list of the top representatives along with their reservation counts. It uses the RepresentiveResource collection to format the data, and it also returns pagination information.

7-**Error Handling:** If an error occurs during the query or any other process, it is caught, and an error response with the exception's details is returned.

**31. Representative Details Controller**

**31.1 getDetails() Method**

**Overview**

This method retrieves the details of a specific representative based on their unique ID. It first checks if the representative exists, then fetches their details from a related table. If any of these resources are not found, appropriate error messages are returned.

### Key Steps:

1-**Retrieving the Representative:** The method begins by searching for a Client with the given representative\_id and ensures the client is of type 1. If no representative is found, it returns an error response indicating that the representative was not found.

2-**Fetching the Representative Details:** Once the representative is found, the method looks for the corresponding entry in the RepresentiveDetials model by using the representative\_id. If no details are found for the representative, an error response is returned indicating that the representative's details were not found.

3-**Returning the Data:** If both the representative and their details are found, the details are wrapped in a RepresentiveDetailsResource for proper data formatting. The method then returns the data along with a success message.

4-**Error Handling:** If any exception occurs while fetching the data, it is caught, and a detailed error message is returned to the client, including the exception’s message.

**31.2 addDetails() Method**

**Overview**

This method adds details for a specific representative, such as passport, license, and residence images, and stores them in the database. It first checks whether the representative exists and whether they already have details associated with them. If everything is valid, it saves the provided details and returns a success response.

### Key Steps:

1-**Checking Representative Existence:** The method first searches for the representative with the given representative\_id and ensures they are of type 1. If no representative is found, it returns an error response indicating that the representative was not found.

2-**Checking if Details Already Exist:** Next, the method checks if the representative already has details in the RepresentiveDetials table by searching for the client\_id. If the details already exist, it returns a 409 error with a message indicating that the details are already present.

3- **Handling File Uploads:** The method checks if the request contains any files (passport\_image, license\_image, and residence\_image). If any of these files are present, they are stored in the uploads/representatives directory in the public disk, and their paths are saved for later use.

4-**Creating Representative Details:** The details are merged with the other form data and stored in the RepresentiveDetials table. If the insertion is successful, the newly created representative details are wrapped in a RepresentiveDetailsResource for proper formatting.

5-**Returning the Data:** Once the representative details are successfully added, the method returns the newly created details along with a success message.

6-**Error Handling:** If any exception occurs while adding the details, it is caught, and an error response is returned, including the exception’s message.

32.3 **updateDetails() Method**

Overview

This method updates the details of a specific representative, including passport, license, and residence images, based on the provided representative\_id. It ensures that the representative and their details exist, handles file uploads for new images, and updates the database with the new or modified data.

**Key Steps:**

1-**Checking Representative Existence:** Similar to the addDetails() method, this method starts by checking if the representative with the given representative\_id exists and has a type of 1. If no such representative is found, it returns a 404 error.

2-**Checking Representative Details Existence:** Next, it checks if the representative already has details stored in the RepresentiveDetials table. If no details are found, it returns a 404 error.

3-**Handling File Updates:** The method retrieves the existing file paths for the passport, license, and residence images from the current representative's details.

If any new files are provided in the request, the method:

* Deletes the old files from storage using Storage::disk('public')->delete().
* Stores the new files in the uploads/representatives directory in the public disk.

4-**Filtering and Merging Data:** The method uses array\_filter() to filter out any null values from the request data before merging the updated file paths into the data array.

5-**Updating Representative Details:** Once the data is prepared, the method updates the representative’s details in the RepresentiveDetials table using the update() method.

6-**Returning the Updated Data:** If the update is successful, the method returns the updated details wrapped in a RepresentiveDetailsResource along with a success message.

7-**Error Handling:** If an error occurs at any point (e.g., during file deletion, file upload, or database update), it is caught, and an error response with the exception message is returned.

**32.4** **allRates() Method**

Overview

This method retrieves all rates associated with a specific representative, paginated for ease of browsing. The representative is identified by their representative\_id. The rates are ordered by the most recent, and a paginated response is returned.

### Key Steps:

1-**Retrieving Rates for a Specific Representative:** The method starts by querying the RepresentiveRate model, filtering for rates that belong to the given representative\_id. It uses the latest() method to ensure the results are ordered by the most recent rates first.

2-**Pagination:** The results are paginated, returning a maximum of 10 records per page. Pagination helps in managing large datasets efficiently and ensures that the API returns data in chunks that are easier to handle.

3-**Returning Paginated Rates:** The method then uses the RepresentiveRateResource resource to transform the collection of rates into the desired format. This is sent back as part of the data with a success message (\_\_('api.rate\_all')).

4-**Error Handling:** If an error occurs during the process (for example, an issue with the database query or pagination), the error is caught, and an error response is returned with the exception message.

**32.5 addRate() Method**

Overview

This method handles adding a new rate for a representative. It ensures that the representative exists in the database and creates a new rate record linked to that representative.

**Key Steps:**

1-**Validate Representative Existence:**

* The method checks whether a representative exists by querying the Client model for a representative\_id where the type is 1. If no such representative is found, an error response is returned with a 404 status and a corresponding message (representative\_not\_found).

2-**Create New Rate:**

* If the representative exists, a new rate is created for that representative using the RepresentiveRate model. The rate and type fields are taken from the incoming request ($request).

3-**Return Success Response:**

* Upon successfully creating the new rate, the method returns the rate data formatted with RepresentiveRateResource, along with a success message (rate\_add).

4-**Error Handling:**

* If any error occurs during the process (e.g., database insertion failure), the method catches the exception and returns an error response with the exception message and a 422 status code.

**32.6 allTypes() Method**

**Overview**

This method retrieves all available types for rates and returns them in a formatted response. It provides the types as an array with IDs and names, where the names are translated using language keys.

**Key Steps:**

1-**Data Preparation:**

* The method prepares a static array of rate types, each represented as an associative array with id and name. The name field uses localization to fetch the corresponding language key values (rate\_type1, rate\_type2, etc.).

2-**Return Data:**

* The method returns the data as a response, formatted into an array with a key types. A success message (rate\_types) is also included in the response.

3-**Error Handling:**

* If any error occurs during the process (e.g., unexpected issues), the method catches the exception and returns an error response with the exception message and a 422 status code.

**32.7 allDiscounts() Method**

**Overview**

The allDiscounts() method retrieves all available discounts for a specific representative and returns them in a formatted response. The discounts are paginated and returned with the discount details in a resource format.

**Key Steps:**

1-**Data Retrieval:**

* The method queries the RepresentiveDiscount model to fetch all the discounts associated with a given representative (identified by representative\_id).
* The discounts are ordered by the latest entries and paginated to ensure that only a limited number of results are returned per request.

2-**Return Data:**

* The retrieved discounts are wrapped in a resource collection (RepresentiveDiscountResource), providing a structured format for each discount.
* The data is returned as a JSON response with the key discounts containing the paginated discounts.

3-**Error Handling:**

* If any error occurs during the process (e.g., database issues or unexpected failures), the method catches the exception and returns an error response.
* The response includes a 422 status code along with the error message and the exception details.

**32.8 addDiscount() Method**

**Overview**

The addDiscount() method is responsible for adding a discount to a specific representative’s account. It checks the representative’s balance, applies the discount, and records the operation in the wallet if applicable. It also provides error handling for various scenarios.

**Key Steps:**

1-**Representative Verification:**

* The method first checks whether the representative exists by querying the Client model, ensuring that the representative is of type 1.
* If the representative is not found, an error response is returned with a 404 status code.

2-**Balance Deduction:**

* Once the representative is verified, the method deducts the discount value from the representative’s balance and saves the updated balance.

3-**Discount Creation:**

* The discount is then created in the RepresentiveDiscount model, associating the discount with the representative and saving it to the database.

4-**Wallet Operation (Optional):**

* The method checks if the application setting allows for wallet operations (via the verify\_wallet\_operations setting).
* If the setting is enabled, the method records the discount transaction in the wallet operations table, providing a detailed description of the operation.

5-**Return Data:**

* The method returns a success response with the created discount data, wrapped in the RepresentiveDiscountResource.

6-**Error Handling:**

* If any exception occurs during the process, it is caught, and a 422 error response is returned with the exception message and details.

**33. Review Controller**

**33.1 all() Method**

**Overview**

The all() method is designed to retrieve a list of reviews based on various filters and user roles. It allows for dynamic filtering by city and area, and applies role-based filtering for investors and admins. The method also includes pagination to handle large sets of data efficiently.

**Key Steps:**

1-**User Identification:**

* The method begins by identifying the currently authenticated user through the request, using the user() method to access user details.

2-**Review Query Initialization:**

* A query builder is initialized for the Review model to start fetching reviews from the database.

3-**Filter Application:**

* **City Filter:**
  + If the city\_id parameter is provided in the request, the method filters reviews based on the associated reservation’s city.
* **Area Filter:**
  + Similarly, if the area\_id parameter is provided, it filters reviews based on the associated reservation’s area.

4-**Role-Based Filtering:**

* **Investor Role:**
  + If the user has an investor role (role\_id == 6), the method checks if the investor exists and is active. If so, it applies additional filters to ensure that the reviews are associated with the investor’s representative.
  + If the investor is not found or is inactive, the method returns an error with a 403 status code (unauthorized).
* **Admin Roles:**
  + If the user has an admin role (either role\_id 17 or 7), the method ensures that the reviews belong to the admin's specific area.

5-**Pagination:**

* The method handles pagination by calling the handlePagination() method, ensuring that the results are appropriately paginated according to the user’s request.

6-**Error Handling:**

* If any error or exception occurs during the execution of the method (e.g., database issues or incorrect parameters), it is caught, and a 422 error response is returned with the exception message.

**33.2 handlePagination() Method**

**Overview**

The handlePagination() method is responsible for determining how to paginate or retrieve data based on the request parameters. It handles both paginated and non-paginated responses, ensuring flexibility in how review data is returned, based on the user's preference.

**Key Steps:**

1-**Check for Pagination Preference:**

* The method first checks if the paginate parameter is present in the request. This helps determine if the user wants paginated data or all data at once.

2-**Paginated Response:**

* If the paginate value is set to 1, the method retrieves the latest 10 records from the query, paginated. This ensures that only a subset of data (10 records per page) is returned.
* The data is then returned using the returnPaginatedData() method, which formats the response for pagination.

3-**Non-Paginated Response:**

* If the paginate value is set to 0, the method retrieves all records from the query without pagination.
* The data is returned using the returnData() method, and the results are formatted as a collection of ReviewResource items.

4-**Default Behavior (Paginated Response):**

* If no pagination preference is specified in the request, the method defaults to paginating the results (10 records per page), ensuring that a paginated response is returned by default.

5-**Return Data:**

* Depending on the pagination choice (or the default behavior), the method returns either a paginated response or a non-paginated collection of reviews.

**33.3 returnPaginatedData() Method**

**Overview**

The returnPaginatedData() method is responsible for formatting and returning the data along with pagination details in a structured format. It works specifically with paginated results to provide both the data and pagination metadata in the response.

**Key Steps:**

1-**Pagination Data Calculation:**

* The method begins by extracting the pagination details from the paginated data:
  + total: The total number of records available.
  + per\_page: The number of records shown per page.
  + current\_page: The current page number.
  + total\_pages: The total number of pages available based on the per\_page value.

2-**Prepare the Response Data:**

* The method then prepares a response containing two main pieces of data:
  + reviews: The collection of paginated reviews, formatted using the ReviewResource for consistent response structure.
  + pagination: An array containing the pagination details calculated earlier (total, per\_page, current\_page, and total\_pages).

3-**Return the Response:**

* The data, including both the reviews and the pagination information, is wrapped into a structured response.
* The success message api.review\_all is included to indicate that the reviews have been fetched successfully.

**33.4 destroy() Method**

**Overview**

The destroy() method is responsible for deleting a specific review from the database. It first checks if the review exists, and if so, it deletes it. Upon successful deletion, it returns a success message. If the review is not found, it returns an error response.

**Key Steps:**

1-**Find the Review:**

* The method attempts to find the review by its review\_id using the Review::find() method.
* If the review exists, it proceeds to the next step. If not, it returns an error response indicating that the review was not found.

2-**Delete the Review:**

* If the review is found, the delete() method is called to remove the review from the database.
* After successful deletion, the method returns a success response with a message confirming the deletion.

3-**Error Handling:**

* If the review cannot be found, an error message is returned indicating that the review does not exist.
* In case of any unexpected issues (e.g., database errors), the method catches the exception and returns a generic error message along with the exception details.

**33.5 update\_represnetative() Method**

**Overview**

The update\_represnetative() method is designed to update the representative\_id field in the reviews table. It iterates through all the reviews, retrieves the associated reservation, and updates the review's representative\_id based on the reservation's representative\_id. This ensures that all reviews are properly linked to the correct representative based on the reservation.

**Key Steps:**

1-**Fetch All Reviews:**

* The method retrieves all reviews using Review::get(), which fetches all records from the reviews table.

2-**Iterate Through Reviews:**

* It loops through each review and finds the corresponding reservation using the reservation\_id from the review.
* If the reservation exists, the method proceeds to update the review's representative\_id field to match the representative\_id from the reservation.

3-**Count Updates:**

* A counter is used to track how many reviews are updated with a new representative\_id. After the loop, the method echoes the total count of updated reviews.

4-**Error Handling:**

* If any unexpected issues arise during the process (e.g., database errors), the method catches the exception and returns an error response with a 422 status code and the exception details.

**34. Reward Controller**

**34.1 all() Method**

**Overview**

The all() method is responsible for retrieving a list of rewards. It can either return the data in a paginated format or as a complete list, based on the user's request. It uses conditional logic to check if the user has specified a preference for pagination or not. If no preference is provided, it defaults to paginated results.

**Key Steps:**

1-**Check for Pagination Request:**

* The method first checks if the paginate parameter is present in the request. It uses this parameter to determine whether to return a paginated list of rewards or a complete list.
* If the paginate parameter is set to 1, it retrieves the rewards with pagination (10 items per page).
* If the paginate parameter is set to 0, it retrieves all rewards without pagination.

2-**Retrieve Rewards:**

* If pagination is enabled (paginate == 1), the method fetches the latest rewards with the pagination feature, including total count, current page, and total pages.
* If pagination is disabled (paginate == 0), it fetches all the rewards without pagination.

3-**Return Data:**

* If pagination is enabled, the method returns the rewards along with the pagination data, which includes:
  + Total number of rewards
  + Number of rewards per page
  + Current page number
  + Total number of pages
* If pagination is disabled, it simply returns the rewards in a non-paginated format.

4-**Error Handling:**

* If an error occurs during the data retrieval process (such as a database issue or an unexpected exception), the method catches the error and returns a 422 status code along with the error message.

**35. Role Controller**

**35.1 all() Method**

Overview

The all() method is responsible for fetching a list of roles from the database. It includes the ability to paginate the results, filter out specific roles, or return all available roles based on user input in the request. The method checks for various query parameters to determine the appropriate filtering and pagination logic.

**Key Steps:**

1-**Check for Pagination Request:**

* The method first checks if the paginate parameter is present in the request.
* If the paginate parameter is set to 1, it retrieves the roles with pagination (10 roles per page).
* If the paginate parameter is set to 0, it retrieves all roles without pagination.

2-**Additional Filtering (Role Exclusion):**

* If the role parameter is set to 0, the method applies additional filters to exclude specific roles (roles with IDs 1, 6, 7, and 8).

3-**Retrieve Roles:**

* Depending on the pagination setting, the method either fetches paginated results or all available roles based on the criteria:
  + Paginated Data: Includes the total count, current page, total pages, and roles per page.
  + Non-Paginated Data: Returns all roles without pagination, depending on the given filtering conditions.

4-**Return Data:**

* The method returns a structured response:
  + If pagination is applied, it returns the roles along with the pagination details.
  + If pagination is not applied, it returns the full list of roles based on the filtering.

5-**Error Handling:**

* If any error occurs during the process (e.g., database issues or invalid inputs), the method catches the exception and returns an error response with a 422 status code and the exception message.

**35.2 get() Method**

**Overview**

The get() method is responsible for retrieving a specific role by its ID from the database. It checks if the role exists, and if so, it returns the role's data in a structured response. If the role is not found, it returns an error response indicating the role could not be located.

**Key Steps:**

1-**Fetch Role by ID:**

* The method attempts to find a role by its ID from the database using the Role::find() method.

2-**Check for Role Existence:**

* If the role is found, the method proceeds to return the role's details in the response using a custom resource (RoleResource).
* If the role does not exist (i.e., it returns null), the method returns a 404 Not Found error indicating the role could not be found.

3-**Return Data:**

* If the role is found, the method returns the role data wrapped in a custom resource format with a success message.
* If the role is not found, it returns an error message with a 404 status code indicating the role was not found.

4-**Error Handling:**

* In the event of an exception or any issues during the process (such as database errors), the method catches the exception and returns an error response with a 422 status code and the exception message.

**35.3 add() Method**

**Overview**

The add() method is used to add a new role to the database along with its associated permissions. The method takes in a StoreRoleRequest containing the role's name and the permissions assigned to the role, processes the permissions, and stores the role information in the database. Additionally, it supports storing permissions in both English and Arabic.

**Key Steps:**

1-**Initialize Data Arrays:**

* The method initializes an empty array $data for storing permission keys.
* It also prepares an empty array $permission\_ar to store Arabic permissions.

2-**Retrieve Available Permissions:**

* It loads the available permissions from the configuration file config('trans\_ar.premissions') and stores them in the $permissions array.

3-**Process Permissions:**

* The method iterates over the incoming permissions from the request ($request->permissions).
* For each permission in the request, it checks whether it exists in the $permissions array. If found, the corresponding Arabic permission is added to the $permission\_ar array.

4-**Convert Permissions to JSON:**

* The permissions in Arabic ($permission\_ar) are encoded into JSON format using json\_encode, with the option JSON\_UNESCAPED\_UNICODE to ensure proper encoding of Arabic characters.

5-**Insert Role Data into the Database:**

* The method inserts the new role data into the roles table, including the role's name and both the English and Arabic permissions (as JSON data).
* This is done using a database query via DB::table('roles')->insert().

6-**Return Success Response:**

* If the role is added successfully, the method returns a success response with a message indicating that the role was successfully added.

7-**Error Handling:**

* If an error occurs during any part of the process (such as a database error or permission handling issue), the method catches the exception and returns a 422 error with the exception's message.

**35.4 update() Method**

**Overview**

The update() method is responsible for updating an existing role in the database based on the provided ID. The method checks if the role exists, validates the incoming data (such as permissions), and updates the role's name and permissions (both English and Arabic). It returns an updated role resource if the update is successful, or an error message if the role is not found or an error occurs.

**Key Steps:**

1-**Role Lookup:**

* The method attempts to find the role by its ID using Role::find($id).
* If the role does not exist, the method returns a 404 error response indicating that the role could not be found.

2-**Permissions Data Preparation:**

* The method retrieves the available permissions from the configuration file config('trans\_ar.premissions') and stores them in the $permissions array.
* It also prepares an empty array $permission\_ar to store the Arabic permissions.

3-**Process Incoming Permissions:**

* The method iterates through the permissions provided in the request ($request->permissions).
* For each permission in the request, it checks whether the permission exists in the available permissions list ($permissions). If it does, the corresponding Arabic permission is added to the $permission\_ar array.

4-**Update Role Information:**

* The role is updated with the new name, the English permissions (permissions\_en), and the Arabic permissions (permissions\_ar).
* The permissions are encoded in JSON format before storing them in the database (json\_encode).

5-**Return Updated Role Resource:**

* If the role is successfully updated, the method returns the updated role information wrapped in a RoleResource object.
* A success message (api.role\_update) is returned along with the updated role.

6-**Error Handling:**

* If an exception is thrown during the update process (e.g., due to a database issue or invalid data), the method catches the exception and returns an error response with a 422 status code and the exception message.

**35.5** **destroy() Method**

**Overview**

The destroy() method is responsible for deleting a role from the database based on the provided role ID. The method ensures that the role exists before attempting deletion and returns an appropriate success or error response depending on whether the operation is successful.

**Key Steps:**

1-**Role Lookup:**

* The method attempts to find the role by its ID using Role::find($id).
* If the role is not found, it returns a 404 error indicating that the role does not exist.

2-**Role Deletion:**

* If the role exists, it is deleted using the delete() method on the role instance.
* After deletion, the method returns a success response indicating that the role has been successfully deleted, with a message (api.role\_delete).

3-**Error Handling:**

* If an exception is thrown during the deletion process (for example, due to a database issue), the method catches the exception and returns an error response with a 422 status code along with the exception message.

**35.6 permissions() Method**

The permissions() method is responsible for retrieving a list of permissions from the configuration file and returning them in a structured format. It handles any exceptions that may occur during the retrieval process and ensures a proper response is provided.

**Key Steps:**

1-**Retrieve Permissions:**

* The method retrieves the permissions from the trans\_ar.premissions configuration, which likely holds a mapping of permission keys and their corresponding values.
* It then loops through each permission and structures the data into a key-value pair format.

2-**Format Data:**

* Each permission is structured in an array with two keys: key (the permission identifier) and value (the permission's description or label).
* The permissions are then organized into an array, $data, which will be returned as part of the response.

3-**Return Permissions:**

* After formatting the permissions, the method returns the permissions data within a structured API response. The response includes the key 'permissions' for the data and a message 'تم الحصول على جميع الصلاحيات' indicating that all permissions have been retrieved successfully.

4-**Error Handling:**

* If any exception occurs during the process (for example, due to missing configuration or data issues), the method catches the exception and returns an error response. The error response includes a 422 status code and the exception message.

**36. ScheduledNotifications Controller**

**36.1 all() Method**

**Overview**

The all() method is designed to retrieve a collection of notifications, either with or without pagination, based on the query parameter provided by the user. It processes the request and returns either a paginated or non-paginated response, depending on the user's preferences.

**Key Steps:**

1-**Check for Pagination Preference:**

* The method checks if the user has specified a pagination preference through the paginate parameter in the request.
* It first checks if the paginate parameter is set to 1, which indicates that the user wants paginated data.
* If the paginate parameter is set to 0, the method will return all notifications without pagination.
* If the paginate parameter is not provided or does not match 0 or 1, it defaults to paginated results.

2-**Retrieve Notifications:**

* If pagination is requested, the method fetches the latest notifications with pagination using ScheduledNotifications::latest()->paginate(10).
* If no pagination is needed (when paginate is set to 0), it fetches all notifications using Notification::latest()->get().

3-**Prepare Pagination Data:**

* When pagination is used, the method prepares a pagination array containing:
  + total: Total number of notifications.
  + per\_page: Number of notifications per page.
  + current\_page: Current page number.
  + total\_pages: Total number of pages.

4-**Return Data:**

* The method returns a JSON response that includes the notification data and pagination information if pagination is requested.
* If pagination is not needed, the method only returns the notifications without the pagination data.
* The response is wrapped with a success message (\_\_('api.notification\_all')) indicating that the operation was successful.

5-**Error Handling:**

* If any error occurs during the process (such as issues with database retrieval or incorrect data), the method catches the exception and returns an error response with a 422 status code and the exception message.

**36.2 get() Method**

**Overview**

The get() method is responsible for retrieving a single notification based on its ID. If the notification exists, it returns the data; otherwise, it handles the error gracefully.

**Key Steps:**

1-**Find Notification by ID:**

* The method attempts to find a notification by its ID using the ScheduledNotifications::find($id) query.
* If the notification is not found (i.e., the result is null), it returns a 404 error response indicating that the notification could not be found.

2-**Return Data:**

* If the notification is successfully found, the method returns the notification data wrapped inside the response.
* The notification data is transformed using ScheduledNotificationsResource, which is a resource class, likely used for formatting the response data in a structured way.

3-**Return Success Message:**

* The response is accompanied by a success message (\_\_('api.notification\_get')), which will be translated appropriately based on the configured language.

4-**Error Handling:**

* If any exception occurs during the process (e.g., issues with the database query or internal errors), the method catches the exception and returns a 422 error with the exception message.

**36.3 add() Method**

**Overview**

The add() method is designed to handle the creation of multiple notifications based on a dynamic set of client IDs. It provides logic for notifying different client groups, such as those with reservations or packages, based on the IDs provided in the request.

**Key Steps:**

1-**Predefined General Case IDs:**

* An array of general case IDs ($generalCaseIds = [0, 1, 2, 3, 4, 5]) is defined to handle specific notification types for predefined groups of clients.

2-**Dynamic Extraction of Remaining IDs:**

* The method dynamically filters out the predefined IDs from the provided client\_ids in the request. This ensures that any client IDs that don't fall into the predefined categories are handled separately.

3-**Handling Notifications Based on Client Types:**

* The method checks whether certain client types exist in the request (using in\_array()). If a client type exists, it creates a new ScheduledNotifications instance and sets the notification type accordingly.
  + Each notification is then filled with the necessary data (excluding client\_ids) and saved to the database.

4-**Handling Remaining Client IDs Dynamically:**

* After handling the predefined client types, the method checks if there are any remaining client IDs that don't match the predefined ones.
* If such IDs are found, it queries the database for clients of type 0 and status 1 (active clients). It then loops through these clients, creating a notification for each and setting the appropriate client\_id.

5-**Saving Notifications:**

* For every notification, whether predefined or dynamically created for remaining clients, a new instance of ScheduledNotifications is created, filled with the request data, and saved to the database.

6-**Return Success or Error:**

* Once the notifications are successfully created, the method returns a success response with a message (via trans('dashboard.added\_successfully')).
* If any errors occur during the process, an exception is caught, and a 422 error with the error message is returned.

**36.4 getClientIdsByGroup() Method**

**Overview**

The getClientIdsByGroup() method is designed to retrieve a specific set of client IDs based on the given group number. It divides the clients into groups of a predefined size and allows fetching clients from any given group dynamically.

**Key Steps:**

1-**Defining Group Size:**

* A constant group size ($groupSize = 500) is defined, indicating how many clients are in each group. This allows efficient processing and fetching of client data in manageable chunks.

2-**Calculating Offset:**

* The method calculates the offset for the query based on the provided groupNumber. The offset determines which set of clients to fetch. For example, if the groupNumber is 1, the offset will be 0, which fetches the first 500 clients. If the groupNumber is 2, the offset will be 500, fetching the next 500 clients, and so on.

3-**Retrieving Client IDs:**

* The method then constructs a query on the Client model to get the clients with type = 0. It orders the results by the id field to ensure a consistent and predictable order.
* The skip() function is used to apply the calculated offset, and the take() function limits the results to the group size (500).
* Finally, it uses pluck('id') to retrieve only the id column of the clients and converts the result to an array using toArray().

4-**Return Client IDs:**

* The method returns the array of client IDs corresponding to the given group number.

**36.5 destroy() Method**

**Overview**

The destroy() method is used to delete a specific notification based on its unique identifier ($id). If the notification exists, it is deleted; otherwise, an error response is returned.

**Key Steps:**

1-**Finding the Notification:**

* The method attempts to find the ScheduledNotifications record using the provided $id. This is done using the find() method.
* If the notification is not found, a 404 error is returned with the message indicating the notification was not found.

2-**Deleting the Notification:**

* If the notification is found, the delete() method is called on the notification instance to remove it from the database.

3-**Returning Success or Error Response:**

* If the deletion is successful, a success message (api.notification\_delete) is returned using the returnSuccess() method.
* If any error occurs during the process (including failure to find the notification), the exception is caught, and a 422 error with the error message is returned.

**36.6 sendNotification() Method**

**Overview**

The sendNotification() method is used to send a notification to a specific client identified by their unique client\_id. It retrieves the client's device token and sends the notification via a predefined notification system.

**Key Steps:**

1-**Retrieving the Client:**

* The method uses the provided client\_id to search for the client in the Client model with the where('id', $client\_id) query.
* If the client exists (i.e., it is not null), the method proceeds to send the notification.

2-**Sending the Notification:**

* If the client exists, the send\_notification() function is called with the client’s device\_token and the notification details (title and body) from the $request.
* The notification is sent to the client's device using their device token.

3-**Returning True:**

* The method returns true after the notification is sent, indicating that the operation was successful, regardless of whether the client existed or not.

**37. Service Controller**

**37.1 all() Method**

**Overview**

The all() method is designed to retrieve a collection of services from the Service model based on various filtering criteria, including service type and pagination. The method handles requests based on the provided parameters, such as whether to filter services by type (main or additional) and whether to paginate the results.

**Key Steps:**

1-**Filter by Service Type (Main or Additional):**

* The method checks if the type parameter is provided in the request.
* If type == 1, it retrieves services of type 1 (main services), otherwise, it retrieves services of type 0 (additional services).
* If no type is provided, it retrieves all services without filtering by type.

2-**Pagination Handling:**

* If the paginate parameter is set to 1, the method paginates the results, retrieving 10 services per page and returning pagination data along with the services.
* If paginate is set to 0, it retrieves all active services (status = 1) without pagination.
* If no pagination option is provided, the method paginates the services by default and returns pagination details.

3-**Service Retrieval:**

* The method constructs the query using Service::latest() to retrieve services in descending order based on creation date.
* Depending on the filtering (type or pagination), it either returns a paginated result or a full list of active services.

4-**Pagination Data:**

* If pagination is enabled, the method returns the following data:
  + total: The total number of services.
  + per\_page: The number of services per page.
  + current\_page: The current page number.
  + total\_pages: The total number of pages.

5-**Response Handling:**

* Upon success, the method returns the list of services along with pagination details, formatted using the ServiceResource collection.
* If there is any error, it catches the exception and returns a response with an error message.

**37.2 get() Method**

**Overview**

The get() method is used to retrieve a specific service from the Service model based on the provided service\_id. It handles errors gracefully, returning appropriate responses in case the service is not found or an exception occurs during the process.

**Key Steps:**

1-**Service Retrieval:**

* The method attempts to fetch a service from the database using the service\_id passed in the request.
* It uses the Service::where('id', $service\_id)->first() query to find the service by its ID.

2-**Check if Service Exists:**

* If the service with the provided service\_id does not exist, the method returns an error response with a 404 status code and a message indicating that the service was not found.

3-**Return Service Data:**

* If the service is found, the method returns the service data using the ServiceResource to structure the response. This ensures the service data is returned in the desired format.

4-**Error Handling:**

* If any exception occurs during the service retrieval or processing, it is caught, and the method returns an error response with a 422 status code.
* The error message includes the exception details for debugging purposes.

**37.3 add() Method**

**Overview**

The add() method is used to create a new service based on the data provided in the StoreServiceRequest. It handles file uploads, service creation, and city-specific pricing for the service, with error handling to ensure smooth operation and graceful failure in case of issues.

**Key Steps:**

1-**Handle Image Upload (if any):**

* If an image is provided in the request, the method attempts to process the image using the handleFile() function. This function is responsible for saving the image and generating the appropriate file path.

2-**Service Creation:**

* The service is created using the data provided in the request, including:
  + title: The title of the service.
  + description: The description of the service.
  + image: The image file path, if an image is provided.
  + type: The type of the service.
  + status: The status of the service.
  + available: Availability of the service (defaults to 0 if not provided).

3-**Handle City-Specific Pricing (if applicable):**

* If city-specific pricing is provided in the request (city\_prices), the method processes the data. It:
  + Decodes the JSON strings into arrays.
  + Iterates over each city and its pricing information.
  + Creates a record in the CityServicesPrices table for each city and its associated pricing, linking the service with the city's price and optional previous cost.

4-**Return Success Response:**

* After successfully creating the service, the method returns a success response with the newly created service data structured using the ServiceResource. This ensures the response is formatted properly.

5-**Error Handling:**

* If any errors occur during the process, such as during file upload, database operations, or processing of city prices, the method catches the exception and returns an error response with a 422 status code.
* The error message includes the exception message to help with debugging.

**37.4 update() Method**

**Overview**

The update() method is designed to handle the update of an existing service. It ensures that the service information is updated based on the provided data and also manages the optional image upload and city-specific pricing updates. It also ensures proper error handling for the smooth execution of the update operation.

**Key Steps:**

1-**Find Service to Update:**

* The method attempts to find the service with the provided service\_id. If the service is not found, it returns a 404 error with a message indicating that the service does not exist.

2-**Handle Image Upload (if provided):**

* If an image is provided in the request, the method processes the image upload using the handleFile() function, which saves the image and generates the file path. If no image is provided, the existing image for the service is retained.

3-**Update Service Data:**

* The service data is updated with the provided request data, including:
  + title: The updated title, or the current title if no new value is provided.
  + description: The updated description, or the current description if no new value is provided.
  + image: The new image file path (if provided) or the existing image.
  + type: The updated type.
  + status: The updated status.
  + available: The updated availability, with a default of the current availability if no new value is provided.

4-**Update City-Specific Pricing (if applicable):**

* If city-specific pricing is provided in the request (city\_prices), the method first deletes any existing city-specific pricing records for the service.
* Then, it processes the city prices:
  + Decodes the city price data from JSON format.
  + Iterates over the city prices and updates the CityServicesPrices table, creating new records for each city's pricing information associated with the service.

5-**Return Success Response:**

* After the successful update of the service, the method returns a success response with the updated service data, structured using the ServiceResource.

6-**Error Handling:**

* If any exception occurs during the update process (e.g., database errors, file upload issues), the method catches the error and returns a 422 status code with the error message. The error message includes the exception details to help identify the cause of the issue.

**37.5 destroy() Method**

**Overview**

The destroy() method is used to delete a service from the system, ensuring that both the service data and any associated media (e.g., images) are properly handled before deletion. This method incorporates error handling and returns an appropriate response depending on the outcome of the operation.

**Key Steps:**

1-**Find Service to Delete:**

* The method attempts to find the service with the provided service\_id. If the service is not found, it returns a 404 error with a message indicating that the service was not found.

2-**Delete Associated Image (if present):**

* If the service has an associated image, it uses Laravel's Storage facade to delete the image from the public storage disk. This ensures that no orphaned files are left behind after deleting the service.

3-**Delete the Service:**

* After handling any associated media, the service is deleted from the database using the delete() method. This removes the service record and its associated data from the database.

4-**Return Success Response:**

* If the service is successfully deleted, the method returns a success response confirming that the service was deleted. This is accompanied by a message indicating the successful deletion.

5-**Error Handling:**

* If any exception occurs during the deletion process (e.g., database errors, file deletion issues), the method catches the error and returns a 422 status code with the error message. The error message includes the exception details to help identify the issue.

**38. Setting Controller**

**38.1 all() Method**

**Overview**

The all() method is used to retrieve the first application setting from the database and return it as a response. It ensures that the requested setting is formatted correctly and handles any potential errors gracefully.

**Key Steps:**

1-**Retrieve Setting:**

* The method attempts to retrieve the first setting from the Application model. It is assumed that there is only one setting record in the database. If found, this setting is returned in the response. If not found, it will return a null setting, but the structure will still be returned correctly.

2-**Return the Setting:**

* The setting is wrapped inside a SettingResource, which ensures that it is formatted correctly before being returned to the client. The returnData() method is used to wrap the response and include a data key containing the setting.

3-**Error Handling:**

* If any exception occurs during the process (such as issues with retrieving the setting from the database), the method will catch the exception and return an error response with a 422 status code. The error message will contain the details of the exception to assist with troubleshooting.

**38.2 get() Method**

The get() method is designed to retrieve a specific application setting by its ID and return it as a structured response. It also includes error handling to ensure that appropriate feedback is provided in case of issues.

**Key Steps:**

1-**Retrieve Setting by ID:**

* The method attempts to find the Application setting by the provided ID using the find() method. If the setting exists, it is returned in the response.

2-**Check if Setting Exists:**

* If the setting does not exist (i.e., the find() method returns null), a 404 Not Found error response is returned with the message Setting not found.

3-**Return Setting:**

* If the setting is successfully found, it is passed to a SettingResource for proper formatting. The response includes the setting in a structured data key, and the operation is accompanied by a success message.

4-**Error Handling:**

* If an error occurs during the execution (for example, due to a database issue or other unforeseen circumstances), the method will catch the exception and return an error response. The error message will be included in the response with a 422 status code.

**38.3 update() Method**

**Overview**

The update() method is designed to handle the updating of an existing application setting. It accepts a request containing new data, updates the relevant fields, handles file uploads (such as logos and images), and returns a structured response with the updated setting or an error message.

**Key Steps:**

1-**Retrieve Setting by ID**:

The method begins by attempting to find the existing Application setting using the provided id. It uses the find() method to retrieve the setting from the database.

2-**Check if Setting Exists**:

If no setting is found (i.e., the find() method returns null), the method returns a 404 Not Found error response with the message "Setting not found."

3-**File Handling (Logo, Pop-up Images)**:

* The method checks if the request contains a file for the logo, pop\_up\_image, or pop\_up\_image\_2.
* If a file is provided, it deletes the old file (if one exists) and uploads the new file to storage using the uploadImage() method.
* This ensures that only the latest image is stored and displayed.

4-**Update Other Fields**:

The method updates various fields in the setting, including:

* **working\_days**: Converted into an appropriate format using the convertWorkingDays() method.
* **payment\_types\_buraydah** and **payment\_types\_riyadh**: Processed and updated using the getPaymentType() method.
* Other fields such as title, description, email, phone, etc., are updated based on the request data.

5-**Return Updated Setting**:

Once the setting is updated, the method returns a structured response with a success message and the updated setting.

6-**Error Handling**:

If an error occurs during the execution of the update (e.g., an issue with the file upload or an unexpected failure), the method will return an error message in the response with an appropriate status code.

**38.4 destroy() Method**

**Overview**

The destroy() method is designed to delete an application setting by its ID. It checks if the setting exists, deletes its associated logo file (if it exists), and then removes the setting from the database. The method handles errors and returns appropriate feedback on the success or failure of the operation.

### Key Steps:

1-**Retrieve Setting by ID**:

The method attempts to retrieve the application setting using the provided id. It uses the find() method to check if the setting exists in the database.

2-**Check if Setting Exists**:

If the setting does not exist (i.e., the find() method returns null), the method returns a 404 Not Found error response with the message "Setting not found."

3-**Delete Associated Logo**:

If the setting has a logo file associated with it, the method deletes the logo file from storage using the Storage::disk('public')->delete() method. This ensures that no orphaned files remain in storage.

4-**Delete the Setting**:

Once the logo is deleted (if it exists), the setting itself is removed from the database using the delete() method.

5-**Return Success Message**:

After successfully deleting the setting, the method returns a success message with a 200 OK response.

6-**Error Handling**:

If an error occurs during the process (e.g., file deletion failure, database issue, or other exceptions), the method will catch the exception and return an error response with the appropriate status code and error message.

**38.5 convertWorkingDays() Method**

**Overview**

The convertWorkingDays() method is designed to convert a list of full day names (such as "Monday", "Tuesday", etc.) into their respective abbreviations (e.g., "Mon", "Tue"). It returns a comma-separated string of the abbreviated day names.

### Key Steps:

1-**Define Day Abbreviations**:

The method starts by defining an array that maps the full day names (e.g., "saturday", "monday") to their respective abbreviations (e.g., "sat", "mon").

2-**Map Days to Abbreviations**:

It then processes the provided list of days by mapping each full day name to its corresponding abbreviation using the defined mapping. If a day is not found in the mapping array, the full name is kept as is.

3-**Join Abbreviated Days into a String**:

After converting the days into their abbreviated forms, the method joins the resulting array into a single, comma-separated string.

4-**Return Abbreviated Days**:

The method returns the comma-separated string of abbreviated days, which represents the working days in a shortened format.

**39. Shift Controller**

**39.1 all() Method**

Overview

The all() method is responsible for retrieving a list of shifts based on various filtering and pagination options. It allows for retrieving data with or without pagination, and can filter by area ID if specified. The method returns the shifts data in a structured format, including pagination details when applicable.

**Key Steps:**

1-**Initialize Shift Query**:

The method starts by creating a query for the Shift model, preparing it to be filtered based on the request data.

2-**Filter by Area ID**:

If the area\_id parameter is provided in the request, the method filters the shifts by the specified area\_id.

3-**Handle Pagination**:

* **Paginate with Custom Settings**: If the paginate parameter is set to 1, the method orders the shifts by the day of the week in a custom order (Saturday through Friday) and then paginates the results with 100 entries per page.
* **Paginate with Default Settings**: If the paginate parameter is set to 0, the method retrieves all shifts without pagination.
* **Default Pagination**: If no pagination settings are specified, the method applies default pagination, returning 10 entries per page.

4-**Return Shifts Data**:

* If pagination is used, the method includes pagination metadata in the response, such as total results, current page, and total pages.
* The shifts are returned in the response using the ShiftResource::collection() method for proper data formatting.

5-**Error Handling**:

If any errors occur during the execution (e.g., database issues), the method catches the exception and returns a structured error response with a 422 status code and the error message.

**39.2 add() Method**

**Overview**

The add() method is designed to create a new shift and assign it to specific clients, with the option to provide the details via a form request. It handles the creation of a shift record and can associate clients with the shift if client IDs are provided.

### Key Steps:

1-**Create New Shift**:

The method begins by creating a new shift record using the data provided in the form request. The required fields for creating the shift are day, from, to, and area\_id.

2-**Assign Clients to Shift**:

If the client\_ids parameter is provided in the request, the method assigns the specified clients to the newly created shift. It uses the sync() method to ensure that the appropriate clients are linked to the shift.

3-**Return Shift Data**:

Upon successful creation, the method returns a structured response that includes the details of the newly created shift. The response uses a ShiftResource for proper formatting of the shift data.

4-**Error Handling**:

If any errors occur during the execution (e.g., database issues or validation failures), the method catches the exception and returns a structured error response with a 422 status code. The error message is included in the response.

**39.3 get() Method**

**Overview**

The get() method is designed to retrieve a specific shift by its ID. It checks if the shift exists in the database, returns the shift data if found, and handles any errors that may occur during the process.

### Key Steps:

1-**Retrieve Shift by ID**:

The method begins by attempting to retrieve the shift using the provided id. If the shift exists in the database, it is fetched successfully.

2-**Check if Shift Exists**:

If the shift is found, it is passed through a ShiftResource for proper formatting and returned in the response. If the shift does not exist, a 404 Not Found error response is returned, indicating that the shift could not be found.

3-**Return Shift Data**:

If the shift is found and successfully retrieved, the response includes the shift data, structured according to the ShiftResource.

4-**Error Handling**:

If an error occurs during the execution (e.g., database issues or other unforeseen circumstances), the method will catch the exception and return an error response. The error message is included in the response with a 422 status code.

**39.4 update() Method**

**Overview**

The update() method is designed to update the details of an existing shift by its ID. It checks if the shift exists, updates its details with the provided request data, and handles any errors that may occur during the process.

### Key Steps:

1-**Retrieve Shift by ID**:

The method begins by attempting to find the shift using the provided id. If the shift exists in the database, it proceeds with updating the shift details.

2-**Update Shift Details**:

If the shift exists, the method updates the shift's attributes (day, from, to, area\_id) using the data provided in the request. If any attribute is not provided, the current value is retained.

3-**Assign Representatives to Shift**:

If the client\_ids field is provided and contains client IDs, the method assigns the clients to the shift by syncing the client\_ids with the shift.

4-**Return Updated Shift Data**:

Once the shift is updated successfully, the response includes the updated shift data, formatted using ShiftResource.

5-**Error Handling**:

If an error occurs (such as the shift not being found or a database issue), the method catches the exception and returns an error response. The error message is included in the response with a 422 status code.

**39.5 destroy() Method**

**Overview**

The destroy() method is designed to delete an existing shift by its ID. It checks if the shift exists, deletes it, and handles any errors that may occur during the process.

### Key Steps:

1-**Retrieve Shift by ID**:

The method begins by attempting to find the shift using the provided id. If the shift exists, it proceeds with the deletion.

2-**Delete Shift**:

If the shift exists, the method deletes the shift from the database.

3- **Return Success Message**:

Once the shift is deleted successfully, the response includes a success message indicating that the shift has been deleted.

4-**Error Handling**:

If an error occurs (such as the shift not being found or a database issue), the method catches the exception and returns an error response. The error message is included in the response with a 422 status code.

**40. Slider Controller**

**40.1 all() Method**

**Overview**

The all() method is responsible for retrieving a list of sliders, with optional filtering by city and support for both paginated and non-paginated results. It ensures a flexible response structure based on the request parameters.

**Key Steps:**

1-**Filter by City (Optional)**:  
If the request includes a city\_id, the method filters the sliders to include only those that match the specified city.

2-**Pagination Handling**:  
The method checks whether pagination is requested:

* If paginate is set to 1, it returns a paginated list of sliders with additional pagination details.
* If paginate is set to 0, it retrieves all matching sliders without pagination.
* If no paginate value is provided, it defaults to paginated results with a standard page size.

3-**Return Sliders**:  
Sliders are returned using a resource collection for consistent formatting, either as a simple list or accompanied by pagination data.

4-**Error Handling**:  
If an exception is thrown during the process (e.g., due to a database issue), the method catches the error and returns an appropriate error message with a 422 status code.

**40.2 add() Method**

**Overview**

The add() method handles the creation of a new slider, including image upload, data validation, and structured API response formatting.

### Key Steps:

1-**Image Upload** (Optional):  
If an image file is included in the request, it is uploaded and the path is stored using a helper method responsible for file handling.

2-**Create Slider**:  
A new slider is created using validated data, which includes fields such as title, description, image, city\_id, and status.

3-**Return Created Slider**:  
After creation, the slider is wrapped in a resource for proper formatting and returned with a success message and a structured key.

4-**Error Handling**:  
If an exception occurs at any stage (e.g., file upload failure, DB error), the method captures it and returns an error message with a 422 status code.

**40.3 get() Method**

**Overview**

The get() method retrieves a single slider by its ID and returns it in a formatted API response.

### Key Steps:

1-**Find Slider by ID**:  
The method attempts to locate the slider using the provided ID.

2-**Check Existence**:  
If the slider does not exist, it returns a 404 error response with a localized message indicating that the slider was not found.

3-**Return Slider Resource**:  
If found, the slider is wrapped in a resource for proper structure and returned along with a success message and descriptive key.

4-**Error Handling**:  
Any unexpected errors are caught and a 422 error response is returned with the error message for debugging or logging purposes.

**40.4 update() Method**

**Overview**

The update() method updates the details of an existing slider, including optional image replacement and field updates.

### Key Steps:

1-**Find Slider by ID**:  
It first checks if a slider with the provided ID exists. If not, a 404 error response is returned with a localized message.

2-**Handle Image Update**:  
If a new image file is provided in the request, it is processed and stored. If not, the existing image path is preserved.

3-**Update Slider Fields**:  
Each field is updated if a new value is provided. If a value is not provided, the current one is kept. This ensures partial updates work correctly.

4-**Return Updated Resource**:  
The updated slider is wrapped in a resource and returned with a success message and structured key.

5-**Error Handling**:  
Any exceptions are caught and a 422 error response is returned, including the error message for debugging or logs.

**40.5 destroy() Method**

**Overview**

The destroy() method is responsible for deleting a specific slider by its ID.

### Key Steps:

1-**Find Slider by ID**:  
It attempts to retrieve the slider using the given ID. If not found, it returns a 404 error response with a localized message indicating the slider was not found.

2-**Optional Image Deletion** (commented out):  
There's a placeholder for deleting the slider's associated image from storage. This is currently commented out, indicating potential future use for cleaning up image files.

3-**Delete the Slider**:  
Once found, the slider is deleted from the database.

4-**Return Success Response**:  
A success message is returned using a consistent response format and a localized translation key.

5-**Error Handling**:  
Any exceptions during the process are caught and a 422 error response is returned, along with the exception message for further context.

**41. Terms Controller**

**41.1 all() Method**

**Overview**

The all() method retrieves a list of terms, with optional pagination support.

**Key Steps:**

1-**Pagination Check**:  
It first checks if the request includes the paginate parameter:

* If paginate is set to 1, it fetches the terms using pagination with the latest terms first.
* If paginate is set to 0, it retrieves all terms without pagination.
* If paginate is not provided, it defaults to paginated results.

2-**Pagination Data Construction** *(when applicable)*:  
When paginated results are returned, it constructs a pagination metadata array containing total items, current page, items per page, and total number of pages.

3-**Resource Wrapping**:  
Whether paginated or not, it wraps the retrieved terms using a resource class to standardize the API response format.

4-**Return Response**:  
A consistent success response is returned containing the terms (and pagination data when applicable), with a localized message.

5-**Error Handling**:  
If an error occurs during any of these steps, it catches the exception and returns a 422 error response with a localized message and the error details.

**41.2 get() Method**

**Overview**

The get() method is used to retrieve a specific term by its ID.

**Key Steps:**

1-**Term Retrieval**:  
It attempts to find a term using the provided term\_id.

2-**Conditional Response**:

* If the term **exists**, it wraps the term in a resource and returns a success response with a localized message.
* If the term **does not exist**, it returns a 404 error with a localized "not found" message.

3-**Resource Wrapping**:  
The returned term is transformed using a resource class to maintain a consistent API response format.

4-**Error Handling**:  
Any unexpected errors during the operation are caught, and a 422 error response is returned with a localized message and error details.

**41.3 add() Method**

**Overview**

The add() method is used to create and store a new term in the system.

### Key Steps:

1-**Validation**:  
Uses the StoreTermRequest to validate incoming data before proceeding with creation.

2-**Term Creation**:  
A new term is created using the validated request data.

3-**Resource Wrapping**:  
The created term is wrapped using a resource class to format the API response consistently.

4-**Success Response**:  
Returns the newly created term with a localized success message.

5-**Error Handling**:  
If an error occurs during the process, a 422 error response is returned with a localized error message and the actual exception message.

**41.4 update() Method**

**Overview**

The update() method updates the details of an existing term in the system.

### Key Steps:

1-**Validation**:  
Uses StoreTermRequest to ensure the provided data is valid before applying any changes.

2-**Term Lookup**:  
Finds the term by the provided ID. If not found, a 404 error response is returned.

3-**Update Operation**:  
If the term exists, it is updated using the new validated data.

4-**Resource Wrapping**:  
The updated term is returned in a structured format using a resource class.

5-**Success Response**:  
Returns the updated term with a localized success message.

6-**Error Handling**:  
If an exception occurs, a 422 error response is returned with a localized error message and the exception message.

**41.5 destroy() Method**

**Overview**

The destroy() method handles the deletion of a specific term from the system.

### Key Steps:

1-**Term Lookup**:  
Attempts to find the term using the provided ID. If it doesn't exist, a 404 error response is returned.

2-**Delete Operation**:  
If the term exists, it is deleted from the database.

3-**Success Response**:  
Returns a localized success message indicating the term was deleted successfully.

4-**Error Handling**:  
Any exceptions are caught, and a 422 error response is returned with a localized error message and the exception details.

**42. User Controller**

**42.1 all() Method**

**Overview**

The all() method retrieves a list of admin users based on the role\_id, with support for pagination based on the request.

**Key Steps:**

1-**Check for Pagination**:  
The method first checks if the request includes a paginate parameter.

* If paginate == 1, it returns paginated results (10 admins per page).
* If paginate == 0, it returns all admins without pagination.
* If no paginate parameter is specified, it returns paginated results by default (10 admins per page).

2-**Data Retrieval**:  
Admin users are retrieved by filtering for role\_id = 1, sorted by the latest entries.

3-**Pagination**:  
If pagination is enabled, the response includes pagination metadata:

* total: Total number of admins.
* per\_page: Number of admins per page.
* current\_page: Current page number.
* total\_pages: Total number of pages.

4-**Return Response**:

* The list of admins is wrapped in a resource (ManagerResource::collection) and returned with localized success messages.

5-**Error Handling**:  
If an error occurs during the process, a 422 error response is returned with the exception details.

**42.2** **get() Method**

Overview

The get() method retrieves a specific admin user by their admin\_id, filtering by role\_id.

**Key Steps:**

1-**Find Admin by ID**:  
The method attempts to find an admin user with role\_id = 1 (indicating an admin role) using the provided admin\_id.

2-**Error Handling**:

* If no admin is found (!$admin), the method returns a 404 error with a localized message indicating the admin was not found.
* If the admin is found, the method proceeds to return the admin's details.

3-**Return Response**:

* The admin's details are wrapped in a ManagerResource for structured data presentation and returned with a success message.

4-**Error Handling**:

* If any error occurs during the process (e.g., database query issues), a 422 error response is returned, including the exception message.

**42.3 add() Method**

**Overview**

The add() method is responsible for creating a new admin user with the details provided in the StoreAdminRequest.

**Key Steps:**

1-**Handle File Upload**:

* If the request contains an image file, the method calls handleFile() to process and store the image file. The path of the image is saved in $imagePath.
* If no image is provided, the $imagePath remains null.

2-**Create Admin User**:

* The method uses the validated data from the request (name, email, phone, password, etc.) to create a new admin user in the User model with role\_id = 1, indicating the user is an admin.
* The image path (if available) is stored in the image field; otherwise, it’s set to null.

3-**Return Response**:

* Upon successful creation, the method returns the admin user details wrapped in a ManagerResource. The success response includes the localized message for the admin creation.

4-**Error Handling**:

* If an error occurs during the process (e.g., during file upload or database creation), a 422 error response is returned, including the exception message.

**42.4** **update() Method**

**Overview**

he update() method is responsible for updating the details of an existing admin user, including their personal information and image.

**Key Steps:**

1-**Find Admin User**:

* The method first checks if the admin user exists in the database using the provided $admin\_id. It filters by role\_id = 1 to ensure only admins are updated.
* If the admin user does not exist, a 404 error response is returned.

2-**Handle Image Upload (if provided)**:

* If the request contains a new image file, the method processes and stores the file using the handleFile() method, updating the $imagePath.
* If the admin already has an image, the old image file is deleted from storage (if it exists) using File::delete().

3-**Update Admin Details**:

* The method updates the admin user’s details using the validated data from the request (name, email, phone, image, status, city\_id).
* If any of these values are not provided in the request, the old values are retained.

4-**Update Password (if provided)**:

* If a new password is provided in the request, it is updated for the admin user.

5-**Return Response**:

* After the update is successful, the method returns the updated admin user’s details wrapped in a ManagerResource and a localized success message.

6-**Error Handling**:

* If an error occurs during the process (e.g., file handling or database update), a 422 error response is returned, including the exception message.

**42.5 destroy() Method**

**Overview**

The destroy() method is responsible for deleting an admin user from the system. This includes removing their personal information and associated image file from storage.

**Key Steps:**

1-**Find Admin User**:

* The method first attempts to find the admin user using the provided $admin\_id and filters by role\_id = 1 to ensure the user is an admin.
* If the admin user does not exist, it returns a 404 error response with a localized message indicating that the admin was not found.

2-**Delete Image (if exists)**:

* If the admin has an image associated with their account, the image file is deleted from the file system.
* The path to the image is resolved using base\_path(), and the file is deleted using File::delete() if it exists.

3-**Delete Admin Record**:

* After deleting the image (if applicable), the admin's record is deleted from the database using $admin->delete().

4-**Return Success Response**:

* Upon successful deletion, the method returns a success response with a localized success message indicating the admin has been deleted.

5-**Error Handling**:

* If any errors occur during the process (e.g., while deleting the image or the admin record), the method returns a 422 error response with the exception message.

**43. Wallet Controller**

**43.1 all() Method**

**Overview**

The all() method is used to retrieve wallet records for the current user. The method checks the user's role to determine if the data should be filtered based on an investor's status or if all wallets should be returned. It also handles pagination if specified by the request.

**Key Steps:**

1-**User Role Check**:

* The method first identifies the role of the current user. If the user is an investor (role\_id == 6), it retrieves the investor’s information by matching the email and phone number and ensuring the investor's status is 1 (active).
* If the user is an investor, it fetches wallets associated with that specific investor using their investor\_id.
* If the user is not an investor (i.e., an admin or another role), it fetches all wallets.

2-**Pagination Handling**:

* The method checks if the request contains a paginate parameter to determine whether pagination is required:
  + If paginate == 1: It paginates the results with 10 wallets per page and includes pagination details like total count, current page, and the total number of pages.
  + If paginate == 0: It returns all wallets without pagination.
  + If no paginate parameter is provided, it defaults to paginated results with 10 wallets per page.

3-**Response Return**:

* Based on the pagination setting, it returns the wallet data along with the pagination details (if applicable) or just the list of wallets.
* The wallet data is wrapped in a WalletResource collection to standardize the API response format.

4-**Error Handling**:

* If any errors occur during the data retrieval or processing (e.g., database issues, invalid data), the method returns a 422 error response along with the exception message.

**44.2 get() Method**

**Overview**

The get() method retrieves a specific wallet record based on the provided wallet\_id. If the wallet is found, it returns the data in a structured format. If not, it returns an error.

**Key Steps:**

1-**Wallet Lookup**:

* The method attempts to find a wallet with the specified wallet\_id by querying the Wallet model.
* It uses the where() method to search for a wallet with the matching id and retrieves the first result.

2-**Wallet Not Found Check**:

* If no wallet is found (i.e., the query returns null), the method returns a 404 error response indicating that the wallet was not found. It uses the returnError() method to structure the error message appropriately.

3-**Success Response**:

* If the wallet is found, the method formats the wallet data using the WalletResource resource class. This ensures that the wallet data is returned in the desired API format.
* It then returns the wallet data in a structured response with the appropriate success message.

4-**Error Handling**:

* Any errors encountered during the wallet retrieval process (e.g., database connection issues, unexpected exceptions) are caught using a try-catch block.
* If an error occurs, the method returns a 422 error response along with the error message.

**44.3 add() Method**

**Overview**

The add() method creates a new wallet entry for an investor. It checks if the investor has enough wallet balance, handles the image file upload (if present), creates the wallet entry, and updates the user's wallet balance accordingly.

**Key Steps:**

1-**Check Investor Wallet Balance**:

* The method starts by retrieving the user (investor) based on the investor\_id passed in the request.
* It checks whether the investor's current wallet balance is sufficient to cover the requested total value. If the balance is insufficient, it returns an error message with a 403 status code.

2-**Image File Handling**:

* If the request contains an image file, it uploads the file to the uploads/wallets directory in the public storage, and the file path is stored in the imagePath variable.

3-**Create Wallet Entry**:

* A new wallet entry is created using the Wallet::create() method, where:
  + user\_id is set to the authenticated user's ID.
  + investor\_id is taken from the request.
  + The imagePath (if available) is stored.
  + The total value and status are set.

4-**Update Investor Wallet Balance**:

* The method updates the investor’s wallet balance by subtracting the requested total amount from the current balance using the update() method on the user model.

5-**Return Success**:

* Upon successful wallet creation and balance update, the method returns a success response containing the newly created wallet data formatted using the WalletResource class, along with a success message.

6-**Error Handling**:

* Any errors that occur during the wallet creation process (such as issues with the database or file handling) are caught in the try-catch block.
* If an error occurs, a 422 status code is returned with an error message.

**44.4 destroy() Method**

**Overview**

The destroy() method is responsible for deleting a wallet entry. It checks if the wallet exists, handles the deletion of the wallet image (if present), and performs the deletion of the wallet record.

**Key Steps:**

1-**Check Wallet Existence**:

* The method begins by attempting to retrieve the wallet based on the provided wallet\_id.
* If no wallet is found with the given ID, it returns a 404 error with a message indicating that the wallet was not found.

2-**Delete Wallet Image**:

* If the wallet has an associated image (i.e., an image field is present), the image file is deleted from the public storage using the Storage::disk('public')->delete() method.

3-**Delete Wallet Entry**:

* The wallet entry itself is then deleted using the delete() method.

4-**Return Success**:

* Upon successful deletion of the wallet and its associated image (if applicable), the method returns a success message confirming the wallet's deletion.

5-**Error Handling**:

* Any errors that occur during the deletion process (e.g., issues with file deletion or database issues) are caught in the try-catch block.
* If an error occurs, a 422 status code is returned along with an error message describing the issue.

**44.5 representativeOperations() Method**

**Overview**

The representativeOperations() method is designed to fetch wallet operations for a representative. It retrieves operations that match the given representative\_id, with specific filters like active status and a specific type of notification. It returns the data in a structured format, using a resource for the operations.

**Key Steps:**

1-**Retrieve Operations**:

* The method starts by querying the WalletOperation model to fetch operations for the specified representative\_id with a status of 1 (indicating active or valid operations) and a type\_notify of 1 (indicating a specific type of notification).
* The get() method is used to retrieve all matching operations.

2-**Return Data**:

* The method uses a resource (ClientOperationsResource) to format the retrieved wallet operations.
* The operations are then returned in a structured response, wrapped under the key "operations".

3-**Error Handling**:

* Any errors that occur during the retrieval of the operations are caught in the try-catch block.
* If an error occurs, the method returns a 422 status code along with a message indicating the error.

**44.6 change\_wallet() Method**

**Overview**

The change\_wallet() method is responsible for modifying the balance of a client’s wallet, either adding or subtracting a specific amount based on the type provided in the request. It logs the operation in the ManualWalletOperation table and returns an appropriate response based on whether the balance update is successful or if an error occurs.

**Key Steps:**

1-**Check Client Validity**:

* The method starts by looking for a client with the specified client\_id and ensures that the client is active (status = 1).
* If the client does not exist, it returns an error message with a 404 status code indicating that the client was not found.

2-**Balance Modification (Add or Subtract)**:

* The method checks the type parameter to decide whether to add or subtract the amount from the client's balance.
* **If type == 1 (add balance)**: The balance is increased by the amount provided in the request.
* **If type == 2 (subtract balance)**: The balance is decreased by the amount.

3-**Log the Operation**:

* Regardless of whether the balance is added or subtracted, a new record is created in the ManualWalletOperation table to log the transaction, including details such as the amount, type of transaction, reason, client, and the admin making the change.

4-**Save Changes**:

* After modifying the balance and logging the operation, the client’s updated balance is saved to the database.

5-**Return Success**:

* If the operation completes successfully, a success message is returned, confirming that the client’s balance has been updated.

6-**Error Handling**:

* If any error occurs during the process (such as database issues or missing client), a 422 status code is returned with the error message.

**44.7 all\_manual\_wallet\_operations() Method**

**Overview**

The all\_manual\_wallet\_operations() method is responsible for retrieving all the manual wallet operations, paginated for efficient display. It returns these operations along with pagination details, such as the total number of records, current page, and number of records per page.

**Key Steps:**

1-**Retrieve Manual Wallet Operations**:

* The method fetches the latest manual wallet operations from the ManualWalletOperation model, applying pagination to limit the number of records returned per request (10 in this case).

2-**Pagination Information**:

* After fetching the operations, the method calculates the pagination data, including:
  + total: The total number of operations available.
  + per\_page: The number of operations per page (10 in this case).
  + current\_page: The current page number.
  + total\_pages: The total number of pages available based on the total number of operations and records per page.

3-**Return Data**:

* The method returns the operations data inside a response with pagination information. The ManualClientOperationsResource::collection() method is used to format each manual wallet operation in the response.
* The success response includes both the operations and pagination data.

4-**Error Handling**:

* If any error occurs during the process (e.g., database issues), the method catches the exception and returns a 422 status code with the error message.

**45. Weight Controller**

**45.1 all() Method**

Overview

The all() method is responsible for retrieving all the weights from the Weights model and returning them as a JSON response.

**Key Steps:**

1-**Retrieve Weights**:

* The method fetches all records from the Weights model using the Weights::get() method, which returns all available weight entries from the database.

2-**Resource Transformation**:

* The fetched data is passed through the WeightResource::collection() method. This transforms each weight entry into a consistent format, ensuring that the API response adheres to a specific structure.

3-**Return Data**:

* The method returns a JSON response containing the transformed weights data with a status code of 200. This provides a structured response that can be used in client applications.

4-**Error Handling**:

* If any errors occur during the process (e.g., database connection issues), the method catches the exception and returns a 403 status code along with the error message in the response body.

**45.2 add() Method**

**Overview**

The add() method is used to add a new weight entry into the Weights model. It validates the input data, adds the new entry to the database, and returns a success message if successful, or an error message if something goes wrong.

**Key Steps:**

1-**Validation**:

* The method first defines validation rules for the request. It checks that both the name\_ar (Arabic name) and name\_en (English name) fields are required.
* It uses the Validator::make() function to validate the input data against these rules.
* If validation fails, it returns a JSON response with the first error message and a status code of 403.

2-**Create New Weight**:

* If the validation passes, a new record is created in the Weights model using the data from the request (name\_ar and name\_en).
* The Weights::create() method is used to insert the new weight into the database.

3-**Return Success**:

* After successfully adding the weight, the method returns a success message ('تم اضافة الوزن بنجاح') to indicate that the operation was successful.

4-**Error Handling**:

* If any error occurs during the process, such as a database issue or unexpected error, the method catches the exception and returns a 422 status code with the error message.

**45.3 update() Method**

**Overview**

The update() method is used to update an existing weight entry in the Weights model. It validates the input data, finds the corresponding weight record in the database, updates it, and returns a success message if the operation is successful, or an error message if any issues occur.

**Key Steps:**

1-**Validation**:

* The method begins by defining validation rules for the request. It ensures:
  + The weight\_id is required and exists in the weights table.
  + The name\_ar (Arabic name) and name\_en (English name) are both required.
* It uses the Validator::make() function to validate the request data against these rules.
* If validation fails, it returns a JSON response with the first error message and a status code of 403.

2-**Find and Update Weight**:

* If the validation passes, the method retrieves the Weights record with the given weight\_id from the database using the Weights::find() method.
* Once the record is found, it updates the fields name\_ar and name\_en using the data from the request.

3-**Return Success**:

* After successfully updating the weight, the method returns a success message ('تم تعديل الوزن بنجاح'), indicating that the update was successful.

4-**Error Handling**:

* If any error occurs during the process, such as a database issue or validation failure, the method catches the exception and returns a 422 status code with the error message.

**46.1 delete() Method**

**Overview**

The delete() method is responsible for deleting an existing weight entry from the Weights model. It validates the request, deletes the specified record, and returns an appropriate success or error message.

**Key Steps:**

1-**Validation**:

* The method first defines validation rules for the request. It ensures:
  + The weight\_id is required and exists in the weights table.
* It uses Validator::make() to validate the input data against these rules.
* If the validation fails, a JSON response is returned with the first validation error message and a 403 status code.

2-**Find and Delete Weight**:

* If validation is successful, the method retrieves the Weights record using the weight\_id from the request by calling Weights::find().
* Once the record is found, it is deleted using the delete() method.

3-**Return Success**:

* After the deletion, the method returns a success message ('تم حذف الوزن بنجاح'), indicating that the weight record has been successfully deleted.

4-**Error Handling**:

* If any error occurs during the process (such as a database error), the method catches the exception and returns a 422 status code along with the error message.

**46. Reservation Controller**

**46.1 all() Method**

**Overview**

The all() method retrieves a list of reservations, applying different filters based on the user's role, query parameters, and pagination settings. It returns the data as either a paginated list or a non-paginated list depending on the request.

**Key Steps:**

1-**User Role Check**:

* The method first checks the authenticated user's role using $request->user()->role\_id.
* Based on the user's role (admin or investor), the query for fetching reservations is constructed differently:
  + **Investor Role**: Filters reservations by their association with a specific investor.
  + **Admin or Other Roles**: Applies filters related to area and other request parameters.

2-**Filters**:

* **Date Filters**: If the date\_from and date\_to parameters are provided, the query filters reservations within that date range.
* **Search**: If a search term is provided, the query searches across multiple fields in the client and reservation models.
* **Other Filters**: Filters are applied based on fields like representative\_phone, representative\_id, area\_id, payment\_type, and status. The status filter supports a wide range of possible values, including custom logic like today, archive, delete, etc.

3-**Pagination**:

* The method checks if the paginate parameter is set in the request.
  + If paginate == 1, it returns the results as a paginated response with pagination data.
  + If paginate == 0, it returns the results without pagination.
  + If the paginate parameter is absent, it defaults to paginated results.

4-**Reservation Count**:

* Before returning the data, the method counts the total number of reservations matching the query parameters.

5-**Return Data**:

* The method returns the filtered reservations wrapped in a response, either paginated or not, along with the total count of reservations.

6-**Error Handling**:

* If any error occurs during the process, the method catches the exception and returns a 403 status code with the error message.

**46.2 get() Method**

**Overview**

The get() method retrieves a specific reservation by its ID. It checks if the reservation exists and returns the data if found, or an error if not. It also handles any errors that might occur during the process.

**Key Steps:**

1-**Find Reservation**:

* The method attempts to find the reservation using Reservation::find($id), where $id is the reservation's unique identifier provided in the request.

2-**Check for Existence**:

* If the reservation is not found (!$reservation), it returns a 404 error with the message 'Reservation not found' using the returnError() method.

3-**Return Data**:

* If the reservation is found, the method returns the reservation data wrapped in a ReservationResource for proper formatting. It uses the returnData() method to send the data as a response with the appropriate message 'Reservation retrieved successfully'.

4-**Error Handling**:

* If any error occurs during the process (such as a database issue or a server error), the method catches the exception and returns a 403 error with a generic error message, appending the exception's message for debugging.

**46.3 add() Method**

**Overview**

The add() method handles the process of creating a new reservation. It validates inputs, calculates pricing, finds available time slots, and ensures that the reservation doesn't conflict with others. It also supports coupon usage, calculates discounts, and assigns representatives based on location.

**Key Steps:**

1-**Log Request Data**:

* The method logs the incoming request data for debugging purposes, including reservation details like date, time, and location.

2-**Parse and Format Date/Time**:

* The reservation date and time are parsed and formatted using Carbon. The time is adjusted to a 24-hour format if necessary, and the target time for the reservation is determined.

3-**Determine Location and Area**:

* The location is retrieved based on the provided location ID. The method checks whether the location's coordinates are valid and finds the appropriate geographical area for the reservation.

4-**Find Available Shifts**:

* The available shifts are fetched based on the day of the week and area. Shifts are filtered based on whether the system is configured to send reservations to representatives or not.

5-**Generate Available Time Slots**:

* The method calculates available time slots for each shift in 90-minute intervals based on the shift start and end times.

6-**Check for Matching Reservation Time**:

* The method checks if the requested reservation time matches any of the available slots. If no match is found, an error is returned.

7-**Assign Representative**:

* A representative is assigned to the reservation based on proximity to the provided location. If no representative is found, an error is returned.

8-**Check for Conflicting Reservations**:

* The method checks whether there is already an existing reservation with the same date and time. If a conflict is found, an error is returned.

9-**Calculate Pricing**:

* The service pricing is calculated based on the reservation details, including additional services and the main service selected by the user. If a coupon is provided, the discount is applied to the pricing, and the final price after discount is calculated.

10-**Handle Coupon Logic**:

* If a valid coupon is provided, the method checks whether the coupon is still active and whether it has been used by the client before. If valid, the coupon is applied, and the discount is calculated. The coupon usage is updated, and the coupon is associated with the client.

11-**Create Reservation**:

* A new reservation object is created, and various attributes like client ID, payment type, car details, location, and representative ID are saved to the reservation. The reservation's time and price are also set based on the previous calculations.

12-**Save Services and Send Confirmation**:

* The reservation's associated services are saved. A success message is returned, confirming that the reservation was added successfully.

**46.4 calServicesPrice() Method**

**Overview**

The calServicesPrice() method calculates the total price for additional services requested by the user in a reservation. It checks the provided services and sums up their costs based on the area where the reservation is being made.

**Key Steps:**

1-**Initialize Services Price**:

* The method starts by initializing the services\_price variable to 0, which will accumulate the total price of additional services.

2-**Check for Additional Services**:

* The method checks whether the request contains any additional services by looking for the additional\_services field in the request. If no additional services are provided, the price remains 0.

3-**Loop Through Additional Services**:

* If additional services are provided, the method loops through each service ID and fetches the corresponding Service object from the database.

4-**Fetch Service Cost**:

* For each service, the method retrieves the price of the service from the CityServicesPrices model, which stores the cost of services in specific cities. The service's cost is added to the total services\_price.

5-**Return Total Services Price**:

* After iterating over all the additional services, the method returns the total accumulated services\_price, which represents the total cost of all the additional services selected for the reservation.

**46.5 saveReservationServices() Method**

**Overview**

The saveReservationServices() method is responsible for associating a list of selected services with a reservation. It handles both storing and updating scenarios depending on the type of operation being performed.

**Key Steps:**

1-**Check for Provided Services**:

* The method first verifies whether any services are provided and that the list is not empty. If no services are given, the method does nothing and returns true.

2-**Determine Operation Type (Store or Update)**:

* The method checks the value of the $type parameter to determine if it's handling a new reservation creation (store) or updating an existing one.

3-**Handle Store Operation**:

* In the case of a store operation, the method loops through each service ID in the list and attaches the service to the reservation by calling a helper method designed to handle the attachment process.

4-**Handle Update Operation**:

* If the operation type is update, the method first removes all previously attached services from the reservation to ensure no outdated services remain.
* It then reattaches the currently selected services one by one using the same helper method used during the store process.

5-**Return Value**:

* After completing the necessary service attachments, the method returns true to indicate successful execution.

**46.6 attachServiceToReservation() Method**

**Overview**

The attachServiceToReservation() method is responsible for linking a specific service to a reservation, along with storing the associated service cost based on the city where the reservation is located.

**Key Steps:**

1-**Find the Service**:

* The method begins by locating the service using the provided service\_id.
* If the service does not exist, the method exits without taking any action.

2-**Retrieve City-Specific Pricing**:

* Once the service is found, the method searches the CityServicesPrices table to find the cost of that service in the specific city related to the given area.
* This ensures that the price is accurate based on the city where the service will be provided.

3-**Attach the Service to the Reservation**:

* If pricing information is available, the method attaches the service to the reservation and records the cost in the pivot table.
* This allows the reservation to reflect not just the services selected but also their corresponding charges based on location.

**46.7 checkCoupon() Method**

**Overview**

The checkCoupon() method is responsible for validating a discount coupon provided by a client and determining whether it can be applied to their reservation based on several rules including code validity, usage limits, and date range.

**Key Steps:**

**1-Validate the Input:**

* The method starts by validating the incoming request to ensure it contains:
  + A coupon code
  + A valid client\_id that exists in the database
  + A date to check coupon availability
* If any of these validations fail, the method returns an error with details.

**2-Verify the Coupon Code:**

* It looks up the coupon using the provided code and checks:
  + That the coupon exists.
  + That the coupon is active (status is 1).
  + That the coupon has not been used up (times ≠ used).
* If the coupon does not meet these criteria, an error is returned indicating the coupon is invalid or exhausted.

**3-Check for Single-Use Restriction:**

* If the coupon is restricted to a single use per user (valid\_for\_one\_user is 1), the method checks if this user has already used the coupon.
* If the coupon has already been used by the client, an error is returned indicating the coupon is no longer valid for them.

**4-Evaluate Date Validity:**

* The method then checks if the current date (provided in the request) falls within the coupon's active period using the active\_from and active\_to fields.
* Several scenarios are handled:
  + No date restrictions → always valid.
  + Both start and end dates exist → must fall within that range.
  + Only start date exists → valid from that date onward.
  + Only end date exists → valid until that date.
* If the date is valid, coupon details are returned including the code, discount amount, and the services it applies to.

**5-Return the Result:**

* If all checks pass, the method returns the coupon data in a structured format.
* If any condition fails (invalid code, already used, or date mismatch), a meaningful error response is sent back to the client.

**46.8** **formatService() Method**

**Overview**

The formatService() method is responsible for formatting a collection of services into a simplified array structure that includes only essential information, such as the service ID and its localized title.

**Key Steps:**

**1-Initialize an Empty Array:**

* The method begins by creating an empty array called services\_array to store the formatted service data.

**2-Loop Through Each Service:**

* It iterates over the list of service objects provided as input.
* For each service, it extracts:
  + The service's id
  + The translated title based on the current application locale (e.g., Arabic or English)

**3-Store the Formatted Data:**

* Each formatted service is added to the services\_array using a simple associative array structure.
* This ensures the response includes only necessary and user-readable information.

**4-Return the Result:**

* After processing all services, the method returns the services\_array.
* This array can be used in API responses or views where a clean list of service data is needed, localized for the end user.

**46.9 checkReservation() Method**

**Overview**

The checkReservation() method determines whether a reservation already exists for a given representative on a specific date and time slot. It is used to avoid scheduling conflicts when creating or updating a reservation.

**Key Steps:**

**1-Handle Update Case:**

* If an id is provided, the method retrieves the existing reservation data using that ID.
* It checks whether the representative ID in the request differs from the one associated with the existing reservation.
* If the representative has changed, it searches for any other reservation with the new representative, date, and time (excluding canceled reservations where status is 6).

**2-Handle Create Case:**

* If no id is given, the method assumes a new reservation is being created.
* It directly checks if a reservation already exists for the given representative, date, and time, excluding canceled ones.

**3-Return Conflict Status:**

* If a reservation is found, the method returns true, indicating a conflict (i.e., the slot is already booked).
* If no reservation is found, it returns false, meaning the slot is available for booking.

**49.10 newReservationMail() Method**

**Overview**

The newReservationMail() method is responsible for sending an email notification when a new reservation is created. This helps keep the application administrators or stakeholders informed about incoming bookings.

**Key Steps:**

**1-Retrieve Application Email Configuration:**

* The method fetches the first record from the Application model, which holds configuration data, including the notification email address.

**2-Check for Valid Application:**

* It verifies that the application configuration exists before proceeding.
* If the application data is missing, the method exits without sending an email.

**3-Send the Email Notification:**

* If the application configuration is valid, the method sends an email to the address specified in email\_1.
* It uses the NewReservationMail Mailable class to format and deliver the email with the reservation details.

**46.11 cancelRequest() Method**

The cancelRequest() method is responsible for canceling an existing reservation request. It updates the reservation's status and stores relevant cancellation details.

**Key Steps:**

**1-Retrieve the Reservation:**

* The method locates the reservation using the provided reservation\_id from the request.
* If the reservation does not exist, it throws a 404 error automatically via findOrFail.

**2-Update Cancellation Details:**

* Once the reservation is found, it updates the following fields:
  + cancel\_reason: the reason provided by the user for canceling the reservation.
  + status: sets the reservation status to 6, indicating it has been canceled.
  + cancel\_id: stores the ID of the user who performed the cancellation.

**3-Return Success Response:**

* After the update, the method returns a JSON success response confirming that the reservation was canceled successfully.

**46.12** **update() Method**

**Overview**

The update() method is responsible for updating an existing reservation with new details. It ensures that the provided reservation data is valid, checks for any conflicting reservations, and adjusts the reservation accordingly.

**Key Steps:**

**1-Log the Update Request:**

* The method logs the incoming request data for debugging and tracking purposes.

**2-Retrieve the Reservation and Related Data:**

* The method retrieves the reservation using the reservation\_id from the request, including soft-deleted entries if applicable.
* It then checks if the provided from time needs to be converted (12-hour to 24-hour format).

**3-Check for Reservation Conflicts:**

* The method verifies if any other reservation exists with the same date, time, and representative. If a conflict is found, it returns an error.

**4-Validate Representative Availability:**

* It checks if the representative is available during the provided time by checking shifts for that day.
* If the representative is not assigned to the shift, an error is returned.

**5-Validate Payment Method:**

* If the payment\_type is provided, the method checks if it's valid and assigns it to the reservation.

**6-Calculate Service Costs:**

* It calculates the costs of services associated with the reservation, factoring in the city-specific pricing and any applicable discounts from coupons.

**7-Update Reservation Data:**

* The reservation's data is updated with the new information, including the client ID, representative ID, car details, location, payment type, service costs, and more.

**8-Update Subscription Services (If Applicable):**

* If the reservation is part of a package, it updates the corresponding package subscription and service counts for the client.

**9-Send Notifications:**

* A push notification is sent to the client informing them of the reservation update.

**10- Save the Updated Reservation:**

* The updated reservation is saved, and a success message is returned.

**46.13 updateReservationStatusNotification() Method**

The updateReservationStatusNotification() method is responsible for sending notifications based on the current status of a reservation. The method customizes the message according to the reservation status and triggers a push notification to the client.

**🔑 Key Steps:**

**1-Define the Notification Message Based on Status:**

* The method first checks the status of the reservation.
* It then defines a corresponding notification message, including a title and body, based on the status:
  + **Status 0**: Rejected reservation
  + **Status 1**: Pending reservation
  + **Status 2**: Approved reservation
  + **Status 3**: Completed reservation
  + **Status 5**: Representative work started

**2-Save the Notification:**

* Once the appropriate message is determined, the method saves the notification for the client using the saveNotification() function. This logs the notification for later reference.

**3-Send Firebase Push Notification:**

* After saving the notification, the method creates a new instance of FireBasePushNotification and sends the notification to the client’s device using their device\_token.
* The notification includes the title and body message specific to the reservation’s status.

**46.14 delete() Method**

**Overview**

The delete() method is responsible for marking a reservation as deleted in the system. This is done by updating the reservation’s status and setting a deletion timestamp. The method returns a success response once the deletion is processed.

**Key Steps:**

**1-Retrieve the Reservation:**

* The method fetches the reservation from the database using the provided reservation\_id from the request.

**2-Update the Reservation:**

* The reservation’s status is updated to 20 (indicating that it is deleted).
* The deleted\_at field is set to the current timestamp using now(), marking the reservation as soft-deleted.

**3-Return Success Response:**

* After successfully marking the reservation as deleted, the method returns a success response with a status code of 200 and a message indicating that the deletion was successful.

**46.15 restore() Method**

**Overview**

The restore() method is designed to restore a previously soft-deleted reservation from the database. If successful, it will return a success message; otherwise, it will handle any errors gracefully and return an error message.

**Key Steps:**

**1-Try to Restore the Reservation:**

* The method attempts to find the reservation by its id, even if it has been soft-deleted, using the withTrashed() method.
* Once the reservation is located, the restore() method is called to bring it back to its active state.

**2-Return Success Response:**

* If the restoration is successful, the method returns a success response with a status code of 200 and a message indicating that the reservation has been restored successfully.

**3-Handle Errors:**

* If an error occurs during the restoration process (such as the reservation not being found or any other exception), the method will catch the exception and return an error response with a 403 status code and a message stating that the reservation cannot be restored.

**46.16 statuses() Method**

**Overview**

The statuses() method retrieves a list of predefined status options for reservations or other relevant entities. It returns the status options along with their associated labels, which are translated into the appropriate language. The method also handles potential errors gracefully.

**Key Steps:**

**1-Define Statuses:**

* The method defines an array of status options, where each option has an id and a name. The name is a translated label that corresponds to the status description. The translation is done via the \_\_() function, which uses the localization files.

**2-Return Success Response:**

* The method returns a success response with a data key, which contains the list of status options (statuses). The response is formatted with the message 'statuses' using the translation function.

**3-Handle Errors:**

* If any error occurs during the process (e.g., a runtime error or issue fetching statuses), the method catches the exception and returns an error response with a 403 status code and the error message.

**46.17 statusUpdated() Method**

**Overview**

The statusUpdated() method is responsible for fetching a list of updated status options, where each status has an associated id and translated name. This method is useful for displaying updated status options for reservations or other related entities. It also includes error handling in case something goes wrong during the execution.

**Key Steps:**

**1-Define Updated Statuses:**

* The method defines an array of updated status options, where each option contains an id (the unique identifier of the status) and a name (the label for the status). The label is translated into the relevant language using the \_\_() function for localization.

**2-Return Success Response:**

* The method returns a structured success response, encapsulating the statuses in a data key. The response also includes the translated label for 'statuses' as part of the message.

**3-Handle Errors:**

* In the event of an error or exception during execution, the method catches the exception and returns an error response with a 403 status code, accompanied by the error message.

**46.18 current\_hour\_reservations() Method**

**Overview**

The current\_hour\_reservations() method retrieves all reservations that fall within the current hour, based on the server's local time. It is useful for checking reservations that are scheduled to start or end within the current time window. The method handles different time formats and provides pagination for large result sets.

**Key Steps:**

**1-Get Current Time:**

* The method uses the Carbon library to fetch the current date and time. It calculates both the start and end times of the current hour, formatting them as 24-hour strings (H:i:s) for comparison in the database.

**2-Query Reservations:**

* The method queries the Reservation model for reservations that match the current date and whose times overlap with the current hour.
  + The query checks if the reservation’s from or to times fall within the current hour.
  + It handles reservations that span across the current hour (i.e., start before and end after the current hour).

**3-Paginate Results:**

* The results are paginated, with a limit of 10 reservations per page, ensuring that the system does not overload with too many results at once.

**4-Return Data:**

* A success response is returned with the retrieved reservations formatted as a collection of ReservationResource. The response includes a translated label for the current\_hour\_reservations.

**5-Error Handling:**

* If an error occurs during the process, the method catches the exception and returns a 422 status code with the error message.

**46.19 coming\_hour\_reservations() Method**

**Overview**

The coming\_hour\_reservations() method retrieves reservations that are scheduled to occur in the next hour, based on the current time or a specific hour passed via the request. It also returns a list of representatives who are active clients and eligible for assignments.

**Key Steps:**

**1-Time Calculation:**

* The method first fetches the current time using the Carbon library.
* It checks if an hour is provided in the request. If provided, it calculates the start and end of the next hour based on the given hour. If no hour is provided, it calculates the next hour based on the current time.

**2-Query Reservations:**

* The method queries the Reservation model to retrieve all reservations:
  + For the current date (date).
  + Within the time range of the next hour (from time between the next hour's start and end).
  + It excludes reservations with the status 6 (canceled).
* If the authenticated user is an admin or has a specific role (i.e., role ID 7 or 17), the method restricts the results to only reservations within the user's area (area\_id).

**3-Fetch Active Representatives:**

* The method also fetches all active representatives (clients with type 1 and status 1).

**4-Return Data:**

* The response includes a success status with the reservation data (formatted via ReservationResource) and the list of representatives (formatted via RepresentiveResource).

**5-Error Handling:**

* In case of any errors, the method catches the exception and returns a 403 error with a custom message.

**46.20 notify() Method**

**Overview**

The notify() method sends a push notification to the representative of a reservation once the notification is triggered by the completion of a certain task related to the reservation. It saves the notification to the database and sends a push notification using Firebase.

**Key Steps:**

**1-Retrieve Reservation:**

* The method first fetches the reservation from the Reservation model using the provided reservation\_id.

**2-Prepare Notification:**

* A notification message is created with a title and body, both of which are translated using Laravel’s trans() helper function to handle localization.

**3-Save Notification:**

* The notification is saved in the database by calling the saveNotification() method, passing the representative\_id of the reservation and the prepared message.

**4-Send Firebase Notification:**

* A new instance of the FireBasePushNotification controller is created.
* A push notification is sent to the device\_token of the client associated with the reservation, using Firebase. The notification includes the title and body.

**46.21 saveNotification() Method**

**Overview**

The saveNotification() method is responsible for saving a notification in the database for a specific client. It takes a client ID and a message array as parameters, then stores the title and body of the message in the Notification model.

**Key Steps:**

**1-Create New Notification:**

* A new instance of the Notification model is created.

**2-Assign Values:**

* The client\_id, title, and body fields of the Notification model are populated with the respective values provided in the method parameters:
  + client\_id: The ID of the client who will receive the notification.
  + title: The title of the notification.
  + body: The content of the notification message.

**3-Save Notification:**

* The notification is then saved to the database using the save() method of the Notification model.

**46.22 getReservationTypes() Method**

**Overview**

The getReservationTypes() method is designed to return various types of reservation data based on the user's role. It calculates the total count of reservations across different categories, such as "all", "today", "archive", "delete unpaid", and "canceled by client", and returns them in the response.

**Key Steps:**

**1-Role-Based Filtering:**

* Depending on the user's role, the method determines the scope of the reservation counts:
  + **Role 6**: Checks reservations related to a specific investor by matching email and phone.
  + **Role 7 or 17**: Filters reservations based on the user's area.
  + **Other roles**: Retrieves reservation counts for all records.

**2-Reservation Counts:**

The following counts are calculated based on the role:

* **All Count**: The total number of reservations.
* **Today Count**: The number of reservations for the current day.
* **Archive Count**: The number of archived (deleted) reservations.
* **Delete Unpaid**: The number of reservations with a "delete unpaid" status.
* **Deleted Count**: The number of canceled reservations.

**3-Response:**

The method organizes these counts into an array of reservation types and returns the data in a structured format. The response includes:

* **Name**: The label for the reservation type (e.g., "All", "Today", etc.)
* **Value**: The identifier for the reservation type (e.g., 'all', 'today', etc.)
* **Count**: The total count of reservations for the type.

**4-Error Handling:**

If an error occurs during the process, it catches the exception and returns an error message with a 422 status code.

**46.23 not\_finished\_today\_comapred\_to\_all\_of\_today() Method**

**Overview**

The not\_finished\_today\_comapred\_to\_all\_of\_today() method calculates and returns the total number of reservations for today, as well as the number of reservations that are not finished. It provides a comparison between the total reservations and the ones that have not yet finished.

**Key Steps:**

**1-Reservation Counts for Today:**

* **Not Finished Count**: The method filters reservations with statuses indicating that they are not finished (statuses 2 and 7) and counts them.
* **Total Today Count**: The method counts the total number of reservations for today.

**2-Retrieve Reservations:**

* **Today Reservations**: It fetches all the reservations for today, allowing detailed information to be returned.

**3-Response:**

The method returns the following data:

* **Today Reservations**: A list of all reservations for today.
* **Not Finished Reservations**: The count of reservations that are not finished.
* **Total Reservations for Today**: The total count of all reservations for today.

**4-Error Handling:**

If an error occurs during the execution, the method catches the exception and returns an error message with a 422 status code.

**46.24** **forward\_to\_representative() Method**

Overview

The forward\_to\_representative() method is responsible for updating the representative for a list of reservations and notifying the selected representative about the new assignment. It ensures that the representative exists and handles any errors appropriately.

**Key Steps:**

**1-Validate Representative Existence:**

* **Check Representative**: The method first checks if the given representative exists and is of type 1 (representative type) using the representative\_id provided in the request.
* If no representative is found, it returns a 404 error indicating the representative was not found.

**2-Retrieve and Validate Reservations:**

* **Fetch Reservations**: The method retrieves the reservations based on the provided reservations\_ids.
* **Check Reservations**: If no reservations are found for the provided IDs, it returns a 404 error indicating that no reservations were found.

**3-Update Representative for Reservations:**

* **Bulk Update**: Once validated, the method updates the representative\_id for all the selected reservations in a single query.

**4-Send Notification:**

* **Notification Message**: The method prepares a notification message for the representative informing them about the new reservation assignments.
* **Save Notification**: The notification is saved in the system.
* **Send Notification**: A push notification is sent to the representative's device using the send\_notification() helper function.

**5-Success Response:**

* If the operation is successful, it returns a success message indicating that the reservations were successfully forwarded to the representative.

**6-Error Handling:**

* If any error occurs during the process, the method catches the exception and returns an error message with a 422 status code.

**46.25 getCars() Method**

**Overview**

The getCars() method is responsible for retrieving a list of cars associated with a specific client. It fetches the cars based on the provided client\_id and returns the results in a structured format. If any error occurs during the process, it returns an appropriate error message.

**Key Steps:**

**1-Retrieve Client Cars:**

* **Fetch Cars**: The method uses the client\_id from the request to filter and retrieve cars associated with that specific client from the database.
* **Sorting**: The cars are sorted in descending order based on their creation date (latest()).

**2-Return Data:**

* **Car Resources**: The cars are transformed using the CarResource to ensure the response follows the desired structure.
* **Return Response**: The method returns the list of cars with a success message ('تم استرجاع الداتا بنجاح'), indicating that the data has been successfully retrieved.

**3-Error Handling:**

* If an error occurs during the process (e.g., database issues or invalid request parameters), the method catches the exception and returns a 422 error along with the error message.

**46.26 mainServices() Method**

**Overview**

The mainServices() method is responsible for retrieving a list of services that are categorized as the main services, based on their type. It filters the services by type = 0, transforms the data into a structured format, and returns the list to the user. If any error occurs during the process, it returns an appropriate error message.

**Key Steps:**

**1-Retrieve Main Services:**

* **Filter Services**: The method queries the Service model to get services where the type is equal to 0, which denotes the main services.
* **Transformation**: The services are transformed using the ServiceResource to ensure that the response follows the desired format.

**2-Return Data:**

* **Return Response**: The method returns the list of main services along with a success message (\_\_('api.service\_all')), indicating the successful retrieval of services.

**3-Error Handling:**

* If an error occurs during the process (e.g., database issues or other unexpected errors), the method catches the exception and returns a 422 error along with the error message.

**46.27 additional\_services()**

**Overview**

The additional\_services() method is responsible for retrieving a list of additional services available to the client. It checks whether a subscription\_id is provided in the request. If it is, it fetches the services associated with the subscription. Otherwise, it retrieves all available additional services. The method returns the service data in a structured format, and if any error occurs, an appropriate error message is returned.

**Key Steps:**

**1-Check for Subscription ID:**

* **Subscription Check**: The method first checks if a subscription\_id is provided in the request.
  + If the subscription exists, the method retrieves the services linked to the subscription where the count is greater than 0, ensuring that the client has access to those services.
  + If no subscription is found or the subscription\_id does not match any existing subscription, an error message is returned.

**2-Retrieve Additional Services:**

* **No Subscription**: If no subscription\_id is provided, the method retrieves all services of type = 1, which are categorized as additional services.

**3-Return Data:**

* **Return Response**: The method returns the list of additional services to the client. The data is formatted using ServiceResource, which structures the response appropriately.

**4-Error Handling:**

* **Subscription Not Found**: If the subscription cannot be found, the method returns an error with a 404 status and the message الاشتراك غير موجود ("Subscription not found").
* **General Errors**: If any other error occurs during the process (e.g., database issues), it is caught and a 422 error is returned with the message of the exception.

**46.28 getMonthlyReservationStats() Method**

**Overview**

The getMonthlyReservationStats() method retrieves the reservation statistics for a given month and year, grouping the data by day. The method calculates several key metrics for each day, including the total number of reservations, finished reservations, the percentage of finished reservations, and reservations that used a coupon. It ensures that the data returned includes all days of the month, even if no reservations were made on certain days.

**Key Steps:**

**1-Fetch Reservation Data:**

* **SQL Query**: The method executes a query to fetch reservation data, grouped by day, for a specific month and year. The query calculates:
  + total\_reservations: The total number of reservations on each day.
  + finished\_reservations: The number of reservations that are finished (status = 3).
  + reservations\_using\_coupon: The number of reservations that used a coupon.
  + finished\_reservations\_using\_coupon: The number of finished reservations that used a coupon.

**2-Get the Total Days in the Month:**

* The method calculates the total number of days in the given month using cal\_days\_in\_month(), ensuring that it considers months with different lengths (28, 29, 30, or 31 days).

**3-Initialize Data Array:**

* **Initialization**: An array ($data) is initialized to include all days of the month, with placeholders for all metrics (total reservations, finished reservations, etc.), setting the initial values to 0.

**4-Merge Fetched Data:**

* **Merge Statistics**: The reservation statistics fetched in step 1 are merged into the initialized $data array. For each day, the statistics are updated with the corresponding values, and the finished\_ratio is calculated as the percentage of finished reservations out of total reservations.

**5-Return Data:**

* **Return Structured Data**: The data is structured in an array and returned in the response. The statistics are ordered sequentially by day, and the method returns the result as a part of the API response.

**6-Error Handling:**

* **Error Handling**: If any error occurs while fetching or processing the data, an appropriate error message is returned. However, no specific error handling is shown in the provided code, as it assumes that the query will succeed.

**46.29 get\_monthly\_reservations\_report() Method**

**Overview**

The get\_monthly\_reservations\_report() method retrieves a report of reservation statistics for a specific month and year, grouping the data by day. The report provides details about the total number of reservations, finished reservations, the use of coupons, and the ratio of finished reservations for each day of the selected month.

**Key Steps:**

**1-Fetch Reservation Data:**

* **SQL Query**: The method executes a query to fetch reservation data, grouped by day, for the specific month and year. The query calculates:
  + total\_reservations: The total number of reservations on each day.
  + finished\_reservations: The number of finished reservations (where status = 3).
  + reservations\_using\_coupon: The number of reservations that used a coupon.
  + finished\_reservations\_using\_coupon: The number of finished reservations that used a coupon.

**2-Get the Total Days in the Month:**

* The method calculates the total number of days in the given month using cal\_days\_in\_month(). This ensures it accounts for months of varying lengths (28, 29, 30, or 31 days).

**3-Initialize Data Array:**

* **Initialization**: An array ($data) is created, initializing each day of the month with default values for all reservation statistics (set to 0). This provides placeholders for each day, ensuring the report includes all days of the month, even if no reservations were made on specific days.

**4-Merge Fetched Data:**

* **Merge Statistics**: The reservation statistics obtained from the query are merged into the $data array. For each day, the statistics are updated with the corresponding values, and the finished\_ratio is calculated as the percentage of finished reservations relative to the total reservations for that day.

**5-Return Data:**

* **Return Structured Data**: The final data, which is an array containing the daily statistics for the month, is returned as part of the API response. The statistics are organized sequentially by day.

**6-Error Handling:**

* **Error Handling**: If an error occurs during the data fetching or processing, a relevant error message is returned. The provided code assumes the query runs successfully, and error handling is built into the response structure.

**46.30 getReservationsUsingCoupon() Method**

**Overview**

The getReservationsUsingCoupon() method retrieves all reservations that used a specific coupon, including soft-deleted records, and paginates the results. It provides the total reservations, the pagination details, and a list of reservations that used the given coupon.

**Key Steps:**

**1-Fetch Reservations by Coupon:**

* **SQL Query**: The method retrieves reservations from the database that match the specified coupon. It uses the withTrashed() method to include both active and soft-deleted reservations.

**2-Pagination:**

* **Paginate Results**: The method uses the paginate(10) method to fetch the reservations in paginated form. This means only 10 reservations will be shown per page. It automatically handles pagination for the API response.

**3-Generate Pagination Data:**

* **Pagination Info**: The method constructs an array with pagination information, including:
  + total: The total number of reservations matching the coupon.
  + per\_page: The number of reservations shown per page (10 in this case).
  + current\_page: The current page number of the paginated results.
  + total\_pages: The total number of pages based on the results.

**4-Return Data:**

* **Return Structured Data**: The method returns a response containing both the paginated reservations and the pagination information. The response is structured in a way that can be easily consumed by the frontend.

**5-Error Handling:**

* **Error Handling**: If an error occurs during the fetching or pagination process, the method catches the exception and returns a relevant error message.

**46.31 finished\_reservations() Method**

**Overview**

The finished\_reservations() method retrieves all reservations that have been completed (with status code 3) for a specific representative within a specified month and year. The method supports pagination to efficiently handle large result sets.

**Key Steps:**

**1-Filter by Representative and Date:**

* **Representative ID**: Filters reservations based on the given representative\_id from the request.
* **Status Check**: Only includes reservations that are marked as finished (status = 3).
* **Date Filtering**: Applies additional filters to only include reservations made in the specified year and month using the date column.

**2-Paginate Results:**

* **Pagination Logic**: Uses the paginate(10) method to return 10 reservations per page, sorted by the latest entries. This ensures better performance and user experience when browsing the list.

**3-Prepare Pagination Metadata:**

* **Pagination Structure**: Constructs an array that includes:
  + total: Total number of finished reservations for the representative in the selected month.
  + per\_page: The number of reservations shown per page (10).
  + current\_page: The current page number in the pagination.
  + total\_pages: The total number of pages available for the results.

**4-Return Data:**

* **Structured API Response**: Returns a structured response containing the reservations list (wrapped in a resource collection) along with pagination details for front-end use.

**5-Handle Errors:**

* **Try-Catch Block**: If any exception occurs during the query or processing, the method catches the exception and returns a formatted error response with status code 422.

**46.32 reservations\_revenue\_report() Method**

**Overview**

The reservations\_revenue\_report() method generates a financial summary of all reservations within a specified date range. It provides insights into the total number of reservations, overall revenue, and the breakdown of revenue by different payment types.

**Key Steps:**

**1-Aggregate Financial Data:**

* **Total Reservations**: Counts the total number of reservations within the given date range.
* **Total Revenue (total\_money)**: Sums up the total\_price of all reservations, rounding to two decimal places for currency accuracy.
* **Revenue by Payment Type**:
  + **Electronic Payments**: payment\_type = 1
  + **Wallet Payments**: payment\_type = 2
  + **Network Payments**: payment\_type = 3
  + Each of these is calculated separately and rounded to two decimal places.

**2-Filter by Date and Status:**

* **Date Range Filter**: Uses the whereBetween() clause to restrict results to the provided startDate and endDate.
* **Status Filter**: Excludes reservations that were canceled or failed by ensuring status != 6.

**3-Return Revenue Summary:**

* **Structured Output**: Returns a single object containing the financial metrics, making it easy to display in dashboards or reports.

**4-Handle Errors Gracefully:**

* **Exception Handling**: If any issue occurs during the query or processing, the error is caught and returned as a 422 error with the message.

**46.33 avg\_times\_taken\_in\_finishing\_reservations() Method**

**Overview**

The avg\_times\_taken\_in\_finishing\_reservations() method provides a detailed report on the durations taken to complete reservations, comparing the actual time with the expected washing time. It supports filtering by date and representative, and includes calculated fields like the total time taken and lateness per reservation.

**Key Steps:**

**1-Input Validation:**

* Validates optional input fields:
  + day: Specific day for filtering (format YYYY-MM-DD)
  + representative\_id: Specific representative for filtering
  + page: Pagination size (default is 10 if not provided)

**2-Retrieve Standard Washing Time:**

* Fetches the predefined washing time value from the application settings to use as a reference for measuring lateness.

**3-Dynamic Filtering:**

* Applies filters conditionally:
  + If day is provided, filters reservations by that specific date.
  + If representative\_id is provided, filters reservations for that representative.

**4-Reservation Selection:**

* Filters to only completed reservations (status = 3) for analysis.

**5-Time Calculation Per Reservation:**

* For each reservation, it calculates:
  + time\_taken: The time in minutes from representative\_start\_at to representative\_end\_at.
  + lateness: Difference between time\_taken and standard washingTime.
    - A positive value means the representative exceeded the expected time.
    - A negative value is formatted as + followed by the absolute value, indicating they finished early.

**6-Pagination and Transformation:**

* Paginates the result set (200 records per page).
* Each reservation is enhanced with the computed time\_taken and lateness fields before being returned.

**7-Output Construction:**

* Wraps reservations in a standardized resource format.
* Includes pagination metadata for frontend handling.

**8-Error Handling:**

* Any exception is caught and returned as a structured error response with a 422 status code.

**46.34 waiting\_for\_electronic\_payment() Method**

**Overview**

The waiting\_for\_electronic\_payment() method retrieves a list of reservations that are currently marked as "waiting for electronic payment." It supports filtering by the authenticated user's role and area, and also allows optional pagination.

**Key Steps:**

**1-Role-Based Area Filtering:**

* If the authenticated user has a role\_id of 7 or 17 (e.g., admin or supervisor roles), the query is restricted to their associated area\_id.

**2-Optional Area Filter:**

* If an area\_id is provided in the request, the query is further filtered by that area, overriding or supplementing the role-based area filter.

**3-Status Condition:**

* Filters the reservations by status = 7, which indicates they are "waiting for electronic payment."

**4-Soft-Deleted Records:**

* Uses withTrashed() to include soft-deleted reservations in the result set.

**5-Pagination Handling:**

* If paginate is present and set to 1, the method paginates the result set with up to 200 reservations per page.
* Pagination metadata is included in the response:
  + Total count
  + Items per page
  + Current page
  + Total number of pages

**6-Data Transformation:**

* The results are wrapped in a standardized ReservationResource collection for consistent API formatting.

**7-Message Localization:**

* Returns a translated message for the label "waiting\_for\_electronic\_payment" using localization helpers.

**8-Error Handling:**

* Catches any thrown exception and returns a structured error response with a 422 status code and the error message.

**46.35 orders\_placed\_on\_weekdays() Method**

**Overview**

The orders\_placed\_on\_weekdays() method provides statistical data showing how many completed reservations (orders) were placed on each day of the week within a specified date range. It supports localization of weekdays and allows filtering by a custom date range.

**Key Steps:**

**1-Date Range Filtering:**

* Accepts two optional inputs:
  + from: Start date for filtering (default: 2000-01-01)
  + to: End date for filtering (default: today)
* Filters reservations using the date field within this range.

**2-Status Filter:**

* Only includes reservations with status = 3, which typically indicates completed or finished reservations.

**3-Soft Deletes:**

* Uses withTrashed() to include reservations that were soft-deleted, ensuring complete statistics.

**4-Grouping by Weekday:**

* Groups the results by the weekday name (DAYNAME(created\_at)), counting how many orders were placed on each day.

**5-Ordered Weekday Output:**

* Uses SQL’s FIELD() function to maintain the correct weekday order from Sunday to Saturday.

**6-Localization:**

* Each weekday is translated from English to Arabic using a predefined dictionary:
  + Sunday → الأحد
  + Monday → الإثنين
  + Tuesday → الثلاثاء
  + Wednesday → الأربعاء
  + Thursday → الخميس
  + Friday → الجمعة
  + Saturday → السبت

**7-Error Handling:**

* Any thrown error is caught, and a structured error response is returned with a 422 status code.

**46.36** **availableTimes() Method**

**Overview**

The availableTimes() method dynamically generates available appointment times based on system settings, the selected date, and optional reservation source. It supports two scheduling strategies and ensures the returned time slots are unique and sorted.

**Key Steps:**

**1-System Configuration Check:**

* Retrieves the first application setting using Application::first().
* Determines if the scheduling logic should follow the **new way** (sending\_reservations\_to\_representatives == 1) or the **old way**.

**2-Reservation Source Condition:**

* If from\_reservation = 1, it:
  + Uses the appropriate scheduling method (workHoursUsingAreas or workHours).
  + Removes any duplicate appointment times.
  + Returns the appointment data immediately as a JSON response with HTTP status 200.

**3-Appointment Date Validation:**

* If date is provided:
  + Converts the date to a lowercase day abbreviation (e.g., Mon → mon).
  + Checks if the given day is a valid working day via workDays().

**4-Scheduling Strategy:**

* Uses workHoursUsingAreas() if the setting sends reservations to representatives.
* Otherwise, uses workHours().

**5-Future Date Check:**

* For normal (non-reservation) calls without the all flag:
  + Ensures the date is today or in the future.

**6-Duplicate Time Removal:**

* Applies removeDuplicateAppointments() to clean up overlapping or repeated slots.

**7-Time Sorting:**

* Flattens the appointment data structure.
* Sorts all time slots in chronological order using strtotime().
* Rebuilds the final structured array for output.

**8-Response Structure:**

* Maps and formats the sorted times into a consistent array structure for frontend use.
* Returns the formatted result using returnData() with a "done" message.

**46.37 workHoursUsingAreas() Method**

**Overview**

The workHoursUsingAreas() method dynamically determines the available working hours (time slots) for a specific area based on a reservation date and location data. It adapts to whether the system uses shifts and whether a location or coordinates are provided.

**Key Steps:**

**1-Area Detection:**

* If location\_id is available in the request:
  + Retrieves the corresponding location from the database.
  + Uses the checkLocation() method with the location’s latitude and longitude to detect the operational area.
* If latitude (lat) and longitude (lng) are provided directly in the request:
  + Uses checkLocation() with the given coordinates.
* If neither are provided:
  + Uses default coordinates to determine the area.

**→ If no area is detected**, the method returns an empty array of hours.

**2-Fetch Application Settings:**

* Retrieves the first record of Application for system-wide settings like work shifts and acceptance policy.

**3-Date Processing:**

* Parses the reservation date using Carbon.
* Extracts the full name of the weekday (e.g., "Monday") to query for corresponding shifts.

**4-Shifts Retrieval:**

* Fetches all shifts for the detected area\_id and weekday from the Shift model.
* Orders the shifts based on the from time (start of shift).

**5-Representative Count:**

* Uses the getRepresentativesCount() helper method to determine how many representatives are available on the given date and time.

**6-Time Slot Generation Logic:**

* If the application uses work shifts (use\_work\_shift == 1) and shifts are defined:
  + Iterates over each shift and uses generateTimeSlots() to create time slots between from and to times.
  + Merges all the resulting time slots into the $hours array.
* If shifts are not used but global from and to times are defined in the application settings:
  + Generates time slots based on the global from and to values.

**🔁 Final Output:**

* Returns a compiled array of available working hours for the specified area and reservation date, considering the business logic and available representatives.

**46.38** **removeDuplicateAppointments() Method**

**Overview**

The removeDuplicateAppointments() method is a utility function designed to **eliminate duplicate time slots** from an array of appointment data based on their time keys.

**Key Steps:**

**1-Initialization:**

* uniqueAppointments: An array to store only distinct appointments.
* seenKeys: An array to keep track of time keys that have already been processed.

**2-Iteration:**

* Loops through each $appointment in the input array.
* Extracts the **first key** of each appointment array, which represents the appointment time.

**3-Uniqueness Check:**

* If the time key has **not been encountered before**:
  + Adds the appointment to uniqueAppointments.
  + Records the time key in seenKeys to avoid duplicates in the future.

**🔁 Final Output:**

* Returns the uniqueAppointments array containing **only one entry per time slot**, effectively removing duplicate entries.

**46.39 workDays() Method**

**Overview**

The workDays() method is a simple helper function that retrieves and returns the **working days** configured in the application's settings.

**Key Steps:**

**1-Retrieve Settings:**

* Fetches the first record from the Application model which holds the system-wide settings.

**2-Extract Working Days:**

* Checks if working\_days is set and not null in the application settings.
* If valid, it splits the comma-separated string of working days into an array using explode(', ', ...).

**🔁 Final Output:**

* Returns the resulting array of working days (e.g., ['Sunday', 'Monday', 'Wednesday']), or an empty array if none are set.

**46.40 workHours() Method**

**Overview**

The workHours() method retrieves the available reservation hours for a given date, considering shift schedules and representative capacity. It dynamically generates time slots for the specified date, allowing for time-based reservation checks. This method handles both shift-based and fixed hour-based reservations.

### Key Steps:

#### 1-Parse Reservation Date:

The method uses the Carbon library to convert the provided reservation date (Y M d format) into a standard Y-m-d format. This ensures consistency and ease when working with date comparisons and shift retrieval.

#### 2-Retrieve Application Settings:

It fetches the first record from the Application model to get global settings like shift usage (use\_work\_shift), working hours (from, to), and the acceptance limit (accepting).

#### 3-Fetch Shifts for the Day:

The method queries the Shift model for shifts that match the day of the week (e.g., "Monday") based on the parsed reservation date. The shifts are sorted by the start time (from) to determine when each shift begins and ends.

#### 4-Determine Acceptance Capacity:

It checks the accepting field in the Application model to know how many reservations the system can handle. For the special case of "2025-03-11", the acceptance value is overridden to "90".

#### 5-Get Available Representatives:

The method calls getRepresentativesCount() to determine how many representatives are available for the specific reservation time and date.

#### 6-Generate Time Slots:

The method generates time slots based on whether shift scheduling is enabled (use\_work\_shift == 1). If shifts exist, it loops through each shift and calls generateTimeSlots() to generate available time slots within the shift’s start and end times. If shifts are not used, the method defaults to the static working hours (from, to).

#### 7-Return Available Hours:

The method returns an array of available time slots, which represent the hours that can be reserved based on the current system settings.

**46.41 getRepresentativesCount() Method**

**Overview**

The getRepresentativesCount() method calculates the number of available representatives for a given date and time based on the shift schedules and the application's global settings. It determines the number of representatives who are available for service at the specified time, considering whether shifts are used and their assigned staff.

**Key Steps:**

**1-Parse the Date and Time:**

The method starts by parsing the provided date into a day of the week (e.g., "Monday"). It also retrieves the current time using the Carbon::now()->toTimeString() method. If a specific time is provided (in 12-hour format), it converts the time to a 24-hour format.

**2-Fetch Current Shift:**

The method then checks the shift for the current day ($currentDay) by querying the Shift model. It looks for shifts that overlap with the provided or current time ($currentTime). This ensures that the system retrieves the active shift for the given time window.

**3-Check for Work Shift Usage:**

If the center ($center) uses work shifts (use\_work\_shift == 1), the method retrieves the representatives assigned to the current shift by looking up the clients related to the current shift.

**4-Count Available Representatives:**

* If shifts are used, the method counts the number of available representatives assigned to the current shift. It filters for clients who are active (status == 1), are of type 1 (representative), and have valid geographic coordinates (latitude and longitude).
* If shifts are not used, the method simply counts all available representatives (clients) who match the same criteria (active, type 1, and valid location).

**5-Return Representative Count:**

The method returns the total count of available representatives based on the shift or default criteria.

**46.42 generateTimeSlots() Method**

**Overview**

The generateTimeSlots() method generates a series of available time slots for appointments, taking into account various factors such as the requested time window, the number of available representatives, and appointment availability. It ensures that the available time slots are consistent with the specified criteria and reservation date.

**Key Steps:**

**1-Initialize Time Variables:**

The method begins by converting the from and to times into timestamps using strtotime(). It also initializes $tNow to the from time, which is used to generate the time slots iteratively.

**2-Loop Through Time Range:**

A while loop is used to iterate through the time range from $from to $to. For each iteration:

* The method formats the current timestamp ($tNow) into a human-readable time string (e.g., 12:00 PM) using date('H:i A', $tNow).

**3-Check Time Slot Availability:**

* If the $all flag is set to 1, the method simply adds the time slot to the list without checking for any appointment conflicts.
* Otherwise, it calls the checkAppointment() method to verify if the time slot is available based on the provided request, reservation\_date, and representatives\_count. If the time slot is available, it adds it to the list of available hours.

**4-Add Formatted Time Slot:**

For each valid time slot, the method formats the timestamp ($tNow) into a 12-hour formatted string (e.g., 01:00 PM) using Carbon::createFromTimestamp($tNow)->formatLocalized('%I:%M %p'). This formatted time is then added to the $hours array.

**5-Increment Time by Interval:**

The method increments the $tNow timestamp by the specified $acceptingInterval (defaulting to 60 minutes) to move to the next time slot.

**6-Return Time Slots:**

Once the loop completes, the method returns the $hours array containing all the valid time slots, formatted and ready for use in appointment scheduling.

**46.43 checkAppointment() Method**

**Overview**

The checkAppointment() method verifies if a given time slot for an appointment is available based on various conditions such as shifts, the number of representatives, and the reservation date. It ensures that the selected time does not conflict with existing appointments, shift availability, and other constraints.

**Key Steps:**

**1-Parse Time and Date:**

The method first parses the provided reservation\_date and time. If the time includes "AM" or "PM," it adjusts the format for comparison. A targetTime is created using the reservation date and the formatted time.

**2-Shift-Based Appointment Check:**

If the application uses work shifts ($app->use\_work\_shift == 1):

* The method checks if the selected time falls within an active shift for the specified currentDay.
* It retrieves the representatives assigned to the current shift and counts the number of existing reservations for these representatives on the selected date and time.
* If there is a conflict (e.g., the number of appointments exceeds available representatives), the method returns false.

**3-General Appointment Check:**

If the application does not use work shifts:

* The method counts all existing reservations for the specified date and time (from and to times) and checks for any conflicts.
* It compares the existing appointment count with the number of available representatives to ensure there is availability.

**4-Time Constraints:**

The method checks if the current time has passed the selected appointment time or if the time difference between the current time and the target time is less than 15 minutes. If either condition is met, the method returns false to prevent bookings that are too close to the current time.

**5-Shift Handling and Location Check:**

* If the application uses work shifts, the method checks the location based on either the location\_id, latitude, and longitude from the request, or a default location.
* It retrieves shifts based on the day and, if applicable, the area of the location. For each shift, the method divides it into smaller time slots (e.g., 90-minute intervals) and checks if the requested time fits within these slots.
* It compares the reservation count for the selected time slot with the available representatives for that shift.

**6-Return Availability:**

* If the time slot is available (i.e., the number of existing appointments is within the limit of available representatives), the method returns true.
* If no available slots are found or the time conflicts with existing appointments, the method returns false.

**46.44 reservations\_with\_online\_payment\_report() Method**

**Overview**

The reservations\_with\_online\_payment\_report() method retrieves a paginated list of reservations that have been completed and paid for online. It filters the reservations based on a provided online payment ID or returns all reservations with status "3" (indicating successful payments). The method also handles pagination for large result sets and formats the response with additional pagination data.

**Key Steps:**

**1-Handle Request Parameters:**

The method first checks if an online\_payment\_id is provided in the request. If it is, the method filters the reservations based on this payment method and status "3" (indicating completed payments). If no online\_payment\_id is provided, it simply retrieves all reservations with status "3".

**2-Query Reservations:**

The method uses the Reservation model to query the database for reservations that match the given conditions (payment method and status). It then sorts the reservations by the most recent (latest()) and paginates the results with 10 reservations per page. The paginate() method is used to handle large data sets and ensure efficient loading.

**3-Pagination Data:**

To enhance the user experience and facilitate easy navigation through the results, the method prepares pagination information:

* **total**: The total number of reservations.
* **per\_page**: The number of reservations per page.
* **current\_page**: The current page being viewed.
* **total\_pages**: The total number of pages available based on the pagination.

**4-Return Data:**

Once the data is fetched and paginated, the method returns a success response containing:

* A collection of ReservationResource instances representing the reservations.
* The pagination data to assist in navigating through the results. The response is returned with a translated message for the key api.reservation\_all.

**5-Error Handling:**

If any exception occurs during the execution (such as a database error), the method catches the error and returns a 422 status code along with the error message.

**46.45 available\_times\_reservation\_report() Method**

**Overview**

This method is designed to generate a report showing the availability and status of reservations for specific times on a given date. It calculates the number of reservations for each time slot and provides insights into the total, finished, and not finished reservations. Additionally, it calculates the work capacity based on the available representatives.

### ****Key Steps:****

#### ****1-Handling Request Parameters:****

The method accepts a date parameter from the incoming request. The date is then formatted into the Y-m-d format using Carbon::createFromFormat(), ensuring it is in the appropriate format for querying the database.

#### ****2-Fetching Available Appointment Times:****

The method calls an internal method (availableTimes()) to retrieve available appointment times. The response is decoded from JSON and contains a list of available appointment times.

#### ****3-Calculating Reservation Status for Each Time Slot:****

For each available appointment time:

* The method queries the Reservation model to count the total reservations (totalCount) for that specific time on the provided date.
* It also counts how many of those reservations have finished (finishedCount, where status is 3) and how many are not finished (notFinishedCount).
* The data for each time slot is stored in an array (reservationsData), including the time, total reservations, finished reservations, and not finished reservations.

#### ****4-Calculating Work Capacity:****

The method calculates the number of available representatives:

* It queries the Client model to count the active representatives (representatives\_count), excluding 2 (likely reserved for special tasks).
* The work\_capacity is calculated based on the total number of available representatives and the number of available time slots (count\_of\_times).

#### ****5-Returning the Data:****

The method returns a structured response:

* A collection of reservation data, including time, total reservations, finished, and not finished counts.
* The number of available representatives (representatives\_count - 2).
* The calculated work capacity, which reflects how much work can be done based on the available representatives and time slots.

The response is returned in a standardized format using the returnData() method with a data key containing the report data.

#### ****6-Error Handling:****

If any exception or error occurs during execution, the method catches it and returns a 422 error response with the exception message.

**47. Blog Controller**

**47.1 all() Method**

**Overview**

This method retrieves blog entries from the database and returns them either paginated or as a complete list, depending on the input from the request. It ensures flexibility in how the frontend can receive the blog data by checking the presence and value of the paginate parameter.

### ****🔑 Key Steps:****

#### ****1-Checking the**** paginate ****Parameter:****

The method begins by checking if the request contains a paginate parameter using the filled() method. Based on its value, it determines how to return the data:

* If paginate is set to 1, the method returns a paginated list.
* If paginate is set to 0, the method returns the full list of blog entries without pagination.
* If the paginate parameter is not set or has an unexpected value, the method defaults to returning a paginated list.

#### ****2-Retrieving Blog Entries:****

Depending on the result of the conditional checks:

* For paginated results: it retrieves the latest blogs using latest()->paginate(10), which fetches 10 results per page.
* For full results: it uses latest()->get() to retrieve all blog entries.

#### ****3-Structuring the Pagination Data:****

When paginated results are returned, the method constructs a pagination metadata array that includes:

* The total number of blog entries.
* The number of entries per page.
* The current page number.
* The total number of pages.

This metadata helps the frontend manage navigation and display of paginated content.

#### ****4-Returning the Data:****

The method returns the data using the standardized returnData() format. It wraps the blog data with the appropriate resource (BlogResource) and includes the pagination data when relevant. It also attaches a localized message using the \_\_('api.blog\_all') translation key.

#### ****5-Error Handling:****

If any error occurs during execution (such as database issues or unexpected input), the method catches the exception and returns a 403 error response with the error message, ensuring graceful failure.

**47.2 listPage() Method**

**Overview**

This method retrieves all blog entries from the database and passes them to a view for display in the dashboard. It is designed for internal use in the admin or dashboard interface where pagination is not required.

**Key Steps:**

**1-Retrieving Blog Entries:**

The method uses Blog::latest()->get() to fetch all blog records from the database, ordered by the most recent ones first. This ensures that the newest blogs appear at the top of the list when displayed.

**2-Returning the View:**

It returns a view named dashboard.blogs.index, passing the retrieved blogs using the compact() helper. This helper automatically creates an array with the key 'blogs' and the value being the $blogs collection. The view can then use this data to render the blog list.

**47.3 detailsPage($id) Method**

**Overview**

This method retrieves the details of a single blog entry based on its ID and displays it in a dedicated view within the dashboard. It ensures that only existing blog entries can be viewed, otherwise it throws an error.

**Key Steps:**

**1-Finding the Blog Entry:**

The method uses Blog::findOrFail($id) to look up the blog by its ID. If a blog with the given ID exists, it returns the corresponding model instance. If not, it throws a 404 error automatically, preventing further execution and showing a "Not Found" error page.

**2-Returning the View:**

It returns a view named dashboard.blogs.show, passing the retrieved blog object to the view using the compact() helper. This allows the view to access the blog's data directly and render its details such as title, content, creation date, etc.

**47.4 show($id) Method**

**Overview**

This method is responsible for retrieving a specific blog entry by its ID through an API endpoint and returning a structured JSON response. It includes error handling for cases where the blog does not exist.

**Key Steps:**

**1-Retrieving the Blog:**

It attempts to find a blog record using Blog::find($id). This method returns the blog model if found or null if not.

**2-Handling Missing Blog:**

If the blog is not found, it returns a standardized error response using $this->returnError() with a 404 status code and a message indicating that the blog was not found.

**3-Returning Blog Data:**

If the blog is successfully found, it returns the data using $this->returnData(), with the blog data included in a data key and labeled as blog.

**47.5 add(Request $request) Method**

**Overview**

This method handles the creation of a new blog post through an API request. It validates the input data and returns a structured JSON response containing the newly created blog.

**Key Steps:**

**1-Validating Input:**

It ensures that both title and body fields are present and are strings. If validation fails, Laravel automatically returns a validation error response.

**2-Creating the Blog:**

Once validated, it creates a new blog post using the create() method and only the title and body fields from the request.

**3-Returning Blog Data:**

After creation, it returns a success response using $this->returnData(), which includes the newly created blog inside the data key under blog.

**47.6 update(Request $request, $id) Method**

**Overview**

This method is responsible for updating an existing blog post by its ID. It performs input validation, ensures the blog exists, and then updates the specified fields.

**Key Steps:**

**1-Finding the Blog:**

It uses the provided $id to locate the blog post. If no matching blog is found, it returns an error response with a 404 status.

**2-Validating Input:**

It checks that the title and body fields are present in the request and are of type string. If validation fails, Laravel automatically handles the response.

**3-Updating the Blog:**

After validation and finding the blog, it updates the title and body fields using the update() method.

**4-Returning Updated Data:**

It returns a success response with the updated blog post included inside the data key under blog.

**47.7 destroy($id) Method**

**Overview**

This method handles the deletion of a blog post by its unique identifier. It ensures the blog exists before attempting to delete it and returns appropriate feedback.

**Key Steps:**

**1-Locating the Blog:**

It tries to find the blog using the provided $id. If the blog does not exist, it returns a 404 error response indicating that the blog was not found.

**2-Deleting the Blog:**

If the blog is found, it proceeds to delete it from the database using the delete() method.

**3-Returning a Success Response:**

After successful deletion, it returns a response indicating that the blog has been deleted successfully.