# Introduction

## This documentation provides an overview of the QT Broker application, a web application built using the Laravel framework.

## **For Users:**

## **Account Summary:** View a comprehensive overview of your financial activity.

## **Portfolio:** Monitor and manage your investment portfolio.

## **Wallet:** Access and manage your digital wallet.

## **Deposit:** Easily deposit funds into your account.

## **Withdraw:** Withdraw funds from your account.

## **Request Refund:** Initiate refund requests for investments that are not yet matured.

## **Refer a Friend:** Earn rewards by referring friends to the platform.

## **For Admins:**

## **Trade Management:** Oversee and manage all trade activities.

## **Investment Management:** Monitor and manage investment portfolios.

## **Share Claim Processing:** Handle and process share claims efficiently.

## **User Management:** Create, edit, and manage user accounts.

## **Deposit Processing:** Process and verify user deposits.

## **Withdrawal Processing:** Process and authorize user withdrawals.

## **Refund Management:** Handle and process refunds.

## **Referral Program:** Monitor and manage the referral program.

## **Investment Plan Management:** Create, edit, and manage investment plans.

## **Payment Option Configuration:** Configure and manage available payment options.

## **Technical Stack:**

## **Backend:** Laravel Framework

## **Frontend:** Tailwind CSS, Vite

## **Authentication:** Laravel Breeze

## **Local Development Setup:**

## To set up the project locally, follow these steps:

## **Clone the Repository:**

## git clone https://github.com/your-username/your-project-name.git1

## **Install Dependencies:**

## cd your-project-name

## composer install

## npm install

## **Configure Environment Variables:** Create a .env file based on the .env.example file, and populate it with your database credentials and other necessary environment variables.

## **Database Setup:** Create a database for your project and update the database credentials in your .env file. Run the database migrations:

## php artisan migrate

## **Start the Development Server:** Start the Laravel development server and Vite:

## php artisan serve

## 

## 

## 

# Routing Structure:

**Public Routes:**

* /notification: This route likely retrieves notifications for the user and displays them on the homepage. (Controller: HomeController)
* /admin/\*: These routes handle various admin functionalities like signup, login, onboarding, and profile management. (Controller: AdminController)
* Routes for deposits, withdrawals, claiming shares, rating, and viewing the wallet: These routes handle user interactions related to financial transactions and account management. (Controllers: TransactionController, SharesController, ReviewController, WalletController)
* /dashboard: This route displays the user dashboard, likely requiring both authentication and email verification. (Implemented with anonymous function)
* Routes for trading, investments, portfolio, refunds, and referral: These routes manage user interactions related to investments, trading, and other financial activities. (Controllers: TradeController, InvestmentController, PortfolioController, RefundController, ReferralController)

**Authenticated Routes (require login):**

These routes are only accessible to users who have logged in successfully.

* Routes starting with /profile: These routes handle user profile management functionalities like editing, updating, and deleting the profile. (Controller: ProfileController)

**Admin Routes (require login and admin role):**

These routes are only accessible to users with the "admin" role.

* Routes starting with /admin: These routes handle various admin functionalities like managing deposits, withdrawals, payment options, trades, investments, claimed shares, refunds, users, referrals, etc. (Controllers: TransactionController, PaymentOptionController, TradeController, InvestmentController, SharesController, RefundController, UserController, ReferralController)
* Routes for protected files and claim files: These routes handle retrieving protected files and claim files stored on the server. (Implemented with anonymous functions)

# Controller Logic Breakdown

## Transaction Controller

This code defines a TransactionController in a Laravel application, handling various deposit and withdrawal transactions and approval and rejection for admin:

**1.** generateUniqueTransactionId()**:**

* Generates a unique transaction ID using a combination of 'TRD' and a random 8-character string.
* Ensures uniqueness by checking the database for existing transactions with the same ID.

**2.** create\_deposit() **and** create\_withdrawal()**:**

* These methods retrieve payment options and generate a unique transaction ID.
* They then render the deposit and withdrawal views, respectively, passing the transaction ID and payment options.

**3.** storeTransaction()**:**

* Validates the incoming request data (amount, type, and other relevant fields).
* Creates a new Transaction record, associating it with the authenticated user.
* Redirects the user to the appropriate route (deposit or withdrawal) with a success message.

**4.** adminViewDeposits() **and** adminViewWithdrawals()**:**

* These methods retrieve all deposits or withdrawals, respectively, and pass them to the corresponding admin views.

**5.** admin\_update()**:**

* Updates a deposit or withdrawal based on the provided input (status, amount disbursed).
* Redirects the admin to the deposits or withdrawals list with a success message.

**6.** approve()**:**

* Approves a transaction and updates its status.
* If the transaction is a deposit:
  + Creates a wallet for the user if it doesn't exist.
  + Increments the user's wallet balance.
  + If it's the user's first deposit and they have a referral code, creates a referral record.
  + Sends an email notification to the user.
* If the transaction is a withdrawal:
  + Decrements the user's wallet balance.
  + Sends an email notification to the user.

**7.** decline()**:**

* Declines a transaction and updates its status.

**Key Points:**

* **Transaction ID Generation:** Ensures unique transaction IDs.
* **Transaction Creation:** Creates new transactions with necessary details.
* **Admin View:** Provides admin interface to view and manage deposits and withdrawals.
* **Transaction Approval/Decline:** Updates transaction status, modifies wallet balances, and sends notifications.
* **Referral System:** Handles referral bonuses for first-time deposits.

## Trade Controller

**User-Facing Features:**

1. index()**:**
   * Renders the trade page, providing a form for users to input trade details.
   * Fetches available payment options to display on the form.
2. tradeHistory()**:**
   * Retrieves a list of all trades, ordered by their creation date.
   * Renders the trade history page, displaying the list of trades.
3. store()**:**
   * Validates the incoming trade data (amount, market, pair, timer, trade action, stop loss).
   * Decrements the user's wallet balance by the trade amount.
   * Creates a new Trade record, storing the user ID, trade details, and other relevant information.
   * Redirects the user back to the trade page with a success message.

**Admin-Facing Features:**

1. admin\_view\_trades()**:**
   * Retrieves all trades, ordered by their creation date.
   * Renders the admin trade view, displaying the list of trades.
2. admin\_update\_trade()**:**
   * Validates the incoming update data (PNL, PNL amount).
   * Updates the trade record with the new PNL and status.
   * Updates the user's wallet balance based on the PNL (profit or loss).
   * Redirects the admin back to the trades list with a success message.

**Key Points:**

* **Trade Creation:** Handles the creation of new trades, validating input and updating the user's wallet balance.
* **Trade History:** Provides a view for users and admins to see past trades.
* **Admin Trade Management:** Allows admins to update trade status, PNL, and wallet balances.
* **Wallet Integration:** Interacts with the Wallet model to update balances.

## Shares Controller

**Claiming Shares:**

1. claimShares()**:**
   * Validates the user's uploaded documents (letter explanation, signed contract, breach proof).
   * Generates unique filenames with timestamps for uploaded documents.
   * Stores the uploaded documents in designated folders (uploads/letter\_explanations, etc.).
   * Creates a new ClaimedShares record with user ID, document paths, and other relevant data.
   * Decrements the user's wallet balance by a fixed amount (currently 2228, assuming a dynamic value is needed).
   * Redirects the user to the dashboard with success and claim ID messages.

**Admin Management:**

1. admin\_view\_claimed\_shares()**:**
   * Retrieves all claimed shares with associated user information, ordered by creation date.
   * Renders the admin view for claimed shares, displaying the list of claims.
2. approveClaim()**:**
   * Finds the specific claim by ID.
   * Updates the claim status to 'approved' (or desired status).
   * Increments the user's wallet's share count by a fixed amount (currently 5000, assuming a dynamic value is needed).
   * Saves the updated claim.
   * Sends an email notification to the user using ClaimApprovedMail.
   * Redirects the admin back to the claimed shares list with a success message.
3. declineClaim()**:**
   * Finds the claim by the provided ClaimedShares object.
   * Updates the claim status to 'rejected'.
   * Increments the user's wallet balance by the claim amount (currently 2228, assuming a dynamic value is needed).
   * Saves the updated claim.
   * Sends an email notification to the user using ClaimDeclinedMail.
   * Redirects the admin back with a success message.

## Refund Controller

**User-Facing Features:**

1. index()**:**
   * Retrieves the user's active investments.
   * Renders the refund page, displaying the list of eligible investments for refund.
2. store()**:**
   * Validates the incoming refund request (investment ID, reason).
   * Checks for existing pending refunds for the same investment.
   * Generates a unique refund ID.
   * Creates a new Refund record with user ID, investment ID, reason, and other relevant data.
   * Redirects the user back to the refund page with a success message and the refund ID.

**Admin-Facing Features:**

1. admin\_view\_refunds()**:**
   * Retrieves all refund requests, ordered by creation date.
   * Renders the admin refund view, displaying the list of refund requests.
2. admin\_update\_refund()**:**
   * Validates the incoming request (ID, type, amount for approval).
   * Finds the specific refund by ID.
   * Based on the type parameter:
     + **Approval:**
       - Updates the investment status to 'refunded'.
       - Updates the refund status to 'approved'.
       - Calculates the refund amount and updates the wallet balance.
       - Saves the updated investment and refund records.
     + **Rejection:**
       - Updates the refund status to 'rejected'.
       - Saves the updated refund record.
   * Redirects the admin back to the refund list with an appropriate success message.

**Key Points:**

* **Refund Request:** Handles user requests for refunds, including validation and creation.
* **Admin Review:** Provides an interface for admins to review and process refund requests.
* **Refund Processing:** Updates investment status, refund status, and user wallet balance.
* **Email Notifications:** While not explicitly shown in the code, consider sending email notifications to users about the status of their refund requests.

## ReferralController

This code defines a ReferralController in a Laravel application, handling functionalities related to referral programs. It includes methods for generating unique referral codes, displaying user's referral link, managing referrals from the admin side, and updating referral status and crediting bonuses.

**Methods:**

1. generateUniqueId()  
   * Generates a unique referral code for the authenticated user.
   * Ensures uniqueness by checking if the code already exists in the database.
2. view()  
   * Retrieves the user's referral code or generates a new one if it doesn't exist.
   * Renders the referral view, displaying the referral code.
3. index()  
   * Retrieves all referral records.
   * Renders the admin view for referrals, displaying the list of referrals.
4. update()  
   * Validates the incoming request, ensuring the amount is provided.
   * Updates the referral status to 'approved'.
   * Increments the referer's wallet bonus by the specified amount.
   * Updates the referral's date disbursed and bonus amount.
   * Saves the updated referral record.
   * Redirects the admin back to the referral list with a success message.

**Key Points:**

* **Referral Code Generation:** Ensures unique referral codes for each user.
* **Referral Tracking:** Stores information about referrals and their status.
* **Admin Interface:** Provides an admin interface to view and manage referrals.
* **Bonus Crediting:** Updates the referer's wallet balance upon referral approval.

## PaymentOptionController

**Overview**

It provides methods for managing payment options, including creating and deleting them.

**Methods:**

1. index()  
   * Retrieves all payment options from the database.
   * Renders the admin view for payment options, displaying the list of options.
2. store()  
   * Validates the incoming request data (name and account details).
   * Creates a new PaymentOption record with the validated data.
   * Redirects the admin back to the payment options list with a success message.
3. destroy()  
   * Deletes the specified payment option.
   * Redirects the admin back to the payment options list with a success message.

**Key Points:**

* **Payment Option Management:** Allows admins to create and delete payment options.
* **Data Validation:** Ensures that the input data for creating a payment option is valid.
* **Error Handling:** While not explicitly shown in this code, it's important to implement error handling to catch potential exceptions and provide informative error messages.

## InvestmentPlanController

**Overview**

It provides methods for managing investment plans, including creating, updating, and deleting them.

**Methods:**

1. index()  
   * Retrieves all investment plans from the database.
   * Renders the admin view for investment plans, displaying the list of plans.
2. store()  
   * Validates the incoming request data (name, description, price, ROI, and duration).
   * Creates a new InvestmentPlan record with the validated data.
   * Redirects the admin back to the investment plans list with a success message.
3. update()  
   * Validates the incoming request data (name, description, price, ROI, and duration).
   * Updates the specified investment plan with the validated data.
   * Redirects the admin back to the investment plans list with a success message.
4. destroy()  
   * Deletes the specified investment plan.
   * Redirects the admin back to the investment plans list with a success message.

## InvestmentController

This handles functionalities related to user investments and admin management of investments.

**User-Facing Features:**

1. generateUniqueTransactionId()  
   * Generates a unique transaction ID for an investment (INV + random string).
2. index()  
   * Retrieves available investment plans.
   * Retrieves the user's wallet balance.
   * Renders the investment page, displaying investment plans and user's balance.
3. store()  
   * Validates the selected investment plan ID.
   * Checks if the user's wallet balance is sufficient for the chosen investment plan.
   * If sufficient balance:
     + Generates a unique transaction ID.
     + Creates a new Investment record with user ID, investment plan ID, transaction ID, and investment amount.
     + Decrements the user's wallet balance by the investment amount.
     + Sends an email notification to the user about the investment (using InvestmentStartedMail).
   * If insufficient balance:
     + Redirects back with an 'Insufficient balance' error message.
   * Redirects back to the portfolio with a success message and investment ID upon successful investment.

**Admin-Facing Features:**

1. admin\_view\_investments()  
   * Retrieves all investment records, ordered by creation date (descending).
   * Renders the admin investment view, displaying the list of investments.
2. admin\_update\_investment()  
   * Validates the request data (investment ID, profit/loss indicator, and amount).
   * Finds the specified investment record based on the ID.
   * Calculates the approved amount based on the investment amount and profit/loss:
     + For loss: approved amount = investment amount - claimed loss amount.
     + For profit: approved amount = investment amount + claimed profit amount.
   * Updates the investment record with:
     + Claimed profit/loss amount.
     + Disbursed amount (approved amount).
     + Profit/loss indicator (loss or profit).
     + Disbursement date and time.
     + Disbursed by (admin user ID).
     + Status ID (presumably set to 'disbursed').
   * Updates the user's wallet balance by adding the approved amount.
   * Sends an email notification to the user about the investment disbursement (using InvestmentDisbursedMail).
   * Redirects back with a success message upon successful update.

**Key Points:**

* **Investment Management:** Allows users to invest in plans and admins to manage investments.
* **Balance Check:** Ensures users have sufficient balance before investing.
* **Transaction ID:** Generates unique IDs for each investment.
* **Profit/Loss Handling:** Enables admins to process profit/loss claims for investments.
* **Email Notifications:** Sends emails to users about investment activity.

## **Sending Email Notifications**

The application utilizes Laravel's built-in email system to send notifications. This system leverages the Mail facade to easily define and send emails.

**Key Components:**

1. **Mail Class:**
   * A dedicated mail class is created for each type of notification (e.g., TransactionApprovedMail).
   * This class extends Laravel's Mailable class.
   * It defines the email's subject, body, and any attachments.
2. **Mail Facade:**
   * The Mail facade provides a convenient way to send emails.
   * It's used to instantiate the mail class and send it to the specified recipient.

**Example:**

PHP

// In TransactionController.php

public function approve(Request $request) {

// ... (other logic)

// Send email notification

\Mail::to($transaction->user->email)->send(new \App\Mail\TransactionApprovedMail($transaction, 'withdrawal'));

// ... (other logic)

}

**Mail Class Example:**

PHP

namespace App\Mail;

use Illuminate\Bus\Queueable;

use Illuminate\Contracts\Queue\ShouldQueue;

use Illuminate\Mail\Mailable;

use Illuminate\Queue\SerializesModels;1

class TransactionApprovedMail extends Mailable

{

use Queueable, SerializesModels;

public $transaction;

public $type;

/\*\*

\* Create a new message instance.

\*

\* @return void

\*/

public function \_\_construct($transaction,2 $type)

{

$this->transaction = $transaction;

$this->type = $type;

}

/\*\*

\* Build the message.

\*

\* @return $this

\*/

public function build()

{

return $this->subject('Your ' . $this->type . ' Transaction is Approved')

->view('emails.transaction\_approved')

->with([

'transaction' => $this->transaction,

'type' => $this->type,

]);

}

}

**Key Points:**

* **Mail Configuration:** Ensure your config/mail.php file is configured with the correct mail driver (e.g., SMTP, Mailgun, Sendmail) and its respective credentials. Right now the app makes use of the SMTP configuration and the credentials are set up in the env SMTP variables.
* **Email Templates:** Create email templates using Blade templates in the resources/views/emails directory.
* **Queueing Emails:** For large-scale applications, consider using Laravel's queue system to asynchronously process email sending.
* **Testing:** Test your email notifications to ensure they are sent correctly and formatted properly.