To implement this wallet management system, we will break down the process into several parts. We'll start with the **database tables**, move to **wallet and transaction management**, handle **concurrency** with locking mechanisms, and lastly, build **unit tests** to verify the correctness of the system.

1. Database Structure

You already have the table structures for wallets and transactions. Below is how they can be defined in Laravel's migration system.

Migration for wallets Table:

```
use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;


class CreateWalletsTable extends Migration

{

   public function up()

   {

      Schema::create('wallets', function (Blueprint $table) {

         $table->id();

         $table->foreignId('user_id')->constrained('users')->onDelete('cascade');

         $table->decimal('balance', 15, 2)->default(0);

         $table->timestamps();

      });

   }


   public function down()

   {

      Schema::dropIfExists('wallets');

   }

}
```

Migration for transactions Table:

```
use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;
```

```
use Illuminate\Support\Facades\Schema;


class CreateTransactionsTable extends Migration
{
    public function up()
    {
        Schema::create('transactions', function (Blueprint $table) {
            $table->id();
            $table->foreignId('wallet_id')->constrained('wallets')->onDelete('cascade');
            $table->enum('type', ['deposit', 'withdrawal', 'rebate']);
            $table->decimal('amount', 15, 2)->nullable();
            $table->timestamps();
        });
    }


    public function down()
    {
        Schema::dropIfExists('transactions');
    }
}
```

2. Wallet Management

We'll now implement the **Wallet** and **Transaction** models, followed by the necessary functions for deposit, withdrawal, and rebate calculation.

Wallet Model

```
namespace App\Models;


use Illuminate\Database\Eloquent\Model;


class Wallet extends Model
{
    protected $fillable = ['user_id', 'balance'];
```

```
public function transactions()
{
    return $this->hasMany(Transaction::class);
}

// Method to perform deposit and rebate calculation
public function deposit($amount)
{
    // Use a database transaction to ensure consistency
    \DB::transaction(function () use ($amount) {
        $this->balance += $amount;
        $this->save();

        // Record the deposit transaction
        Transaction::create([
            'wallet_id' => $this->id,
            'type' => 'deposit',
            'amount' => $amount
        ]);

        // Dispatch rebate job asynchronously
        \App\Jobs\CalculateRebate::dispatch($this, $amount);
    });
}

// Method to perform withdrawal
public function withdraw($amount)
{
    // Ensure the wallet has enough funds
    if ($this->balance < $amount) {
```

```
            throw new \Exception("Insufficient funds");
        }

        // Use a database transaction for atomicity
        \DB::transaction(function () use ($amount) {
            $this->balance -= $amount;
            $this->save();

            // Record the withdrawal transaction
            Transaction::create([
                'wallet_id' => $this->id,
                'type' => 'withdrawal',
                'amount' => $amount
            ]);
        });
    }
}
```

Transaction Model

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Transaction extends Model
{
    protected $fillable = ['wallet_id', 'type', 'amount'];

    public function wallet()
    {
        return $this->belongsTo(Wallet::class);
    }
}
```

3. Job for Rebate Calculation

We will need to create a **Job** to handle the rebate calculation asynchronously.

Create the Job

php artisan make:job CalculateRebate

CalculateRebate Job

```
namespace App\Jobs;

use App\Models\Wallet;
use App\Models\Transaction;

class CalculateRebate extends Job
{
    protected $wallet;
    protected $depositAmount;

    public function __construct(Wallet $wallet, $depositAmount)
    {
        $this->wallet = $wallet;
        $this->depositAmount = $depositAmount;
    }

    public function handle()
    {
        // Calculate 1% rebate
        $rebate = $this->depositAmount * 0.01;

        // Use a transaction for consistency
        \DB::transaction(function () use ($rebate) {
            $this->wallet->balance += $rebate;
            $this->wallet->save();
```

```
            // Record the rebate transaction
            Transaction::create([
                'wallet_id' => $this->wallet->id,
                'type' => 'rebate',
                'amount' => $rebate
            ]);
        });
    }
}
```

4. Concurrency Handling

To ensure **accuracy in concurrent operations** like deposit and withdrawal, we can use **database locking**. Laravel offers two main types of locking mechanisms:

1. **Pessimistic Locking:** This prevents other transactions from accessing the wallet record until the current transaction is complete.
2. **Optimistic Locking:** This allows simultaneous operations but uses versioning or timestamps to resolve conflicts when a transaction is updated.

Pessimistic Locking (example for deposits and withdrawals):

```
// Using lock for deposit
$this->lockForUpdate()->deposit($amount);

// Using lock for withdrawal
$this->lockForUpdate()->withdraw($amount);
```

5. API Endpoints

We will create routes and controllers for the API.

WalletController

```
namespace App\Http\Controllers;

use App\Models\Wallet;
use Illuminate\Http\Request;

class WalletController extends Controller
{
```

```
    public function deposit(Request $request, $walletId)
    {
        $wallet = Wallet::findOrFail($walletId);
        $wallet->deposit($request->amount);
        return response()->json(['message' => 'Deposit successful']);
    }

    public function withdraw(Request $request, $walletId)
    {
        $wallet = Wallet::findOrFail($walletId);
        try {
            $wallet->withdraw($request->amount);
            return response()->json(['message' => 'Withdrawal successful']);
        } catch (\Exception $e) {
            return response()->json(['message' => $e->getMessage()], 400);
        }
    }

    public function balance($walletId)
    {
        $wallet = Wallet::findOrFail($walletId);
        return response()->json(['balance' => $wallet->balance]);
    }

    public function transactionHistory($walletId)
    {
        $wallet = Wallet::findOrFail($walletId);
        return response()->json(['transactions' => $wallet->transactions]);
    }
}
```

Routes

```
Route::post('wallets/{walletId}/deposit', [WalletController::class, 'deposit']);
Route::post('wallets/{walletId}/withdraw', [WalletController::class, 'withdraw']);
Route::get('wallets/{walletId}/balance', [WalletController::class, 'balance']);
Route::get('wallets/{walletId}/transactions', [WalletController::class, 'transactionHistory']);
```

6. Unit Tests

You will write tests for **deposit**, **withdrawal**, and **concurrency handling**.

Deposit with Rebate Test

```
public function testDepositWithRebate()
{
    $wallet = Wallet::create(['user_id' => 1, 'balance' => 0]);

    // Deposit 100
    $wallet->deposit(100);

    $this->assertEquals(100, $wallet->balance);
    $this->assertDatabaseHas('transactions', ['amount' => 100, 'type' => 'deposit']);
    $this->assertDatabaseHas('transactions', ['amount' => 1, 'type' => 'rebate']);
}
```

Concurrent Deposits Test

```
public function testConcurrentDeposits()
{
    $wallet = Wallet::create(['user_id' => 1, 'balance' => 0]);

    // Simulate two concurrent deposits
    \Parallel\run(function () use ($wallet) {
        $wallet->deposit(100);
    });

    \Parallel\run(function () use ($wallet) {
        $wallet->deposit(200);
    });
```

```
    $this->assertEquals(300, $wallet->balance); // Correct total balance after both deposits
}
```

7. Documentation on Concurrency Handling

- **Pessimistic Locking:** When performing operations on the wallet, we ensure that the record is locked until the transaction is completed. This prevents race conditions.
- **Optimistic Locking:** This could be used if you'd prefer allowing concurrent reads but checking for conflicts before performing updates.

Final Steps

1. **Run Migrations:** php artisan migrate
2. **Queue Worker:** Ensure the queue is running for the asynchronous rebate job: php artisan queue:work
3. **Test API:** Test deposit, withdrawal, and transaction history endpoints.

This structure ensures the system is robust and scalable for concurrent transactions.