Stripe Connect Marketplace Integration Plan

Overview

This plan integrates Stripe Connect into GoHappyGo to enable marketplace payments where:

- **Requesters (Buyers)** pay for delivery requests
- **Travelers (Sellers)** receive payments upon request fulfillment
- Users can act as both buyers and sellers depending on the scenario

Escrow Payment Model

Critical Payment Flow:

GoHappyGo uses an **escrow payment model** to protect both requesters and travelers:

1. **Payment is Charged Immediately:**
    - For instant travels: When requester creates request
    - For non-instant travels: When traveler accepts request
    - Funds are **charged from requester's card** and **held in escrow** on the platform's Stripe account
2. **Funds are Held in Escrow:**
    - Money is NOT transferred to traveler immediately
    - Platform holds the funds until requester confirms completion
    - This protects requester (ensures service delivery) and traveler (ensures payment)
3. **Transfer Happens on Completion:**
    - When requester clicks "Complete Request" (indicating everything went well)
    - Backend transfers the traveler's portion to their Stripe Connect account
    - Platform fee remains on platform account
    - Only then does the traveler receive payment

Why Escrow?

- □ **Protects Requesters**: Ensures travelers deliver before receiving payment
- □ **Protects Travelers**: Ensures payment is secured before travel
- □ **Dispute Resolution**: Funds can be held if issues arise
- □ **Builds Trust**: Both parties are protected

Payment Flow Summary:

```
Requester Creates Request (Instant) OR Traveler Accepts (Non-Instant)
  ↓
Payment Charged → Funds Held in Escrow (Platform Account)
  ↓
Travel Takes Place
  ↓
Requester Completes Request → Funds Transferred to Traveler
  ↓
Traveler Receives Payment
```

Stripe Terminology Explained

1. **Stripe Connect Account (Connected Account)**

A Stripe account created for each user (traveler) who will receive payments. In marketplace models, these are "Custom" accounts that belong to your platform users.

Types:

- **Express Account**: Simplified onboarding, Stripe handles most compliance
- **Custom Account**: Full control, you handle compliance (required for PSD2 in France)
- **Deferred Account**: Created with minimal info, fully onboarded later

For GoHappyGo: We'll use **Custom Accounts with Deferred Onboarding** - create account during registration with minimal info (firstName, lastName, email, country), complete onboarding later when user wants to receive payments.

2. **Account Link**

A URL that redirects users to Stripe-hosted onboarding pages to complete their account setup. Required for PSD2 compliance in France.

Parameters:

- `account`: The Stripe Connect account ID
- `refresh_url`: Where to redirect if user clicks "Skip" or session expires (e.g., `/settings/payments?refresh=true`)
- `return_url`: Where to redirect after successful onboarding (e.g., `/settings/payments?success=true`)
- `type`: `account_onboarding` for initial setup, `account_update` for updates

Flow:

1. User clicks "Complete Payment Setup"
2. Backend creates Account Link via Stripe API
3. User redirected to Stripe-hosted page
4. User completes KYC/onboarding on Stripe
5. Stripe redirects back to `return_url`
6. Webhook `account.updated` confirms completion

3. **Payment Intent**

Represents a payment attempt. In marketplaces, it's created with `on_behalf_of` to specify which connected account receives the funds.

States:

- `requires_payment_method`: Needs card details
- `requires_confirmation`: Card attached, needs confirmation
- `requires_action`: 3D Secure required
- `processing`: Payment being processed
- `succeeded`: Payment successful
- `canceled`: Payment canceled

For GoHappyGo (ESCROW MODEL):

- Created when requester submits request (instant) or when traveler accepts (non-instant)
- **Created on platform account** (NOT on connected account) - funds held in escrow
- Uses `application_fee_amount` for platform fee
- **NO automatic transfer** - funds stay on platform until request is completed
- When requester completes request, funds are manually transferred via Stripe Transfers API
- This ensures travelers only receive payment after requester confirms everything went well

4. **Deferred Account**

A Stripe Connect account created with minimal information, allowing users to receive payments later after completing full onboarding.

Minimal Info Required:

- `type: 'custom'`
- `country`: ISO country code (e.g., "FR", "US")
- `email`: User email
- `capabilities[card_payments][requested]: true`
- `capabilities[transfers][requested]: true`

Benefits:

- Users can register without full KYC
- Onboarding happens only when they want to receive payments
- Better UX - no friction during registration

5. **Transfer**

Moves funds from platform's Stripe account to a connected account. In marketplaces, this happens automatically via Payment Intent with `on_behalf_of`.

6. **Application Fee (Platform Fee)**

The amount the platform keeps from each transaction. Calculated using `PlatformPricingService.calculateFee()`.

7. **Webhook Events**

Stripe sends events to your webhook endpoint. Key events:

- `payment_intent.succeeded`: Payment completed
- `payment_intent.payment_failed`: Payment failed
- `account.updated`: Connected account updated (onboarding complete)
- `charge.refunded`: Refund processed

Stripe Elements, Payment Methods, and PCI Compliance

What are Stripe Elements?

Stripe Elements is a set of pre-built, customizable UI components that securely collect payment information directly in your frontend application. The key benefit: **card data never touches your servers**, which means you don't need PCI compliance certification.

Components:

- `CardElement`: Single component that collects card number, expiry, CVC, and postal code
- `CardNumberElement`, `CardExpiryElement`, `CardCvcElement`: Individual components for custom layouts
- `PaymentRequestButton`: Apple Pay / Google Pay button

How it works:

1. Frontend loads Stripe.js library with your `STRIPE_PUBLISHABLE_KEY`
2. Stripe Elements creates iframes that securely collect card data
3. Card data is tokenized client-side into a Payment Method
4. Only the `paymentMethodId` (token) is sent to your backend
5. Your backend never sees raw card numbers, expiry dates, or CVCs

What are Payment Methods?

A **Payment Method** is a tokenized representation of a customer's payment instrument (card, bank account, etc.). It's created by Stripe Elements on the frontend and can be:

- **Attached** to a Payment Intent for immediate payment
- **Saved** to a Customer for future use
- **Reused** for multiple payments (with customer consent)

Payment Method Object Structure:

```
{
  "id": "pm_1234567890",
  "type": "card",
  "card": {
    "brand": "visa",
    "last4": "4242",
    "exp_month": 12,
    "exp_year": 2025
  },
  "customer": "cus_xxx" // if saved to customer
}
```

Key Properties:

- `id`: The Payment Method ID (what you store, not card details)
- `type`: Usually "card" for credit/debit cards
- `card.last4`: Last 4 digits (safe to display)
- `card.brand`: Visa, Mastercard, etc. (safe to display)

How This Enables PCI Compliance

PCI DSS (Payment Card Industry Data Security Standard) requires strict security measures if you handle card data. By using Stripe Elements and Payment Methods:

1. **No Card Data on Your Servers**: Card numbers, CVCs, and expiry dates never reach your backend
2. **Reduced PCI Scope**: You're only responsible for handling the tokenized Payment Method ID
3. **Stripe Handles PCI**: Stripe is PCI Level 1 certified, so they handle all sensitive data
4. **Compliance by Design**: Using Stripe Elements automatically keeps you compliant

What You Store:

- ✅ `paymentMethodId` (e.g., "pm_1234567890") - Safe to store
- ✅ `card.last4` - Safe to display to users
- ✅ `card.brand` - Safe to display
- ❌ Never store: card number, CVC, full expiry date

Payment Flow with Elements and Payment Methods

For Instant Travels (Payment at Request Creation - ESCROW):

1. **Frontend:**
   - User fills out request form
   - Stripe Elements collects card details in secure iframe
   - User clicks "Submit Request"
   - Frontend calls `stripe.createPaymentMethod({ type: 'card', card: cardElement })`
   - Frontend receives `paymentMethodId`
   - Frontend sends request data + `paymentMethodId` to backend
2. **Backend:**
   - Receives `paymentMethodId` (not raw card details)
   - Calculates total amount using `PlatformPricingService`
   - Creates Payment Intent on **platform account** (escrow):

        - `payment_method: paymentMethodId`
        - `amount`: Total amount (travelerPayment + fee + TVA)
        - `application_fee_amount`: Platform fee + TVA on fee
        - **NO `on_behalf_of`** - funds stay on platform
        - `metadata`: Store `travelerPaymentAmount` for later transfer
    - Confirms Payment Intent (may require 3D Secure)
    - **Funds are charged and held in escrow**
    - Returns Payment Intent status to frontend

3. **Frontend:**

    - If Payment Intent status is `requires_action` (3D Secure):
        - Call `stripe.handleCardAction(paymentIntent.client_secret)`
        - User completes 3D Secure challenge
    - If `succeeded`: Show success, request created
    - If `failed`: Show error, allow retry

4. **Later - When Requester Completes Request:**

    - Requester clicks "Complete Request" button
    - Frontend calls `PATCH /api/request/{id}/complete`
    - Backend transfers funds to traveler (see Phase 8)
    - Frontend shows success message

For Non-Instant Travels (Payment at Acceptance - ESCROW):

1. **Frontend (Request Creation):**

    - User fills out request form
    - Stripe Elements collects card details
    - Frontend creates Payment Method: `stripe.createPaymentMethod()`
    - Frontend sends request data + `paymentMethodId` to backend
    - Backend stores `paymentMethodId` in `RequestEntity`
    - **No payment processed yet**

2. **Backend (Request Creation):**

    - Creates `RequestEntity` with status `'NEGOTIATING'`
    - Stores `paymentMethodId` in `RequestEntity.paymentMethodId`
    - No payment processed yet

3. **Backend (Traveler Accepts):**

    - When traveler accepts request
    - Calculate total amount using `PlatformPricingService`
    - Create Payment Intent on **platform account** (escrow):
        - `payment_method`: Stored `paymentMethodId`
        - `amount`: Total amount (travelerPayment + fee + TVA)
        - `application_fee_amount`: Platform fee + TVA on fee
        - **NO `on_behalf_of`** - funds stay on platform
        - `metadata`: Store `travelerPaymentAmount` for later transfer
    - Confirm Payment Intent (may require 3D Secure)
    - **Funds are charged and held in escrow**
    - Update transaction status to `'paid'` (escrow)
    - Update request status to `'ACCEPTED'`

4. **Frontend (If 3D Secure Required):**

    - Backend returns Payment Intent with `requires_action` status
    - Frontend calls `stripe.handleCardAction()`
    - User completes 3D Secure

- Webhook confirms payment success

5. **Later - When Requester Completes Request:**
   - Requester clicks "Complete Request" button
   - Frontend calls `PATCH /api/request/{id}/complete`
   - Backend transfers funds to traveler (see Phase 8)
   - Frontend shows success message

How Stripe Elements + Payment Methods Work Together for PCI Compliance

The Complete Flow:

1. **Frontend (Stripe Elements):**

```
// Load Stripe.js
const stripe = Stripe('pk_test_...');
const elements = stripe.elements();

// Create Card Element
const cardElement = elements.create('card', {
  style: { /* custom styling */ }
});
cardElement.mount('#card-element');

// When user submits form
const { paymentMethod, error } = await stripe.createPaymentMethod({
  type: 'card',
  card: cardElement,
  billing_details: {
    name: 'John Doe',
    email: 'john@example.com'
  }
});

// Send paymentMethod.id to backend (NOT card details!)
// paymentMethod.id = "pm_1234567890"
```

2. **Backend (Payment Intent Creation):**

```
// Backend receives paymentMethodId (safe to store)
const paymentIntent = await stripe.paymentIntents.create({
  amount: 2000, // €20.00 in cents
  currency: 'eur',
  payment_method: paymentMethodId, // From frontend
  application_fee_amount: 200, // Platform fee
  on_behalf_of: travelerStripeAccountId, // Connected account
  confirm: true, // Auto-confirm
  return_url: 'https://yourapp.com/return'
});
```

3. **3D Secure Handling (if required):**

```
// Frontend checks Payment Intent status
if (paymentIntent.status === 'requires_action') {
  const { error } = await stripe.handleCardAction(
    paymentIntent.client_secret
  );
  // User completes 3D Secure challenge
}
```

Why This is PCI Compliant:

- □ **Card data stays in Stripe's secure iframe** - Never sent to your server
- □ **Only tokenized Payment Method ID** - Safe to store in database
- □ **Stripe handles all sensitive operations** - Tokenization, encryption, PCI compliance
- □ **No PCI scope for your application** - You only handle tokens, not card data
- □ **Automatic 3D Secure** - Stripe handles SCA requirements

For Instant vs Non-Instant:

- **Instant**: Create Payment Method → Create & Confirm Payment Intent immediately
- **Non-Instant**: Create Payment Method → Store `paymentMethodId` → Create & Confirm Payment Intent when traveler accepts

Benefits of This Approach

1. **Security**: Card data never touches your servers
2. **PCI Compliance**: Reduced scope, Stripe handles sensitive data
3. **Flexibility**: Payment Methods can be saved for future use
4. **Better UX**: Stripe Elements provides optimized card input with validation
5. **International**: Stripe Elements supports international card formats
6. **3D Secure**: Automatic handling of SCA/3D Secure requirements
7. **Mobile Support**: Stripe Elements works seamlessly on mobile devices
8. **Accessibility**: Built-in accessibility features for screen readers

stripeProductId and stripePriceId - NOT NEEDED

Why not needed:

- Stripe Products/Prices are for recurring subscriptions or catalog items
- GoHappyGo has dynamic pricing based on weight × pricePerKg + platform fee
- Each request has unique pricing, so we use Payment Intents directly
- No need to create products/prices in Stripe

What we use instead:

- Payment Intents with `amount` calculated dynamically
- `application_fee_amount` for platform fee
- Direct amount calculation per request

Implementation Plan

Phase 1: User Entity & Registration Updates

Files to modify:

- `backend/src/user/user.entity.ts`
- `backend/src/auth/dto/register.dto.ts`
- `backend/src/auth/auth.service.ts`

Changes:

1. Add to `UserEntity`:
    - `stripeAccountId: string | null` - Stripe Connect account ID
    - `isStripeConnected: boolean` - Whether account is fully onboarded
    - `countryCode: string` - ISO country code (e.g., "FR", "US")
2. Add to `RegisterDto`:
    - `countryCode: string` - Required field with validation (ISO 3166-1 alpha-2)
3. Update `auth.service.ts`:
    - After user creation, call `stripeService.createDeferredAccount(user)`
    - Store `stripeAccountId` in user entity
    - Set `isStripeConnected = false`

Phase 2: Stripe Module & Service

New files:

- `backend/src/stripe/stripe.module.ts`
- `backend/src/stripe/stripe.service.ts`
- `backend/src/stripe/stripe.controller.ts`
- `backend/src/stripe/dto/create-account-link.dto.ts`
- `backend/src/stripe/dto/payment-intent-response.dto.ts`
- `backend/src/stripe/entities/stripe-payment.entity.ts` (optional, for tracking)

Stripe Service Methods:

1. **`createDeferredAccount(user: UserEntity): Promise<string>`**
    - Creates Stripe Connect Custom account
    - Returns `account.id` to store in `user.stripeAccountId`
2. **`createAccountLink(accountId: string, returnUrl: string, refreshUrl: string): Promise<string>`**
    - Creates Account Link for onboarding
    - Returns URL to redirect user
3. **`createPaymentIntent(amount: number, applicationFee: number, metadata: object): Promise<PaymentIntent>`**
    - Creates Payment Intent on **platform account** (NOT on connected account)
    - Sets `application_fee_amount` for platform fee
    - **Funds are held in escrow** on platform account until request is completed
    - Returns Payment Intent for frontend confirmation
    - **Note**: No `on_behalf_of` - funds stay on platform until transfer
4. **`confirmPaymentIntent(paymentIntentId: string, paymentMethodId: string): Promise<PaymentIntent>`**
    - Confirms payment with card details
    - Handles 3D Secure if required
    - **Funds are charged but held in escrow** on platform account
5. **`transferToConnectedAccount(amount: number, connectedAccountId: string, chargeId: string, metadata: object): Promise<Transfer>`**
    - Transfers funds from platform account to traveler's connected account
    - Called when requester completes the request (confirms travel went well)
    - Transfers only the traveler's portion (total amount - platform fee)
    - Platform fee remains on platform account
    - **This is the escrow release mechanism**
6. **`handleWebhook(event: Stripe.Event): Promise<void>`**

- Processes webhook events
- Updates transaction status
- Updates account onboarding status

Phase 3: Payment Flow Integration

Files to modify:

- `backend/src/request/request.service.ts`
- `backend/src/request/dto/createRequestToTravel.dto.ts`
- `backend/src/transaction/transaction.service.ts`
- `backend/src/transaction/transaction.entity.ts`

Payment Flow for Instant Travels (`isInstant = true`) - ESCROW MODEL:

1. Requester submits request with `paymentMethodId` (from Stripe Elements)
2. Calculate amounts:
   - `travelerPayment = weight × pricePerKg`
   - `platformFee = PlatformPricingService.calculateFee(travelerPayment)`
   - `tvaOnFee = (TVA / 100) × platformFee`
   - `totalAmount = travelerPayment + platformFee + tvaOnFee`
3. Create Payment Intent on **platform account** (NOT connected account):
   - `amount`: Total amount (in cents)
   - `currency`: 'eur'
   - `payment_method`: `paymentMethodId` from frontend
   - `application_fee_amount`: Platform fee + TVA on fee (in cents)
   - `metadata`: `{ requestId, travelId, requesterId, travelerAccountId, travelerPayment }`
   - **NO `on_behalf_of`** - funds stay on platform account (escrow)
4. Confirm Payment Intent (may require 3D Secure)
5. If successful:
   - **Funds are charged and held in escrow** on platform account
   - Create `TransactionEntity` with:
     - `status = 'paid'` (charged but not transferred)
     - `stripePaymentIntentId`
     - `stripeChargeId`
     - `amount = totalAmount`
     - `travelerPaymentAmount = travelerPayment` (stored for later transfer)
   - Create `RequestEntity` with status `'ACCEPTED'`
   - Emit events
6. If failed:
   - Return error, don't create request
7. **When Requester Completes Request** (via `PATCH /api/request/{id}/complete`):
   - Calculate traveler's portion: `travelerPayment` (from transaction metadata)
   - Call `stripeService.transferToConnectedAccount()`:
     - Transfer `travelerPayment` to traveler's `stripeAccountId`
     - Use `chargeId` from Payment Intent
     - Update transaction status to `'transferred'` or `'completed'`
   - Update request status to `'COMPLETED'`
   - Emit completion events

Payment Flow for Non-Instant Travels (`isInstant = false`) - ESCROW MODEL:

1. **Request Creation (No Payment Yet):**
   - Requester submits request with `paymentMethodId` (from Stripe Elements)
   - Frontend uses Stripe Elements to collect card
   - Frontend calls `stripe.createPaymentMethod()` to tokenize card
   - Frontend sends `paymentMethodId` (not raw card details) to backend

- Backend stores `paymentMethodId` in `RequestEntity.paymentMethodId`
- Create `RequestEntity` with status `'NEGOTIATING'` and `paymentMethodId`
- **No payment processed yet**

2. **When Traveler Accepts Request:**
   - Calculate amounts:
     - `travelerPayment = weight × pricePerKg`
     - `platformFee = PlatformPricingService.calculateFee(travelerPayment)`
     - `tvaOnFee = (TVA / 100) × platformFee`
     - `totalAmount = travelerPayment + platformFee + tvaOnFee`
   - Create Payment Intent on **platform account** (NOT connected account):
     - `amount`: Total amount (in cents)
     - `currency`: 'eur'
     - `payment_method`: Stored `paymentMethodId`
     - `application_fee_amount`: Platform fee + TVA on fee (in cents)
     - `metadata`: `{ requestId, travelId, requesterId, travelerAccountId, travelerPayment }`
     - **NO `on_behalf_of`** - funds stay on platform account (escrow)
   - Confirm Payment Intent (may require 3D Secure)
   - If successful:
     - **Funds are charged and held in escrow** on platform account
     - Create `TransactionEntity` with:
       - `status = 'paid'` (charged but not transferred)
       - `stripePaymentIntentId`
       - `stripeChargeId`
       - `amount = totalAmount`
       - `travelerPaymentAmount = travelerPayment` (stored for later transfer)
     - Update request status to `'ACCEPTED'`
   - If failed:
     - Keep request in `'NEGOTIATING'`
     - Notify requester to update payment method
     - Return error

3. **When Requester Completes Request** (via `PATCH /api/request/{id}/complete`):
   - Calculate traveler's portion: `travelerPayment` (from transaction metadata)
   - Call `stripeService.transferToConnectedAccount()`:
     - Transfer `travelerPayment` to traveler's `stripeAccountId`
     - Use `chargeId` from Payment Intent
     - Update transaction status to `'transferred'` or `'completed'`
   - Update request status to `'COMPLETED'`
   - Emit completion events

Transaction Entity Updates:

- Add `stripePaymentIntentId: string | null` - Payment Intent ID for tracking
- Add `stripeChargeId: string | null` - Charge ID (needed for transfers)
- Add `stripeTransferId: string | null` - Transfer ID (when funds released to traveler)
- Add `travelerPaymentAmount: number | null` - Amount to transfer to traveler (stored in escrow)
- Add `platformFeeAmount: number | null` - Platform fee amount
- Update `status` enum to include:
  - `'paid'` - Payment charged, funds held in escrow
  - `'transferred'` - Funds transferred to traveler
  - `'completed'` - Full transaction completed
  - `'cancelled'` - Payment failed or request cancelled
- Update `paymentMethod` to `'stripe'` instead of `'platform'`

Phase 4: Webhook Handler

New file:

- `backend/src/stripe/stripe.controller.ts` (webhook endpoint)

Endpoint:

- `POST /webhooks/stripe` - Raw body required, no JSON parsing
- Verify webhook signature using `STRIPE_WEBHOOK_SECRET`
- Handle events:
  - `payment_intent.succeeded`:
    - Update transaction to `'paid'` (funds charged, held in escrow)
    - **DO NOT transfer funds yet** - wait for request completion
  - `payment_intent.payment_failed`:
    - Update transaction to `'cancelled'`
    - Notify requester
    - For non-instant: Keep request in `'NEGOTIATING'`
    - For instant: Don't create request
  - `transfer.created`:
    - Update transaction `stripeTransferId`
    - Update transaction status to `'transferred'`
  - `transfer.failed`:
    - Log error, notify admin
    - Transaction remains `'paid'` (funds still in escrow, need manual intervention)
  - `account.updated`:
    - Check if `charges_enabled && payouts_enabled`
    - Update `isStripeConnected = true`

Phase 5: Onboarding Endpoints

New endpoints in Stripe Controller:

- `GET /api/stripe/onboarding-link` - Get Account Link URL for current user
- `GET /api/stripe/account-status` - Check if user's account is fully onboarded

Flow:

1. User clicks "Complete Payment Setup" in frontend
2. Frontend calls `GET /api/stripe/onboarding-link`
3. Backend creates Account Link, returns URL
4. Frontend redirects user to Stripe
5. User completes onboarding on Stripe
6. Stripe redirects to `return_url` (frontend)
7. Frontend polls or webhook updates `isStripeConnected`

Phase 6: Environment Variables

Add to `.env`:

```
STRIPE_SECRET_KEY=sk_test_...
STRIPE_PUBLISHABLE_KEY=pk_test_...
STRIPE_WEBHOOK_SECRET=whsec_...
STRIPE_APPLICATION_FEE_PERCENTAGE=0 (we calculate fee separately)
FRONTEND_URL=http://localhost:4200 (for return_url)
```

Phase 7: Request DTO Updates - Migration to Payment Methods

Current Implementation (Needs Migration):

The current `CreateRequestToTravelDto` accepts raw card details:

- `cardNumber: string`
- `expiryDate: string`
- `cvc: string`

This is NOT PCI compliant and should be migrated to Payment Methods.

New Implementation (Recommended):

Modify `CreateRequestToTravel.dto.ts`:

```
// Remove these fields:
// cardNumber: string;
// expiryDate: string;
// cvc: string;

// Add this field:
@ApiProperty({
  description: 'Stripe Payment Method ID (created via Stripe Elements on frontend)',
  example: 'pm_1234567890',
  required: true
})
@IsNotEmpty()
@IsString()
paymentMethodId: string;
```

Migration Strategy:

1. **Phase 1 (Backend):**
    - Add `paymentMethodId` field to DTO
    - Keep old fields temporarily (mark as deprecated)
    - Update service to handle both (for backward compatibility during migration)
2. **Phase 2 (Frontend):**
    - Implement Stripe Elements
    - Create Payment Method before submitting request
    - Send `paymentMethodId` instead of raw card details
3. **Phase 3 (Cleanup):**
    - Remove deprecated card fields from DTO
    - Remove card processing logic from backend
    - Update API documentation

Benefits of Migration:

- □ **PCI Compliance**: Card data never touches your servers
- □ **Security**: Reduced attack surface, no card data to protect
- □ **Flexibility**: Payment Methods can be saved for future use
- □ **Better UX**: Stripe Elements provides optimized card input
- □ **International**: Supports international card formats automatically
- □ **3D Secure**: Automatic handling of SCA requirements

For Both Instant and Non-Instant:

- **Instant**: Frontend creates Payment Method → Backend creates & confirms Payment Intent immediately
- **Non-Instant**: Frontend creates Payment Method → Backend stores `paymentMethodId` → Backend creates & confirms Payment Intent when traveler accepts

Phase 8: Complete Request Integration - Escrow Release

Files to modify:

- `backend/src/request/request.service.ts` (update `completeRequest` method)
- `backend/src/stripe/stripe.service.ts` (add `transferToConnectedAccount` method)
- `backend/src/transaction/transaction.service.ts` (update transaction status)

Escrow Release Flow:

1. Requester calls `PATCH /api/request/{id}/complete`
2. Backend validates:
    - Request exists and is in `'ACCEPTED'` status
    - User is the requester (authorized)
    - Transaction exists and is in `'paid'` status (funds in escrow)
3. Calculate transfer amount:
    - Get `travelerPaymentAmount` from transaction
    - This is the amount to transfer to traveler (excluding platform fee)
4. Get traveler's `stripeAccountId` from request (travel.user.stripeAccountId)
5. Call `stripeService.transferToConnectedAccount()`:

```
const transfer = await stripe.transfers.create({
  amount: travelerPaymentAmount, // in cents
  currency: 'eur',
  destination: travelerStripeAccountId,
  source_transaction: chargeId, // From Payment Intent charge
  metadata: {
    requestId: request.id,
    transactionId: transaction.id
  }
});
```

6. Update transaction:
    - `stripeTransferId = transfer.id`
    - `status = 'transferred'`
7. Update request status to `'COMPLETED'`
8. Emit completion events

Important Notes:

- **Platform fee stays on platform account** (not transferred)
- Transfer happens **only when requester confirms** travel went well
- If transfer fails, transaction remains `'paid'` (funds still in escrow)
- Need manual intervention for failed transfers

Phase 9: Error Handling & Edge Cases

Scenarios to handle:

1. **User tries to create request but traveler's account not onboarded**
    - Return error: "Traveler must complete payment setup first"
    - Check `travel.user.isStripeConnected === true` before allowing request
2. **Payment fails during request creation (instant)**
    - Don't create request, return error
    - Frontend shows error, allows retry
3. **Payment fails during acceptance (non-instant)**
    - Keep request in `'NEGOTIATING'`, notify requester
    - Requester can update payment method and retry

4. **Webhook arrives before database transaction commits**
    - Implement idempotency (check if already processed)
    - Use `paymentIntentId` as idempotency key
5. **Transfer fails when completing request**
    - Transaction remains `'paid'` (funds still in escrow)
    - Log error, notify admin
    - Admin can manually retry transfer or refund
    - Consider implementing retry mechanism with exponential backoff
6. **Requester never completes request**
    - Funds remain in escrow indefinitely
    - Consider implementing:
        - Auto-complete after X days (if no disputes)
        - Dispute resolution mechanism
        - Refund policy for uncompleted requests
7. **Request is cancelled after payment**
    - If in `'NEGOTIATING'`: Refund payment (if already charged)
    - If in `'ACCEPTED'`: Refund payment, cancel transfer
    - Update transaction status to `'cancelled'`
8. **Traveler's account is closed/deactivated**
    - Check account status before transfer
    - If account closed: Hold funds, notify admin
    - Admin can manually handle (refund or alternative transfer)

Stripe Platform Settings

1. Enable Connect

- Dashboard → Settings → Connect
- Enable "Custom accounts"
- Set up branding (optional)

2. Configure Onboarding

- Dashboard → Connect → Settings → Onboarding
- Customize fields required
- Set up business profile requirements

3. Webhook Configuration

- Dashboard → Developers → Webhooks
- Add endpoint: `https://yourdomain.com/webhooks/stripe`
- Select events:
    - `payment_intent.succeeded`
    - `payment_intent.payment_failed`
    - `checkout.session.completed`
    - `checkout.session.async_payment_failed`
    - `account.updated`
    - `charge.refunded`

4. PSD2 Compliance (France)

- Stripe automatically handles SCA (Strong Customer Authentication)
- Use Payment Intents (not Charges directly)
- Handle `requires_action` state for 3D Secure
- Stripe-hosted onboarding satisfies PSD2 requirements

Testing Strategy

Using Stripe CLI for Local Development

Stripe CLI allows you to test webhooks locally without deploying your application. It acts as a bridge between Stripe's servers and your local development environment, forwarding webhook events in real-time.

1. Install Stripe CLI

```
# Windows (using Scoop)
scoop install stripe

# macOS
brew install stripe/stripe-cli/stripe

# Linux
# Download from https://github.com/stripe/stripe-cli/releases

# Verify installation
stripe --version
```

2. Login to Stripe CLI

```
stripe login
```

This opens your browser to authenticate with your Stripe account. You'll be redirected to Stripe's website to authorize the CLI.

Alternative (API Key):

```
stripe login --api-key sk_test_...
```

3. Forward Webhooks to Local Server

Basic Forwarding:

```
# Forward all webhook events to your local backend
stripe listen --forward-to http://localhost:3000/webhooks/stripe
```

Output:

```
> Ready! Your webhook signing secret is whsec_xxxxx (^C to quit)
```

Important: Copy the webhook signing secret (`whsec_xxxxx`) and add it to your `.env`:

```
STRIPE_WEBHOOK_SECRET=whsec_xxxxx
```

Filter Specific Events:

```
# Only forward specific events (more efficient)
stripe listen --forward-to http://localhost:3000/webhooks/stripe \
  --events payment_intent.succeeded,payment_intent.payment_failed,account.updated
```

With Logging:

```
# Show detailed logs of forwarded events
stripe listen --forward-to http://localhost:3000/webhooks/stripe --print-json
```

4. Trigger Test Events

You can trigger test events from the CLI to simulate real Stripe events:

Basic Triggers:

```
# Trigger payment_intent.succeeded
stripe trigger payment_intent.succeeded

# Trigger payment_intent.payment_failed
stripe trigger payment_intent.payment_failed

# Trigger account.updated
stripe trigger account.updated
```

Trigger with Custom Data:

```
# Override metadata to match your application
stripe trigger payment_intent.succeeded \
  --override payment_intent:metadata.requestId=123 \
  --override payment_intent:metadata.travelId=456 \
  --override payment_intent:metadata.requesterId=789

# Override amount
stripe trigger payment_intent.succeeded \
  --override payment_intent:amount=2000 \
  --override payment_intent:currency=eur

# Override connected account
stripe trigger payment_intent.succeeded \
  --override payment_intent:on_behalf_of=acct_xxxxx
```

List Available Triggers:

```
# See all available event triggers
stripe trigger --help

# List all triggerable events
stripe trigger --list
```

5. Test Payment Flows Locally

Complete Testing Workflow:

1. **Start Your Backend:**

```
cd backend
npm run start:dev
```

2. **Start Stripe CLI in Separate Terminal:**

```
stripe listen --forward-to http://localhost:3000/webhooks/stripe
```

3. **Test Instant Payment Flow:**

   - Open frontend application
   - Use test card `4242 4242 4242 4242` (any CVC, future expiry)
   - Create request with instant travel
   - Watch Stripe CLI terminal for `payment_intent.succeeded` event
   - Check backend logs for webhook processing
   - Verify database: transaction status = `'paid'`

4. **Test Non-Instant Payment Flow:**

   - Create request with non-instant travel (payment method stored)
   - Accept request as traveler via API or frontend
   - Watch Stripe CLI for `payment_intent.succeeded` event
   - Verify transaction status updated to `'paid'`
   - Verify request status updated to `'ACCEPTED'`

5. **Test 3D Secure Flow:**

   - Use test card `4000 0025 0000 3155` (requires 3D Secure)
   - Create payment intent (instant or on acceptance)
   - Frontend should show 3D Secure challenge modal
   - Complete challenge (use any password in test mode)
   - Watch Stripe CLI for `payment_intent.succeeded` event
   - Verify payment completed

6. **Test Payment Failure:**

   - Use decline card `4000 0000 0000 0002`
   - Attempt payment
   - Watch Stripe CLI for `payment_intent.payment_failed` event
   - Verify error handling in frontend
   - Verify request not created (for instant) or status unchanged (for non-instant)

6. View and Inspect Webhook Events

List Recent Events:

```
# List last 10 events
stripe events list

# List with more details
stripe events list --limit 20

# Filter by event type
stripe events list --type payment_intent.succeeded
```

Retrieve Specific Event:

```
# Get event details
stripe events retrieve evt_xxxxx

# Get full JSON payload
stripe events retrieve evt_xxxxx --format json

# Pretty print JSON
stripe events retrieve evt_xxxxx --format json | jq
```

Resend Event:

```
# Resend a webhook event (useful for testing)
stripe events resend evt_xxxxx
```

7. Test Account Onboarding

Simulate Onboarding Completion:

```
# Create a test connected account first
stripe accounts create --type=custom --country FR

# Then trigger account.updated with onboarding complete
stripe trigger account.updated \
  --override account:id=acct_xxxxx \
  --override account:charges_enabled=true \
  --override account:payouts_enabled=true \
  --override account:details_submitted=true
```

Test Account Link Creation:

```
# Create Account Link via API (test in backend)
stripe account_links create \
  --account acct_xxxxx \
  --refresh_url "http://localhost:4200/settings/payments?refresh=true" \
  --return_url "http://localhost:4200/settings/payments?success=true" \
  --type account_onboarding
```

8. Test Payment Intents Directly

Create Test Payment Intent:

```
# Create a payment intent
stripe payment_intents create \
  --amount=2000 \
  --currency=eur \
  --payment_method=pm_card_visa \
  --on_behalf_of=acct_xxxxx \
  --application_fee_amount=200 \
  --confirm

# Retrieve payment intent
stripe payment_intents retrieve pi_xxxxx

# Confirm payment intent
stripe payment_intents confirm pi_xxxxx
```

9. Monitor Webhook Delivery

View Webhook Logs:

```
# Real-time webhook logs
stripe logs tail

# Filter by endpoint
stripe logs tail --forward-connect-to http://localhost:3000/webhooks/stripe

# View failed deliveries
stripe events list --failures
```

Test Webhook Endpoint:

```
# Test if your endpoint is reachable
curl -X POST http://localhost:3000/webhooks/stripe \
  -H "Content-Type: application/json" \
  -d '{"test": true}'
```

10. Common CLI Commands Reference

```
# Help for any command
stripe --help
stripe listen --help
stripe trigger --help

# Switch between test and live mode
stripe config --set test_mode true
stripe config --set test_mode false

# View current configuration
stripe config --list

# Test API connectivity
stripe balance retrieve

# Create test data
stripe customers create --email test@example.com
stripe payment_methods create --type card --card[number]=4242424242424242 --card[exp_month]=12 --card[exp_year]=2025

# View test data
stripe customers list
stripe payment_intents list
stripe accounts list
```

11. Debugging Tips

Enable Verbose Logging:

```
# See detailed request/response logs
stripe listen --forward-to http://localhost:3000/webhooks/stripe --print-json --verbose
```

Test Webhook Signature Verification:

```
# The webhook secret from `stripe listen` should match your backend's STRIPE_WEBHOOK_SECRET
# If verification fails, check:
# 1. Webhook secret matches in .env
# 2. Raw body is used (not parsed JSON) for signature verification
# 3. Correct header name: 'stripe-signature'
```

Common Issues:

1. **Webhook not received**: Check firewall, ensure backend is running, verify URL is correct
2. **Signature verification fails**: Ensure using raw body, correct webhook secret
3. **Event not triggering**: Check event type matches what you're listening for
4. **Test data not found**: Ensure using test mode (`sk_test_...`), not live mode

12. Integration Testing Workflow

Complete End-to-End Test:

```
# Terminal 1: Start backend
cd backend && npm run start:dev

# Terminal 2: Start Stripe CLI
stripe listen --forward-to http://localhost:3000/webhooks/stripe

# Terminal 3: Run frontend
cd frontend && npm start

# Terminal 4: Monitor logs
# Watch backend logs for webhook processing
# Watch Stripe CLI for forwarded events
```

Test Scenario Checklist:

- [ ] Instant travel payment succeeds
- [ ] Non-instant travel payment on acceptance
- [ ] 3D Secure challenge handled correctly
- [ ] Payment failure handled gracefully
- [ ] Webhook events received and processed
- [ ] Database updated correctly
- [ ] Frontend shows correct status
- [ ] Account onboarding flow works
- [ ] Error messages are user-friendly

Test Cards (Stripe Test Mode)

Success Cards:

- `4242 4242 4242 4242` - Visa, succeeds immediately
- `5555 5555 5555 4444` - Mastercard, succeeds immediately
- `5200 8282 8282 8210` - Mastercard, succeeds immediately

3D Secure Cards:

- `4000 0025 0000 3155` - Requires 3D Secure authentication
- `4000 0027 6000 3184` - Requires 3D Secure, authentication fails

Decline Cards:

- `4000 0000 0000 0002` - Card declined (generic decline)
- `4000 0000 0000 9995` - Insufficient funds

- `4000 0000 0000 0069` - Expired card

Special Test Cards:

- `4000 0000 0000 3220` - 3D Secure required, but authentication unavailable
- `4000 0000 0000 3055` - 3D Secure required, authentication succeeds

Test Card Details:

- **CVC**: Any 3 digits (e.g., `123`)
- **Expiry**: Any future date (e.g., `12/25`)
- **ZIP**: Any 5 digits (e.g., `12345`)

Test Scenarios

1. **Instant Travel Payment Success:**
    - Create request with instant travel
    - Use test card `4242 4242 4242 4242`
    - Verify payment succeeds
    - Check webhook received
    - Verify transaction status = `'paid'`
2. **Non-Instant Travel Payment on Acceptance:**
    - Create request with non-instant travel
    - Store payment method
    - Accept request as traveler
    - Verify payment processed
    - Check webhook received
3. **Payment Failure Handling:**
    - Use decline card `4000 0000 0000 0002`
    - Attempt payment
    - Verify error returned
    - Check webhook `payment_intent.payment_failed` received
4. **3D Secure Flow:**
    - Use 3D Secure card `4000 0025 0000 3155`
    - Create payment intent
    - Verify `requires_action` status
    - Complete 3D Secure challenge
    - Verify payment succeeds
5. **Webhook Processing:**
    - Trigger webhook via CLI
    - Verify webhook signature validation
    - Check database updated correctly
    - Verify idempotency (duplicate webhooks handled)
6. **Account Onboarding Flow:**
    - Create deferred account
    - Generate Account Link
    - Complete onboarding (or trigger via CLI)
    - Verify `isStripeConnected = true`
7. **Refund Processing:**
    - Create successful payment
    - Trigger refund via Stripe Dashboard or API

- Verify `charge.refunded` webhook
- Check transaction status updated

Security Considerations

1. **Never store raw card details** - Use Stripe Payment Methods API (store only `paymentMethodId`)
2. **Use Stripe Elements on frontend** - PCI compliant card input (card data never touches your server)
3. **Verify webhook signatures** - Prevent fake webhooks
4. **Idempotency keys** - Prevent duplicate payments
5. **Validate amounts** - Recalculate on backend, don't trust frontend
6. **Handle 3D Secure** - Required for PSD2 compliance (Payment Intent `requires_action` state)
7. **Tokenization** - Stripe Payment Methods are already tokenized (secure by design)

Database Migrations

1. Add columns to `UserEntity`:
    - `stripeAccountId VARCHAR(255) NULL`
    - `isStripeConnected BOOLEAN DEFAULT FALSE`
    - `countryCode VARCHAR(2) NOT NULL`
2. Add columns to `TransactionEntity`:
    - `stripePaymentIntentId VARCHAR(255) NULL` - Payment Intent ID
    - `stripeChargeId VARCHAR(255) NULL` - Charge ID (needed for transfers)
    - `stripeTransferId VARCHAR(255) NULL` - Transfer ID (when funds released)
    - `travelerPaymentAmount DECIMAL(10,2) NULL` - Amount to transfer to traveler
    - `platformFeeAmount DECIMAL(10,2) NULL` - Platform fee amount
    - Update `status` enum to include: `'paid'`, `'transferred'`, `'completed'`, `'cancelled'`

Frontend Integration Requirements

1. Install Stripe.js

Angular/TypeScript:

```
npm install @stripe/stripe-js
```

React:

```
npm install @stripe/stripe-js @stripe/react-stripe-js
```

Vanilla JavaScript:

```
<script src="https://js.stripe.com/v3/"></script>
```

2. Initialize Stripe

Angular Service Example:

```
// stripe.service.ts
import { Injectable } from '@angular/core';
import { loadStripe, Stripe } from '@stripe/stripe-js';

@Injectable({
  providedIn: 'root'
})
export class StripeService {
  private stripePromise: Promise<Stripe | null>;

  constructor() {
    // Get publishable key from your backend config or environment
    this.stripePromise = loadStripe('pk_test_...');
  }

  async getStripe(): Promise<Stripe | null> {
    return this.stripePromise;
  }
}
```

3. Create Stripe Elements Component

Angular Component Example:

```
// payment-form.component.ts
import { Component, OnInit, ViewChild, ElementRef } from '@angular/core';
import { StripeService } from './stripe.service';
import { Stripe, StripeElements, StripeCardElement } from '@stripe/stripe-js';

@Component({
  selector: 'app-payment-form',
  template: `
    <form (ngSubmit)="onSubmit()" #paymentForm="ngForm">
      <div id="card-element"><!-- Stripe Elements will mount here --></div>
      <div id="card-errors" role="alert"></div>
      <button type="submit" [disabled]="loading">
        {{ loading ? 'Processing...' : 'Submit Payment' }}
      </button>
    </form>
  `
})
export class PaymentFormComponent implements OnInit {
  @ViewChild('paymentForm') paymentForm: any;
  stripe: Stripe | null = null;
  elements: StripeElements | null = null;
  cardElement: StripeCardElement | null = null;
  loading = false;
  paymentMethodId: string | null = null;

  constructor(private stripeService: StripeService) {}

  async ngOnInit() {
    this.stripe = await this.stripeService.getStripe();
    if (!this.stripe) {
      console.error('Stripe failed to load');
      return;
    }

    // Create Elements instance
    this.elements = this.stripe.elements({
      appearance: {
        theme: 'stripe',
        variables: {
          colorPrimary: '#0570de',
        }
      }
    });

    // Create and mount Card Element
    this.cardElement = this.elements.create('card', {
      style: {
        base: {
          fontSize: '16px',
          color: '#424770',
          '::placeholder': {
            color: '#aab7c4',
          },
        },
        invalid: {
          color: '#9e2146',
        },
```

```
      },
    });

    this.cardElement.mount('#card-element');

    // Listen for validation errors
    this.cardElement.on('change', ({error}) => {
      const displayError = document.getElementById('card-errors');
      if (error) {
        displayError!.textContent = error.message;
      } else {
        displayError!.textContent = '';
      }
    });
  }

  async onSubmit() {
    if (!this.stripe || !this.cardElement) {
      return;
    }

    this.loading = true;

    // Create Payment Method from card element
    const { paymentMethod, error } = await this.stripe.createPaymentMethod({
      type: 'card',
      card: this.cardElement,
      billing_details: {
        name: 'User Name', // Get from form
        email: 'user@example.com', // Get from form
      },
    });

    if (error) {
      console.error('Error creating payment method:', error);
      this.loading = false;
      return;
    }

    this.paymentMethodId = paymentMethod.id;

    // Send paymentMethodId to your backend
    // For instant travels: Create request + process payment
    // For non-instant: Store paymentMethodId with request
  }
}
```

4. Handle Instant Travel Payment Flow

Complete Flow for Instant Travels:

```
// request.service.ts (Frontend)
async createRequestToTravel(requestData: CreateRequestDto, paymentMethodId: string) {
  try {
    // Step 1: Create Payment Method (already done in component)

    // Step 2: Send request with paymentMethodId to backend
    const response = await this.http.post('/api/request/to-travel', {
      ...requestData,
      paymentMethodId: paymentMethodId // Instead of cardNumber, expiryDate, cvc
    }).toPromise();

    // Step 3: Backend returns Payment Intent
    const paymentIntent = response.paymentIntent;

    // Step 4: Handle 3D Secure if required
    if (paymentIntent.status === 'requires_action') {
      const stripe = await this.stripeService.getStripe();
      const { error } = await stripe!.handleCardAction(
        paymentIntent.client_secret
      );

      if (error) {
        throw new Error(error.message);
      }

      // Payment Intent is now confirmed
      // Backend webhook will update transaction status
    }

    // Step 5: Show success
    return response;
  } catch (error) {
    console.error('Payment failed:', error);
    throw error;
  }
}
```

5. Handle Non-Instant Travel Payment Flow

Request Creation (Store Payment Method):

```
// request.service.ts (Frontend)
async createRequestToTravel(requestData: CreateRequestDto, paymentMethodId: string) {
  // For non-instant: Just store paymentMethodId, no payment yet
  return await this.http.post('/api/request/to-travel', {
    ...requestData,
    paymentMethodId: paymentMethodId,
    isInstant: false
  }).toPromise();
}
```

Traveler Accepts (Process Payment):

```
// request.service.ts (Frontend)
async acceptRequest(requestId: number) {
  try {
    // Backend processes payment with stored paymentMethodId
    const response = await this.http.post(`/api/request/${requestId}/accept`, {}).toPromise();

    const paymentIntent = response.paymentIntent;

    // Handle 3D Secure if required
    if (paymentIntent && paymentIntent.status === 'requires_action') {
      const stripe = await this.stripeService.getStripe();
      const { error } = await stripe!.handleCardAction(
        paymentIntent.client_secret
      );

      if (error) {
        throw new Error(error.message);
      }
    }

    return response;
  } catch (error) {
    console.error('Payment failed:', error);
    throw error;
  }
}
```

6. Display Saved Payment Methods (Optional)

Show Last 4 Digits:

```
// After creating Payment Method, store metadata
const paymentMethod = await stripe.createPaymentMethod({...});

// Display to user
const cardInfo = {
  last4: paymentMethod.card.last4,
  brand: paymentMethod.card.brand,
  expMonth: paymentMethod.card.exp_month,
  expYear: paymentMethod.card.exp_year
};

// Store in component state or send to backend
```

7. Handle Account Onboarding

Redirect to Stripe Onboarding:

```
// settings.component.ts
async completePaymentSetup() {
  try {
    // Get Account Link from backend
    const response = await this.http.get('/api/stripe/onboarding-link').toPromise();
    const accountLinkUrl = response.url;

    // Redirect user to Stripe
    window.location.href = accountLinkUrl;
  } catch (error) {
    console.error('Failed to get onboarding link:', error);
  }
}

// After redirect back from Stripe
ngOnInit() {
  const urlParams = new URLSearchParams(window.location.search);
  if (urlParams.get('success') === 'true') {
    // Onboarding completed
    this.checkAccountStatus();
  } else if (urlParams.get('refresh') === 'true') {
    // User skipped or session expired
    this.showOnboardingPrompt();
  }
}

async checkAccountStatus() {
  const response = await this.http.get('/api/stripe/account-status').toPromise();
  if (response.isStripeConnected) {
    // Show success message
  }
}
```

8. Error Handling

Common Errors to Handle:

```
// payment-form.component.ts
handleStripeError(error: StripeError) {
  switch (error.type) {
    case 'card_error':
      // Card was declined
      this.showError(error.message);
      break;
    case 'validation_error':
      // Invalid input
      this.showError(error.message);
      break;
    case 'api_error':
      // Stripe API error
      this.showError('Payment service temporarily unavailable. Please try again.');
      break;
    default:
      this.showError('An unexpected error occurred.');
  }
}
```

9. Testing in Frontend

Use Test Cards:

```
// In development, show test card info
const TEST_CARDS = {
  success: '4242 4242 4242 4242',
  threeDSecure: '4000 0025 0000 3155',
  decline: '4000 0000 0000 0002'
};

// Display helper text in development mode
if (environment.production === false) {
  // Show test card numbers to developers
}
```

10. Required Frontend Changes Summary

DTO Updates:

- Remove `cardNumber`, `expiryDate`, `cvc` from `CreateRequestToTravelDto`
- Add `paymentMethodId: string` to `CreateRequestToTravelDto`
- Add optional `paymentMethodId` to `RequestEntity` (for non-instant)

New Components Needed:

1. **PaymentFormComponent**: Stripe Elements card input
2. **PaymentStatusComponent**: Display payment status, handle 3D Secure
3. **OnboardingComponent**: Redirect to Stripe onboarding, handle return

New Services Needed:

1. **StripeService**: Initialize Stripe.js, create Payment Methods
2. **PaymentService**: Handle payment flows, 3D Secure

API Integration:

1. Update `POST /api/request/to-travel` to accept `paymentMethodId`
2. Add `GET /api/stripe/onboarding-link` endpoint call
3. Add `GET /api/stripe/account-status` endpoint call
4. Handle Payment Intent responses from backend
5. Implement 3D Secure handling with `stripe.handleCardAction()`

User Experience:

1. Show loading states during payment processing
2. Display clear error messages for payment failures
3. Guide users through 3D Secure challenges
4. Show payment method details (last 4 digits, brand) after creation
5. Prompt users to complete onboarding before receiving payments

Summary

Key Points:

- □ Deferred account creation during registration (minimal info)
- □ Full onboarding via Account Links when user wants to receive payments
- □ **ESCROW MODEL**: Payment charged immediately, funds held until requester completes request
- □ Payment Intents created on **platform account** (not connected account)
- □ Funds transferred to traveler **only when requester confirms** travel went well

- □ Platform fee stays on platform account (not transferred)
- □ Webhook handling for payment status updates and transfer confirmations
- □ PSD2 compliant with Stripe-hosted onboarding
- □ No need for `stripeProductId` or `stripePriceId` (dynamic pricing)
- □ Platform fee calculated using existing `PlatformPricingService`
- □ Users can be both buyers and sellers (each has their own Connect account)
- □ **PCI Compliant**: Stripe Elements + Payment Methods (card data never touches servers)
- □ **Stripe CLI**: Test webhooks locally with `stripe listen --forward-to`
- □ **Frontend Integration**: Stripe.js, Elements, Payment Methods, 3D Secure handling
- □ **Complete Request Integration**: Escrow release mechanism when requester confirms completion

Next Steps:

1. Implement Phase 1 (User entity updates)
2. Implement Phase 2 (Stripe service with escrow support)
3. Integrate payment flow (Phase 3) - **ESCROW MODEL**: Charge to platform, hold funds
4. Add webhook handler (Phase 4) - Handle transfers and payment status
5. Add onboarding endpoints (Phase 5)
6. **Implement Phase 8**: Complete request integration - Escrow release mechanism
7. **Frontend**: Install Stripe.js, create PaymentFormComponent
8. **Frontend**: Update DTOs to use `paymentMethodId` instead of raw card details
9. **Frontend**: Implement 3D Secure handling
10. **Frontend**: Add onboarding flow
11. **Frontend**: Add "Complete Request" button with payment transfer confirmation
12. Test thoroughly with Stripe CLI (payment, escrow, transfer flows)