Sistema de Wallets Virtuais - App de Entregas

Visão Geral

Sistema de gestão financeira para aplicativo de entregas estilo Uber, com wallets virtuais internas e integração com Asaas para cobrança via PIX e Boleto.

Modelo de Negócio

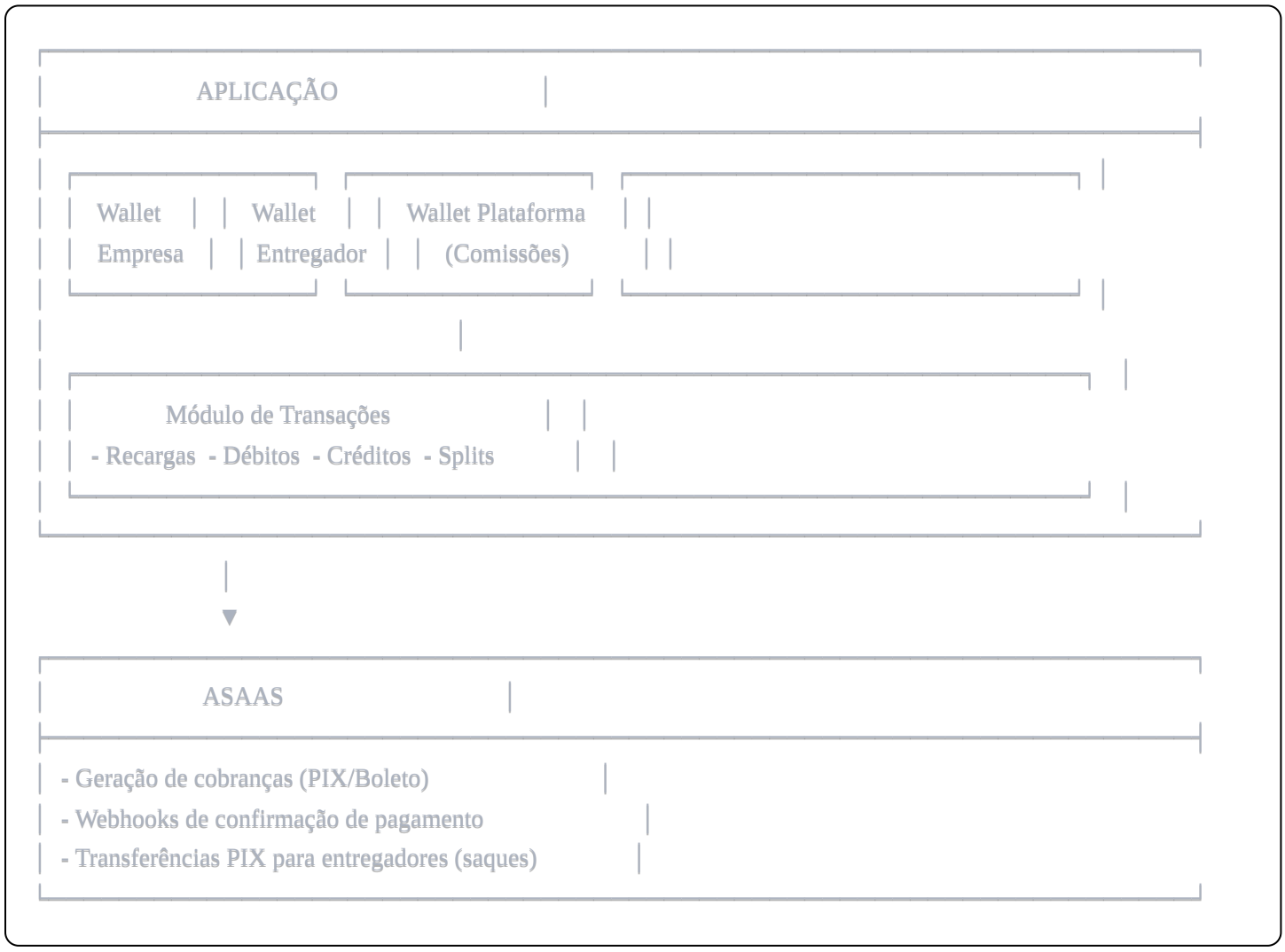
- **Empresas**: Clientes que solicitam entregas (restaurantes, lojas, etc.)
- **Entregadores**: Prestadores de serviço que realizam as entregas
- **Plataforma**: Cobra comissão sobre cada entrega

Modalidades de Pagamento

1. **Pré-pago**: Empresa recarrega saldo antes de solicitar entregas
2. **Pós-pago**: Empresa acumula entregas e recebe cobrança semanal

---

Arquitetura do Sistema



Modelagem do Banco de Dados (PostgreSQL)

Tabela: wallets

Armazena as carteiras virtuais de cada usuário do sistema.

sql

```sql
CREATE TYPE wallet_type AS ENUM ('empresa', 'entregador', 'plataforma');
CREATE TYPE wallet_status AS ENUM ('ativo', 'bloqueado', 'suspenso');

CREATE TABLE wallets (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Relacionamento com usuário (pode ser empresa ou entregador)
    usuario_id UUID NOT NULL,
    tipo wallet_type NOT NULL,

    -- Saldos
    saldo_disponivel DECIMAL(15,2) NOT NULL DEFAULT 0.00,
    saldo_bloqueado DECIMAL(15,2) NOT NULL DEFAULT 0.00, -- Saldo em processamento

    -- Controle
    status wallet_status NOT NULL DEFAULT 'ativo',

    -- Auditoria
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    -- Constraints
    CONSTRAINT saldo_disponivel_positivo CHECK (saldo_disponivel >= 0),
    CONSTRAINT saldo_bloqueado_positivo CHECK (saldo_bloqueado >= 0),
    CONSTRAINT unique_usuario_tipo UNIQUE (usuario_id, tipo)
);

CREATE INDEX idx_wallets_usuario ON wallets(usuario_id);
CREATE INDEX idx_wallets_tipo ON wallets(tipo);
CREATE INDEX idx_wallets_status ON wallets(status);

-- Trigger para updated_at
CREATE OR REPLACE FUNCTION update_updated_at_column()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ language 'plpgsql';

CREATE TRIGGER update_wallets_updated_at
    BEFORE UPDATE ON wallets
    FOR EACH ROW
    EXECUTE FUNCTION update_updated_at_column();
```

Tabela: wallet_transactions

Registra todas as movimentações financeiras do sistema.

```sql
```

```sql
CREATE TYPE transaction_type AS ENUM (
    'recarga',          -- Empresa adicionou saldo (PIX/Boleto)
    'debito_entrega',   -- Empresa pagou por entrega
    'credito_entrega',  -- Entregador recebeu por entrega
    'comissao',         -- Plataforma recebeu comissão
    'saque',            -- Entregador sacou dinheiro
    'estorno',          -- Estorno de transação
    'ajuste_manual',    -- Ajuste feito pelo admin
    'cobranca_pospago'  -- Cobrança semanal pós-pago
);

CREATE TYPE transaction_status AS ENUM (
    'pendente',     -- Aguardando processamento
    'processando',  -- Em processamento
    'concluida',    -- Finalizada com sucesso
    'falhou',       -- Falhou
    'cancelada',    -- Cancelada
    'estornada'     -- Foi estornada
);

CREATE TABLE wallet_transactions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Wallet relacionada
    wallet_id UUID NOT NULL REFERENCES wallets(id),

    -- Tipo e status
    tipo transaction_type NOT NULL,
    status transaction_status NOT NULL DEFAULT 'pendente',

    -- Valores
    valor DECIMAL(15,2) NOT NULL,
    saldo_anterior DECIMAL(15,2) NOT NULL,
    saldo_posterior DECIMAL(15,2) NOT NULL,

    -- Referências
    entrega_id UUID, -- Se relacionado a uma entrega
    cobranca_id UUID, -- Se relacionado a uma cobrança (Asaas)
    transacao_origem_id UUID REFERENCES wallet_transactions(id), -- Para estornos

    -- Detalhes
    descricao TEXT,
    metadata JSONB DEFAULT '{}',

    -- Auditoria
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
```

```
    processed_at TIMESTAMP WITH TIME ZONE,

    -- Constraints
    CONSTRAINT valor_positivo CHECK (valor > 0)
);

CREATE INDEX idx_transactions_wallet ON wallet_transactions(wallet_id);
CREATE INDEX idx_transactions_tipo ON wallet_transactions(tipo);
CREATE INDEX idx_transactions_status ON wallet_transactions(status);
CREATE INDEX idx_transactions_entrega ON wallet_transactions(entrega_id);
CREATE INDEX idx_transactions_cobranca ON wallet_transactions(cobranca_id);
CREATE INDEX idx_transactions_created ON wallet_transactions(created_at);
```

Tabela: cobrancas

Gerencia as cobranças geradas no Asaas.

sql

```sql
CREATE TYPE cobranca_tipo AS ENUM ('recarga', 'pospago');
CREATE TYPE cobranca_forma AS ENUM ('pix', 'boleto');
CREATE TYPE cobranca_status AS ENUM (
    'pendente',
    'aguardando_pagamento',
    'confirmada',
    'vencida',
    'cancelada',
    'estornada'
);

CREATE TABLE cobrancas (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Relacionamentos
    empresa_id UUID NOT NULL,
    wallet_id UUID NOT NULL REFERENCES wallets(id),

    -- Dados Asaas
    asaas_id VARCHAR(100), -- ID da cobrança no Asaas
    asaas_customer_id VARCHAR(100), -- ID do cliente no Asaas

    -- Tipo e forma
    tipo cobranca_tipo NOT NULL,
    forma_pagamento cobranca_forma NOT NULL,

    -- Valores
    valor DECIMAL(15,2) NOT NULL,
    valor_liquido DECIMAL(15,2), -- Valor após taxas Asaas

    -- Datas
    data_vencimento DATE NOT NULL,
    data_pagamento TIMESTAMP WITH TIME ZONE,

    -- Status
    status cobranca_status NOT NULL DEFAULT 'pendente',

    -- Dados do pagamento
    pix_copia_cola TEXT,
    pix_qrcode_url TEXT,
    boleto_url TEXT,
    boleto_codigo_barras VARCHAR(100),
    linha_digitavel VARCHAR(100),

    -- Período (para pós-pago)
    periodo_inicio DATE,
```

```
    periodo_fim DATE,

    -- Metadata
    metadata JSONB DEFAULT '{}',

    -- Auditoria
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    CONSTRAINT valor_positivo CHECK (valor > 0)
);

CREATE INDEX idx_cobrancas_empresa ON cobrancas(empresa_id);
CREATE INDEX idx_cobrancas_wallet ON cobrancas(wallet_id);
CREATE INDEX idx_cobrancas_asaas ON cobrancas(asaas_id);
CREATE INDEX idx_cobrancas_status ON cobrancas(status);
CREATE INDEX idx_cobrancas_vencimento ON cobrancas(data_vencimento);
CREATE INDEX idx_cobrancas_tipo ON cobrancas(tipo);

CREATE TRIGGER update_cobrancas_updated_at
    BEFORE UPDATE ON cobrancas
    FOR EACH ROW
    EXECUTE FUNCTION update_updated_at_column();
```

Tabela: entregas_financeiro

Vincula entregas com as movimentações financeiras (split).

sql

```sql
CREATE TABLE entregas_financeiro (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Entrega
    entrega_id UUID NOT NULL,

    -- Participantes
    empresa_id UUID NOT NULL,
    entregador_id UUID NOT NULL,

    -- Valores
    valor_total DECIMAL(15,2) NOT NULL,        -- Valor cobrado da empresa
    valor_entregador DECIMAL(15,2) NOT NULL,   -- Parte do entregador
    valor_comissao DECIMAL(15,2) NOT NULL,     -- Parte da plataforma
    percentual_comissao DECIMAL(5,2) NOT NULL, -- % da comissão aplicada

    -- Transações relacionadas
    transacao_debito_empresa_id UUID REFERENCES wallet_transactions(id),
    transacao_credito_entregador_id UUID REFERENCES wallet_transactions(id),
    transacao_comissao_id UUID REFERENCES wallet_transactions(id),

    -- Cobrança (se pós-pago)
    cobranca_id UUID REFERENCES cobrancas(id),

    -- Status
    processado BOOLEAN DEFAULT FALSE,
    processado_at TIMESTAMP WITH TIME ZONE,

    -- Auditoria
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    CONSTRAINT valores_positivos CHECK (
        valor_total > 0 AND
        valor_entregador >= 0 AND
        valor_comissao >= 0
    ),
    CONSTRAINT soma_valores CHECK (
        valor_entregador + valor_comissao = valor_total
    )
);

CREATE INDEX idx_entregas_fin_entrega ON entregas_financeiro(entrega_id);
CREATE INDEX idx_entregas_fin_empresa ON entregas_financeiro(empresa_id);
CREATE INDEX idx_entregas_fin_entregador ON entregas_financeiro(entregador_id);
```

```sql
CREATE INDEX idx_entregas_fin_cobranca ON entregas_financeiro(cobranca_id);
CREATE INDEX idx_entregas_fin_processado ON entregas_financeiro(processado);
```

Tabela: saques

Controla os saques dos entregadores.

sql

```sql
CREATE TYPE saque_status AS ENUM (
    'solicitado',
    'processando',
    'concluido',
    'falhou',
    'cancelado'
);

CREATE TABLE saques (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Entregador
    entregador_id UUID NOT NULL,
    wallet_id UUID NOT NULL REFERENCES wallets(id),

    -- Valores
    valor DECIMAL(15,2) NOT NULL,
    taxa DECIMAL(15,2) DEFAULT 0.00, -- Taxa de saque se houver
    valor_liquido DECIMAL(15,2) NOT NULL, -- Valor - taxa

    -- Dados bancários
    tipo_chave_pix VARCHAR(20) NOT NULL, -- cpf, cnpj, email, telefone, aleatoria
    chave_pix VARCHAR(100) NOT NULL,

    -- Asaas
    asaas_transfer_id VARCHAR(100),

    -- Status
    status saque_status NOT NULL DEFAULT 'solicitado',
    motivo_falha TEXT,

    -- Transação relacionada
    transacao_id UUID REFERENCES wallet_transactions(id),

    -- Auditoria
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    processed_at TIMESTAMP WITH TIME ZONE,

    CONSTRAINT valor_positivo CHECK (valor > 0),
    CONSTRAINT valor_liquido_positivo CHECK (valor_liquido > 0)
);

CREATE INDEX idx_saques_entregador ON saques(entregador_id);
CREATE INDEX idx_saques_wallet ON saques(wallet_id);
```

```sql
CREATE INDEX idx_saques_status ON saques(status);
CREATE INDEX idx_saques_created ON saques(created_at);
```

Tabela: empresas_config_financeiro

Configurações financeiras por empresa.

sql

```sql
CREATE TYPE modalidade_pagamento AS ENUM ('prepago', 'pospago');
CREATE TYPE dia_semana AS ENUM ('domingo', 'segunda', 'terca', 'quarta', 'quinta', 'sexta', 'sabado');

CREATE TABLE empresas_config_financeiro (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    empresa_id UUID NOT NULL UNIQUE,

    -- Modalidade
    modalidade modalidade_pagamento NOT NULL DEFAULT 'prepago',

    -- Configurações pós-pago
    dia_fechamento dia_semana DEFAULT 'domingo',
    dia_vencimento_boleto INTEGER DEFAULT 3, -- Dias após fechamento
    limite_credito DECIMAL(15,2) DEFAULT 0.00, -- Limite de crédito (0 = sem limite)

    -- Configurações gerais
    percentual_comissao DECIMAL(5,2) NOT NULL DEFAULT 20.00, -- % de comissão padrão

    -- Asaas
    asaas_customer_id VARCHAR(100), -- ID do cliente no Asaas

    -- Auditoria
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE TRIGGER update_empresas_config_updated_at
    BEFORE UPDATE ON empresas_config_financeiro
    FOR EACH ROW
    EXECUTE FUNCTION update_updated_at_column();
```

Tabela: webhooks_log

Log de todos os webhooks recebidos do Asaas.

sql

```sql
CREATE TABLE webhooks_log (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

    -- Identificação
    provider VARCHAR(50) NOT NULL DEFAULT 'asaas',
    event_type VARCHAR(100) NOT NULL,

    -- Dados
    payload JSONB NOT NULL,
    headers JSONB,

    -- Processamento
    processed BOOLEAN DEFAULT FALSE,
    processed_at TIMESTAMP WITH TIME ZONE,
    error_message TEXT,

    -- Referências
    cobranca_id UUID REFERENCES cobrancas(id),
    saque_id UUID REFERENCES saques(id),

    -- Auditoria
    received_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

CREATE INDEX idx_webhooks_event ON webhooks_log(event_type);
CREATE INDEX idx_webhooks_processed ON webhooks_log(processed);
CREATE INDEX idx_webhooks_received ON webhooks_log(received_at);
CREATE INDEX idx_webhooks_cobranca ON webhooks_log(cobranca_id);
```

Integração com Asaas

Configuração

javascript

```javascript
// config/asaas.js
module.exports = {
  baseUrl: process.env.ASAAS_SANDBOX === 'true'
    ? 'https://sandbox.asaas.com/api/v3'
    : 'https://api.asaas.com/v3',
  apiKey: process.env.ASAAS_API_KEY,
  webhookToken: process.env.ASAAS_WEBHOOK_TOKEN, // Para validar webhooks
};
```

Cliente HTTP para Asaas

javascript

```
// services/asaas/client.js
const axios = require('axios');
const config = require('../../config/asaas');

const asaasClient = axios.create({
  baseURL: config.baseUrl,
  headers: {
    'Content-Type': 'application/json',
    'access_token': config.apiKey,
  },
  timeout: 30000,
});

// Interceptor para logging
asaasClient.interceptors.request.use((request) => {
  console.log(`[ASAAS] ${request.method.toUpperCase()} ${request.url}`);
  return request;
});

asaasClient.interceptors.response.use(
  (response) => response,
  (error) => {
    console.error('[ASAAS] Erro:', error.response?.data || error.message);
    throw error;
  }
);

module.exports = asaasClient;
```

Serviço de Clientes (Asaas)

javascript

```
// services/asaas/customers.js
const asaasClient = require('./client');

/**
 * Cria ou atualiza cliente no Asaas
 * Deve ser chamado quando uma empresa se cadastra
 */
async function criarOuAtualizarCliente(empresa) {
  // Primeiro, verifica se já existe pelo CPF/CNPJ
  const existing = await buscarClientePorCpfCnpj(empresa.cpf_cnpj);

  if (existing) {
    return existing;
  }

  const payload = {
    name: empresa.razao_social || empresa.nome,
    cpfCnpj: empresa.cpf_cnpj.replace(/\D/g, ''),
    email: empresa.email,
    phone: empresa.telefone?.replace(/\D/g, ''),
    mobilePhone: empresa.celular?.replace(/\D/g, ''),
    address: empresa.endereco?.logradouro,
    addressNumber: empresa.endereco?.numero,
    complement: empresa.endereco?.complemento,
    province: empresa.endereco?.bairro,
    postalCode: empresa.endereco?.cep?.replace(/\D/g, ''),
    externalReference: empresa.id, // ID da empresa no nosso sistema
    notificationDisabled: false,
  };

  const response = await asaasClient.post('/customers', payload);
  return response.data;
}

async function buscarClientePorCpfCnpj(cpfCnpj) {
  const cpfCnpjLimpo = cpfCnpj.replace(/\D/g, '');
  const response = await asaasClient.get('/customers', {
    params: { cpfCnpj: cpfCnpjLimpo }
  });

  return response.data.data?.[0] || null;
}

async function buscarClientePorId(asaasCustomerId) {
  const response = await asaasClient.get(`/customers/${asaasCustomerId}`);
  return response.data;
```

```
}

module.exports = {
  criarOuAtualizarCliente,
  buscarClientePorCpfCnpj,
  buscarClientePorId,
};
```

Serviço de Cobranças (Asaas)

javascript

```
// services/asaas/cobrancas.js
const asaasClient = require('./client');

/**
 * Cria cobrança via PIX
 */
async function criarCobrancaPix({
  asaasCustomerId,
  valor,
  descricao,
  externalReference,
  vencimento, // Date object
}) {
  const payload = {
    customer: asaasCustomerId,
    billingType: 'PIX',
    value: valor,
    dueDate: formatarData(vencimento),
    description: descricao,
    externalReference: externalReference,
  };

  const response = await asaasClient.post('/payments', payload);

  // Buscar QR Code do PIX
  const pixData = await buscarQrCodePix(response.data.id);

  return {
    ...response.data,
    pix: pixData,
  };
}

/**
 * Cria cobrança via Boleto
 */
async function criarCobrancaBoleto({
  asaasCustomerId,
  valor,
  descricao,
  externalReference,
  vencimento,
  diasAposVencimento = 3, // Dias de multa/juros após vencimento
}) {
  const payload = {
    customer: asaasCustomerId,
```

```
    billingType: 'BOLETO',
    value: valor,
    dueDate: formatarData(vencimento),
    description: descricao,
    externalReference: externalReference,
    fine: {
      value: 2, // 2% de multa
      type: 'PERCENTAGE',
    },
    interest: {
      value: 1, // 1% de juros ao mês
      type: 'PERCENTAGE',
    },
    postalService: false, // Não enviar boleto físico
  };

  const response = await asaasClient.post('/payments', payload);
  return response.data;
}

/**
 * Busca QR Code do PIX
 */
async function buscarQrCodePix(paymentId) {
  const response = await asaasClient.get(`/payments/${paymentId}/pixQrCode`);
  return response.data;
}

/**
 * Busca linha digitável do boleto
 */
async function buscarLinhaDigitavel(paymentId) {
  const response = await asaasClient.get(`/payments/${paymentId}/identificationField`);
  return response.data;
}

/**
 * Cancela cobrança
 */
async function cancelarCobranca(paymentId) {
  const response = await asaasClient.delete(`/payments/${paymentId}`);
  return response.data;
}

/**
 * Busca cobrança por ID
 */
```

```
async function buscarCobranca(paymentId) {
  const response = await asaasClient.get(`/payments/${paymentId}`);
  return response.data;
}

/**
 * Lista cobranças de um cliente
 */
async function listarCobrancasCliente(asaasCustomerId, filtros = {}) {
  const response = await asaasClient.get('/payments', {
    params: {
      customer: asaasCustomerId,
      ...filtros,
    },
  });
  return response.data;
}

function formatarData(date) {
  return date.toISOString().split('T')[0]; // YYYY-MM-DD
}

module.exports = {
  criarCobrancaPix,
  criarCobrancaBoleto,
  buscarQrCodePix,
  buscarLinhaDigitavel,
  cancelarCobranca,
  buscarCobranca,
  listarCobrancasCliente,
};
```

Serviço de Transferências/Saques (Asaas)

javascript

```
// services/asaas/transferencias.js
const asaasClient = require('./client');

/**
 * Realiza transferência PIX para entregador (saque)
 */
async function realizarTransferenciaPix({
  valor,
  tipoChavePix, // CPF, CNPJ, EMAIL, PHONE, EVP (aleatória)
  chavePix,
  descricao,
  externalReference,
}) {
  const payload = {
    value: valor,
    pixAddressKey: chavePix,
    pixAddressKeyType: tipoChavePix.toUpperCase(),
    description: descricao,
    externalReference: externalReference,
  };

  const response = await asaasClient.post('/transfers', payload);
  return response.data;
}

/**
 * Busca transferência por ID
 */
async function buscarTransferencia(transferId) {
  const response = await asaasClient.get(`/transfers/${transferId}`);
  return response.data;
}

/**
 * Lista transferências realizadas
 */
async function listarTransferencias(filtros = {}) {
  const response = await asaasClient.get('/transfers', { params: filtros });
  return response.data;
}

/**
 * Consulta saldo disponível na conta Asaas
 */
async function consultarSaldo() {
  const response = await asaasClient.get('/finance/balance');
```

```
  return response.data;
}

module.exports = {
  realizarTransferenciaPix,
  buscarTransferencia,
  listarTransferencias,
  consultarSaldo,
};
```

Serviços de Wallet (Lógica de Negócio)

Serviço Principal de Wallet

javascript

```
// services/wallet/walletService.js
const db = require('../../database');
const { v4: uuidv4 } = require('uuid');

class WalletService {
  /**
   * Cria wallet para um usuário
   */
  async criarWallet(usuarioId, tipo) {
    const result = await db.query(
      `INSERT INTO wallets (usuario_id, tipo)
       VALUES ($1, $2)
       ON CONFLICT (usuario_id, tipo) DO NOTHING
       RETURNING *`,
      [usuarioId, tipo]
    );

    // Se já existia, busca
    if (result.rows.length === 0) {
      return this.buscarWallet(usuarioId, tipo);
    }

    return result.rows[0];
  }

  /**
   * Busca wallet de um usuário
   */
  async buscarWallet(usuarioId, tipo) {
    const result = await db.query(
      `SELECT * FROM wallets WHERE usuario_id = $1 AND tipo = $2`,
      [usuarioId, tipo]
    );
    return result.rows[0];
  }

  /**
   * Busca wallet por ID
   */
  async buscarWalletPorId(walletId) {
    const result = await db.query(
      `SELECT * FROM wallets WHERE id = $1`,
      [walletId]
    );
    return result.rows[0];
  }
```

```
/**
 * Credita valor na wallet (aumenta saldo)
 * Usado para: recargas, créditos de entrega
 */
async creditar(walletId, valor, tipo, opcoes = {}) {
  const client = await db.getClient();

  try {
    await client.query('BEGIN');

    // Lock na wallet para evitar race condition
    const walletResult = await client.query(
      `SELECT * FROM wallets WHERE id = $1 FOR UPDATE`,
      [walletId]
    );

    const wallet = walletResult.rows[0];
    if (!wallet) {
      throw new Error('Wallet não encontrada');
    }

    if (wallet.status !== 'ativo') {
      throw new Error('Wallet não está ativa');
    }

    const saldoAnterior = parseFloat(wallet.saldo_disponivel);
    const saldoPosterior = saldoAnterior + parseFloat(valor);

    // Atualiza saldo
    await client.query(
      `UPDATE wallets SET saldo_disponivel = $1 WHERE id = $2`,
      [saldoPosterior, walletId]
    );

    // Registra transação
    const transacao = await client.query(
      `INSERT INTO wallet_transactions
       (wallet_id, tipo, status, valor, saldo_anterior, saldo_posterior,
        entrega_id, cobranca_id, descricao, metadata, processed_at)
       VALUES ($1, $2, 'concluida', $3, $4, $5, $6, $7, $8, $9, NOW())
       RETURNING *`,
      [
        walletId,
        tipo,
        valor,
        saldoAnterior,
```

```
        saldoPosterior,
        opcoes.entregaId || null,
        opcoes.cobrancaId || null,
        opcoes.descricao || null,
        JSON.stringify(opcoes.metadata || {}),
      ]
    );

    await client.query('COMMIT');

    return {
      wallet: { ...wallet, saldo_disponivel: saldoPosterior },
      transacao: transacao.rows[0],
    };
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

/**
 * Debita valor da wallet (diminui saldo)
 * Usado para: pagamento de entregas, saques
 */
async debitar(walletId, valor, tipo, opcoes = {}) {
  const client = await db.getClient();

  try {
    await client.query('BEGIN');

    // Lock na wallet
    const walletResult = await client.query(
      `SELECT * FROM wallets WHERE id = $1 FOR UPDATE`,
      [walletId]
    );

    const wallet = walletResult.rows[0];
    if (!wallet) {
      throw new Error('Wallet não encontrada');
    }

    if (wallet.status !== 'ativo') {
      throw new Error('Wallet não está ativa');
    }
```

```
  const saldoAnterior = parseFloat(wallet.saldo_disponivel);

  // Verifica saldo suficiente (exceto para pós-pago que permite negativo)
  if (!opcoes.permitirNegativo && saldoAnterior < parseFloat(valor)) {
    throw new Error('Saldo insuficiente');
  }

  const saldoPosterior = saldoAnterior - parseFloat(valor);

  // Atualiza saldo
  await client.query(
    `UPDATE wallets SET saldo_disponivel = $1 WHERE id = $2`,
    [saldoPosterior, walletId]
  );

  // Registra transação
  const transacao = await client.query(
    `INSERT INTO wallet_transactions
     (wallet_id, tipo, status, valor, saldo_anterior, saldo_posterior,
      entrega_id, cobranca_id, descricao, metadata, processed_at)
     VALUES ($1, $2, 'concluida', $3, $4, $5, $6, $7, $8, $9, NOW())
     RETURNING *`,
    [
      walletId,
      tipo,
      valor,
      saldoAnterior,
      saldoPosterior,
      opcoes.entregaId || null,
      opcoes.cobrancaId || null,
      opcoes.descricao || null,
      JSON.stringify(opcoes.metadata || {}),
    ]
  );

  await client.query('COMMIT');

  return {
    wallet: { ...wallet, saldo_disponivel: saldoPosterior },
    transacao: transacao.rows[0],
  };
} catch (error) {
  await client.query('ROLLBACK');
  throw error;
} finally {
  client.release();
}
```

```
}

/**
 * Bloqueia parte do saldo (move de disponível para bloqueado)
 */
async bloquearSaldo(walletId, valor) {
  const client = await db.getClient();

  try {
    await client.query('BEGIN');

    const walletResult = await client.query(
      `SELECT * FROM wallets WHERE id = $1 FOR UPDATE`,
      [walletId]
    );

    const wallet = walletResult.rows[0];
    if (parseFloat(wallet.saldo_disponivel) < parseFloat(valor)) {
      throw new Error('Saldo insuficiente para bloqueio');
    }

    await client.query(
      `UPDATE wallets
       SET saldo_disponivel = saldo_disponivel - $1,
           saldo_bloqueado = saldo_bloqueado + $1
       WHERE id = $2`,
      [valor, walletId]
    );

    await client.query('COMMIT');

    return this.buscarWalletPorId(walletId);
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

/**
 * Desbloqueia saldo
 */
async desbloquearSaldo(walletId, valor) {
  const client = await db.getClient();

  try {
```

```
    await client.query('BEGIN');

    await client.query(
      `UPDATE wallets
       SET saldo_disponivel = saldo_disponivel + $1,
           saldo_bloqueado = saldo_bloqueado - $1
       WHERE id = $2 AND saldo_bloqueado >= $1`,
      [valor, walletId]
    );

    await client.query('COMMIT');

    return this.buscarWalletPorId(walletId);
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

/**
 * Busca extrato da wallet
 */
async buscarExtrato(walletId, opcoes = {}) {
  const { dataInicio, dataFim, tipo, limit = 50, offset = 0 } = opcoes;

  let query = `
    SELECT * FROM wallet_transactions
    WHERE wallet_id = $1
  `;
  const params = [walletId];
  let paramIndex = 2;

  if (dataInicio) {
    query += ` AND created_at >= $${paramIndex}`;
    params.push(dataInicio);
    paramIndex++;
  }

  if (dataFim) {
    query += ` AND created_at <= $${paramIndex}`;
    params.push(dataFim);
    paramIndex++;
  }

  if (tipo) {
```

```
    query += ` AND tipo = $${paramIndex}`;
    params.push(tipo);
    paramIndex++;
  }

  query += ` ORDER BY created_at DESC LIMIT $${paramIndex} OFFSET $${paramIndex + 1}`;
  params.push(limit, offset);

  const result = await db.query(query, params);
  return result.rows;
 }

 /**
  * Busca a wallet da plataforma (singleton)
  */
 async getWalletPlataforma() {
  // ID fixo para a plataforma
  const PLATAFORMA_ID = '00000000-0000-0000-0000-000000000001';

  let wallet = await this.buscarWallet(PLATAFORMA_ID, 'plataforma');

  if (!wallet) {
    wallet = await this.criarWallet(PLATAFORMA_ID, 'plataforma');
  }

  return wallet;
 }
}

module.exports = new WalletService();
```

Serviço de Recarga

javascript

```
// services/wallet/recargaService.js
const db = require('../../database');
const walletService = require('./walletService');
const asaasCustomers = require('../asaas/customers');
const asaasCobrancas = require('../asaas/cobrancas');

class RecargaService {
 /**
  * Gera uma recarga via PIX
  */
 async gerarRecargaPix(empresaId, valor) {
  // Busca configuração financeira da empresa
  const config = await this.getConfigEmpresa(empresaId);

  // Busca ou cria wallet da empresa
  let wallet = await walletService.buscarWallet(empresaId, 'empresa');
  if (!wallet) {
   wallet = await walletService.criarWallet(empresaId, 'empresa');
  }

  // Busca ou cria cliente no Asaas
  let asaasCustomerId = config.asaas_customer_id;
  if (!asaasCustomerId) {
   const empresa = await this.getEmpresa(empresaId);
   const customer = await asaasCustomers.criarOuAtualizarCliente(empresa);
   asaasCustomerId = customer.id;

   // Salva o ID do cliente Asaas
   await db.query(
    `UPDATE empresas_config_financeiro
     SET asaas_customer_id = $1
     WHERE empresa_id = $2`,
    [asaasCustomerId, empresaId]
   );
  }

  // Cria cobrança no Asaas
  const vencimento = new Date();
  vencimento.setDate(vencimento.getDate() + 1); // Vence amanhã

  const cobrancaAsaas = await asaasCobrancas.criarCobrancaPix({
   asaasCustomerId,
   valor,
   descricao: `Recarga de créditos - App Entregas`,
   externalReference: `recarga_${empresaId}_${Date.now()}`,
   vencimento,
```

```
  });

  // Salva cobrança no banco
  const cobranca = await db.query(
    `INSERT INTO cobrancas
     (empresa_id, wallet_id, asaas_id, asaas_customer_id, tipo, forma_pagamento,
      valor, data_vencimento, status, pix_copia_cola, pix_qrcode_url)
     VALUES ($1, $2, $3, $4, 'recarga', 'pix', $5, $6, 'aguardando_pagamento', $7, $8)
     RETURNING *`,
    [
      empresaId,
      wallet.id,
      cobrancaAsaas.id,
      asaasCustomerId,
      valor,
      vencimento,
      cobrancaAsaas.pix.payload, // PIX copia e cola
      cobrancaAsaas.pix.encodedImage, // QR Code em base64
    ]
  );

  return {
    cobranca: cobranca.rows[0],
    pix: {
      copiaCola: cobrancaAsaas.pix.payload,
      qrCodeBase64: cobrancaAsaas.pix.encodedImage,
      expiresAt: cobrancaAsaas.pix.expirationDate,
    },
  };
}

/**
 * Gera uma recarga via Boleto
 */
async gerarRecargaBoleto(empresaId, valor) {
  const config = await this.getConfigEmpresa(empresaId);

  let wallet = await walletService.buscarWallet(empresaId, 'empresa');
  if (!wallet) {
    wallet = await walletService.criarWallet(empresaId, 'empresa');
  }

  let asaasCustomerId = config.asaas_customer_id;
  if (!asaasCustomerId) {
    const empresa = await this.getEmpresa(empresaId);
    const customer = await asaasCustomers.criarOuAtualizarCliente(empresa);
    asaasCustomerId = customer.id;
```

```
    await db.query(
      `UPDATE empresas_config_financeiro
       SET asaas_customer_id = $1
       WHERE empresa_id = $2`,
      [asaasCustomerId, empresaId]
    );
  }

  const vencimento = new Date();
  vencimento.setDate(vencimento.getDate() + 3); // Vence em 3 dias

  const cobrancaAsaas = await asaasCobrancas.criarCobrancaBoleto({
    asaasCustomerId,
    valor,
    descricao: `Recarga de créditos - App Entregas`,
    externalReference: `recarga_${empresaId}_${Date.now()}`,
    vencimento,
  });

  // Busca linha digitável
  const boletoInfo = await asaasCobrancas.buscarLinhaDigitavel(cobrancaAsaas.id);

  const cobranca = await db.query(
    `INSERT INTO cobrancas
     (empresa_id, wallet_id, asaas_id, asaas_customer_id, tipo, forma_pagamento,
      valor, data_vencimento, status, boleto_url, boleto_codigo_barras, linha_digitavel)
     VALUES ($1, $2, $3, $4, 'recarga', 'boleto', $5, $6, 'aguardando_pagamento', $7, $8, $9)
     RETURNING *`,
    [
      empresaId,
      wallet.id,
      cobrancaAsaas.id,
      asaasCustomerId,
      valor,
      vencimento,
      cobrancaAsaas.bankSlipUrl,
      boletoInfo.barCode,
      boletoInfo.identificationField,
    ]
  );

  return {
    cobranca: cobranca.rows[0],
    boleto: {
      url: cobrancaAsaas.bankSlipUrl,
      codigoBarras: boletoInfo.barCode,
```

```
      linhaDigitavel: boletoInfo.identificationField,
      vencimento: vencimento,
    },
  };
}

/**
 * Confirma pagamento de recarga (chamado pelo webhook)
 */
async confirmarPagamentoRecarga(cobrancaId, dadosPagamento) {
  const client = await db.getClient();

  try {
    await client.query('BEGIN');

    // Busca cobrança
    const cobrancaResult = await client.query(
      `SELECT * FROM cobrancas WHERE id = $1 FOR UPDATE`,
      [cobrancaId]
    );

    const cobranca = cobrancaResult.rows[0];
    if (!cobranca) {
      throw new Error('Cobrança não encontrada');
    }

    if (cobranca.status === 'confirmada') {
      // Já foi processada, idempotência
      await client.query('COMMIT');
      return { jaProcessado: true };
    }

    // Atualiza status da cobrança
    await client.query(
      `UPDATE cobrancas
       SET status = 'confirmada',
           data_pagamento = $1,
           valor_liquido = $2,
           metadata = metadata || $3
       WHERE id = $4`,
      [
        dadosPagamento.dataPagamento || new Date(),
        dadosPagamento.valorLiquido || cobranca.valor,
        JSON.stringify(dadosPagamento),
        cobrancaId,
      ]
    );
```

```
    await client.query('COMMIT');

    // Credita na wallet (fora da transação para evitar deadlock)
    const resultado = await walletService.creditar(
      cobranca.wallet_id,
      cobranca.valor,
      'recarga',
      {
        cobrancaId: cobranca.id,
        descricao: `Recarga via ${cobranca.forma_pagamento.toUpperCase()}`,
        metadata: dadosPagamento,
      }
    );

    return resultado;
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

async getConfigEmpresa(empresaId) {
  const result = await db.query(
    `SELECT * FROM empresas_config_financeiro WHERE empresa_id = $1`,
    [empresaId]
  );

  if (result.rows.length === 0) {
    // Cria configuração padrão
    const insert = await db.query(
      `INSERT INTO empresas_config_financeiro (empresa_id) VALUES ($1) RETURNING *`,
      [empresaId]
    );
    return insert.rows[0];
  }

  return result.rows[0];
}

async getEmpresa(empresaId) {
  // Implemente conforme sua tabela de empresas
  const result = await db.query(
    `SELECT * FROM empresas WHERE id = $1`,
    [empresaId]
```

```
    );
    return result.rows[0];
  }
}

module.exports = new RecargaService();
```

Serviço de Entregas (Financeiro)

javascript

```
// services/wallet/entregaFinanceiroService.js
const db = require('../../database');
const walletService = require('./walletService');

class EntregaFinanceiroService {
 /**
  * Processa o pagamento de uma entrega
  * Fluxo PRÉ-PAGO: debita empresa, credita entregador e plataforma
  */
 async processarEntregaPrepago(entrega) {
  const client = await db.getClient();

  try {
   await client.query('BEGIN');

   // Busca configuração da empresa
   const configResult = await client.query(
    `SELECT * FROM empresas_config_financeiro WHERE empresa_id = $1`,
    [entrega.empresa_id]
   );
   const config = configResult.rows[0];

   if (config.modalidade !== 'prepago') {
    throw new Error('Empresa não está em modalidade pré-pago');
   }

   // Calcula valores
   const valorTotal = parseFloat(entrega.valor);
   const percentualComissao = parseFloat(config.percentual_comissao) / 100;
   const valorComissao = Math.round(valorTotal * percentualComissao * 100) / 100;
   const valorEntregador = valorTotal - valorComissao;

   // Busca wallets
   const walletEmpresa = await walletService.buscarWallet(entrega.empresa_id, 'empresa');
   const walletEntregador = await walletService.buscarWallet(entrega.entregador_id, 'entregador');
   const walletPlataforma = await walletService.getWalletPlataforma();

   if (!walletEmpresa || !walletEntregador) {
    throw new Error('Wallet não encontrada');
   }

   // Verifica saldo da empresa
   if (parseFloat(walletEmpresa.saldo_disponivel) < valorTotal) {
    throw new Error('Saldo insuficiente');
   }
```

```
    await client.query('COMMIT');

    // Processa transações (cada uma em sua própria transação)
    const transacaoDebito = await walletService.debitar(
      walletEmpresa.id,
      valorTotal,
      'debito_entrega',
      {
        entregaId: entrega.id,
        descricao: `Entrega #${entrega.codigo}`,
      }
    );

    const transacaoCredito = await walletService.creditar(
      walletEntregador.id,
      valorEntregador,
      'credito_entrega',
      {
        entregaId: entrega.id,
        descricao: `Entrega #${entrega.codigo}`,
      }
    );

    const transacaoComissao = await walletService.creditar(
      walletPlataforma.id,
      valorComissao,
      'comissao',
      {
        entregaId: entrega.id,
        descricao: `Comissão entrega #${entrega.codigo}`,
      }
    );

    // Registra split na tabela de controle
    await db.query(
      `INSERT INTO entregas_financeiro
       (entrega_id, empresa_id, entregador_id, valor_total, valor_entregador,
        valor_comissao, percentual_comissao, transacao_debito_empresa_id,
        transacao_credito_entregador_id, transacao_comissao_id, processado, processado_at)
       VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9, $10, true, NOW())`,
      [
        entrega.id,
        entrega.empresa_id,
        entrega.entregador_id,
        valorTotal,
        valorEntregador,
        valorComissao,
```

```
        config.percentual_comissao,
        transacaoDebito.transacao.id,
        transacaoCredito.transacao.id,
        transacaoComissao.transacao.id,
      ]
    );

    return {
      valorTotal,
      valorEntregador,
      valorComissao,
      transacoes: {
        debito: transacaoDebito.transacao,
        credito: transacaoCredito.transacao,
        comissao: transacaoComissao.transacao,
      },
    };
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

/**
 * Registra entrega para cobrança posterior (pós-pago)
 * Não movimenta dinheiro, apenas registra para fechamento semanal
 */
async registrarEntregaPospago(entrega) {
  // Busca configuração
  const configResult = await db.query(
    `SELECT * FROM empresas_config_financeiro WHERE empresa_id = $1`,
    [entrega.empresa_id]
  );
  const config = configResult.rows[0];

  if (config.modalidade !== 'pospago') {
    throw new Error('Empresa não está em modalidade pós-pago');
  }

  // Calcula valores
  const valorTotal = parseFloat(entrega.valor);
  const percentualComissao = parseFloat(config.percentual_comissao) / 100;
  const valorComissao = Math.round(valorTotal * percentualComissao * 100) / 100;
  const valorEntregador = valorTotal - valorComissao;
```

```
    // Busca ou cria wallet do entregador (para creditar depois)
    let walletEntregador = await walletService.buscarWallet(entrega.entregador_id, 'entregador');
    if (!walletEntregador) {
      walletEntregador = await walletService.criarWallet(entrega.entregador_id, 'entregador');
    }

    // Registra entrega pendente de cobrança
    const result = await db.query(
      `INSERT INTO entregas_financeiro
      (entrega_id, empresa_id, entregador_id, valor_total, valor_entregador,
       valor_comissao, percentual_comissao, processado)
      VALUES ($1, $2, $3, $4, $5, $6, $7, false)
      RETURNING *`,
      [
        entrega.id,
        entrega.empresa_id,
        entrega.entregador_id,
        valorTotal,
        valorEntregador,
        valorComissao,
        config.percentual_comissao,
      ]
    );

    return result.rows[0];
  }
}

module.exports = new EntregaFinanceiroService();
```

Serviço de Fechamento Semanal (Pós-Pago)

javascript

```
// services/wallet/fechamentoService.js
const db = require('../../database');
const walletService = require('./walletService');
const recargaService = require('./recargaService');
const asaasCobrancas = require('../asaas/cobrancas');

class FechamentoService {
  /**
   * Processa fechamento semanal de uma empresa pós-pago
   */
  async processarFechamentoEmpresa(empresaId) {
    const client = await db.getClient();

    try {
      await client.query('BEGIN');

      // Busca configuração
      const configResult = await client.query(
        `SELECT * FROM empresas_config_financeiro WHERE empresa_id = $1`,
        [empresaId]
      );
      const config = configResult.rows[0];

      if (config.modalidade !== 'pospago') {
        throw new Error('Empresa não está em modalidade pós-pago');
      }

      // Busca entregas não processadas (não cobradas ainda)
      const entregasResult = await client.query(
        `SELECT * FROM entregas_financeiro
         WHERE empresa_id = $1
         AND processado = false
         AND cobranca_id IS NULL
         FOR UPDATE`,
        [empresaId]
      );

      const entregas = entregasResult.rows;

      if (entregas.length === 0) {
        await client.query('COMMIT');
        return { message: 'Nenhuma entrega para cobrar' };
      }

      // Calcula total
      const valorTotal = entregas.reduce(
```

```
  (sum, e) => sum + parseFloat(e.valor_total),
  0
);

// Define período
const periodoInicio = entregas.reduce(
  (min, e) => e.created_at < min ? e.created_at : min,
  entregas[0].created_at
);
const periodoFim = new Date();

await client.query('COMMIT');

// Busca wallet da empresa
let wallet = await walletService.buscarWallet(empresaId, 'empresa');
if (!wallet) {
  wallet = await walletService.criarWallet(empresaId, 'empresa');
}

// Gera cobrança
const vencimento = new Date();
vencimento.setDate(vencimento.getDate() + config.dia_vencimento_boleto);

// Verifica se tem cliente Asaas
let asaasCustomerId = config.asaas_customer_id;
if (!asaasCustomerId) {
  const empresa = await recargaService.getEmpresa(empresaId);
  const customer = await require('../asaas/customers').criarOuAtualizarCliente(empresa);
  asaasCustomerId = customer.id;

  await db.query(
    `UPDATE empresas_config_financeiro SET asaas_customer_id = $1 WHERE empresa_id = $2`,
    [asaasCustomerId, empresaId]
  );
}

// Cria cobrança no Asaas (pode ser boleto ou PIX)
const cobrancaAsaas = await asaasCobrancas.criarCobrancaBoleto({
  asaasCustomerId,
  valor: valorTotal,
  descricao: `Entregas período ${this.formatarData(periodoInicio)} a ${this.formatarData(periodoFim)}`,
  externalReference: `pospago_${empresaId}_${Date.now()}`,
  vencimento,
});

const boletoInfo = await asaasCobrancas.buscarLinhaDigitavel(cobrancaAsaas.id);
```

```
// Salva cobrança
const cobrancaResult = await db.query(
  `INSERT INTO cobrancas
   (empresa_id, wallet_id, asaas_id, asaas_customer_id, tipo, forma_pagamento,
    valor, data_vencimento, status, boleto_url, boleto_codigo_barras,
    linha_digitavel, periodo_inicio, periodo_fim, metadata)
   VALUES ($1, $2, $3, $4, 'pospago', 'boleto', $5, $6, 'aguardando_pagamento',
           $7, $8, $9, $10, $11, $12)
   RETURNING *`,
  [
    empresaId,
    wallet.id,
    cobrancaAsaas.id,
    asaasCustomerId,
    valorTotal,
    vencimento,
    cobrancaAsaas.bankSlipUrl,
    boletoInfo.barCode,
    boletoInfo.identificationField,
    periodoInicio,
    periodoFim,
    JSON.stringify({ entregas_ids: entregas.map(e => e.id) }),
  ]
);

const cobranca = cobrancaResult.rows[0];

// Vincula entregas à cobrança
await db.query(
  `UPDATE entregas_financeiro SET cobranca_id = $1 WHERE id = ANY($2)`,
  [cobranca.id, entregas.map(e => e.id)]
);

return {
  cobranca,
  boleto: {
    url: cobrancaAsaas.bankSlipUrl,
    codigoBarras: boletoInfo.barCode,
    linhaDigitavel: boletoInfo.identificationField,
    vencimento,
  },
  resumo: {
    qtdEntregas: entregas.length,
    valorTotal,
    periodoInicio,
    periodoFim,
  },
```

```
    };
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

/**
 * Confirma pagamento de cobrança pós-pago
 * Credita os entregadores e a comissão da plataforma
 */
async confirmarPagamentoPospago(cobrancaId, dadosPagamento) {
  const client = await db.getClient();

  try {
    await client.query('BEGIN');

    // Busca cobrança
    const cobrancaResult = await client.query(
      `SELECT * FROM cobrancas WHERE id = $1 FOR UPDATE`,
      [cobrancaId]
    );

    const cobranca = cobrancaResult.rows[0];
    if (!cobranca) {
      throw new Error('Cobrança não encontrada');
    }

    if (cobranca.status === 'confirmada') {
      await client.query('COMMIT');
      return { jaProcessado: true };
    }

    // Busca entregas vinculadas
    const entregasResult = await client.query(
      `SELECT * FROM entregas_financeiro
       WHERE cobranca_id = $1 AND processado = false
       FOR UPDATE`,
      [cobrancaId]
    );

    const entregas = entregasResult.rows;

    // Atualiza status da cobrança
    await client.query(
```

```
      `UPDATE cobrancas
       SET status = 'confirmada',
           data_pagamento = $1,
           valor_liquido = $2
       WHERE id = $3`,
      [
        dadosPagamento.dataPagamento || new Date(),
        dadosPagamento.valorLiquido || cobranca.valor,
        cobrancaId,
      ]
    );

    await client.query('COMMIT');

    // Processa cada entrega (credita entregadores e comissão)
    const walletPlataforma = await walletService.getWalletPlataforma();

    for (const entrega of entregas) {
      // Busca ou cria wallet do entregador
      let walletEntregador = await walletService.buscarWallet(entrega.entregador_id, 'entregador');
      if (!walletEntregador) {
        walletEntregador = await walletService.criarWallet(entrega.entregador_id, 'entregador');
      }

      // Credita entregador
      const transacaoCredito = await walletService.creditar(
        walletEntregador.id,
        entrega.valor_entregador,
        'credito_entrega',
        {
          entregaId: entrega.entrega_id,
          cobrancaId: cobranca.id,
          descricao: `Entrega (pós-pago)`,
        }
      );

      // Credita comissão
      const transacaoComissao = await walletService.creditar(
        walletPlataforma.id,
        entrega.valor_comissao,
        'comissao',
        {
          entregaId: entrega.entrega_id,
          cobrancaId: cobranca.id,
          descricao: `Comissão entrega (pós-pago)`,
        }
      );
```

```
      // Marca como processada
      await db.query(
        `UPDATE entregas_financeiro
         SET processado = true,
             processado_at = NOW(),
             transacao_credito_entregador_id = $1,
             transacao_comissao_id = $2
         WHERE id = $3`,
        [
          transacaoCredito.transacao.id,
          transacaoComissao.transacao.id,
          entrega.id,
        ]
      );
    }

    return {
      cobranca,
      entregasProcessadas: entregas.length,
    };
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

/**
 * Job para rodar fechamento de todas as empresas pós-pago
 * Deve ser agendado para rodar no dia definido (ex: todo domingo)
 */
async executarFechamentoGeral() {
  // Busca todas as empresas pós-pago que têm entregas pendentes
  const empresasResult = await db.query(
    `SELECT DISTINCT ef.empresa_id
     FROM entregas_financeiro ef
     JOIN empresas_config_financeiro ecf ON ecf.empresa_id = ef.empresa_id
     WHERE ef.processado = false
     AND ef.cobranca_id IS NULL
     AND ecf.modalidade = 'pospago'`
  );

  const resultados = [];

  for (const row of empresasResult.rows) {
```

```
      try {
        const resultado = await this.processarFechamentoEmpresa(row.empresa_id);
        resultados.push({
          empresaId: row.empresa_id,
          sucesso: true,
          resultado,
        });
      } catch (error) {
        resultados.push({
          empresaId: row.empresa_id,
          sucesso: false,
          erro: error.message,
        });
      }
    }

    return resultados;
  }

  formatarData(date) {
    return new Date(date).toLocaleDateString('pt-BR');
  }
}

module.exports = new FechamentoService();
```

Serviço de Saques

javascript

```
// services/wallet/saqueService.js
const db = require('../../database');
const walletService = require('./walletService');
const asaasTransferencias = require('../asaas/transferencias');

class SaqueService {
 /**
  * Valor mínimo para saque
  */
 static VALOR_MINIMO_SAQUE = 10.00;

 /**
  * Taxa de saque (0 = sem taxa)
  */
 static TAXA_SAQUE = 0.00;

 /**
  * Solicita saque para entregador
  */
 async solicitarSaque(entregadorId, valor, dadosPix) {
  // Validações
  if (valor < SaqueService.VALOR_MINIMO_SAQUE) {
   throw new Error(`Valor mínimo para saque é R$ ${SaqueService.VALOR_MINIMO_SAQUE.toFixed(2)}`);
  }

  // Busca wallet
  const wallet = await walletService.buscarWallet(entregadorId, 'entregador');
  if (!wallet) {
   throw new Error('Wallet não encontrada');
  }

  if (wallet.status !== 'ativo') {
   throw new Error('Wallet não está ativa');
  }

  const saldoDisponivel = parseFloat(wallet.saldo_disponivel);
  if (saldoDisponivel < valor) {
   throw new Error('Saldo insuficiente');
  }

  // Calcula valor líquido
  const taxa = SaqueService.TAXA_SAQUE;
  const valorLiquido = valor - taxa;

  // Bloqueia o saldo
  await walletService.bloquearSaldo(wallet.id, valor);
```

```
try {
  // Cria registro do saque
  const saqueResult = await db.query(
    `INSERT INTO saques
     (entregador_id, wallet_id, valor, taxa, valor_liquido,
      tipo_chave_pix, chave_pix, status)
     VALUES ($1, $2, $3, $4, $5, $6, $7, 'processando')
     RETURNING *`,
    [
      entregadorId,
      wallet.id,
      valor,
      taxa,
      valorLiquido,
      dadosPix.tipo,
      dadosPix.chave,
    ]
  );

  const saque = saqueResult.rows[0];

  // Realiza transferência no Asaas
  const transferencia = await asaasTransferencias.realizarTransferenciaPix({
    valor: valorLiquido,
    tipoChavePix: dadosPix.tipo,
    chavePix: dadosPix.chave,
    descricao: `Saque App Entregas`,
    externalReference: saque.id,
  });

  // Atualiza saque com ID da transferência
  await db.query(
    `UPDATE saques SET asaas_transfer_id = $1 WHERE id = $2`,
    [transferencia.id, saque.id]
  );

  // Debita da wallet (remove do bloqueado)
  const transacao = await this.finalizarSaque(saque.id, transferencia);

  return {
    saque: { ...saque, asaas_transfer_id: transferencia.id },
    transferencia,
    transacao,
  };
} catch (error) {
  // Em caso de erro, desbloqueia o saldo
```

```
    await walletService.desbloquearSaldo(wallet.id, valor);

    // Atualiza status do saque
    await db.query(
      `UPDATE saques
       SET status = 'falhou', motivo_falha = $1
       WHERE entregador_id = $2 AND status = 'processando'`,
      [error.message, entregadorId]
    );

    throw error;
  }
}

/**
 * Finaliza saque após confirmação
 */
async finalizarSaque(saqueId, dadosTransferencia) {
  const client = await db.getClient();

  try {
    await client.query('BEGIN');

    // Busca saque
    const saqueResult = await client.query(
      `SELECT * FROM saques WHERE id = $1 FOR UPDATE`,
      [saqueId]
    );

    const saque = saqueResult.rows[0];
    if (!saque) {
      throw new Error('Saque não encontrado');
    }

    if (saque.status === 'concluido') {
      await client.query('COMMIT');
      return { jaProcessado: true };
    }

    // Busca wallet
    const walletResult = await client.query(
      `SELECT * FROM wallets WHERE id = $1 FOR UPDATE`,
      [saque.wallet_id]
    );

    const wallet = walletResult.rows[0];
```

```
    // Remove do saldo bloqueado (já foi transferido)
    const saldoBloqueadoAnterior = parseFloat(wallet.saldo_bloqueado);
    const saldoBloqueadoPosterior = saldoBloqueadoAnterior - parseFloat(saque.valor);

    await client.query(
      `UPDATE wallets SET saldo_bloqueado = $1 WHERE id = $2`,
      [saldoBloqueadoPosterior, wallet.id]
    );

    // Registra transação
    const saldoDisponivel = parseFloat(wallet.saldo_disponivel);

    const transacaoResult = await client.query(
      `INSERT INTO wallet_transactions
       (wallet_id, tipo, status, valor, saldo_anterior, saldo_posterior,
        descricao, metadata, processed_at)
       VALUES ($1, 'saque', 'concluida', $2, $3, $4, $5, $6, NOW())
       RETURNING *`,
      [
        wallet.id,
        saque.valor,
        saldoDisponivel + saldoBloqueadoAnterior, // Saldo total antes
        saldoDisponivel + saldoBloqueadoPosterior, // Saldo total depois
        `Saque PIX`,
        JSON.stringify(dadosTransferencia || {}),
      ]
    );

    // Atualiza saque
    await client.query(
      `UPDATE saques
       SET status = 'concluido',
           processed_at = NOW(),
           transacao_id = $1
       WHERE id = $2`,
      [transacaoResult.rows[0].id, saqueId]
    );

    await client.query('COMMIT');

    return transacaoResult.rows[0];
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
```

```
  }

  /**
   * Lista saques de um entregador
   */
  async listarSaques(entregadorId, opcoes = {}) {
    const { status, limit = 50, offset = 0 } = opcoes;

    let query = `SELECT * FROM saques WHERE entregador_id = $1`;
    const params = [entregadorId];

    if (status) {
      query += ` AND status = $2`;
      params.push(status);
    }

    query += ` ORDER BY created_at DESC LIMIT $${params.length + 1} OFFSET $${params.length + 2}`;
    params.push(limit, offset);

    const result = await db.query(query, params);
    return result.rows;
  }
}

module.exports = new SaqueService();
```

Webhooks (Asaas)

Controller de Webhooks

javascript

```
// controllers/webhookController.js
const db = require('../database');
const recargaService = require('../services/wallet/recargaService');
const fechamentoService = require('../services/wallet/fechamentoService');

/**
 * Eventos do Asaas que nos interessam:
 * - PAYMENT_CONFIRMED: Pagamento confirmado (PIX/Boleto)
 * - PAYMENT_RECEIVED: Pagamento recebido (similar ao confirmed)
 * - PAYMENT_OVERDUE: Pagamento vencido
 * - PAYMENT_DELETED: Pagamento deletado/cancelado
 * - PAYMENT_REFUNDED: Pagamento estornado
 * - TRANSFER_CONFIRMED: Transferência confirmada (saques)
 * - TRANSFER_FAILED: Transferência falhou
 */

async function handleAsaasWebhook(req, res) {
  try {
    const { event, payment, transfer } = req.body;

    // Log do webhook
    const logResult = await db.query(
      `INSERT INTO webhooks_log (provider, event_type, payload, headers)
       VALUES ('asaas', $1, $2, $3)
       RETURNING id`,
      [event, JSON.stringify(req.body), JSON.stringify(req.headers)]
    );
    const webhookLogId = logResult.rows[0].id;

    let resultado;

    switch (event) {
      case 'PAYMENT_CONFIRMED':
      case 'PAYMENT_RECEIVED':
        resultado = await processarPagamentoConfirmado(payment);
        break;

      case 'PAYMENT_OVERDUE':
        resultado = await processarPagamentoVencido(payment);
        break;

      case 'PAYMENT_DELETED':
      case 'PAYMENT_REFUNDED':
        resultado = await processarPagamentoCancelado(payment, event);
        break;
```

```
      case 'TRANSFER_CONFIRMED':
        resultado = await processarTransferenciaConfirmada(transfer);
        break;

      case 'TRANSFER_FAILED':
        resultado = await processarTransferenciaFalhou(transfer);
        break;

      default:
        console.log(`Evento não tratado: ${event}`);
    }

    // Atualiza log como processado
    await db.query(
      `UPDATE webhooks_log
       SET processed = true, processed_at = NOW()
       WHERE id = $1`,
      [webhookLogId]
    );

    res.status(200).json({ received: true, resultado });
  } catch (error) {
    console.error('Erro no webhook:', error);

    // Salva erro no log
    await db.query(
      `UPDATE webhooks_log
       SET error_message = $1
       WHERE id = (SELECT id FROM webhooks_log ORDER BY received_at DESC LIMIT 1)`,
      [error.message]
    );

    // Retorna 200 mesmo com erro para não ficar retentando
    res.status(200).json({ received: true, error: error.message });
  }
}

async function processarPagamentoConfirmado(payment) {
  // Busca cobrança pelo ID do Asaas
  const cobrancaResult = await db.query(
    `SELECT * FROM cobrancas WHERE asaas_id = $1`,
    [payment.id]
  );

  const cobranca = cobrancaResult.rows[0];
  if (!cobranca) {
    console.log(`Cobrança não encontrada: ${payment.id}`);
```

```
    return { ignorado: true, motivo: 'Cobrança não encontrada' };
  }

  const dadosPagamento = {
    dataPagamento: payment.confirmedDate || payment.paymentDate,
    valorPago: payment.value,
    valorLiquido: payment.netValue,
    formaPagamento: payment.billingType,
    asaasData: payment,
  };

  if (cobranca.tipo === 'recarga') {
    return await recargaService.confirmarPagamentoRecarga(cobranca.id, dadosPagamento);
  } else if (cobranca.tipo === 'pospago') {
    return await fechamentoService.confirmarPagamentoPospago(cobranca.id, dadosPagamento);
  }

  return { processado: true };
}

async function processarPagamentoVencido(payment) {
  await db.query(
    `UPDATE cobrancas SET status = 'vencida' WHERE asaas_id = $1`,
    [payment.id]
  );

  return { atualizado: true };
}

async function processarPagamentoCancelado(payment, event) {
  const status = event === 'PAYMENT_REFUNDED' ? 'estornada' : 'cancelada';

  await db.query(
    `UPDATE cobrancas SET status = $1 WHERE asaas_id = $2`,
    [status, payment.id]
  );

  // TODO: Se necessário, reverter créditos

  return { atualizado: true, status };
}

async function processarTransferenciaConfirmada(transfer) {
  // Busca saque pelo ID da transferência
  const saqueResult = await db.query(
    `SELECT * FROM saques WHERE asaas_transfer_id = $1`,
    [transfer.id]
```

```
  );

  const saque = saqueResult.rows[0];
  if (!saque) {
    return { ignorado: true };
  }

  // Já deve ter sido processado no momento da solicitação
  // Mas podemos atualizar status se necessário
  await db.query(
    `UPDATE saques SET status = 'concluido', processed_at = NOW() WHERE id = $1`,
    [saque.id]
  );

  return { processado: true };
}

async function processarTransferenciaFalhou(transfer) {
  const saqueResult = await db.query(
    `SELECT * FROM saques WHERE asaas_transfer_id = $1`,
    [transfer.id]
  );

  const saque = saqueResult.rows[0];
  if (!saque) {
    return { ignorado: true };
  }

  // Desbloqueia saldo
  const walletService = require('../services/wallet/walletService');
  await walletService.desbloquearSaldo(saque.wallet_id, saque.valor);

  // Atualiza status
  await db.query(
    `UPDATE saques
     SET status = 'falhou', motivo_falha = $1
     WHERE id = $2`,
    [transfer.failReason || 'Falha na transferência', saque.id]
  );

  return { processado: true, falha: true };
}

module.exports = {
  handleAsaasWebhook,
};
```

Rota do Webhook

javascript

```
// routes/webhooks.js
const express = require('express');
const router = express.Router();
const { handleAsaasWebhook } = require('../controllers/webhookController');

// Middleware para validar token do Asaas (opcional mas recomendado)
function validarWebhookAsaas(req, res, next) {
  const token = req.headers['asaas-access-token'];

  if (process.env.ASAAS_WEBHOOK_TOKEN && token !== process.env.ASAAS_WEBHOOK_TOKEN) {
    console.warn('Token de webhook inválido');
    return res.status(401).json({ error: 'Unauthorized' });
  }

  next();
}

router.post('/asaas', validarWebhookAsaas, handleAsaasWebhook);

module.exports = router;
```

APIs REST

Rotas de Wallet (Empresa)

javascript

```
// routes/empresa/wallet.js
const express = require('express');
const router = express.Router();
const walletService = require('../../services/wallet/walletService');
const recargaService = require('../../services/wallet/recargaService');

// Middleware de autenticação (implemente conforme seu sistema)
const authEmpresa = require('../../middlewares/authEmpresa');

router.use(authEmpresa);

/**
 * GET /empresa/wallet
 * Retorna saldo e dados da wallet da empresa
 */
router.get('/', async (req, res) => {
  try {
    const empresaId = req.empresa.id;

    let wallet = await walletService.buscarWallet(empresaId, 'empresa');

    if (!wallet) {
      wallet = await walletService.criarWallet(empresaId, 'empresa');
    }

    res.json({
      saldo_disponivel: wallet.saldo_disponivel,
      saldo_bloqueado: wallet.saldo_bloqueado,
      status: wallet.status,
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * GET /empresa/wallet/extrato
 * Retorna extrato de transações
 */
router.get('/extrato', async (req, res) => {
  try {
    const empresaId = req.empresa.id;
    const { data_inicio, data_fim, tipo, page = 1, limit = 50 } = req.query;

    const wallet = await walletService.buscarWallet(empresaId, 'empresa');
```

```
    if (!wallet) {
      return res.json({ transacoes: [] });
    }

    const extrato = await walletService.buscarExtrato(wallet.id, {
      dataInicio: data_inicio,
      dataFim: data_fim,
      tipo,
      limit: parseInt(limit),
      offset: (parseInt(page) - 1) * parseInt(limit),
    });

    res.json({ transacoes: extrato });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * POST /empresa/wallet/recarga/pix
 * Gera recarga via PIX
 */
router.post('/recarga/pix', async (req, res) => {
  try {
    const empresaId = req.empresa.id;
    const { valor } = req.body;

    if (!valor || valor < 10) {
      return res.status(400).json({ error: 'Valor mínimo de recarga é R$ 10,00' });
    }

    const resultado = await recargaService.gerarRecargaPix(empresaId, valor);

    res.json(resultado);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * POST /empresa/wallet/recarga/boleto
 * Gera recarga via Boleto
 */
router.post('/recarga/boleto', async (req, res) => {
  try {
    const empresaId = req.empresa.id;
    const { valor } = req.body;
```

```
    if (!valor || valor < 10) {
      return res.status(400).json({ error: 'Valor mínimo de recarga é R$ 10,00' });
    }

    const resultado = await recargaService.gerarRecargaBoleto(empresaId, valor);

    res.json(resultado);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * GET /empresa/wallet/cobrancas
 * Lista cobranças da empresa
 */
router.get('/cobrancas', async (req, res) => {
  try {
    const empresaId = req.empresa.id;
    const { status, tipo, page = 1, limit = 20 } = req.query;

    let query = `SELECT * FROM cobrancas WHERE empresa_id = $1`;
    const params = [empresaId];

    if (status) {
      query += ` AND status = $${params.length + 1}`;
      params.push(status);
    }

    if (tipo) {
      query += ` AND tipo = $${params.length + 1}`;
      params.push(tipo);
    }

    query += ` ORDER BY created_at DESC LIMIT $${params.length + 1} OFFSET $${params.length + 2}`;
    params.push(parseInt(limit), (parseInt(page) - 1) * parseInt(limit));

    const result = await db.query(query, params);

    res.json({ cobrancas: result.rows });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

```
module.exports = router;
```

Rotas de Wallet (Entregador)

javascript

```
// routes/entregador/wallet.js
const express = require('express');
const router = express.Router();
const walletService = require('../../services/wallet/walletService');
const saqueService = require('../../services/wallet/saqueService');

const authEntregador = require('../../middlewares/authEntregador');

router.use(authEntregador);

/**
 * GET /entregador/wallet
 * Retorna saldo e dados da wallet do entregador
 */
router.get('/', async (req, res) => {
  try {
    const entregadorId = req.entregador.id;

    let wallet = await walletService.buscarWallet(entregadorId, 'entregador');

    if (!wallet) {
      wallet = await walletService.criarWallet(entregadorId, 'entregador');
    }

    res.json({
      saldo_disponivel: wallet.saldo_disponivel,
      saldo_bloqueado: wallet.saldo_bloqueado,
      status: wallet.status,
      valor_minimo_saque: saqueService.constructor.VALOR_MINIMO_SAQUE,
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * GET /entregador/wallet/extrato
 * Retorna extrato de transações
 */
router.get('/extrato', async (req, res) => {
  try {
    const entregadorId = req.entregador.id;
    const { data_inicio, data_fim, tipo, page = 1, limit = 50 } = req.query;

    const wallet = await walletService.buscarWallet(entregadorId, 'entregador');
```

```
    if (!wallet) {
      return res.json({ transacoes: [] });
    }

    const extrato = await walletService.buscarExtrato(wallet.id, {
      dataInicio: data_inicio,
      dataFim: data_fim,
      tipo,
      limit: parseInt(limit),
      offset: (parseInt(page) - 1) * parseInt(limit),
    });

    res.json({ transacoes: extrato });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

/**
 * POST /entregador/wallet/saque
 * Solicita saque
 */
router.post('/saque', async (req, res) => {
  try {
    const entregadorId = req.entregador.id;
    const { valor, tipo_chave_pix, chave_pix } = req.body;

    if (!valor || !tipo_chave_pix || !chave_pix) {
      return res.status(400).json({
        error: 'Informe valor, tipo_chave_pix e chave_pix'
      });
    }

    const tiposValidos = ['cpf', 'cnpj', 'email', 'telefone', 'aleatoria'];
    if (!tiposValidos.includes(tipo_chave_pix.toLowerCase())) {
      return res.status(400).json({
        error: 'Tipo de chave PIX inválido'
      });
    }

    const resultado = await saqueService.solicitarSaque(
      entregadorId,
      valor,
      { tipo: tipo_chave_pix, chave: chave_pix }
    );

    res.json(resultado);
```

```
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
});

/**
 * GET /entregador/wallet/saques
 * Lista saques do entregador
 */
router.get('/saques', async (req, res) => {
  try {
    const entregadorId = req.entregador.id;
    const { status, page = 1, limit = 20 } = req.query;

    const saques = await saqueService.listarSaques(entregadorId, {
      status,
      limit: parseInt(limit),
      offset: (parseInt(page) - 1) * parseInt(limit),
    });

    res.json({ saques });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

module.exports = router;
```

Jobs Agendados

Job de Fechamento Semanal

javascript

```javascript
// jobs/fechamentoSemanal.js
const cron = require('node-cron');
const fechamentoService = require('../services/wallet/fechamentoService');

/**
 * Executa todo domingo às 23:00
 * Gera cobranças para todas as empresas pós-pago
 */
function iniciarJobFechamento() {
  // Formato: segundo minuto hora dia-do-mes mes dia-da-semana
  cron.schedule('0 0 23 * * 0', async () => {
    console.log('Iniciando fechamento semanal...');

    try {
      const resultados = await fechamentoService.executarFechamentoGeral();

      console.log('Fechamento concluído:');
      console.log(`- Empresas processadas: ${resultados.length}`);
      console.log(`- Sucessos: ${resultados.filter(r => r.sucesso).length}`);
      console.log(`- Falhas: ${resultados.filter(r => !r.sucesso).length}`);

      // Log detalhado de falhas
      resultados
        .filter(r => !r.sucesso)
        .forEach(r => console.error(`Falha empresa ${r.empresaId}: ${r.erro}`));

    } catch (error) {
      console.error('Erro no fechamento semanal:', error);
    }
  }, {
    timezone: 'America/Sao_Paulo'
  });

  console.log('Job de fechamento semanal agendado para domingos às 23:00');
}

module.exports = { iniciarJobFechamento };
```

Job de Verificação de Cobranças Vencidas

javascript

```
// jobs/verificarCobrancasVencidas.js
const cron = require('node-cron');
const db = require('../database');

/**
 * Executa todo dia às 08:00
 * Atualiza status de cobranças vencidas
 */
function iniciarJobVerificacaoVencidas() {
  cron.schedule('0 0 8 * * *', async () => {
    console.log('Verificando cobranças vencidas...');

    try {
      const result = await db.query(
        `UPDATE cobrancas
         SET status = 'vencida'
         WHERE status = 'aguardando_pagamento'
         AND data_vencimento < CURRENT_DATE
         RETURNING id`
      );

      console.log(`${result.rowCount} cobranças marcadas como vencidas`);

      // TODO: Enviar notificações para empresas com cobranças vencidas

    } catch (error) {
      console.error('Erro na verificação de vencidas:', error);
    }
  }, {
    timezone: 'America/Sao_Paulo'
  });

  console.log('Job de verificação de vencidas agendado para 08:00 diariamente');
}

module.exports = { iniciarJobVerificacaoVencidas };
```

Variáveis de Ambiente

```
env
```

```
# .env

# Banco de dados
DATABASE_URL=postgresql://user:password@localhost:5432/entregas

# Asaas
ASAAS_API_KEY=sua_api_key_aqui
ASAAS_SANDBOX=true
ASAAS_WEBHOOK_TOKEN=token_secreto_para_webhooks

# App
NODE_ENV=development
PORT=3000

# Timezone
TZ=America/Sao_Paulo
```

Checklist de Implementação

Fase 1 - Infraestrutura

- [ ] Criar tabelas no banco de dados
- [ ] Configurar cliente HTTP do Asaas
- [ ] Implementar serviço de clientes Asaas
- [ ] Implementar serviço de cobranças Asaas
- [ ] Implementar serviço de transferências Asaas

Fase 2 - Wallets

- [ ] Implementar WalletService (CRUD, crédito, débito)
- [ ] Implementar RecargaService (PIX e Boleto)
- [ ] Implementar SaqueService
- [ ] Criar rotas de API para empresas
- [ ] Criar rotas de API para entregadores

Fase 3 - Entregas

- [ ] Integrar EntregaFinanceiroService com fluxo de entregas existente
- [ ] Implementar processamento pré-pago
- [ ] Implementar registro pós-pago

Fase 4 - Fechamento e Webhooks

- [ ] Implementar FechamentoService

- [ ] Configurar endpoint de webhook
- [ ] Implementar handlers de eventos
- [ ] Configurar jobs agendados

Fase 5 - Testes e Deploy

- [ ] Testar fluxo completo em sandbox
- [ ] Configurar webhooks no painel Asaas
- [ ] Migrar para produção
- [ ] Monitorar primeiras transações

Configuração do Webhook no Asaas

1. Acesse o painel do Asaas
2. Vá em Integrações > Webhooks
3. Adicione um novo webhook:
   - **URL**: `https://seudominio.com/webhooks/asaas`
   - **Eventos**:
     - PAYMENT_CONFIRMED
     - PAYMENT_RECEIVED
     - PAYMENT_OVERDUE
     - PAYMENT_DELETED
     - PAYMENT_REFUNDED
     - TRANSFER_CONFIRMED
     - TRANSFER_FAILED
4. Copie o token gerado e configure em `ASAAS_WEBHOOK_TOKEN`

Considerações de Segurança

1. **Validação de Webhooks**: Sempre valide o token do webhook
2. **Idempotência**: Todas as operações de webhook devem ser idempotentes
3. **Transações**: Use transações de banco para operações críticas
4. **Locks**: Use `FOR UPDATE` para evitar race conditions
5. **Logs**: Mantenha log de todas as operações financeiras
6. **Auditoria**: Nunca delete registros, use soft delete quando necessário