



# Análise Completa da Arquitetura - Timming LoveU

**Data da Análise:** 23 de Outubro de 2024

**Versão do Projeto:** 1.0.0

**Analista:** DeepAgent - Abacus.AI



## Sumário Executivo

O **Timming LoveU** é uma aplicação web moderna desenvolvida para casais criarem páginas românticas personalizadas. A aplicação permite:

- Cronômetro do tempo de relacionamento
- Galeria de fotos e vídeos
- Música personalizada
- Compartilhamento via link único
- Sistema de assinatura/planos (estrutura criada, pagamento a implementar)



## Stack Tecnológico

### Frontend

- **Framework:** Next.js 14.2.33 (App Router)
- **Linguagem:** TypeScript 5.2.2
- **UI Library:** React 18.2.0
- **Estilização:**
  - Tailwind CSS 3.3.3
  - Radix UI (componentes primitivos)
  - Lucide React (ícones)
  - Framer Motion (animações)

### Backend

- **Framework:** Next.js API Routes (serverless)
- **Autenticação:** NextAuth.js 4.24.11
- **ORM:** Prisma 6.7.0
- **Database:** PostgreSQL 15+
- **Hash de Senhas:** bcryptjs 2.4.3
- **Validação:** Zod 3.23.8, Yup 1.3.0

### DevOps & Infraestrutura

- **Containerização:** Docker & Docker Compose
- **CI/CD:** Scripts automatizados (pre-deploy, post-deploy)
- **Testes:** Jest 30.2.0 + React Testing Library
- **Linting:** ESLint 9.38.0 + TypeScript ESLint

- **Reverse Proxy:** Nginx (opcional)
  - **Cache:** Redis 7 (opcional, estrutura criada)
-

## Estrutura do Projeto

---



```

next.config.js      # Configuração do Next.js
tsconfig.json       # Configuração TypeScript
tailwind.config.ts  # Configuração Tailwind
package.json        # Dependências e scripts
.env               # Variáveis de ambiente

```

## Modelo de Dados (Prisma Schema)

### Entidades Principais

#### 1. User (Usuário)

```

- id: String (PK)
- email: String (unique)
- password: String (hashed)
- firstName: String
- lastName: String
- planoAtivo: Boolean (indica se tem plano ativo)
- dataExpiracaoPlano: DateTime
- isAdmin: Boolean
- createdAt: DateTime
- updatedAt: DateTime

```

#### Relacionamentos:

- 1:N com Account (NextAuth)
- 1:N com Session (NextAuth)
- 1:N com CouplePage
- 1:N com Media

#### 2. CouplePage (Página do Casal)

```

- id: String (PK)
- userId: String (FK -> User)
- nomeCasal: String
- mensagem: String (opcional)
- dataInicioRelacao: DateTime
- musicaUrl: String (opcional)
- musicaFile: String (opcional)
- bannerUrl: String (opcional)
- galeria: String[] (array de URLs)
- videos: String[] (array de URLs)
- slugPublico: String (unique - URL da página)
- ativo: Boolean
- views: Int (contador de visualizações)
- createdAt: DateTime
- updatedAt: DateTime

```

#### Relacionamentos:

- N:1 com User
- 1:N com Media

### 3. Media (Fotos/Vídeos)

```
- id: String (PK)
- couplePageId: String (FK -> CouplePage)
- userId: String (FK -> User)
- tipo: String ('image' ou 'video')
- url: String (path ou URL do arquivo)
- fileName: String
- fileSize: Int (tamanho em bytes)
- mimeType: String
- titulo: String (opcional)
- descricao: String (opcional)
- dataEvento: DateTime (data da foto/vídeo)
- ordem: Int (ordem de exibição)
- ativo: Boolean
- createdAt: DateTime
- updatedAt: DateTime
```

#### Relacionamentos:

- N:1 com CouplePage
- N:1 com User

### 4. PlanoAssinatura (Planos de Assinatura)

```
- id: String (PK)
- nome: String
- preco: Decimal
- descricao: String (opcional)
- duracaoMeses: Int (default: 1)
- maxPaginas: Int (default: 1)
- stripeProductId: String (opcional - para integração Stripe)
- stripePriceId: String (opcional - para integração Stripe)
- ativo: Boolean
- createdAt: DateTime
- updatedAt: DateTime
```

**⚠ IMPORTANTE:** Atualmente **não há relação direta** entre User e PlanoAssinatura. O campo planoAtivo no User é apenas um booleano. Para implementar o sistema de pagamento, será necessário:

1. Criar tabela UserSubscription para relacionar User com PlanoAssinatura
2. Armazenar informações de pagamento (Stripe Customer ID, Subscription ID)
3. Implementar webhooks do Stripe para sincronização

### 5. NextAuth Models

- Account : Contas OAuth/providers
  - Session : Sessões de usuário
  - VerificationToken : Tokens de verificação de email
-

# Sistema de Autenticação

## Implementação Atual

**Provider:** Credentials (Email/Senha)

- **Localização:** `/lib/auth/auth-options.ts`
- **Adapter:** Prisma Adapter
- **Estratégia:** JWT (sem database sessions)
- **Hash:** bcryptjs com salt rounds = 12

## Fluxo de Autenticação

1. Usuário envia email/senha  
↓
2. **NextAuth** valida credenciais via `authorize()`  
↓
3. Busca usuário no banco (Prisma)  
↓
4. Compara hash da senha (`bcrypt.compare`)  
↓
5. Retorna dados do usuário (se válido)  
↓
6. JWT é criado com dados do usuário  
↓
7. Cookie de sessão é setado

## Proteção de Rotas

**Server-side:**

```
const session = await getServerSession(authOptions)
if (!session) redirect('/login')
```

**Client-side:**

```
import { useSession } from 'next-auth/react'
const { data: session, status } = useSession()
```

## Dados na Sessão

```
{
  user: {
    id: string
    email: string
    name: string
    firstName: string
    lastName: string
    planoAtivo: boolean
    isAdmin: boolean
  }
}
```

## API Routes

---

### Endpoints Principais

#### 1. Autenticação

- `POST /api/auth/[...nextauth]` - NextAuth endpoints (login, logout, session)
- `POST /api/signup` - Registro de novos usuários

#### 2. Couple Management

- `GET /api/couple/stats` - Estatísticas da página do casal
- Endpoints adicionais podem existir (não listados explicitamente no código revisado)

#### 3. Media Management

- `POST /api/media/upload` - Upload de fotos/vídeos
- `GET /api/media/list` - Listar mídia
- `DELETE /api/media/delete` - Deletar mídia

#### 4. Contact

- `POST /api/contact` - Formulário de contato

#### 5. Health Check

- `GET /api/health` - Status da aplicação e banco de dados
- `HEAD /api/health` - Health check simplificado (load balancers)

#### Resposta do Health Check:

```
{
  "status": "healthy",
  "timestamp": "2024-10-23T10:00:00.000Z",
  "uptime": 12345,
  "responseTime": "45ms",
  "checks": {
    "database": "healthy"
  },
  "version": "1.0.0",
  "environment": "production"
}
```

---

## Frontend Architecture

---

### Padrões de Design

#### 1. App Router (Next.js 14)

- Server Components por padrão
- Client Components marcados com `'use client'`
- Streaming e Suspense para loading states

#### 2. Componentes Principais

##### Dashboard:

- `DashboardClient` - Container principal (client component)
- `CoupleInfo` - Informações do casal
- `RelationshipStats` - Estatísticas do relacionamento



- `MediaGallery` - Galeria de fotos/vídeos
- `MediaUpload` - Upload de mídia
- `Milestones` - Marcos importantes

#### UI Components:

- Baseados em Radix UI primitives
- Estilizados com Tailwind CSS
- Tema configurável (ThemeProvider)

### 3. State Management

- React Hooks (useState, useEffect)
- SWR para data fetching (2.2.4)
- Zustand para estado global (5.0.3)
- React Hook Form para formulários (7.53.0)

### 4. Estilização

- **Tailwind CSS:** Utility-first
- **CSS Modules:** `globals.css`
- **Variáveis CSS:** Temas e cores
- **Animações:** Framer Motion

### 5. Fontes

- **Playfair Display:** Fonte romântica/títulos
- **Inter:** Fonte corpo/UI
- Google Fonts com `next/font`



## Segurança

### Implementações de Segurança

#### 1. Headers HTTP (next.config.js)

- ✓ X-DNS-Prefetch-Control: on
- ✓ Strict-Transport-Security: HSTS habilitado
- ✓ X-Frame-Options: SAMEORIGIN (proteção clickjacking)
- ✓ X-Content-Type-Options: nosniff
- ✓ X-XSS-Protection: 1; mode=block
- ✓ Referrer-Policy: strict-origin-when-cross-origin
- ✓ Permissions-Policy: camera=(), microphone=(), geolocation=()
- ✓ Cache-Control: no-store para APIs

#### 2. Autenticação

- ✓ Senhas hashadas com bcrypt (12 rounds)
- ✓ JWT para sessões
- ✓ NEXTAUTH\_SECRET obrigatório
- ✓ Timeout de sessão

#### 3. Database

- ✓ Prisma ORM (proteção contra SQL injection)
- ✓ Prepared statements automáticos

- ☒ Connection pooling

#### 4. Rate Limiting

- ⚠️ Estrutura criada ( lib/rate-limit.ts )
- ⚠️ Não implementado nas rotas ainda

#### 5. Validação

- ☒ Validação server-side (Zod, Yup)
- ☒ Validação client-side (React Hook Form)
- ☒ Sanitização de inputs

#### 6. CORS

- ⚠️ Configuração não explícita (Next.js padrão)

#### 7. Environment Variables

- ☒ Variáveis sensíveis em .env
- ☒ .env não commitado (.gitignore)
- ☒ Validação de variáveis obrigatórias



## Docker & Deploy

### Docker Compose Stack

```

Services:
├── db (PostgreSQL 15)
│   ├── Port: 5432
│   ├── Volume: postgres_data
│   └── Health check: pg_isready
├── app (Next.js)
│   ├── Port: 3000
│   ├── Depends on: db
│   └── Health check: /api/health
├── redis (Redis 7)
│   ├── Port: 6379
│   ├── Volume: redis_data
│   └── AOF persistence
└── nginx (Alpine)
    ├── Ports: 80, 443
    ├── SSL support
    └── Profile: with-nginx (opcional)
  
```

### Multi-stage Dockerfile

#### Stages:

1. **deps** - Instala dependências
2. **builder** - Build da aplicação
3. **runner** - Imagem final otimizada

#### Características:

- ☒ Non-root user (nextjs:nodejs)

- ☒ Alpine Linux (imagem leve)
- ☒ dumb-init para signal handling
- ☒ Health check integrado
- ☒ Multi-platform support (ARM64)

## Opções de Deploy

### 1. Vercel (Recomendado)

- ☒ Otimizado para Next.js
- ☒ Deploy automático (Git push)
- ☒ Edge Functions
- ☒ Escalabilidade automática

### 2. Docker (VPS/Cloud)

- ☒ Portabilidade total
- ☒ Controle completo
- ☒ Self-hosted

### 3. Manual (Node.js)

- ☒ PM2 ecosystem
- ☒ Nginx reverse proxy
- ☒ Systemd service



## Análise de Performance

## Otimizações Implementadas

### 1. Build Optimization

- ☒ SWC Minification (mais rápido que Babel)
- ☒ Tree shaking automático
- ☒ Code splitting por rota

### 2. Images

- ☒ Next.js Image optimization
- ☒ AVIF e WebP formats
- ☒ Lazy loading automático
- ☒ Responsive images




### 3. Caching

- ☒ Compression habilitado
- ☒ Static assets cache
- ⚠️ Redis configurado mas não utilizado ainda

### 4. Database

- ☒ Connection pooling (Prisma)
- ☒ Índices criados (couplePageld, userId)
- ⚠️ Queries podem ser otimizadas (ex: uso de `select` específico)

## 5. Fonts

-  Google Fonts otimizado
-  Font display: swap
-  Preload automático

---

## Testes

### Setup de Testes

**Framework:** Jest 30.2.0

**Testing Library:** React Testing Library 16.3.0

### Estrutura de Testes

```
__tests__/  
├── integration/  
│   ├── api-couple-stats.test.ts  
│   └── api-media.test.ts  
└── components/dashboard/__tests__/  
    ├── ...  
    └── ...
```

### Scripts de Teste

```
npm test           # Rodar todos os testes  
npm run test:watch # Watch mode  
npm run test:coverage # Cobertura  
npm run test:unit   # Testes unitários  
npm run test:integration # Testes de integração
```

### Mocks

- NextAuth ( `__mocks__/next-auth.ts` )
  - Prisma ( `__mocks__/prisma.ts` )
-

## Dependências Importantes

### Production Dependencies (principais)

Pacote	Versão	Uso
next	14.2.33	Framework principal
react	18.2.0	UI library
next-auth	4.24.11	Autenticação
@prisma/client	6.7.0	ORM/Database
bcryptjs	2.4.3	Hash de senhas
zod	3.23.8	Validação de schemas
tailwindcss	3.3.3	Estilização
framer-motion	10.18.0	Animações
sonner	2.0.7	Notificações toast
date-fns	3.6.0	Manipulação de datas




### Dev Dependencies (principais)

Pacote	Versão	Uso
typescript	5.2.2	Type checking
jest	30.2.0	Testes
eslint	9.38.0	Linting
prisma	6.7.0	CLI do Prisma

## Integrações Planejadas

### Sistema de Pagamento (NÃO IMPLEMENTADO)

#### Estrutura Preparada:

-  Tabela `PlanoAssinatura` no banco
-  Campos `planoAtivo` e `dataExpiracaoPlano` no User
-  Campos `stripeProductId` e `stripePriceId` no PlanoAssinatura

#### O que PRECISA ser implementado:

## 1. Integração Stripe

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

// Variáveis de ambiente
STRIPE_SECRET_KEY=sk_...
STRIPE_PUBLISHABLE_KEY=pk_...
STRIPE_WEBHOOK_SECRET=whsec_...
```

## 2. API Routes Necessários

```
POST /api/payment/create-checkout - Criar sessão de checkout
POST /api/payment/webhook - Webhook do Stripe
POST /api/payment/portal - Portal do cliente Stripe
GET /api/payment/plans - Listar planos disponíveis
```

## 3. Tabela de Assinaturas

```
model UserSubscription {
  id String @id @default(cuid())
  userId String @unique
  planoId String
  stripeCustomerId String?
  stripeSubscriptionId String?
  stripePriceId String?
  status String // active, canceled, past_due, etc.
  currentPeriodEnd DateTime
  cancelAtPeriodEnd Boolean @default(false)
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  user User @relation(fields: [userId], references: [id])
  plano PlanoAssinatura @relation(fields: [planoId], references: [id])
}
```

## 4. Webhook Handler

- Eventos a escutar:
- checkout.session.completed
- customer.subscription.updated
- customer.subscription.deleted
- invoice.payment\_succeeded
- invoice.payment\_failed

## 5. UI Components

- Página de planos
  - Formulário de pagamento (Stripe Elements)
  - Portal de gerenciamento de assinatura
  - Status de pagamento no dashboard
-



## Fluxo de Dados Principal

### Criação de Página de Casal

1. Usuário faz login  
↓
2. Acessa Dashboard (/dashboard)  
↓
3. Clica em "Criar Página"  
↓
4. Preenche formulário (CoupleInfo)
  - Nome do casal
  - Data início relacionamento
  - Mensagem
  - Banner
  - Música
 ↓
5. POST /api/couple/create  
↓
6. Prisma cria CouplePage
  - Gera slugPublico **Ú**nico
 ↓
7. Retorna URL da página  
↓
8. Usuário pode:
  - Upload de fotos (MediaUpload)
  - Ver estatísticas (RelationshipStats)
  - Compartilhar link

### Upload de Mídia

1. Usuário seleciona arquivo  
↓
2. Validação client-side
  - Tipo de arquivo
  - Tamanho (ex: **max** 10MB)
 ↓
3. FormData **é** criado  
↓
4. POST /api/media/upload  
↓
5. Validação server-side  
↓
6. Arquivo **é** salvo em /public/uploads  
↓
7. Prisma cria registro Media
  - URL do arquivo
  - Metadata
 ↓
8. Retorna dados do upload  
↓
9. UI atualiza gallery

## Visualização Pública

1. Alguém acessa /[slug]  
↓
2. Busca CouplePage por slugPublico  
↓
3. Incrementa contador views  
↓
4. Renderiza página pública
  - Cronômetro em tempo **real**
  - Galeria de fotos
  - Música de fundo
  - Informações do casal
 ↓
5. JavaScript calcula tempo decorrido



## Problemas e Limitações Identificadas



### Críticos

1. **Sistema de Pagamento Incompleto**
  - Tabela PlanoAssinatura existe mas não é usada
  - Sem integração Stripe
  - Sem validação de plano ativo
2. **Upload de Mídia**
  - Armazenamento local (/public/uploads)
  - Não escalável em produção
  - **Recomendação:** Integrar S3, Cloudinary ou similar
3. **Rate Limiting**
  - Código existe mas não está aplicado nas rotas
  - Vulnerável a abuse



### Médios

1. **Caching**
  - Redis configurado mas não utilizado
  - Sem cache de queries do Prisma
2. **Validação de Email**
  - Tabela VerificationToken existe
  - Sem implementação de envio de email
3. **Monitoramento**
  - Sem logging estruturado
  - Sem APM (Application Performance Monitoring)
  - Sem alertas
4. **Backup**
  - Sem backup automático do banco
  - Sem política de retenção



## Menores

### 1. Testes

- Cobertura limitada
- Poucos testes de integração

### 2. Documentação

- Falta documentação de API
- Falta README completo para desenvolvedores

### 3. Acessibilidade

- Componentes UI podem não estar 100% acessíveis
  - Falta testes de acessibilidade
- 






## Métricas e KPIs





---

### Métricas Disponíveis




#### User Level:

-  Número de usuários registrados
-  Data de criação de conta
-  Status do plano (boolean)







#### Couple Page Level:

-  Views da página
-  Número de mídias (fotos/vídeos)
-  Data de criação
-  Duração do relacionamento (calculado)

#### Media Level:

-  Total de uploads
-  Tamanho dos arquivos
-  Tipo de mídia

### Métricas Faltantes

-  Conversão de cadastro → criação de página
  -  Retenção de usuários
  -  Churn rate
  -  Receita (MRR, ARR)
  -  CAC (Customer Acquisition Cost)
  -  LTV (Lifetime Value)
-

## Recomendações para Implementação de Pagamento

### Fase 1: Setup Stripe

```
# 1. Instalar dependências
npm install stripe @stripe/stripe-js

# 2. Criar conta Stripe
# https://dashboard.stripe.com

# 3. Configurar webhooks
# URL: https://seu-dominio.com/api/payment/webhook
# Eventos: checkout.session.completed, customer.subscription.*

# 4. Criar produtos no Stripe Dashboard
# Ex: Plano Mensal - R$ 9,90
```

### Fase 2: Database Schema

```
// Adicionar ao schema.prisma

model UserSubscription {
  id          String    @id @default(cuid())
  userId      String    @unique
  planoId     String
  stripeCustomerId String? @unique
  stripeSubscriptionId String? @unique
  stripePriceId String?
  status      String    // active, canceled, past_due, trialing
  currentPeriodStart DateTime
  currentPeriodEnd  DateTime
  cancelAtPeriodEnd Boolean @default(false)
  trialEnd          DateTime?
  createdAt         DateTime @default(now())
  updatedAt         DateTime @updatedAt

  user User @relation(fields: [userId], references: [id], onDelete: Cascade)
  plano PlanoAssinatura @relation(fields: [planoId], references: [id])

  @@map("user_subscriptions")
}

// Atualizar modelo User
model User {
  // ... campos existentes
  subscription UserSubscription?
}

// Atualizar modelo PlanoAssinatura
model PlanoAssinatura {
  // ... campos existentes
  subscriptions UserSubscription[]
}
```

### Fase 3: API Implementation

Estrutura de arquivos:

```

app/api/payment/
├── create-checkout/route.ts    # Criar sessão Stripe
├── webhook/route.ts           # Receber eventos Stripe
├── portal/route.ts             # Portal do cliente
├── plans/route.ts              # Listar planos
└── subscription-status/route.ts # Status da assinatura

```

#### Exemplo - create-checkout:

```

import Stripe from 'stripe'
import { NextRequest, NextResponse } from 'next/server'
import { getServerSession } from 'next-auth'
import { authOptions } from '@lib/auth/auth-options'

const stripe = new Stripe(process.env.STRIPE_SECRET_KEY!, {
  apiVersion: '2023-10-16'
})

export async function POST(req: NextRequest) {
  const session = await getServerSession(authOptions)
  if (!session?.user) {
    return NextResponse.json({ error: 'Unauthorized' }, { status: 401 })
  }

  const { priceId } = await req.json()

  // Buscar ou criar Stripe Customer
  const user = await prisma.user.findUnique({
    where: { id: session.user.id },
    include: { subscription: true }
  })

  let customerId = user?.subscription?.stripeCustomerId

  if (!customerId) {
    const customer = await stripe.customers.create({
      email: user!.email,
      metadata: { userId: user!.id }
    })
    customerId = customer.id
  }

  // Criar Checkout Session
  const checkoutSession = await stripe.checkout.sessions.create({
    customer: customerId,
    mode: 'subscription',
    payment_method_types: ['card'],
    line_items: [{ price: priceId, quantity: 1 }],
    success_url: `${process.env.NEXTAUTH_URL}/dashboard?session_id={CHECKOUT_SESSION_ID}`,
    cancel_url: `${process.env.NEXTAUTH_URL}/pricing`,
    metadata: {
      userId: user!.id
    }
  })

  return NextResponse.json({ url: checkoutSession.url })
}

```

#### Exemplo - webhook:

```

import Stripe from 'stripe'
import { NextRequest, NextResponse } from 'next/server'
import { prisma } from '@lib/db'

const stripe = new Stripe(process.env.STRIPE_SECRET_KEY!, {
  apiVersion: '2023-10-16'
})

const webhookSecret = process.env.STRIPE_WEBHOOK_SECRET!

export async function POST(req: NextRequest) {
  const body = await req.text()
  const signature = req.headers.get('stripe-signature')!

  let event: Stripe.Event

  try {
    event = stripe.webhooks.constructEvent(body, signature, webhookSecret)
  } catch (err) {
    return NextResponse.json({ error: 'Webhook signature verification failed' }, { status: 400 })
  }

  switch (event.type) {
    case 'checkout.session.completed': {
      const session = event.data.object as Stripe.Checkout.Session
      await handleCheckoutCompleted(session)
      break
    }
    case 'customer.subscription.updated': {
      const subscription = event.data.object as Stripe.Subscription
      await handleSubscriptionUpdated(subscription)
      break
    }
    case 'customer.subscription.deleted': {
      const subscription = event.data.object as Stripe.Subscription
      await handleSubscriptionDeleted(subscription)
      break
    }
  }

  return NextResponse.json({ received: true })
}

async function handleCheckoutCompleted(session: Stripe.Checkout.Session) {
  const userId = session.metadata?.userId
  if (!userId) return

  const subscription = await stripe.subscriptions.retrieve(session.subscription as string)

  await prisma.userSubscription.upsert({
    where: { userId },
    create: {
      userId,
      planoId: 'plano-id-from-price',
      stripeCustomerId: session.customer as string,
      stripeSubscriptionId: subscription.id,
      stripePriceId: subscription.items.data[0].price.id,
      status: subscription.status,
      currentPeriodStart: new Date(subscription.current_period_start * 1000),
      currentPeriodEnd: new Date(subscription.current_period_end * 1000)
    }
  })
}

```

```

    },
    update: {
      stripeSubscriptionId: subscription.id,
      stripePriceId: subscription.items.data[0].price.id,
      status: subscription.status,
      currentPeriodStart: new Date(subscription.current_period_start * 1000),
      currentPeriodEnd: new Date(subscription.current_period_end * 1000)
    }
  })

  await prisma.user.update({
    where: { id: userId },
    data: {
      planoAtivo: true,
      dataExpiracaoPlano: new Date(subscription.current_period_end * 1000)
    }
  })
}

async function handleSubscriptionUpdated(subscription: Stripe.Subscription) {
  // Similar à handleCheckoutCompleted
}

async function handleSubscriptionDeleted(subscription: Stripe.Subscription) {
  await prisma.userSubscription.update({
    where: { stripeSubscriptionId: subscription.id },
    data: { status: 'canceled' }
  })

  const userSub = await prisma.userSubscription.findUnique({
    where: { stripeSubscriptionId: subscription.id }
  })

  if (userSub) {
    await prisma.user.update({
      where: { id: userSub.userId },
      data: { planoAtivo: false }
    })
  }
}

```

## Fase 4: Frontend Components

### Página de Planos:

```
// app/pricing/page.tsx
'use client'

import { useState } from 'react'
import { Button } from '@components/ui/button'
import { Card } from '@components/ui/card'
import { Check } from 'lucide-react'

const plans = [
  {
    id: 'mensal',
    name: 'Plano Mensal',
    price: 'R$ 9,90',
    stripePriceId: 'price_xxx',
    features: [
      '1 página personalizada',
      'Galeria ilimitada de fotos',
      'Upload de vídeos',
      'Música personalizada',
      'Cronômetro do relacionamento',
      'Compartilhamento ilimitado'
    ]
  },
  {
    id: 'anual',
    name: 'Plano Anual',
    price: 'R$ 99,00',
    priceDescription: 'Economize 17%',
    stripePriceId: 'price_yyy',
    features: [
      'Tudo do Plano Mensal',
      '2 meses grátis',
      'Suporte prioritário',
      'Novidades em primeira mão'
    ]
  }
]

export default function PricingPage() {
  const [loading, setLoading] = useState<string | null>(null)

  async function handleSubscribe(priceId: string, planId: string) {
    setLoading(planId)
    try {
      const res = await fetch('/api/payment/create-checkout', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ priceId })
      })
      const data = await res.json()
      window.location.href = data.url
    } catch (error) {
      console.error(error)
    } finally {
      setLoading(null)
    }
  }

  return (
    <div className="container mx-auto px-4 py-16">
      <h1 className="text-4xl font-bold text-center mb-4">
        Escolha Seu Plano
      </h1>
    </div>
  )
}
```

```

</h1>
<p className="text-center text-gray-600 mb-12">
  Teste grátis por 7 dias. Cancele quando quiser.
</p>

<div className="grid md:grid-cols-2 gap-8 max-w-4xl mx-auto">
  {plans.map(plan => (
    <Card key={plan.id} className="p-8">
      <h3 className="text-2xl font-bold mb-2">{plan.name}</h3>
      <div className="text-4xl font-bold mb-1">{plan.price}</div>
      {plan.priceDescription && (
        <p className="text-sm text-green-600 mb-6">{plan.priceDescription}</p>
      )}
      <ul className="space-y-3 mb-8">
        {plan.features.map(feature => (
          <li key={feature} className="flex items-start gap-2">
            <Check className="w-5 h-5 text-green-500 shrink-0 mt-0.5" />
            <span>{feature}</span>
          </li>
        ))}
      </ul>
      <Button
        size="lg"
        className="w-full"
        onClick={() => handleSubscribe(plan.stripePriceId, plan.id)}
        disabled={loading !== null}
      >
        {loading === plan.id ? 'Processando...' : 'Assinar Agora'}
      </Button>
    </Card>
  ))}
</div>
</div>
)
}

```

## Fase 5: Middleware de Validação

```
// middleware/requireSubscription.ts
import { getServerSession } from 'next-auth'
import { authOptions } from '@lib/auth/auth-options'
import { prisma } from '@lib/db'
import { redirect } from 'next/navigation'

export async function requireActiveSubscription() {
  const session = await getServerSession(authOptions)
  if (!session?.user) {
    redirect('/login')
  }

  const user = await prisma.user.findUnique({
    where: { id: session.user.id },
    include: { subscription: true }
  })

  const now = new Date()
  const hasActivePlan =
    user?.planoAtivo &&
    user.dataExpiracaoPlano &&
    user.dataExpiracaoPlano > now

  if (!hasActivePlan) {
    redirect('/pricing?expired=true')
  }

  return { user, session }
}
```



## Fase 6: Testing

```
// __tests__/api/payment.test.ts
import { POST } from '@app/api/payment/create-checkout/route'
import { getServerSession } from 'next-auth'

jest.mock('next-auth')
jest.mock('stripe')

describe('Payment API', () => {
  it('creates checkout session for authenticated user', async () => {
    (getServerSession as jest.Mock).mockResolvedValue({
      user: { id: 'user-123', email: 'test@example.com' }
    })

    const req = new Request('http://localhost:3000/api/payment/create-checkout', {
      method: 'POST',
      body: JSON.stringify({ priceId: 'price_123' })
    })

    const res = await POST(req as any)
    const data = await res.json()

    expect(data).toHaveProperty('url')
    expect(res.status).toBe(200)
  })

  it('returns 401 for unauthenticated user', async () => {
    (getServerSession as jest.Mock).mockResolvedValue(null)

    const req = new Request('http://localhost:3000/api/payment/create-checkout', {
      method: 'POST',
      body: JSON.stringify({ priceId: 'price_123' })
    })

    const res = await POST(req as any)
    expect(res.status).toBe(401)
  })
})
```



## Checklist de Implementação de Pagamento

### Setup Inicial

- [ ] Criar conta Stripe
- [ ] Instalar `stripe` e `@stripe/stripe-js`
- [ ] Configurar variáveis de ambiente (`STRIPE_SECRET_KEY`, etc.)
- [ ] Criar produtos/preços no Stripe Dashboard

### Database

- [ ] Atualizar `schema.prisma` com `UserSubscription`
- [ ] Executar migration
- [ ] Seed inicial de `PlanoAssinatura`

## Backend

- [ ] Implementar `/api/payment/create-checkout`
- [ ] Implementar `/api/payment/webhook`
- [ ] Implementar `/api/payment/portal`
- [ ] Implementar `/api/payment/plans`
- [ ] Implementar middleware de validação de plano
- [ ] Configurar webhook URL no Stripe Dashboard
- [ ] Testar webhooks localmente (Stripe CLI)

## Frontend

- [ ] Criar página `/pricing`
- [ ] Criar componente de planos
- [ ] Integrar Stripe Checkout
- [ ] Criar página de sucesso `/dashboard?session_id=xxx`
- [ ] Criar página de cancelamento `/pricing?canceled=true`
- [ ] Adicionar badge de plano no dashboard
- [ ] Adicionar portal de gerenciamento de assinatura

## Segurança

- [ ] Validar webhook signature
- [ ] Validar plano ativo em rotas protegidas
- [ ] Rate limiting em endpoints de pagamento
- [ ] Logging de eventos de pagamento

## Testes

- [ ] Testes unitários dos endpoints
- [ ] Testes de integração com Stripe (modo test)
- [ ] Testes do webhook handler
- [ ] Testes de validação de plano

## Deploy

- [ ] Configurar variáveis de ambiente em produção
- [ ] Configurar webhook em produção
- [ ] Testar fluxo completo em produção
- [ ] Monitorar logs do Stripe



## Recursos e Documentação

### Documentação Oficial

- **Next.js:** <https://nextjs.org/docs>
- **Prisma:** <https://www.prisma.io/docs>
- **NextAuth:** <https://next-auth.js.org>
- **Stripe:** <https://stripe.com/docs>
- **Tailwind CSS:** <https://tailwindcss.com/docs>

## Guias Internos do Projeto

- `DEPLOYMENT_GUIDE.md` - Guia completo de deploy
- `SECURITY_CHECKLIST.md` - Checklist de segurança
- `TESTING_GUIDE.md` - Guia de testes
- `PRODUCTION_README.md` - Setup de produção



## Ambiente de Desenvolvimento

### Pré-requisitos

```
# Node.js 18+
node --version

# PostgreSQL 15+
psql --version

# Git
git --version
```

### Setup Local

```
# 1. Clonar repositório
cd /home/ubuntu/timiming_loveu

# 2. Instalar dependências
npm install

# 3. Configurar .env
cp .env.example .env
# Editar .env com valores reais

# 4. Setup do banco
npx prisma generate
npx prisma migrate dev
npx prisma db seed

# 5. Iniciar desenvolvimento
npm run dev

# Aplicação rodando em http://localhost:3000
```

## Comandos Úteis

```
# Desenvolvimento
npm run dev           # Dev server
npm run build         # Build produção
npm run start         # Iniciar prod build
npm run lint          # Linter
npm run type-check    # Type checking

# Banco de Dados
npm run prisma:generate # Gerar Prisma Client
npm run prisma:migrate  # Migrations
npm run prisma:studio   # Prisma Studio UI
npm run prisma:seed     # Seed database

# Testes
npm test              # Rodar testes
npm run test:watch    # Watch mode
npm run test:coverage # Coverage

# Docker
npm run docker:build  # Build imagem
npm run docker:up     # Iniciar containers
npm run docker:down   # Parar containers
```

## Próximos Passos Recomendados

### Prioridade Alta

- 1. Implementar Sistema de Pagamento Stripe**
  - Seguir checklist acima
  - Estimativa: 2-3 dias
- 2. Migrar Upload de Mídia para Cloud Storage**
  - AWS S3 ou Cloudinary
  - Atualizar URL handling
  - Estimativa: 1-2 dias
- 3. Aplicar Rate Limiting**
  - Usar código existente em `lib/rate-limit.ts`
  - Aplicar em rotas sensíveis (signup, login, upload)
  - Estimativa: 4-6 horas

### Prioridade Média

- 1. Implementar Caching com Redis**
  - Cache de queries do Prisma
  - Cache de páginas públicas
  - Estimativa: 1 dia
- 2. Sistema de Email**
  - Verificação de email
  - Reset de senha

- Notificações de pagamento
- Estimativa: 1-2 dias

### 3. Melhorar Testes

- Aumentar cobertura para 80%+
- Testes E2E com Playwright
- Estimativa: 2-3 dias

## Prioridade Baixa

### 1. Logging e Monitoring

- Winston/Pino para logging
- Sentry para error tracking
- Posthog/Mixpanel para analytics
- Estimativa: 1 dia

### 2. Backup Automático

- Backup diário do PostgreSQL
- Retenção de 30 dias
- Estimativa: 4 horas

### 3. Documentação API

- Swagger/OpenAPI
- Postman collection
- Estimativa: 1 dia

---

## Suporte e Manutenção

---

### Logs

```
# Logs da aplicação
tail -f logs/app.log

# Logs do Docker
docker-compose logs -f app

# Logs do Prisma
# Configurado em lib/db.ts: log: ['query']
```

### Health Checks

```
# Local
curl http://localhost:3000/api/health

# Produção
curl https://seu-dominio.com/api/health
```

## Backup Manual

```
# PostgreSQL
pg_dump -h host -U user database > backup_$(date +%Y%m%d).sql

# Docker
docker-compose exec db pg_dump -U timing timing_loveu > backup.sql
```

## Restore

```
# PostgreSQL
psql -h host -U user database < backup.sql

# Docker
cat backup.sql | docker-compose exec -T db psql -U timing timing_loveu
```



## Conclusão

O **Timing LoveU** é uma aplicação bem estruturada, moderna e com fundação sólida para crescimento. A arquitetura baseada em Next.js 14 + Prisma + PostgreSQL é escalável e mantível.

### Pontos Fortes ✓

- ✓ Stack tecnológico moderno e consolidado
- ✓ Código TypeScript bem tipado
- ✓ Estrutura de pastas organizada
- ✓ Segurança básica implementada
- ✓ Docker pronto para produção
- ✓ Testes configurados
- ✓ CI/CD scripts prontos

### Áreas de Melhoria 🔧

- 🔧 Sistema de pagamento precisa ser implementado
- 🔧 Upload de mídia precisa ser movido para cloud
- 🔧 Rate limiting precisa ser aplicado
- 🔧 Caching não está sendo utilizado
- 🔧 Cobertura de testes pode ser melhorada

## Viabilidade para Produção

**Status:** ⚠️ **Quase Pronto** (com ressalvas)

A aplicação pode ser colocada em produção para testes beta ou MVP, mas para lançamento comercial completo, é **essencial** implementar:

1. Sistema de pagamento (Stripe)
2. Cloud storage para mídia
3. Rate limiting

**Estimativa de tempo para produção completa:** 1-2 semanas de desenvolvimento focado.

## Informações Técnicas Resumidas

Item	Detalhe
Framework	Next.js 14.2.33 (App Router)
Linguagem	TypeScript 5.2.2
Database	PostgreSQL 15+
ORM	Prisma 6.7.0
Autenticação	NextAuth 4.24.11
Estilização	Tailwind CSS 3.3.3
Deploy	Docker / Vercel
Node Version	18+
Package Manager	npm

**Relatório gerado por:** DeepAgent (Abacus.AI)  
**Data:** 23/10/2024  
**Versão do Relatório:** 1.0

## Contato

- Para questões sobre este relatório ou o projeto:
- Revisar documentação interna: `DEPLOYMENT_GUIDE.md` , `SECURITY_CHECKLIST.md`
  - Verificar logs: `/logs/` ou `docker-compose logs`
  - Health check: `GET /api/health`

**FIM DO RELATÓRIO**