🏗️ Arquitectura del Proyecto

Este documento detalla las decisiones de arquitectura, patrones de diseño y mejores prácticas implementadas en la aplicación de Finanzas Personales.

📋 Tabla de Contenidos

🎯 Visión General

La aplicación sigue una **arquitectura modular en capas** con separación clara de responsabilidades. Esta arquitectura permite:

- **Mantenibilidad**: Código organizado y fácil de mantener
- **Testabilidad**: Componentes aislados y fáciles de testear
- **Escalabilidad**: Fácil agregar nuevas funcionalidades
- **Reutilización**: Componentes y lógica reutilizable
- **Claridad**: Flujo de datos predecible

📚 Arquitectura por Capas

1. Capa de Presentación (UI Layer)

Responsabilidad: Renderizar la interfaz de usuario y manejar interacciones.

```
components/
├── ui/                  # Componentes base reutilizables
├── dashboard/           # Componentes específicos del dashboard
├── transactions/        # Componentes de transacciones
├── budgets/            # Componentes de presupuestos
└── auth-layout.tsx     # Layouts compartidos
```

Características:
- Componentes React funcionales
- Tipado estricto con TypeScript
- Props interfaces bien definidas

- Componentes presentacionales puros
- Uso de Radix UI para accesibilidad

Ejemplo:

```tsx
interface TransactionCardProps {
  transaction: Transaction;
  onEdit?: (id: string) => void;
  onDelete?: (id: string) => void;
}

export function TransactionCard({
  transaction,
  onEdit,
  onDelete
}: TransactionCardProps) {
  // Lógica de renderizado pura
  return (
    // JSX
  );
}
```

2. Capa de Lógica de Negocio (Business Logic Layer)

Responsabilidad: Encapsular la lógica de negocio reutilizable.

```
hooks/
├── use-toast.ts        # Notificaciones
├── use-confetti.ts     # Animaciones
└── use-transactions.ts # Lógica de transacciones

lib/
├── utils.ts            # Funciones utilitarias
├── types.ts            # Tipos compartidos
└── i18n/               # Internacionalización
```

Características:
- Custom Hooks para lógica reutilizable
- Pure functions para cálculos
- Validación de datos con Zod
- Gestión de estado con Context API

Ejemplo:

```tsx
export function useTransactions() {
  const [transactions, setTransactions] = useState<Transaction[]>([]);
  const [loading, setLoading] = useState(true);

  const fetchTransactions = async () => {
    // Lógica de fetch
  };

  const createTransaction = async (data: TransactionInput) => {
    // Lógica de creación
  };

  return { transactions, loading, fetchTransactions, createTransaction };
}
```

3. Capa de API (API Layer)

Responsabilidad: Manejar requests HTTP y comunicación con la base de datos.

```
app/api/
├── auth/              # Autenticación
├── transactions/      # CRUD de transacciones
├── budgets/          # CRUD de presupuestos
├── accounts/         # CRUD de cuentas
├── categories/       # Gestión de categorías
├── analysis/         # Análisis con IA
└── settings/         # Configuración de usuario
```

Características:
- API Routes de Next.js
- Validación de inputs
- Rate limiting
- Error handling consistente
- Logging y auditoría

Ejemplo:

```
export async function POST(req: Request) {
  try {
    // 1. Validar autenticación
    const session = await getServerSession(authOptions);
    if (!session?.user?.id) {
      return NextResponse.json(
        { error: "No autorizado" },
        { status: 401 }
      );
    }

    // 2. Validar input
    const body = await req.json();
    const validatedData = transactionSchema.parse(body);

    // 3. Rate limiting
    const rateLimitResult = await checkRateLimit(session.user.id,
"create_transaction");
    if (!rateLimitResult.success) {
      return NextResponse.json(
        { error: "Demasiadas peticiones" },
        { status: 429 }
      );
    }

    // 4. Lógica de negocio
    const transaction = await prisma.transaction.create({
      data: {
        ...validatedData,
        userId: session.user.id,
      },
    });

    // 5. Auditoría
    await logAudit({
      userId: session.user.id,
      action: "CREATE_TRANSACTION",
      details: { transactionId: transaction.id },
    });

    return NextResponse.json(transaction);
  } catch (error) {
    return handleApiError(error);
  }
}
```

4. Capa de Datos (Data Layer)

Responsabilidad: Abstracción de acceso a datos.

```
lib/
└── db.ts              # Cliente de Prisma

prisma/
├── schema.prisma      # Esquema de la base de datos
└── migrations/        # Migraciones
```

Características:

- Prisma ORM para type-safety

- Migraciones versionadas
- Seeds para datos de prueba
- Relaciones bien definidas

Ejemplo:

```
model User {
  id            String         @id @default(uuid())
  email         String         @unique
  password      String
  name          String?
  createdAt     DateTime       @default(now())
  updatedAt     DateTime       @updatedAt

  transactions  Transaction[]
  budgets       Budget[]
  accounts      Account[]
  categories    Category[]
  settings      UserSettings?
  auditLogs     AuditLog[]
}

model Transaction {
  id          String    @id @default(uuid())
  type        String    // "INCOME" | "EXPENSE"
  amount      Float
  description String
  date        DateTime
  createdAt   DateTime  @default(now())
  updatedAt   DateTime  @updatedAt

  userId      String
  user        User      @relation(fields: [userId], references: [id], onDelete: Cas-
cade)

  categoryId  String
  category    Category  @relation(fields: [categoryId], references: [id])

  accountId   String
  account     Account   @relation(fields: [accountId], references: [id])
}
```

🎨 Patrones de Diseño

1. Container/Presenter Pattern

Separación entre componentes con lógica (containers) y componentes de presentación (presenters).

Container Component:

```tsx
// containers/DashboardContainer.tsx
export function DashboardContainer() {
  const { transactions, loading } = useTransactions();
  const { budgets } = useBudgets();
  const { accounts } = useAccounts();

  if (loading) return <DashboardSkeleton />;

  return (
    <DashboardPresenter
      transactions={transactions}
      budgets={budgets}
      accounts={accounts}
    />
  );
}
```

Presenter Component:

```tsx
// components/DashboardPresenter.tsx
interface DashboardPresenterProps {
  transactions: Transaction[];
  budgets: Budget[];
  accounts: Account[];
}

export function DashboardPresenter({
  transactions,
  budgets,
  accounts,
}: DashboardPresenterProps) {
  return (
    <div>
      {/* Renderizado puro sin lógica */}
    </div>
  );
}
```

2. Custom Hooks Pattern

Encapsulación de lógica reutilizable en hooks personalizados.

```tsx
// hooks/use-transactions.ts
export function useTransactions(filters?: TransactionFilters) {
  const [transactions, setTransactions] = useState<Transaction[]>([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState<Error | null>(null);

  useEffect(() => {
    fetchTransactions();
  }, [filters]);

  const fetchTransactions = async () => {
    try {
      setLoading(true);
      const response = await fetch('/api/transactions');
      const data = await response.json();
      setTransactions(data);
    } catch (err) {
      setError(err as Error);
    } finally {
      setLoading(false);
    }
  };

  return { transactions, loading, error, refetch: fetchTransactions };
}
```

3. Context Provider Pattern

Gestión de estado global con Context API.

```tsx
// lib/i18n/context.tsx
interface LanguageContextType {
  language: Language;
  currency: Currency;
  setLanguage: (lang: Language) => void;
  setCurrency: (curr: Currency) => void;
  t: (key: string) => string;
}

const LanguageContext = createContext<LanguageContextType | undefined>(undefined);

export function LanguageProvider({ children }: { children: React.ReactNode }) {
  const [language, setLanguage] = useState<Language>('es');
  const [currency, setCurrency] = useState<Currency>('EUR');

  const t = useCallback((key: string) => {
    return translations[language][key] || key;
  }, [language]);

  return (
    <LanguageContext.Provider value={{ language, currency, setLanguage, setCurrency,
t }}>
      {children}
    </LanguageContext.Provider>
  );
}

export const useLanguage = () => {
  const context = useContext(LanguageContext);
  if (!context) throw new Error('useLanguage must be used within LanguageProvider');
  return context;
};
```

4. Repository Pattern

Abstracción de acceso a datos.

```typescript
// lib/repositories/transaction-repository.ts
export class TransactionRepository {
  async findAll(userId: string, filters?: TransactionFilters) {
    return prisma.transaction.findMany({
      where: {
        userId,
        ...this.buildWhereClause(filters),
      },
      include: {
        category: true,
        account: true,
      },
      orderBy: { date: 'desc' },
    });
  }

  async create(userId: string, data: CreateTransactionInput) {
    return prisma.transaction.create({
      data: {
        ...data,
        userId,
      },
    });
  }

  private buildWhereClause(filters?: TransactionFilters) {
    // Lógica de construcción de filtros
  }
}
```

5. Middleware Pattern

Procesamiento de requests en capas.

```
// lib/security/middleware.ts
export function withAuth(handler: ApiHandler) {
  return async (req: Request, context: any) => {
    const session = await getServerSession(authOptions);

    if (!session) {
      return NextResponse.json(
        { error: 'No autorizado' },
        { status: 401 }
      );
    }

    return handler(req, context, session);
  };
}

export function withRateLimit(limit: number, window: number) {
  return function (handler: ApiHandler) {
    return async (req: Request, context: any) => {
      const session = await getServerSession(authOptions);
      const result = await checkRateLimit(session.user.id, req.url, limit, window);

      if (!result.success) {
        return NextResponse.json(
          { error: 'Demasiadas peticiones' },
          { status: 429 }
        );
      }

      return handler(req, context);
    };
  };
}

// Uso:
export const POST = withAuth(
  withRateLimit(10, 60)(
    async (req, context, session) => {
      // Lógica del handler
    }
  )
);
```

🔧 Principios SOLID

Single Responsibility Principle (SRP)

Cada módulo debe tener una única razón para cambiar.

✅ **Bueno**:

```tsx
// Componente con una única responsabilidad
export function TransactionList({ transactions }: TransactionListProps) {
  return (
    <ul>
      {transactions.map(transaction => (
        <TransactionItem key={transaction.id} transaction={transaction} />
      ))}
    </ul>
  );
}

// Hook con una única responsabilidad
export function useTransactionFilters() {
  const [filters, setFilters] = useState<TransactionFilters>({});

  const updateFilter = (key: string, value: any) => {
    setFilters(prev => ({ ...prev, [key]: value }));
  };

  return { filters, updateFilter };
}
```

❌ **Malo**:

```tsx
// Componente con múltiples responsabilidades
export function TransactionPage() {
  // Lógica de fetch
  const [transactions, setTransactions] = useState([]);

  // Lógica de filtrado
  const [filters, setFilters] = useState({});

  // Lógica de paginación
  const [page, setPage] = useState(1);

  // Lógica de ordenamiento
  const [sortBy, setSortBy] = useState('date');

  // Renderizado
  return (
    // JSX complejo
  );
}
```

Open/Closed Principle (OCP)

Los módulos deben estar abiertos para extensión pero cerrados para modificación.

✅ **Bueno**:

```tsx
// Base component extensible
interface BaseCardProps {
  title: string;
  children: React.ReactNode;
  actions?: React.ReactNode;
}

export function BaseCard({ title, children, actions }: BaseCardProps) {
  return (
    <div className="card">
      <div className="card-header">
        <h3>{title}</h3>
        {actions}
      </div>
      <div className="card-content">{children}</div>
    </div>
  );
}

// Extensión sin modificar el componente base
export function TransactionCard({ transaction }: { transaction: Transaction }) {
  return (
    <BaseCard
      title={transaction.description}
      actions={<TransactionActions transaction={transaction} />}
    >
      <TransactionDetails transaction={transaction} />
    </BaseCard>
  );
}
```

Liskov Substitution Principle (LSP)

Los objetos de una clase derivada deben poder reemplazar objetos de la clase base.

✅ **Bueno**:

```tsx
interface ChartProps {
  data: any[];
  width?: number;
  height?: number;
}

export function BarChart({ data, width = 400, height = 300 }: ChartProps) {
  // Implementación
}

export function LineChart({ data, width = 400, height = 300 }: ChartProps) {
  // Implementación
}

// Ambos pueden usarse intercambiablemente
function DashboardChart({ type, data }: { type: 'bar' | 'line', data: any[] }) {
  const Chart = type === 'bar' ? BarChart : LineChart;
  return <Chart data={data} />;
}
```

Interface Segregation Principle (ISP)

Los clientes no deben depender de interfaces que no usan.

✅ **Bueno**:

```
// Interfaces específicas
interface Readable {
  read(): Promise<Transaction[]>;
}

interface Writable {
  create(data: TransactionInput): Promise<Transaction>;
  update(id: string, data: TransactionInput): Promise<Transaction>;
}

interface Deletable {
  delete(id: string): Promise<void>;
}

// Implementaciones específicas
class TransactionReader implements Readable {
  async read() { /* ... */ }
}

class TransactionWriter implements Writable {
  async create(data: TransactionInput) { /* ... */ }
  async update(id: string, data: TransactionInput) { /* ... */ }
}
```

Dependency Inversion Principle (DIP)

Depender de abstracciones, no de concreciones.

✅ **Bueno**:

```typescript
// Abstracción
interface IStorageService {
  save(key: string, value: any): Promise<void>;
  load(key: string): Promise<any>;
}

// Implementaciones concretas
class LocalStorageService implements IStorageService {
  async save(key: string, value: any) {
    localStorage.setItem(key, JSON.stringify(value));
  }

  async load(key: string) {
    const item = localStorage.getItem(key);
    return item ? JSON.parse(item) : null;
  }
}

class ApiStorageService implements IStorageService {
  async save(key: string, value: any) {
    await fetch('/api/storage', {
      method: 'POST',
      body: JSON.stringify({ key, value }),
    });
  }

  async load(key: string) {
    const response = await fetch(`/api/storage/${key}`);
    return response.json();
  }
}

// Uso - depende de la abstracción
function useStorage(storage: IStorageService) {
  const save = async (key: string, value: any) => {
    await storage.save(key, value);
  };

  const load = async (key: string) => {
    return await storage.load(key);
  };

  return { save, load };
}
```

📊 Gestión del Estado

Niveles de Estado

1. **Estado Local** (useState, useReducer)
   - Datos específicos de un componente
   - No necesitan compartirse

2. **Estado Compartido** (Context API)
   - Idioma y moneda
   - Tema (claro/oscuro)
   - Sesión de usuario

3. **Estado del Servidor** (SWR, React Query)
   - Transacciones
   - Presupuestos
   - Cuentas

Ejemplo de Gestión de Estado

```
// Estado local
function TransactionForm() {
  const [formData, setFormData] = useState<TransactionInput>({
    type: 'EXPENSE',
    amount: 0,
    description: '',
  });

  return (
    // Formulario
  );
}

// Estado compartido (Context)
function App() {
  return (
    <LanguageProvider>
      <ThemeProvider>
        <SessionProvider>
          {children}
        </SessionProvider>
      </ThemeProvider>
    </LanguageProvider>
  );
}

// Estado del servidor (SWR)
function TransactionList() {
  const { data: transactions, error, isLoading } = useSWR(
    '/api/transactions',
    fetcher
  );

  if (isLoading) return <Skeleton />;
  if (error) return <Error />;

  return <List items={transactions} />;
}
```

🔒 Seguridad

Ver SECURITY.md (./nextjs_space/SECURITY.md) para detalles completos.

Capas de Seguridad

1. **Autenticación**
   - NextAuth.js con JWT
   - Sesiones seguras
   - Hashing con bcrypt

2. **Autorización**
   - Verificación en cada request
   - Ownership de recursos
   - Roles y permisos

3. **Validación**
   - Zod schemas en cliente y servidor
   - Sanitización de inputs
   - Type safety con TypeScript

4. **Rate Limiting**
   - Límites por endpoint
   - Protección contra brute force
   - Throttling de requests

5. **Auditoría**
   - Logging de acciones críticas
   - Tracking de cambios
   - Monitoreo de seguridad

⚡ Performance

Optimizaciones Implementadas

1. **Server Components por Defecto**
   ```typescript
   // app/dashboard/page.tsx
   export default async function DashboardPage() {
     const data = await fetchDashboardData();
     return <Dashboard data={data} />;
   }
   ```

2. **Client Components Solo Cuando Necesario**
   ```typescript
   'use client';

export function InteractiveChart({ data }: ChartProps) {
// Componente con interactividad
}
```

1. **Lazy Loading**
   ```typescript
   const AnalysisModal = lazy(() => import('./AnalysisModal'));
   ```

2. **Memoización**
   ```typescript
   const expensiveCalculation = useMemo(() => {
     return calculateTotals(transactions);
   }, [transactions]);
   ```

3. **Debouncing**
   ```
   typescript
   const debouncedSearch = useDebounce(searchTerm, 300);
   ```

📈 Escalabilidad

Preparado para Crecer

1. **Arquitectura Modular**
   - Fácil agregar nuevas features
   - Componentes reutilizables
   - Código desacoplado

2. **Base de Datos**
   - Índices en campos frecuentes
   - Queries optimizadas
   - Paginación en listados

3. **API**
   - Rate limiting
   - Caching
   - Versionado de endpoints

4. **Frontend**
   - Code splitting
   - Lazy loading
   - Server components

📚 Recursos Adicionales

- Next.js Documentation (https://nextjs.org/docs)
- React Best Practices (https://react.dev/learn)
- TypeScript Handbook (https://www.typescriptlang.org/docs/)
- Prisma Guides (https://www.prisma.io/docs)
- SOLID Principles (https://en.wikipedia.org/wiki/SOLID)

Este documento es un trabajo en progreso y se actualizará a medida que evolucione la aplicación.