RendiFlow - Documentación de Arquitectura

□ Índice

□ Resumen Ejecutivo

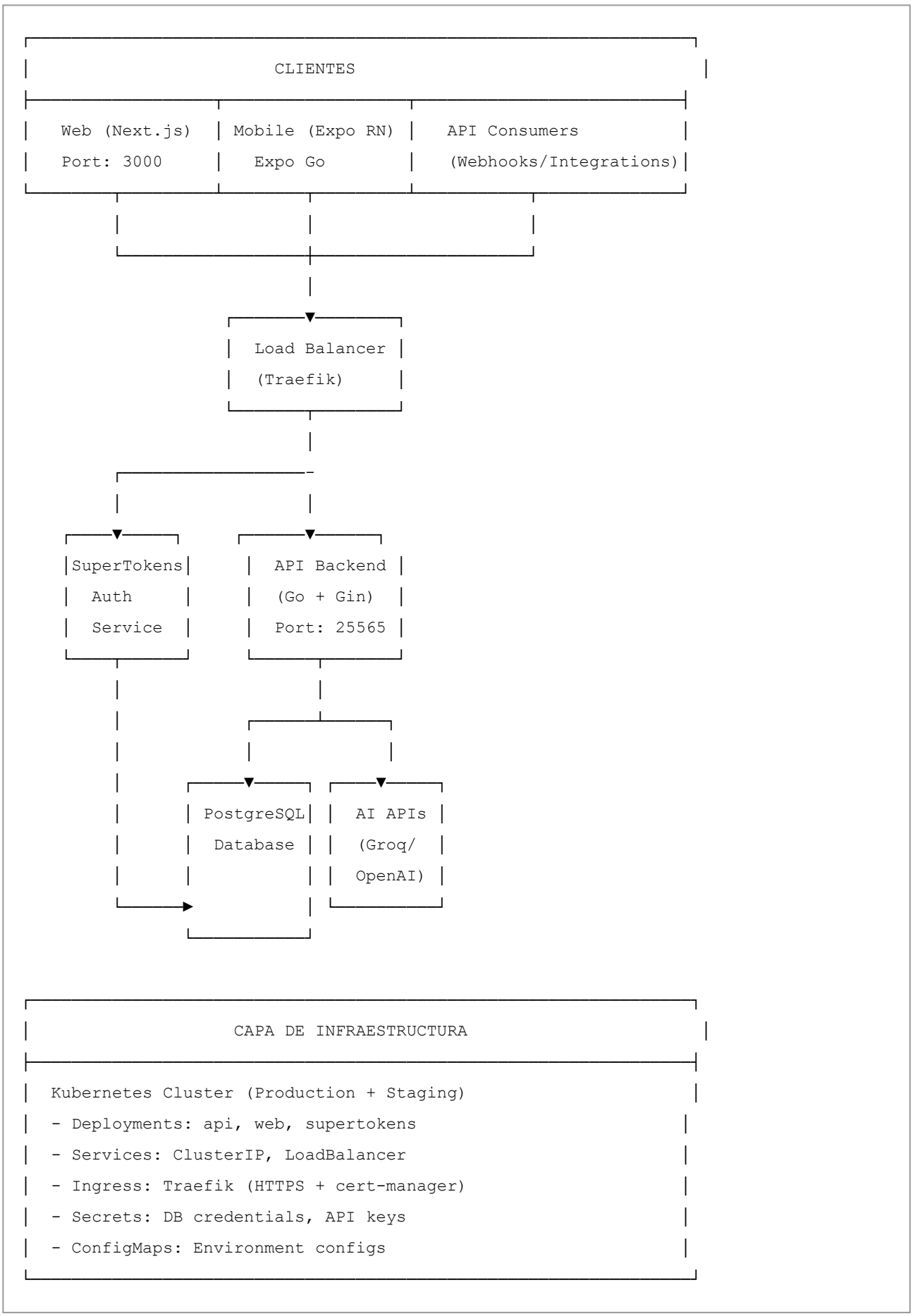
RendiFlow es una plataforma SaaS multi-tenant para la gestión automatizada de boletas y gastos empresariales en Chile, con capacidad de expansión internacional. El sistema utiliza **OCR con IA** (Vision AI) para extraer datos de recibos fiscales chilenos (SII) y documentos financieros globales.

Propuesta de Valor

- **Automatización**: OCR inteligente elimina entrada manual de datos
- **Multi-tenant**: Arquitectura que soporta múltiples empresas aisladas
- **Multi-plataforma**: Web (Next.js) + Móvil (React Native/Expo)
- **Escalable**: Kubernetes + microservicios en Go
- **Segura**: Autenticación enterprise con SuperTokens

□ Arquitectura del Sistema

Arquitectura General: **Clean Architecture + Microservicios**


CLIENTES
Web (Next.js)
Port: 3000
Mobile (Expo RN)
Expo Go
API Consumers
(Webhooks/Integrations)
Load Balancer
(Traefik)
SuperTokens
Auth
Service
API Backend
(Go + Gin)
Port: 25565
PostgreSQL
Database
AI APIs
(Groq/
OpenAI)
CAPA DE INFRAESTRUCTURA
Kubernetes Cluster (Production + Staging)
- Deployments: api, web, supertokens
- Services: ClusterIP, LoadBalancer
- Ingress: Traefik (HTTPS + cert-manager)
- Secrets: DB credentials, API keys
- ConfigMaps: Environment configs

Stack Tecnológico

Backend (API)

Tecnología	Propósito	Versión
Go	Lenguaje principal	1.24
Gin	Framework HTTP	1.10.1
Huma v2	OpenAPI + validación automática	2.34.1
Bun	ORM + Query Builder	1.2.15
PostgreSQL	Base de datos relacional	-
Uber Fx	Dependency Injection	1.24.0
SuperTokens	Autenticación y sesiones	0.25.1
Zap	Logging estructurado	1.27.0
LangChainGo	Integración con LLMs	0.1.13

Frontend Web

Tecnología	Propósito	Versión
Next.js	Framework React SSR	14.x
React	UI Library	18.x
TypeScript	Type safety	5.x
TailwindCSS	Estilos utility-first	3.x
HeroUI	Componentes UI	-
SuperTokens React	Cliente de autenticación	-

Frontend Mobile

Tecnología	Propósito	Versión
Expo	Framework React Native	53.0.20
React Native	Framework móvil multiplataforma	0.79.5
React	UI Library	19.0.0
TypeScript	Type safety	5.8.3
NativeWind	TailwindCSS para RN	4.1.23
Gluestack UI	Sistema de componentes	3.0.2
Expo Router	Navegación basada en archivos	5.1.4
SuperTokens RN	Cliente de autenticación	5.1.5

Infraestructura

Tecnología	Propósito
Kubernetes	Orquestación de contenedores
Docker	Containerización
Traefik	Ingress Controller + Load Balancer
cert-manager	Gestión automática de certificados SSL

Tecnología	Propósito
GitHub Actions	CI/CD pipelines
GitHub Container Registry	Registry de imágenes Docker

Servicios Externos

- **Groq / OpenAI**: APIs de Vision AI para OCR
- **SuperTokens Cloud**: Gestión de autenticación
- **PostgreSQL Cloud**: Base de datos managed

□ Componentes Principales

1. Backend API (Go)

Arquitectura en Capas (Clean Architecture)

```
api/
├── cmd/
│   ├── main.go              # Entry point
│   ├── app/
│   │   ├── app.go           # Fx application setup
│   │   ├── configuration.go # Config loading
│   │   ├── router.go        # HTTP routing (Huma + Gin)
│   │   └── health.go        # Health checks
│   ├── handler/             # ← CAPA DE PRESENTACIÓN
│   │   ├── auth.go          # Auth endpoints
│   │   ├── user.go          # User profile
│   │   ├── tenant.go        # Multi-tenancy
│   │   ├── receipts.go      # Receipt management
│   │   └── membership.go    # RBAC & memberships
│   └── migrate/
│       └── main.go          # DB migrations CLI
├── internal/
│   ├── entity/              # ← CAPA DE DOMINIO
│   │   ├── user/
│   │   │   ├── model.go     # User entity
│   │   │   └── repository.go # User repo interface
│   │   ├── company/         # Tenant entity
│   │   ├── receipt/         # Receipt entity + items
│   │   ├── membership/      # User-Tenant relationships
│   │   └── models.go        # Shared types
│   ├── database/            # ← CAPA DE INFRAESTRUCTURA
│   │   ├── database.go      # DB interface
│   │   ├── postgres.go      # PostgreSQL implementation
│   │   └── migrations.go    # Schema migrations
│   ├── groq/                # ← SERVICIOS EXTERNOS
│   │   ├── client.go        # AI client wrapper
│   │   └── image.go         # OCR processing
│   ├── identity/            # ← DOMINIO: AUTENTICACIÓN
│   │   ├── service.go       # SuperTokens integration
│   │   └── protect.go       # Middleware & guards
│   └── lib/                 # ← UTILIDADES
│       ├── request/         # Request context
│       ├── security/        # Crypto utilities
│       ├── metrics/         # Observability
│       └── helper/          # Helpers
└── docs/
    └── MIGRATIONS.md
```

Patrones Implementados

1. **Repository Pattern**: Abstracción de acceso a datos
    - Interfaces en `entity/*/repository.go`
    - Implementaciones concretas con `RepositoryWithDB`
2. **Dependency Injection**: Uber Fx

```
fx.Provide(
    NewDatabase,
    NewUserProvider,
    NewAuthService,
    handler.NewUserHandler,
)
```

3. **Middleware Chain**:
    - CORS
    - Request logging
    - SuperTokens session verification
    - Request context injection
    - Metrics collection
4. **Domain-Driven Design (DDD)**:
    - Entities con lógica de negocio
    - Repositories para persistencia
    - Services para casos de uso complejos

---

2. Frontend Web (Next.js)

```
web/
├── app/                    # Next.js App Router
│   ├── layout.tsx          # Root layout
│   ├── page.tsx            # Landing page
│   ├── globals.css         # Estilos globales
│   ├── auth/
│   │   └── [[...path]]/    # SuperTokens auth pages
│   ├── dashboard/          # Dashboard principal
│   ├── components/         # Componentes React
│   │   ├── AppRoot.tsx     # App wrapper
│   │   ├── sessionAuthForNextJS.tsx
│   │   └── supertokensProvider.tsx
│   └── config/             # Configuración de SuperTokens
├── public/                 # Assets estáticos
├── next.config.js          # Configuración Next.js
└── tsconfig.json           # TypeScript config
```

Características:

- **Server-Side Rendering (SSR)**: Para SEO y performance
- **Static Generation**: Para páginas públicas
- **API Routes**: Proxy a backend API
- **Standalone build**: Para deployment en contenedores

---

3. Frontend Mobile (Expo)

```
web_mobile/
├── app/                      # Expo Router (file-based)
│   ├── _layout.tsx           # Root layout con providers
│   ├── index.tsx             # Landing/redirect
│   ├── (auth)/               # Auth stack (no protegido)
│   │   ├── _layout.tsx
│   │   ├── sign-in.tsx
│   │   └── sign-up.tsx
│   ├── (tabs)/               # Tab navigation (protegido)
│   │   ├── _layout.tsx       # Tabs config
│   │   ├── home/index.tsx    # Dashboard
│   │   ├── upload/index.tsx # Camera/upload
│   │   ├── support/index.tsx
│   │   └── profile/index.tsx
│   ├── history/              # Screens fuera de tabs
│   ├── receipt/
│   └── reports/
├── components/
│   ├── ui/                   # Gluestack UI wrappers
│   │   ├── box/
│   │   ├── button/
│   │   ├── input/
│   │   ├── text/
│   │   └── ...
│   └── home/                 # Feature components
├── hooks/
│   └── useAuth.ts            # Custom auth hook
├── providers/
│   └── AuthProvider.tsx      # Context provider
├── lib/
│   ├── api.ts                # Axios client
│   ├── auth/
│   │   └── supertokens.ts    # SuperTokens init
│   └── ocr.ts                # OCR helpers
└── assets/
```

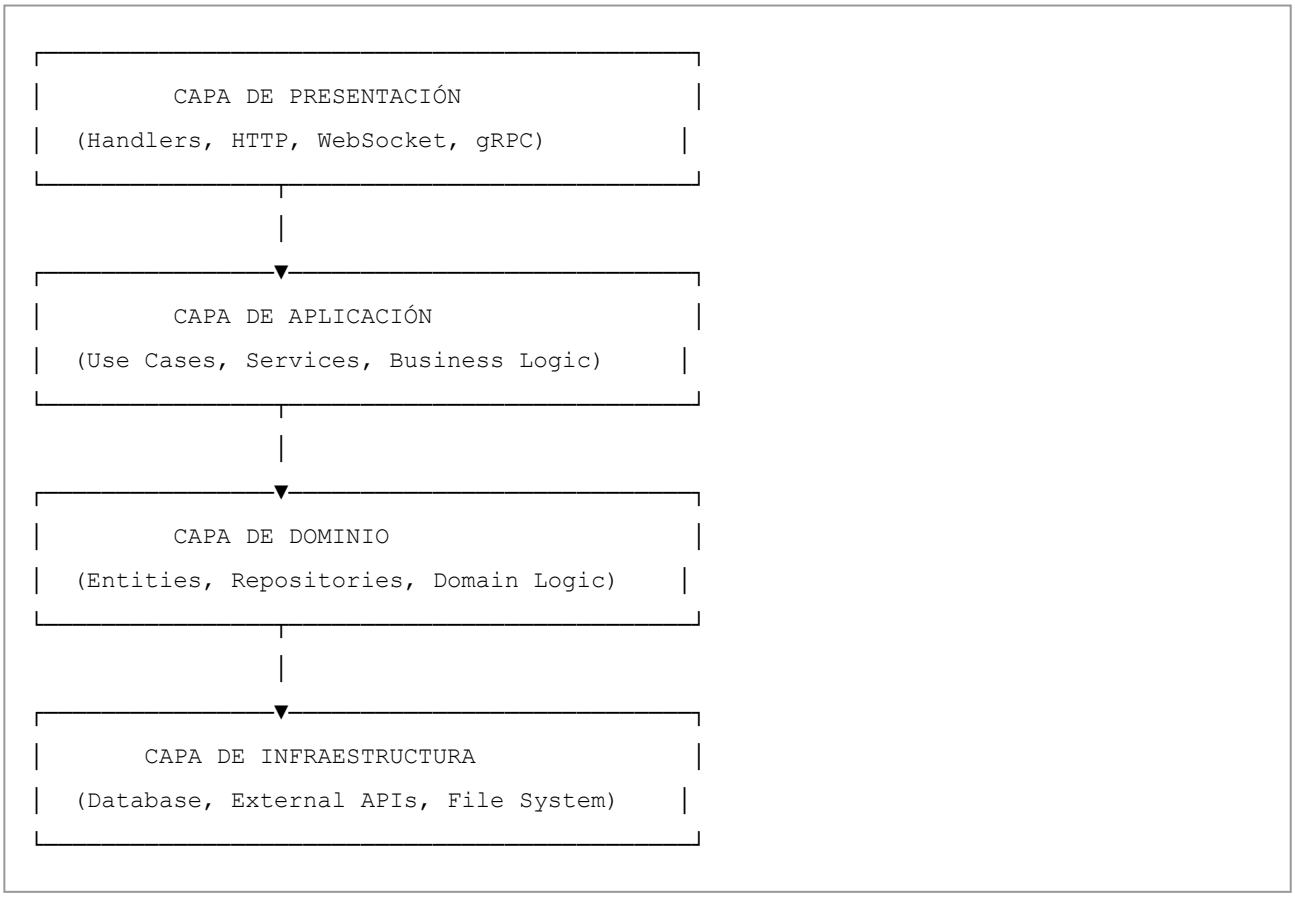
Características:

- **File-based routing**: Expo Router v5
- **Protected routes**: Auth guards en `(tabs)/_layout.tsx`
- **Context API**: Estado global con `AuthProvider`
- **Cross-platform**: iOS, Android, Web
- **NativeWind**: TailwindCSS con React Native

□ Patrones de Diseño

Backend Patterns

1. **Clean Architecture (Arquitectura Hexagonal)**



Ventajas:

- □ Testeable: Cada capa se puede testear independientemente
- □ Mantenible: Cambios en una capa no afectan otras
- □ Escalable: Fácil agregar nuevas funcionalidades
- □ Desacoplado: Bajo acoplamiento entre capas

2. **Repository Pattern**

```
// Interface (domain layer)
type Repository interface {
    Create(ctx *request.Ctx, user *User) (*User, error)
    FindByID(ctx *request.Ctx, id uuid.UUID) (*User, error)
    Update(ctx *request.Ctx, user *User, data UpdateStruct) error
    Delete(ctx *request.Ctx, user *User) error
}

// Implementation (infrastructure layer)
type RepositoryWithDB struct {
    db database.Database
}
```

Ventajas:

- □ Abstracción del almacenamiento de datos
- □ Fácil cambiar de PostgreSQL a MongoDB sin afectar lógica de negocio
- □ Testeable con mocks

3. **Dependency Injection (Uber Fx)**

```
fx.New(
    fx.Provide(
        NewDatabase,           // Inyecta dependencias
        NewUserProvider,       // en el orden correcto
        NewAuthService,
        handler.NewUserHandler,
    ),
    fx.Invoke(StartServer),    // Ejecuta al iniciar
)
```

Ventajas:

- □ Acoplamiento bajo
- □ Ciclo de vida automático
- □ Testeable

4. **Middleware Chain Pattern**

```
// Cadena de middlewares
ginApp.Use(gin.Recovery())
ginApp.Use(cors.New(corsConfig))
ginApp.Use(identity.GinMiddleware(ginApp))
ginApp.Use(MetricsGinMiddleware(instrumenter))

// En Huma
api.UseMiddleware(request.InjectRequestCtxMiddleware)
api.UseMiddleware(ErrorLoggingMiddleware)
api.UseMiddleware(identity.VerifySessionMiddleware(nil))
```

5. **Service Layer Pattern**

```
type IdentityService interface {
    CreateUser(email, password string) (*User, error)
    VerifyEmail(token string) error
    ResetPassword(token, newPassword string) error
}
```

Frontend Patterns

1. **Component Composition (React)**

```
// Atomic Design
<Button onPress={submit} isDisabled={!canSubmit}>
  <ButtonText>Entrar</ButtonText>
</Button>

<Input variant="outline" size="md">
  <InputField placeholder="Email" />
</Input>
```

2. **Context API + Custom Hooks**

```
// Provider
export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);
  // ...
  return <AuthContext.Provider value={...}>{children}</AuthContext.Provider>
}

// Hook
export const useAuth = () => useContext(AuthContext);

// Uso
const { user, loginEmailPassword, logout } = useAuth();
```

3. **Protected Routes Pattern**

```
// web_mobile/app/(tabs)/_layout.tsx
export default function TabsLayout() {
  const { ready, user } = useAuth();
  const router = useRouter();

  useEffect(() => {
    if (!ready) return;
    if (!user) {
      router.replace("/(auth)/sign-in");
    }
  }, [ready, user, router]);

  if (!ready) return <LoadingScreen />;
  // ...
}
```

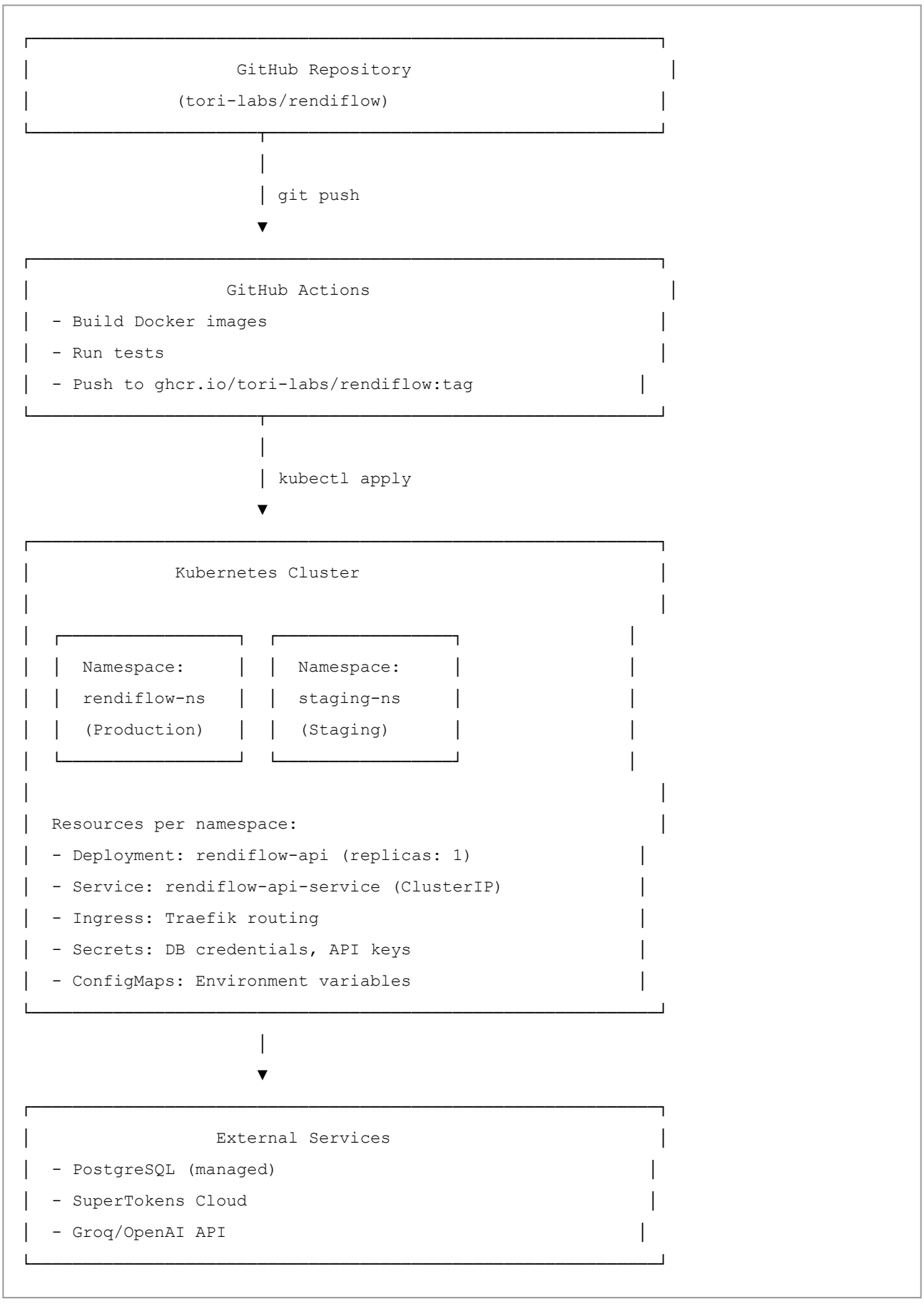
4. **API Client Pattern (Axios)**

```
// lib/api.ts
export const api = axios.create({
  baseURL: `${API_DOMAIN}${API_BASE_PATH}`,
  withCredentials: true,
  headers: { 'Content-Type': 'application/json' },
});

// Interceptors
api.interceptors.response.use(
  (res) => res,
  (err) => Promise.reject(normalizeError(err))
);
```

□ Infraestructura y DevOps

Arquitectura de Deployment



Configuración de Kubernetes

Deployment API (Production)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rendiflow-api
  namespace: rendiflow-ns
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rendiflow
  template:
    spec:
      containers:
        - name: app
          image: ghcr.io/tori-labs/rendiflow:api-stable
          ports:
            - containerPort: 25565
          env:
            - name: DB_DSN
              valueFrom:
                secretKeyRef:
                  name: rendiflow-vars
                  key: DB_DSN
            - name: ENVIRONMENT
              value: "production"
          resources:
            requests:
              cpu: 180m
              memory: 256Mi
            limits:
              cpu: 400m
              memory: 512Mi
          livenessProbe:
            httpGet:
              path: /v1/internal/health
              port: 25565
          readinessProbe:
            httpGet:
              path: /v1/internal/readiness
              port: 25565
```

Ingress (Traefik)

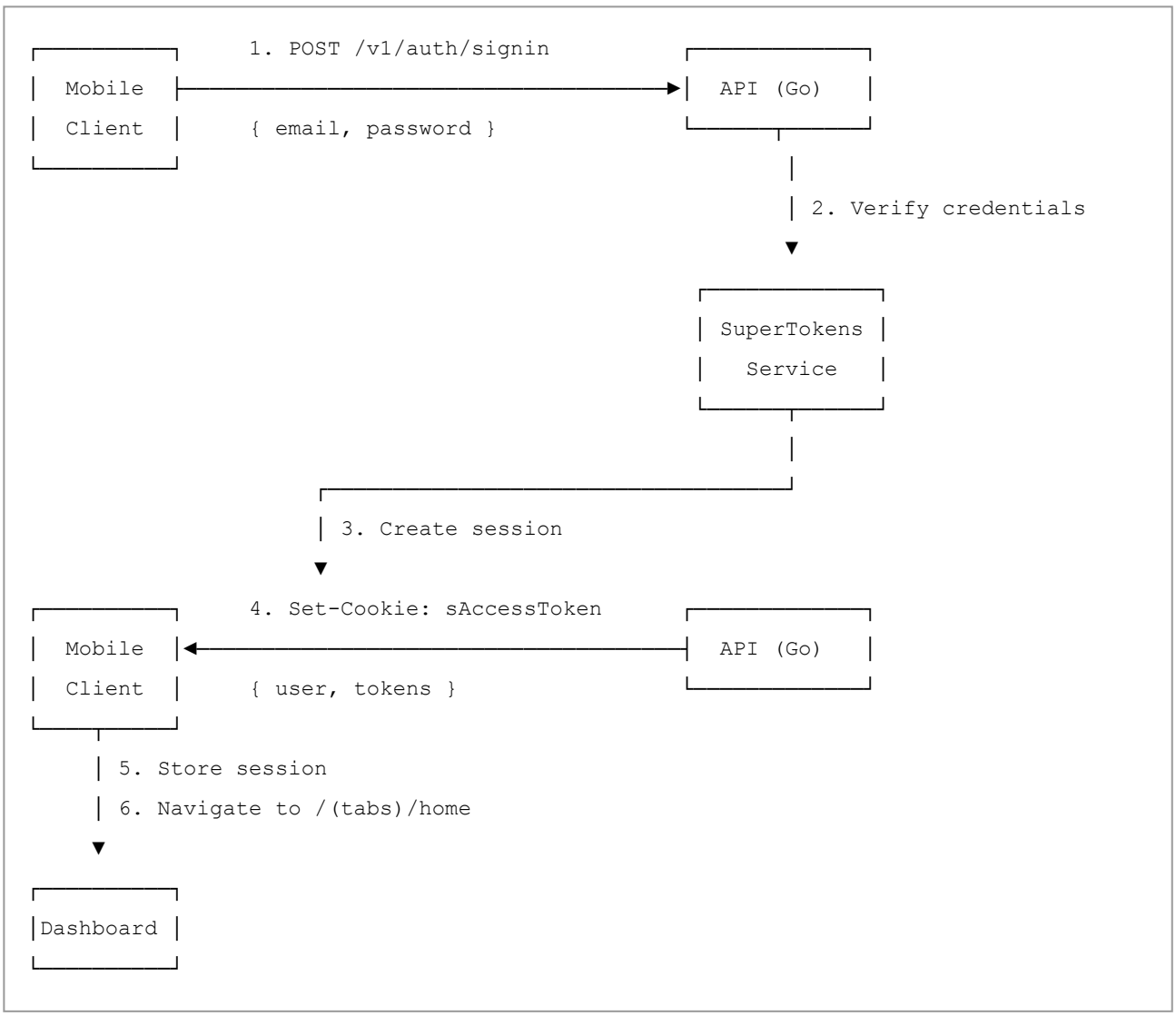
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: rendiflow-ingress
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  ingressClassName: traefik
  tls:
    - hosts:
        - api.rendiflow.com
      secretName: api-tls-cert
  rules:
    - host: api.rendiflow.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: rendiflow-api-service
                port:
                  number: 25565
```

Ambientes

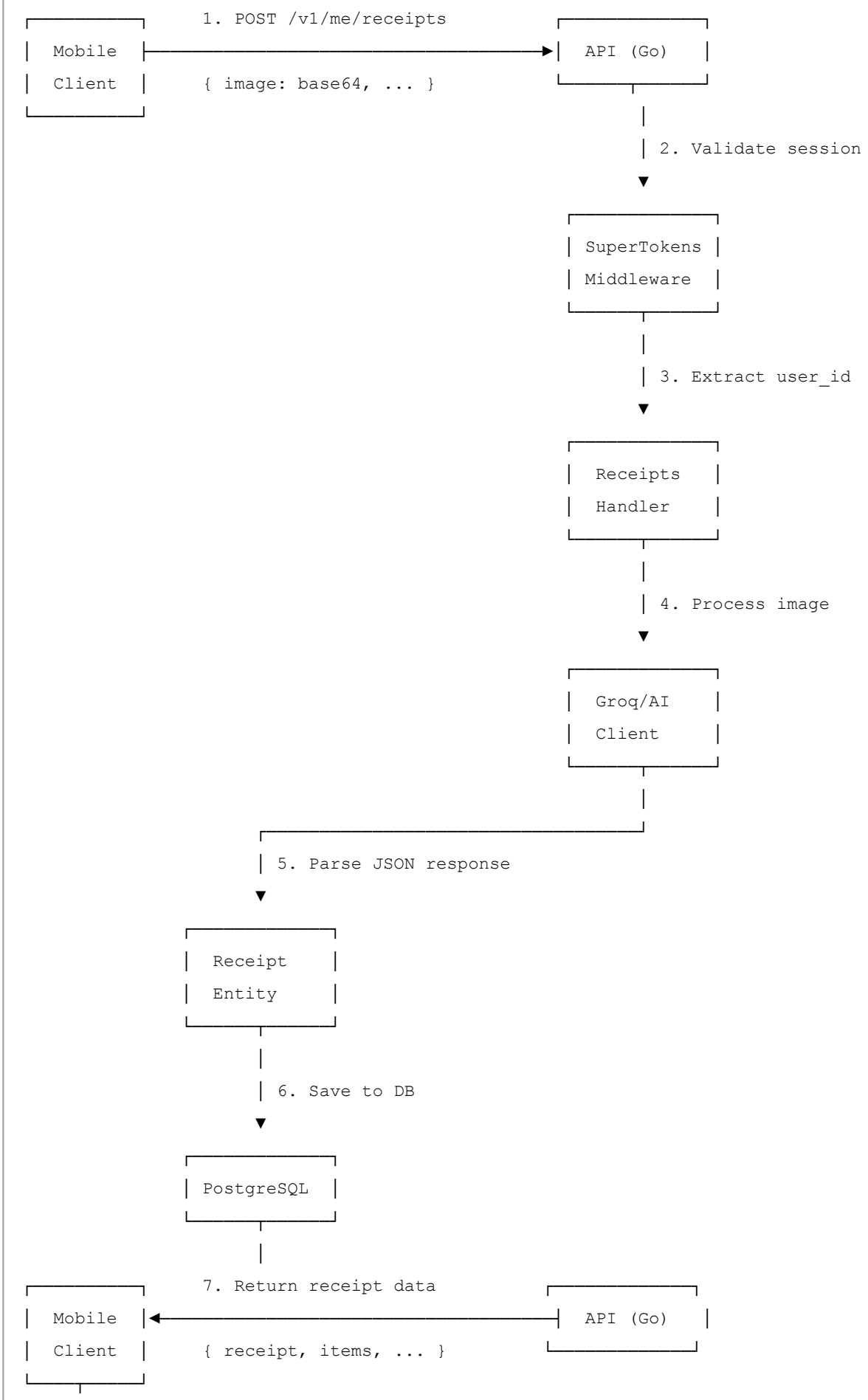
Ambiente	URL	Branch	Deployment
Production	`api.rendiflow.com`	`main`	Kubernetes (rendiflow-ns)
Staging	`staging-api.rendiflow.com`	`develop`	Kubernetes (staging-ns)
Local	`localhost:25565`	feature/*	Docker Compose

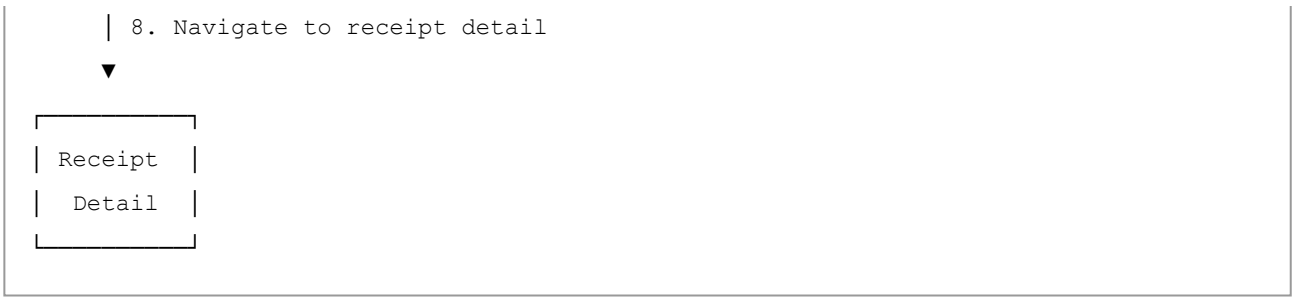
□ Flujos de Datos

1. Flujo de Autenticación (Sign In)

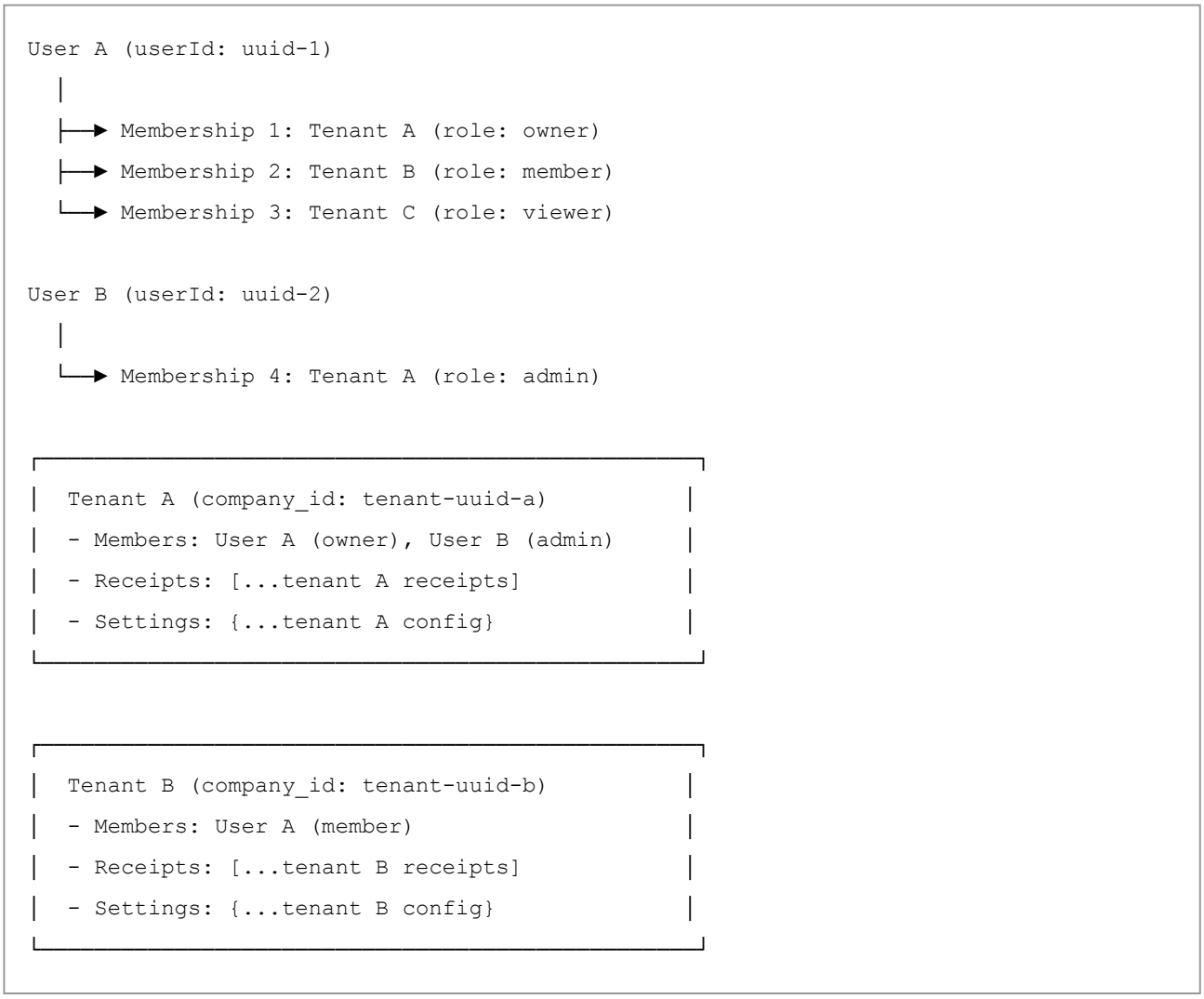


2. Flujo de Carga de Boleta con OCR


Mobile Client
1. POST /v1/me/receipts
{ image: base64, ... }
API (Go)
2. Validate session
SuperTokens Middleware
3. Extract user_id
Receipts Handler
4. Process image
Groq/AI Client
5. Parse JSON response
Receipt Entity
6. Save to DB
PostgreSQL
7. Return receipt data
{ receipt, items, ... }
Mobile Client
API (Go)



3. Flujo Multi-Tenant



Aislamiento de datos:

```
-- Todas las queries filtran por tenant
SELECT * FROM receipts
WHERE company_id = :tenant_id
AND user_id = :user_id;

-- Middleware verifica membership
func VerifyTenantAccess(tenantID uuid.UUID, userID uuid.UUID) {
    membership := FindMembership(userID, tenantID)
    if membership == nil {
        return ErrForbidden
    }
    // Check permissions based on role
}
```

□ Seguridad y Autenticación

SuperTokens Integration

Arquitectura de Autenticación:

Medidas de Seguridad

1. **Session-based Auth** (no JWT en localStorage)
    - Cookies HTTP-only
    - CSRF protection
    - Auto-refresh de tokens
2. **HTTPS Everywhere**
    - cert-manager + Let's Encrypt
    - TLS 1.2+
    - HSTS headers
3. **CORS Restrictivo**

```
if environment == Production {
    allowOrigins = []string{
        "https://rendiflow.com",
        "https://app.rendiflow.com",
    }
}
```

4. **Input Validation**
    - Huma v2: Schema validation automática
    - Sanitización en entities
5. **Database Security**
    - Prepared statements (Bun ORM)
    - Connection pooling
    - Secrets en Kubernetes
6. **Rate Limiting** (TODO)
    - Por IP
    - Por usuario
    - Por endpoint

□ Escalabilidad

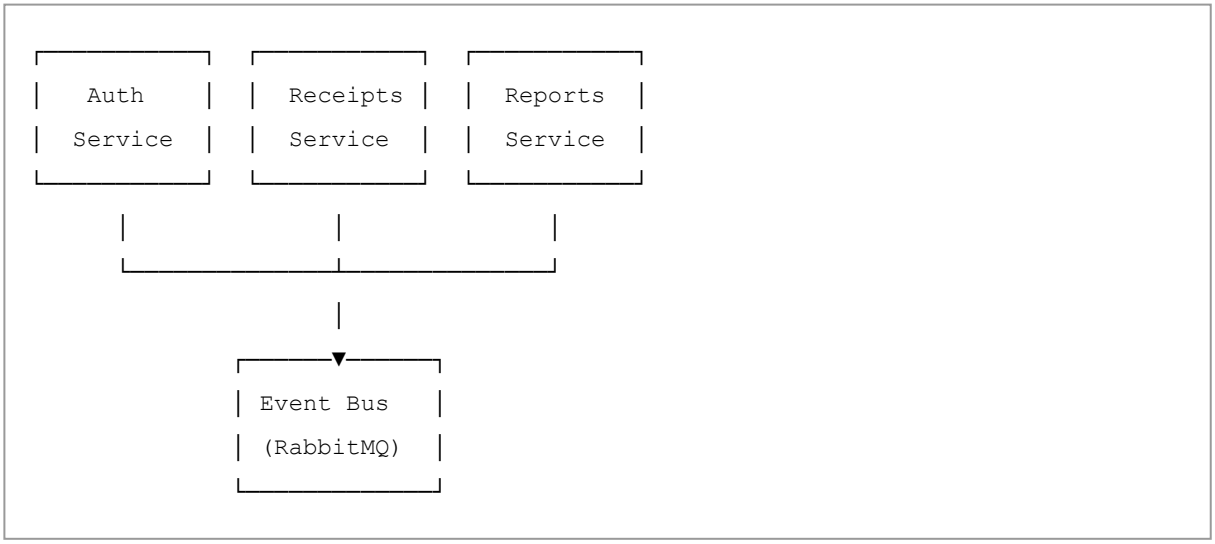
Estrategias Implementadas

1. **Horizontal Scaling**

```
# Kubernetes HPA (TODO)
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
spec:
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 70
```

2. **Database Optimization**
    - Índices en columnas frecuentes
    - Connection pooling
    - Read replicas (futuro)

3. **Caching Strategy** (TODO)
    - Redis para sesiones
    - CDN para assets estáticos
    - Query result caching
4. **Asynchronous Processing** (TODO)
    - Queue para OCR jobs
    - Background workers
    - Event-driven architecture
5. **Microservices Future**



□ Conclusiones

Fortalezas de la Arquitectura

□ **Modular**: Clean Architecture permite cambios sin afectar todo el sistema □ **Escalable**: Kubernetes + microservicios preparados para crecer □ **Segura**: SuperTokens + HTTPS + validación automática □ **Multi-plataforma**: Web + Mobile con código compartido (React) □ **Developer-friendly**: TypeScript + Go con type safety □ **Observable**: Logs estructurados + métricas + health checks □ **Testeable**: Dependency Injection + interfaces permiten mocks

---

□ Referencias

- Clean Architecture - Robert C. Martin (https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html)
- Uber Fx Documentation (https://uber-go.github.io/fx/)
- Huma v2 Docs (https://huma.rocks/)
- SuperTokens Architecture (https://supertokens.com/docs/architecture)

- Kubernetes Best Practices (https://kubernetes.io/docs/concepts/configuration/overview/)
- Expo Documentation (https://docs.expo.dev/)
- Next.js Documentation (https://nextjs.org/docs)

Última actualización: Octubre 2025

Versión: 2.0

Autor: Equipo RendiFlow (Tori Labs)