Table of Contents

SOLUCION DE ARQUITECTURA DE SISTEMAS

Nombre: Alexander Merino

Diseñar la arquitectura de integración de alto nivel

Diagrama de Contexto (C4 Nivel 1)

```
@startuml diagrama de componentes
!define RECTANGLE class
skinparam backgroundColor #EEEBDC
skinparam classBackgroundColor White
skinparam shadowing false
skinparam class {
   BackgroundColor White
   BorderColor Black
}

RECTANGLE BancoTradicional {
   :Clientes;
   :Regulaciones;
}

RECTANGLE ServiciosExternos {
   :Servicios de Pago;
   :APIs de Terceros (Open Finance);
}

RECTANGLE SistemasInternos {
   :Core Bancario Tradicional;
   :Nuevo Core Bancario Digital;
   :Banca Web;
   :Banca Móvil;
   :Sistema de Gestión de Riesgos;
   :Sistema de Prevención de Fraudes;
}

BancoTradicional --> SistemasInternos : Transacciones
SistemasInternos --> ServiciosExternos : Integración de pagos, APIs de terceros
ServiciosExternos --> BancoTradicional : Información de pago, Respuesta de APIs

@enduml
```

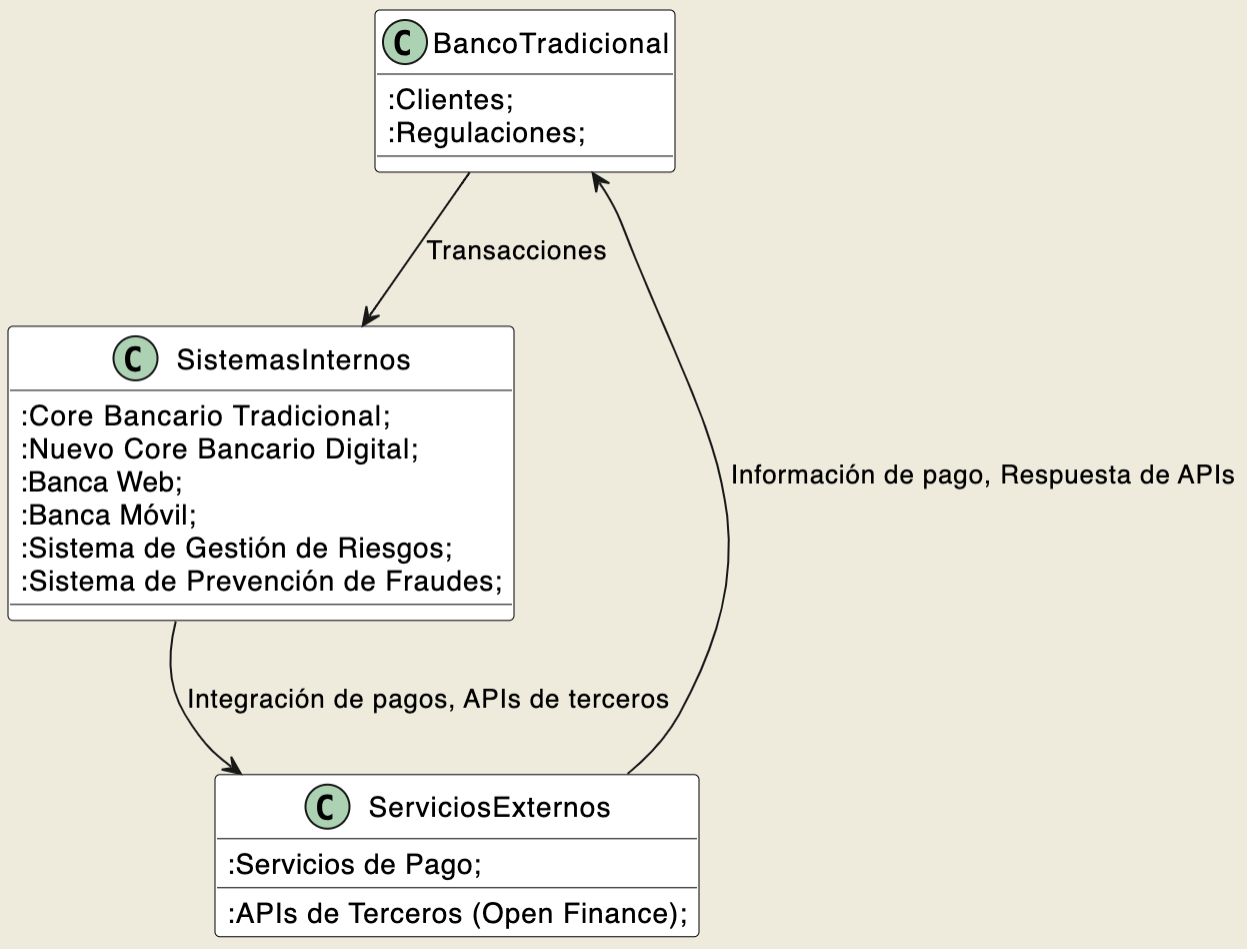


Diagrama de Contenedores (C4 Nivel 2)

```
@startuml diagrama de contenedores
!define RECTANGLE class
skinparam backgroundColor #EEEBDC
skinparam classBackgroundColor White
skinparam shadowing false
skinparam class {
   BackgroundColor White
   BorderColor Black
}

package "Banco Tradicional" {
   RECTANGLE Clientes
```

```
  RECTANGLE Regulaciones
}

package "Plataforma del Banco" {
  RECTANGLE "Core Bancario Tradicional" as CoreTradicional
  RECTANGLE "Nuevo Core Bancario Digital" as CoreDigital
  RECTANGLE "Banca Web" as BancaWeb
  RECTANGLE "Banca Móvil" as BancaMovil
  RECTANGLE "API Gateway" as APIGateway
  RECTANGLE "Sistema de Gestión de Riesgos" as GestionRiesgos
  RECTANGLE "Sistema de Prevención de Fraudes" as PrevencionFraudes
  RECTANGLE "Base de Datos Transaccional" as BDTransaccional
}

package "Servicios Externos" {
  RECTANGLE "Plataforma de Servicios de Pago" as Pago
  RECTANGLE "APIs de Terceros (Open Finance)" as OpenFinance
}

Clientes --> APIGateway : Realiza operaciones bancarias
APIGateway --> CoreTradicional : Acceso a cuentas tradicionales
APIGateway --> CoreDigital : Acceso a nuevas cuentas digitales
BancaWeb --> APIGateway : Operaciones a través de Web
BancaMovil --> APIGateway : Operaciones a través de Móvil
APIGateway --> Pago : Procesa pagos externos
APIGateway --> OpenFinance : Integra con APIs de terceros
CoreTradicional --> BDTransaccional : Consulta y actualiza datos
CoreDigital --> BDTransaccional : Consulta y actualiza datos
APIGateway --> GestionRiesgos : Analiza riesgos de transacciones
APIGateway --> PrevencionFraudes : Monitorea actividades sospechosas

@enduml
```

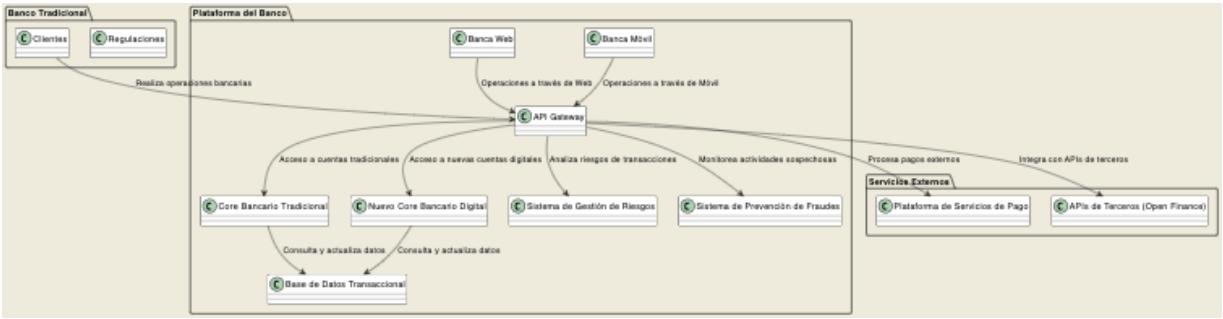


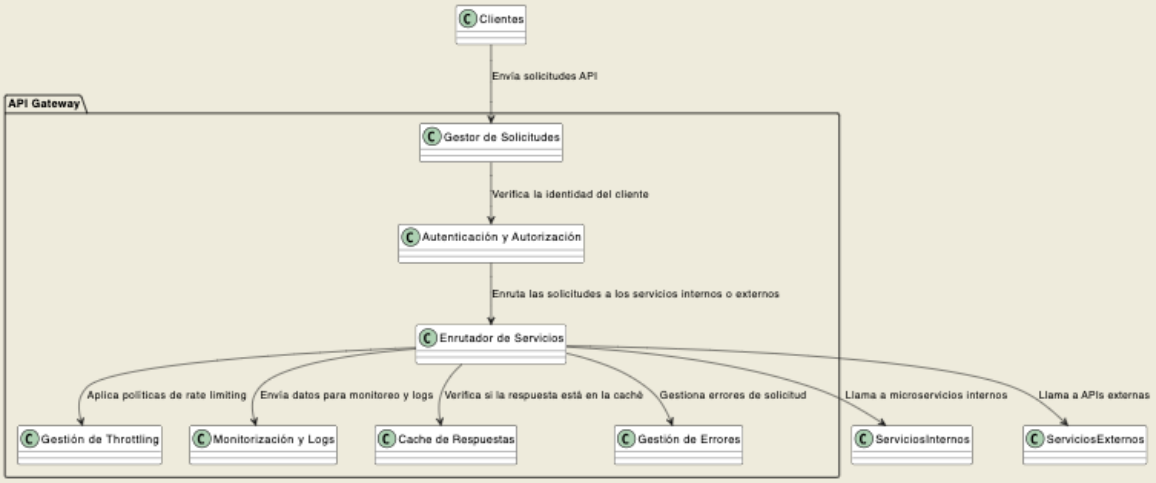
Diagrama de Componentes (C4 Nivel 3)

```
@startuml diagrama de componentes
!define RECTANGLE class
skinparam backgroundColor #EEEBDC
skinparam classBackgroundColor White
skinparam shadowing false
skinparam class {
  BackgroundColor White
  BorderColor Black
}

package "API Gateway" {
  RECTANGLE "Gestor de Solicitudes" as RequestHandler
  RECTANGLE "Autenticación y Autorización" as Auth
  RECTANGLE "Enrutador de Servicios" as ServiceRouter
  RECTANGLE "Gestión de Throttling" as Throttling
  RECTANGLE "Monitorización y Logs" as Monitoring
  RECTANGLE "Cache de Respuestas" as ResponseCache
  RECTANGLE "Gestión de Errores" as ErrorHandling
}

Clientes --> RequestHandler : Envía solicitudes API
RequestHandler --> Auth : Verifica la identidad del cliente
Auth --> ServiceRouter : Enruta las solicitudes a los servicios internos o externos
ServiceRouter --> Throttling : Aplica políticas de rate limiting
ServiceRouter --> ResponseCache : Verifica si la respuesta está en la caché
ServiceRouter --> ServiciosInternos : Llama a microservicios internos
ServiceRouter --> ServiciosExternos : Llama a APIs externas
ServiceRouter --> ErrorHandling : Gestiona errores de solicitud
ServiceRouter --> Monitoring : Envía datos para monitoreo y logs

@enduml
```



Patrones de integración y tecnologías

Patrones de Integración

API Gateway: Un patrón que actúa como un único punto de entrada para todas las solicitudes de los clientes. Se encarga de enrutar solicitudes a los servicios correspondientes, aplicar políticas de seguridad, y gestionar el tráfico.

Service Registry: Un patrón que permite al API Gateway descubrir y enrutar solicitudes a servicios que pueden estar distribuidos y escalados dinámicamente.

Circuit Breaker: Un patrón que ayuda a manejar fallos en servicios dependientes, evitando que una falla en cascada afecte a todo el sistema.

Throttling: Limita la tasa de solicitudes para proteger servicios backend de sobrecargas.

Caching: Almacena respuestas de solicitudes frecuentes para mejorar el rendimiento y reducir la carga en los servicios backend.

Patron de Tecnología

Arquitectura General

1. **Frontend (React.js)**:
    - **Componentes de UI**: Los componentes de React.js manejarán la interfaz de usuario (UI) y se comunicarán con el backend a través de llamadas a APIs RESTful.
    - **State Management**: Puedes utilizar Redux o Context API para manejar el estado global de la aplicación.
    - **Routing**: Usa react-router para manejar la navegación entre las distintas vistas de la aplicación.
2. **Backend (Java Spring Boot)**:
    - **Controladores REST (REST Controllers)**: Los controladores manejarán las solicitudes HTTP y devolverán respuestas JSON al frontend.
    - **Servicios**: Contendrán la lógica de negocio y servirán como intermediarios entre los controladores y los repositorios.
    - **Repositorios (Repositories)**: Acceden a la base de datos utilizando JPA/Hibernate.
    - **Seguridad**: Spring Security puede usarse para manejar autenticación y autorización, con JWT (JSON Web Tokens) para sesiones sin estado.

Patrones de Diseño y Arquitectura

1. **Frontend - React.js**:
    - **Container-Presenter Pattern**: Separa la lógica de negocio (Container) de la presentación (Presenter). Los componentes containers conectan con Redux o Context API y manejan el estado, mientras que los presentadores se encargan de la renderización.
    - **Atomic Design**: Organiza los componentes en átomos, moléculas, organismos, plantillas y páginas, para crear una estructura más manejable y reutilizable.
2. **Backend - Spring Boot**:
    - **MVC (Model-View-Controller)**: Es un patrón esencial en Spring Boot, donde:
        - **Model**: Repositorios y entidades de JPA.
        - **View**: En un proyecto con React.js, la vista es reemplazada por el frontend.
        - **Controller**: Controladores REST que manejan las solicitudes HTTP.
    - **Service Layer**: Implementa un patrón de servicios que encapsula la lógica de negocio.
    - **Repository Pattern**: Utiliza Spring Data JPA para implementar el acceso a los datos.
    - **DTO (Data Transfer Object)**: Utiliza DTOs para transferir datos entre el frontend y backend, asegurando que solo se envíen los datos necesarios.
    - **Singleton**: Utiliza el patrón Singleton en servicios donde una única instancia es suficiente y compartida entre los controladores.

Código

```
@startuml diagrama de patrones de tecnología
!define RECTANGLE class
skinparam backgroundColor #EEEBDC
```

```
skinparam classBackgroundColor White
skinparam shadowing false
skinparam class {
  BackgroundColor White
  BorderColor Black
}

package "Frontend (React.js)" {
  RECTANGLE "UI Components" as UIComponents
  RECTANGLE "State Management (Redux/Context)" as StateManagement
  RECTANGLE "API Calls (Axios/Fetch)" as APICalls
  RECTANGLE "Routing (react-router)" as Routing
}

package "Backend (Spring Boot)" {
  RECTANGLE "REST Controllers" as Controllers
  RECTANGLE "Service Layer" as Services
  RECTANGLE "Data Transfer Objects (DTOs)" as DTOs
  RECTANGLE "Repositories (JPA)" as Repositories
  RECTANGLE "Spring Security (JWT)" as Security
}

UIComponents --> StateManagement : Maneja el estado global
UIComponents --> Routing : Navegación entre vistas
UIComponents --> APICalls : Llama a APIs RESTful

APICalls --> Controllers : Envía solicitudes HTTP

Controllers --> Services : Lógica de negocio
Services --> Repositories : Acceso a datos
Services --> DTOs : Transfiere datos entre frontend y backend
Services --> Security : Autenticación y Autorización
Repositories --> BaseDeDatos : Consulta y actualización de datos

@enduml
```

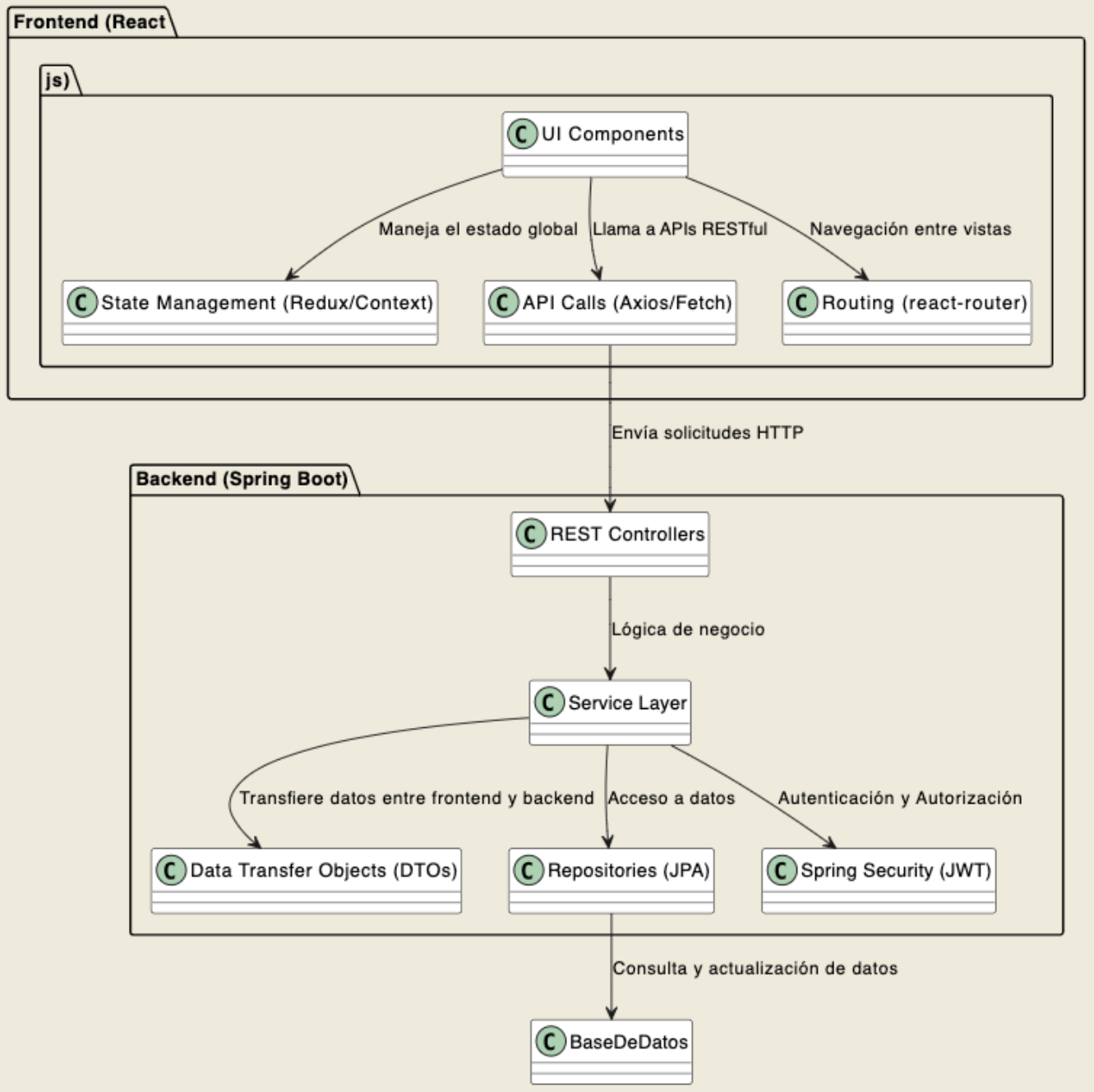


Diagrama de integración en AWS

Código

```
@startuml integracion_aws
!define RECTANGLE class
skinparam backgroundColor #EEEBDC
```

```
skinparam classBackgroundColor White
skinparam shadowing false
skinparam class {
  BackgroundColor White
  BorderColor Black
}

package "AWS Infrastructure" {
  RECTANGLE "VPC" as VPC {
    RECTANGLE "Public Subnets (Multi-AZ)" as PublicSubnets
    RECTANGLE "Private Subnets (Multi-AZ)" as PrivateSubnets
    RECTANGLE "NAT Gateways" as NATGateways
    RECTANGLE "Internet Gateway" as InternetGateway
  }
}

package "Kubernetes Cluster (EKS)" {
  RECTANGLE "EKS Control Plane" as EKSControlPlane
  RECTANGLE "Node Group (Multi-AZ)" as NodeGroup
  RECTANGLE "Load Balancer (ELB)" as LoadBalancer
  RECTANGLE "Auto Scaling" as AutoScaling
  RECTANGLE "Persistent Storage (EFS/S3)" as PersistentStorage
}

package "Database and Storage" {
  RECTANGLE "Amazon RDS (Multi-AZ)" as RDS
  RECTANGLE "Amazon S3 (Cross-Region)" as S3
  RECTANGLE "Amazon EFS" as EFS
}

package "Security" {
  RECTANGLE "IAM Roles & Policies" as IAM
  RECTANGLE "Security Groups" as SecurityGroups
  RECTANGLE "KMS (Key Management)" as KMS
  RECTANGLE "WAF (Web Application Firewall)" as WAF
}

PublicSubnets --> EKSControlPlane : Control Plane
PrivateSubnets --> NodeGroup : Worker Nodes
NATGateways --> NodeGroup : Acceso seguro a Internet
InternetGateway --> LoadBalancer : Exponer servicios
NodeGroup --> PersistentStorage : Almacenamiento persistente
NodeGroup --> RDS : Acceso a base de datos
PersistentStorage --> EFS : Almacenamiento compartido
```

PersistentStorage --> S3 : Almacenamiento de objetos

LoadBalancer --> WAF : Protección de aplicaciones
NodeGroup --> SecurityGroups : Control de tráfico
NodeGroup --> IAM : Gestión de permisos
NodeGroup --> KMS : Cifrado de datos

RDS --> S3 : Backups y replicación
S3 --> CrossRegion : Replicación de datos