**Optimización en sistemas financieros**

**1. Estrategias para mejorar el rendimiento y la escalabilidad**

**a. Concurrencia**

* Implementar un modelo de concurrencia optimista para permitir múltiples transacciones simultáneamente sin bloqueos innecesarios.
* Utilizar técnicas de control de versiones para evitar conflictos de datos.
* Implementar colas de mensajes como RabbitMQ o Kafka para procesar tareas asincrónicas y reducir la carga en la base de datos.

**b. Caché**

* Implementar un sistema de caché en memoria (Redis o Memcached) para reducir la carga en la base de datos.
* Configurar estrategias de expiración de caché y políticas de invalidación para mantener la coherencia de los datos.
* Utilizar caché de respuesta en endpoints críticos para mejorar la velocidad de respuesta.

**c. Patrones de diseño**

* Implementar **CQRS (Command Query Responsibility Segregation)** para separar operaciones de lectura y escritura, optimizando cada una de ellas.
* Aplicar **Event Sourcing** para registrar cambios en los datos y mejorar la trazabilidad.
* Usar **Factory y Repository Pattern** en la capa de acceso a datos para mejorar la modularidad y mantenibilidad del código.

**2. Seguridad en APIs financieras**

**a. Protección de la API**

* **Inyección SQL:** Usar consultas parametrizadas y ORM (Entity Framework).
* **CSRF (Cross-Site Request Forgery):** Implementar tokens CSRF en los formularios y validarlos en el servidor.
* **XSS (Cross-Site Scripting):** Sanitizar entradas del usuario y aplicar **Content Security Policy (CSP)**.
* **Autenticación y autorización:**
  + Implementar OAuth 2.0 con JWT para sesiones seguras.
  + Utilizar API Gateway para gestionar la autenticación centralizada.
  + Aplicar **roles y permisos** a nivel de endpoints.
* **Cifrado:**
  + Forzar el uso de HTTPS para proteger la comunicación.
  + Almacenar contraseñas con hashing seguro como **bcrypt o Argon2**.

**3. Transacciones en sistemas distribuidos**

**a. Consistencia y manejo de errores**

* **Consistencia eventual:**
  + Implementar patrones como **Eventual Consistency** para garantizar sincronización progresiva.
  + Usar colas de mensajes (RabbitMQ, Kafka) para garantizar la entrega de eventos.
* **Manejo de errores:**
  + Implementar el **patrón Saga** para gestionar transacciones distribuidas.
  + Diseñar estrategias de **rollback y compensación** en caso de fallos.
  + Monitorear transacciones con herramientas como **Prometheus o ELK Stack**.

**4. Pruebas unitarias y de integración**

**a. Diseño de una suite de pruebas**

* **Pruebas unitarias:**
  + Utilizar **NUnit o xUnit** en .NET, **JUnit** en Java.
  + Aplicar **Mocks y Stubs** con frameworks como **Moq o Mockito**.
* **Pruebas de integración:**
  + Verificar endpoints con **Postman o Swagger**.
  + Utilizar **TestContainers** para probar bases de datos en entornos aislados.
* **CI/CD:**
  + Automatizar pruebas con **GitHub Actions, Jenkins o Azure DevOps**.

**5. Front-end**

**a. Gestión del estado y autenticación**

* **Estado:**
  + Utilizar **Redux o Context API** en React para manejar el estado global.
  + Implementar **NgRx** en Angular para mantener el estado predecible.
* **Autenticación:**
  + Implementar autenticación basada en tokens (JWT) para validar usuarios.
  + Almacenar los tokens en **HttpOnly Cookies** para mayor seguridad.
  + Aplicar **refresh tokens** para evitar expiraciones prematuras.
* **Seguridad y coherencia:**
  + Incluir el token de autenticación en cada solicitud HTTP.
  + Manejar errores de autenticación con redirección a la página de login.
  + Implementar **CORS (Cross-Origin Resource Sharing)** correctamente.