1. Optimización en sistemas financieros:

Para optimizar el rendimiento y la escalabilidad en una aplicación bancaria que procesa transacciones en tiempo real, implementaría técnicas de concurrencia utilizando programación asíncrona con async/await en .NET, lo cual permite manejar múltiples solicitudes en paralelo sin bloquear el hilo principal. Adicionalmente, integraría un sistema de caché distribuido como Redis para reducir el acceso constante a la base de datos, mejorando los tiempos de respuesta. En cuanto a patrones de diseño, aplicaría CQRS (Command Query Responsibility Segregation) para separar las operaciones de lectura y escritura, optimizando el rendimiento y permitiendo escalabilidad independiente de cada operación.

2. Seguridad en APIs financieras:

Para proteger una API financiera que maneja información sensible, aplicaría múltiples capas de seguridad. En primer lugar, para evitar inyección SQL, utilizaría un ORM como Entity Framework en .NET y consultas parametrizadas. Contra ataques CSRF, implementaría el uso de anti-forgery tokens en cada solicitud sensible, mientras que para mitigar XSS, validaría y sanitizaría todos los datos de entrada en formularios y endpoints. Además, reforzaría la autenticación y autorización con JWT (JSON Web Tokens), asegurando que solo usuarios autorizados puedan acceder a los recursos críticos de la aplicación.

3. Transacciones en sistemas distribuidos:

En un sistema bancario distribuido, para garantizar la consistencia y el manejo de errores al procesar transferencias entre cuentas en diferentes servicios, aplicaría el patrón Saga para manejar transacciones distribuidas. Este patrón permite coordinar múltiples microservicios, asegurando que todas las operaciones se completen correctamente o se reviertan en caso de error. Además, integraría un mecanismo de compensación para revertir cambios parciales en caso de fallo y así mantener la consistencia de los datos en todo el sistema.

4. Pruebas unitarias y de integración:

Para diseñar una suite de pruebas que garantice el correcto funcionamiento de una API bancaria, desarrollaría pruebas unitarias para validar de forma aislada cada componente de la API, utilizando herramientas como xUnit o MSTest. Complementaría con pruebas de integración para evaluar la interacción entre los servicios y la base de datos, asegurando que el flujo completo funcione correctamente. Además, implementaría mocks con Moq para simular dependencias externas sin afectar el entorno real, manteniendo las pruebas rápidas y seguras.

5. Front-end:

En una aplicación bancaria que muestra el saldo de las cuentas, gestionaría el estado utilizando React Context o Redux para asegurar que los datos se mantengan sincronizados en toda la aplicación. La autenticación la manejaría con JWT, garantizando que solo usuarios autenticados puedan acceder y visualizar la información de sus cuentas. Además, integraría WebSockets o SSE (Server-Sent Events) para actualizar el saldo en tiempo real después de cada transacción, proporcionando una experiencia fluida y segura para el usuario.