Lumen

-Por Daniel Álvarez Morales y Miguel Ángel Montero Benítez.
DAM 2ºA.
IES Albarregas.
(0492 - Proyecto DAM - 2025/2026)
Docente: Mercedes Martinez Fragoso.
Fecha: [dd/mm/aaaa]
Versión documento: v1.0

Enlace a repositorio: EnlaceRepositoriGitHub
Enlace a tablero: EnlaceTableroTrello
Enlace a figma: EnlaceFigmaInterfaz
Enlace a documento: Enlace_PDF_DefincionProyecto

ÍNDICE

2 Resumen ejecutivo.

En la actualidad, nadie nos enseña educación financiera. Por eso, en Lumen hemos creado una aplicación que ayuda a cumplir tus objetivos de ahorro, estableciendo límites en los diferentes sectores de gasto. Nuestro objetivo principal es ayudar al ciudadano promedio a ahorrar en una época donde parece que el dinero se esfuma.

Nuestros usuarios destinatarios son la clase obrera, familias y cualquier persona que quiera ahorrar para invertir, todos ellos habituales usuarios de smartphones. La solución que proponemos es una aplicación móvil que analiza tus gastos, genera gráficas de en qué gastas tu dinero y permite establecer limitadores diarios para no excederte.

2.1. Problema/oportunidad

En la actualidad, la mayoría de las personas no reciben una educación financiera adecuada. Esto provoca que muchos ciudadanos, especialmente dentro de la clase trabajadora y las familias, tengan dificultades para controlar sus gastos y alcanzar sus metas de ahorro. En una época donde el costo de vida aumenta y el dinero parece desaparecer rápidamente, existe una clara oportunidad de ofrecer una herramienta digital que fomente la educación financiera práctica y el ahorro consciente.

2.2. Propuesta de solución (1–2 párrafos)

Lumen es una aplicación móvil diseñada para ayudar a los usuarios a gestionar mejor su economía personal. La app analiza los gastos del usuario, genera gráficas interactivas que muestran en qué se está utilizando el dinero y permite establecer límites personalizados en diferentes categorías de consumo. De esta forma, el usuario puede visualizar fácilmente su comportamiento financiero y adoptar hábitos de ahorro más sostenibles.

El desarrollo de Lumen se realizará utilizando Kotlin/Flutter para la interfaz y el entorno Android, con una base de datos MongoDB para el almacenamiento eficiente de la información financiera. Con una interfaz intuitiva y herramientas de análisis claras, Lumen convierte el ahorro en un proceso sencillo, educativo y accesible para todos.

2.3. Objetivo de producto y objetivo de E1

Ayudar a personas con poca o nula educación financiera a que de una manera simple e intuitiva, puedan llegar a sus objetivos de ahorros, teniendo , mediante una interfaz muy sencilla, un control de gastos e ingresos

2.4. Alcance de la demo de E1 (qué se muestra y qué no)

En nuestra primera demo, queremos mostrar, registro de usuario, gestión de las cuentas de usuario, el panel principal (donde se muestra la vista general del gráfico de balance de ingresos y gastos). Queremos mostrar un apartado que no esté disponible para todos los usuarios, si no que sea una funcionalidad únicamente para los Premium (aún no sabemos la funcionalidad premium a implementar)

3 Justificación

Hemos elegido este modelo de aplicación porque nos parece interesante y sus desarrollo aporta los siguientes beneficios:

3.1. Beneficios esperados (técnicos, educativos y de uso)

Beneficio técnico y educativo:

Nos permite aprender a desarrollar gráficos dentro de un programa, así como a interpretar operaciones financieras mediante una representación visual clara de los gastos e ingresos.

Beneficio de uso:

Ayuda al usuario a gestionar sus recursos monetarios a través de una interfaz amigable e intuitiva, mejorando la toma de decisiones financieras a largo plazo.

3.2. Integración con la industria extremeña (sectores, clústeres, casos de uso)

La aplicación puede integrarse en distintos ámbitos de la economía extremeña, especialmente entre pymes, autónomos y emprendedores, para facilitar la planificación y gestión de sus recursos financieros. Su uso puede adaptarse a varios sectores clave de la región:

Sectores agrícolas:

La aplicación podría ayudar a agricultores y cooperativas a registrar gastos en semillas, abonos, combustible o maquinaria, y a controlar los ingresos derivados de la venta de productos.
Esto facilita la planificación de campañas agrícolas y el control del flujo de caja de cada temporada

Ejemplo de caso de uso: Un agricultor utiliza la aplicación para registrar los gastos de gasóleo y abono, y conocer los beneficios obtenidos tras la venta de la cosecha.

PYMES:

Las pymes pueden utilizar la aplicación para llevar un control simple y visual de sus ingresos, gastos e inversiones menores, sin necesidad de software contable complejo.
Les permitiría tomar decisiones rápidas sobre presupuestos, detectar meses con más gasto y mejorar su rentabilidad.

Ejemplo de caso de uso: Una pyme de productos locales controla los ingresos de ventas semanales y los gastos de transporte para mejorar la planificación de pedidos.

Autónomos:

Los trabajadores por cuenta propia podrían usarla para organizar sus facturas y movimientos económicos diarios, separando gastos personales y profesionales.
Esto les ayudaría a planificar impuestos trimestrales y a mantener una visión clara de su actividad económica.

Ejemplo de caso de uso: Un autónomo del sector servicios utiliza la app para visualizar sus ingresos y gastos mensuales, facilitando la presentación de impuestos.

Clústeres

La aplicación también podría integrarse con clústeres regionales, como el Clúster TIC Extremadura o el Clúster Agroalimentario, para fomentar la digitalización y modernización de la gestión económica en las empresas asociadas.
De esta manera, el proyecto contribuiría a los objetivos de innovación y eficiencia que estos grupos empresariales promueven en la región.

3.3. Análisis de productos similares (benchmark breve: 3–5 referencias)

El análisis de nuestro benchmark nos deja con los siguientes productos extremeños similares, aunque todos ellos pertenecen al ámbito de la banca:

A) **Caja Rural de Extremadura – App Ruralvía**
Aplicación de banca digital que permite consultar movimientos, realizar transferencias y gestionar tarjetas.
Aunque es funcional para la administración básica del dinero, no ofrece una **clasificación automática de gastos** ni análisis visual avanzado.
Su principal enfoque es la operativa bancaria, no la interpretación financiera del usuario.

B) **Caja Almendralejo – App de Banca Móvil**
Permite la gestión de cuentas y operaciones habituales desde el móvil.
La aplicación es estable y cercana para usuarios de la región, pero su enfoque sigue siendo transaccional y **carece de herramientas de planificación financiera** como presupuestos o alertas inteligentes.

C) **Unicaja Banco (antes Liberbank en Extremadura) – App Unicaja**
App con funciones modernas como pagos móviles, bizum y gestión de productos financieros.
A pesar de ello, su función de análisis del gasto es **limitada y poco visual** en comparación con gestores financieros especializados.
No incentiva el ahorro ni permite proyecciones de gastos a futuro.

D) **Ibercaja – App Ibercaja (ampliamente presente en el suroeste peninsular)**
Incluye gráficos de balance y categorización básica.
Sin embargo, la clasificación no siempre es automática o precisa, y la herramienta **no permite establecer metas de ahorro personalizadas**.

3.4. Participación en ODS (mapa ODS ↔ funcionalidad/impacto)

Nuestra app recoge todas estas ods de la agenda 2030:

ODS 4 → Educación de calidad

La app enseña a los usuarios a interpretar sus finanzas personales, fomentando la educación económica y la toma de decisiones responsables.

ODS 8 →Trabajo decente y crecimiento económico.

Ayuda a autónomos, pymes y emprendedores a gestionar mejor sus recursos financieros y mantener la estabilidad económica.

ODS 9 → Industria, innovación e infraestructura.

Promueve la digitalización de la gestión financiera en pequeñas empresas y sectores tradicionales (como el agrícola).

ODS 12 → Producción y consumo responsables.

Fomenta el control del gasto y la planificación, evitando el consumo impulsivo o irresponsable.

4. Historias de usuario

Los usuarios que tienen estas demandas expresadas en “historias de usuario” son las que satisfacen nuestras funcionalidades.

Historias principales de usuario del programa:

HU-001: Visualizar gráfico general

Como usuario altamente ocupado,
quiero ver un gráfico sencillo escueto con mis gastos, ingresos y ahorros,
para poder controlar y planificar mi dinero según el periodo que seleccione (semana, mes o año).

HU-002: Añadir gastos hipotéticos

Como usuario preocupado,
quiero introducir posibles gastos en la gráfica de finanzas
para poder calcular con seguridad el pago de deudas ante posibles imprevistos.

HU-003: ver lista de gastos

Como usuario altamente ocupado,
quiero una lista que refleje todos los gastos de un día, semana o mes.
para identificar o tener en cuenta los días o picos más costosos.

HU-004: Crear gráficas alternativas

Como usuario que desea explorar diferentes escenarios,
quiero poder crear gráficas alternativas con distintos gastos o ingresos simulados,

para visualizar cómo cambiaría mi línea de ahorros sin modificar mi gráfica original.

HU-005: Elementos armoniosos

Como usuario con dificultades visuales,
quiero que los elementos visuales de la aplicación (gráficos, botones y colores) mantengan una apariencia armoniosa y coherente,
para sentir una experiencia agradable y facilitar la comprensión de mi información financiera.

HU-006: Carga rápida

Como usuario con tiempo limitado,
quiero que la aplicación cargue los gráficos y los datos en menos de 3 segundos,
para poder visualizar mi situación financiera sin esperas y mejorar mi experiencia de uso.

HU-008: Gráficas por categorías
Como usuario que busca identificar sus hábitos de gasto,
quiero ver gráficas que muestren en qué categorías gasto más,
para poder detectar y reducir los gastos innecesarios.

4.1. Convenciones de numeración y formato (p. ej., HU-001, HU-002…)

En este documento se utilizarán las siguientes convenciones:

– hu-### para identificar historias de usuario (ej.: hu-001).
– ca asociados al mismo código de la hu correspondiente.
– versiones del documento con formato vX.Y (ej.: v1.0).

4.2. Ejemplos de historias (3–5) con Criterios de Aceptación (Given/When/Then)

HU-001: Visualizar gráfico general

Dado que el usuario tiene datos registrados,
cuando accede a la pantalla principal,
entonces se muestra un gráfico con los gastos, ingresos y ahorros del periodo actual.

Dado que el usuario selecciona un periodo distinto (semana, mes o año),
cuando confirma su elección,
entonces el gráfico debe actualizarse automáticamente con los datos correspondientes.

Dado que no existen datos registrados,
cuando el usuario abre la aplicación,
entonces el gráfico debe mostrar un mensaje informativo o estar vacío sin causar errores.

HU-002: Añadir gastos

Dado que el usuario se encuentra en la vista de gastos,
cuando pulsa el botón Añadir gasto el icono '+',
entonces se abre un formulario con los campos necesarios (concepto, fecha, valor importe y si es real (puede ser real o hipotético)).

Dado que el usuario completa correctamente los campos del formulario,
cuando confirma la acción,
entonces el nuevo gasto se guarda y aparece reflejado tanto en la lista de gastos como en el gráfico de finanzas.

Dado que el usuario deja campos vacíos o introduce datos no válidos,
cuando intenta guardar el gasto,
entonces la aplicación debe mostrar un mensaje de error o advertencia solicitando corregir los datos.

HU-004: Crear gráficas alternativas
Dado que el usuario ya tiene una gráfica principal,
cuando selecciona la opción "Crear gráfica alternativa",
entonces el sistema genera una copia editable de la gráfica actual.

Dado que el usuario ya tiene 4 gráficas alternativas,
cuando intenta guardar una nueva,
entonces aparece una ventana emergente que le invita a suscribirse a la versión premium.

HU-005: Elementos armoniosos

Dado que el usuario navega entre distintas secciones de la aplicación,
cuando se muestran los gráficos y botones,
entonces los elementos deben mantener una paleta de colores coherente y equilibrada.

Dado que el usuario cambia el tema de la aplicación (modo claro/oscuro),
cuando se actualiza la interfaz,
entonces todos los componentes deben adaptarse correctamente sin perder legibilidad ni

contraste.

Dado que el usuario visualiza un gráfico o panel,
cuando se cargan los datos,
entonces los textos, iconos y colores deben ser consistentes con el resto de la interfaz.

HU-006: Carga rápida

Dado que el usuario inicia la aplicación,
cuando accede a la pantalla principal,
entonces el tiempo de carga no debe superar los 3 segundos antes de mostrar el contenido inicial.

Dado que el usuario cambia de sección (por ejemplo, de “Gastos” a “Ingresos”),
cuando realiza la acción,
entonces la nueva vista debe mostrarse sin bloqueos ni retrasos perceptibles.

Dado que la conexión del usuario es estable,
cuando la aplicación solicita datos del servidor o archivo local,
entonces los gráficos deben renderizarse progresivamente sin congelar la interfaz.

4.3. Backlog priorizado (MoSCoW/Kano o similar)- Must/Should/Could → Imprescindible/Importante/Conveniente

HU-001
Ver gráfico de gastos, ingresos y ahorros → **Must** (Imprescindible)

HU-002
Añadir gastos hipotéticos a la gráfica de finanzas → **Could** (Conveniente)

HU-003
ver lista detallada de gastos → **Should** (Importante)

HU-004
Crear gráficas alternativas (modo premium) → **Could** (Conveniente)

4.4. Trazabilidad HU ↔ CA ↔ Pruebas ↔ Figma

Trazabilidad HU

hu-001: visualizar gráfico general

Criterios aceptación (CA)
- El usuario puede seleccionar el período (día/semana/mes/año).
- Se muestra un gráfico claro con ingresos, gastos y ahorros.
- El gráfico se actualiza automáticamente al cambiar el período.

Pruebas
- Seleccionar distintos períodos y comprobar que la gráfica se actualiza.
- Verificar que los datos mostrados coinciden con los registros reales.
- Comprobar que la carga es rápida y no se congela.

Figma
Enlaces prototipo hu-001: EnlaceGrafico

Trazabilidad HU

hu-002: añadir gastos hipotéticos

Criterios aceptación (CA)
- El usuario puede introducir un gasto manualmente con cantidad y categoría.
- El gasto hipotético se refleja visualmente sin alterar los datos reales.
- El usuario puede eliminar o editar el gasto simulado.

Pruebas
- Introducir un gasto ficticio y comprobar que se añade a la simulación.
- Verificar que no modifica la base de datos real.
- Borrar el gasto y confirmar que desaparece de la vista.

Figma
Enlaces prototipo hu-002: EnlaceGastosHipoteticos

Trazabilidad HU

hu-003: ver lista de gastos

Criterios aceptación (CA)
- Se muestra una lista de gastos filtrable por día, semana o mes.
- La lista está ordenada por fecha o categoría.
- Los valores mostrados son correctos y legibles.

Pruebas
- probar filtros por día/semana/mes y verificar resultados.
- revisar que los datos coinciden con lo almacenado.
- comprobar la usabilidad al desplazarse por la lista.

Figma
Enlaces prototipo hu-003: EnlaceListaGastos

Trazabilidad HU

hu-004: crear gráficas alternativas

Criterios aceptación (CA)
- el usuario puede crear escenarios financieros sin modificar el gráfico principal.
- cada escenario puede almacenarse temporalmente o descartarse.
- se puede comparar el gráfico original con el alternativo.

Pruebas
- crear un escenario y verificar que se genera correctamente.
- comparar visualmente el escenario vs. gráfico base.
- descartar escenario y confirmar que el gráfico principal queda intacto.

Figma
Enlaces prototipo hu-004: Enlace Gráficas Alternativas

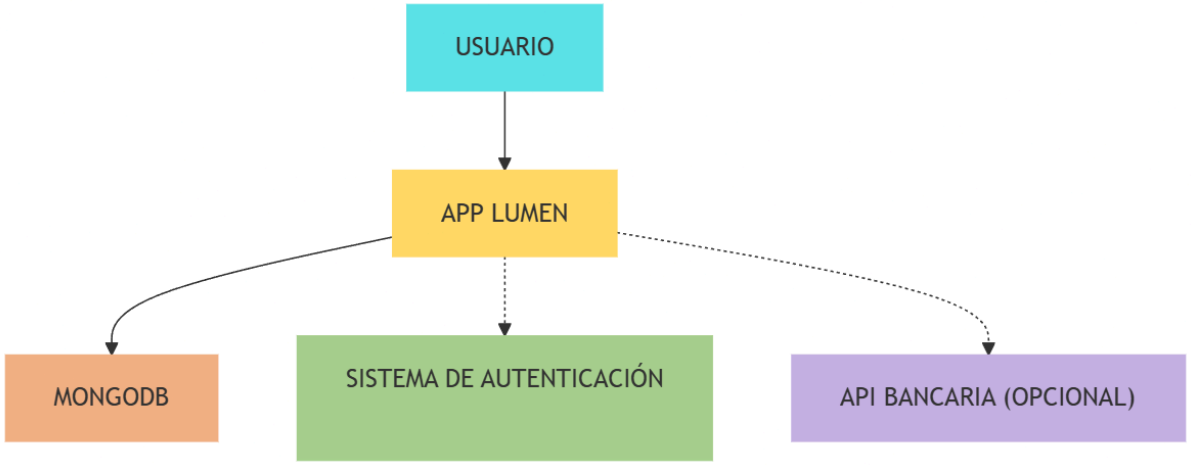
5. Arquitectura

5.1. Diagrama (C1/C2: contexto y contenedores)

Lumen es una aplicación móvil que ayuda a los usuarios a gestionar sus finanzas personales: registrar ingresos y gastos, establecer límites, generar gráficas y alcanzar objetivos de ahorro.

En el diagrama de contexto, hay varios agentes:

-Usuario final: individuos, familias o trabajadores que quieren controlar y optimizar su dinero.
-App Lumen: sistema central donde se procesan los datos financieros y se generan informes.
-Base de datos MongoDB: almacenamiento de información financiera de manera segura y eficiente.
-Sistema de autenticación: gestiona el login y protege los datos del usuario.
-API bancaria (opcional/futuro): permite integrar cuentas bancarias para análisis automático de movimientos.



Podemos observar la rama principal que compone **Usuario-App Lumen-Sistema de autenticacion**

Y de ella cuelga la subrama **Sistema de autenticación-Api**

USUARIO —— APP LUMEN —— MONGODB
                    |
                    ├── SISTEMA DE AUTENTICACIÓN
                    └── API BANCARIA (OPCIONAL)

5.2. Decisiones de arquitectura (ADR) y numeración (p. ej., ADR-001, ADR-002)

ADR-001: Arquitectura Monolítica con Spring Boot

Decisión arquitectónica: Desarrollaremos una API monolítica spring boot que se comunique directamente con mySQL. Todo las operaciones de sistema en un solo artefacto desplegable.

Contexto: Planteamos una organización simple en la aplicación para desarrollarla con éxito.

Razones: Es simple y básico para el despliegue y mantenimiento al concentrar todas las operaciones en un solo artefacto.

Tarea 1:
Crear un proyecto Spring Boot dependencias: Web, Validation, JDBC, MyBatis y Flyway.

Tarea 2:
Estructura packages→ controller/, service/, dao/, model/ y dto/.

Tarea 3:
Implementar un endpoint /api/v1/health que devuelva un JSON con el estado del sistema.

Alternativas: Arquitectura microServicios o Monolito modular.

Consecuencia: Una aplicación fácil de programar debido a su organización aunque puede limitarse al momento de escalar a una app más grande y compleja.

ADR-002: Seguridad con API de google

Decisión arquitectónica: Desarrollaremos una API monolítica spring boot que se comunique directamente con mySQL. Todo las operaciones de sistema en un solo artefacto desplegable.

Contexto: Planteamos un sistema de seguridad la aplicación para el login de las cuentas con google.

Razones: Será sencillo de implementar para registrar las cuentas de los usuario con la API de google.

Tarea 1:
Crear un proyecto Spring Boot dependencias: Web, Validation, MyBatis y MyBatis Migration.

Tarea 2:
Estructura packages→ controller/, service/, dao/, model/ y dto/.

Tarea 3:
Implementar un endpoint /api/v1/health que devuelva un JSON con el estado del sistema.

Alternativas: Api key o JWT.

Consecuencia: Planteamos un sistema de seguridad simple para los usuarios simple pero el acceso estará sujeto a la disponibilidad y políticas de Google.

//Preguntar si se puede cambiar más adelante

ADR-003: Acceso a datos con MyBatis

Decisión arquitectónica:Implementaremos MyBatis para el acceso a datos, en lugar de JDBC Template o Hibernate.
Esta decisión además de permitir escribir SQL explícito, controla exactamente qué consultas se ejecutan contra la base de datos y los mapeos de los objetos son automáticos con las anotaciones o xml.

Contexto: Necesitamos un sistema de acceso a datos para la aplicación eficiente y flexible para interactuar con MySQL, manejando consultas complejas y relaciones de datos de forma predecible.

Razones: Será sencillo de implementar porque permite escribir SQL explícito y controlar exactamente qué consultas se ejecutan contra la base de datos, al mismo tiempo que mapea automáticamente los resultados a objetos Java mediante XML o anotaciones.

Tarea 1:
Configurar MyBatis: definir mappers usando XML o anotaciones para mapear consultas SQL a objetos Java.

Tarea 2
Implementar un endpoint de prueba: desarrollar un endpoint /api/v1/health que ejecute una consulta simple mediante MyBatis y devuelva un JSON con el estado del sistema, verificando que la conexión y los mapeos funcionen correctamente.

Alternativas: spring jdbc template o hibernate.

Consecuencia: Mantenemos control total sobre el SQL y contamos con mapeo automático a objetos Java, lo que facilita el desarrollo y hace el código más limpio y mantenible. Las relaciones complejas siguen requiriendo manejo manual, pero las consultas explícitas permiten adaptaciones futuras fácilmente.

ADR-004: versionado de esquemas con MyBatis Migrations.

Decisión arquitectónica:

//Preguntar si se puede cambiar más adelante

Utilizaremos **MyBatis Migrations** para gestionar el versionado del esquema de la base de datos.

Con esta herramienta controlaremos cada cambio estructural mediante scripts SQL explícitos, garantizando que todos los entornos utilicen la misma versión del esquema.

Contexto:
Necesitamos mantener sincronizada la estructura de la base de datos entre los diferentes desarrolladores y entornos. Debido a que el acceso a datos se realiza con SQL manual mediante MyBatis, resulta natural utilizar una herramienta que siga la misma filosofía y permita controlar claramente qué cambios se aplican en cada versión.

Razones:
MyBatis Migrations es sencillo de usar, ligero y consistente con el enfoque SQL explícito de MyBatis. Cada cambio se define en un archivo SQL versionado, lo que facilita entender y revisar todas las modificaciones realizadas sobre el esquema. Además, evita la complejidad de herramientas más pesadas y se integra bien en proyectos donde no se usa un ORM.

Tarea 1:
Crear la carpeta de migraciones y generar el archivo inicial de control (init) usando MyBatis Migrations.

Tarea 2:
Crear scripts SQL versionados para cada cambio de esquema con la nomenclatura estándar (por ejemplo, *001_create_tables.sql*, *002_add_columns.sql*, etc.).

Tarea 3:
Configurar el arranque del proyecto para que ejecute automáticamente las migraciones pendientes antes de iniciar la aplicación.

Alternativas:
Flyway o Liquibase como herramientas más completas de migración, o el uso de scripts SQL manuales sin sistema de versionado.

Consecuencia:
Obtenemos un sistema simple y transparente para controlar la evolución del esquema. Aunque no dispone de tantas funcionalidades como otras herramientas, garantiza claridad, facilidad de mantenimiento y un control total sobre cada cambio aplicado a la base de datos.

ADR-005: Validaciones y Manejo de Errores JSON

Decisión arquitectónica:
Utilizaremos Bean Validation (Hibernate Validator) para validar los datos de entrada y @ControllerAdvice para gestionar las excepciones y devolver respuestas de error en formato JSON de forma centralizada.
Este enfoque garantiza validaciones consistentes y un manejo unificado de errores en toda la API.

Contexto:
La aplicación necesita validar datos provenientes de peticiones REST (JSON) y asegurar respuestas claras cuando ocurren errores, ya sean de validación, negocio o del propio sistema.
Se requiere un mecanismo estándar, sencillo y totalmente integrado con Spring Boot, sin depender de librerías externas adicionales.

Razones:
Bean Validation es la solución estándar en Java para validar objetos mediante anotaciones y Spring Boot lo integra de forma nativa, permitiendo aplicar validaciones en DTOs sin código manual repetitivo.
Por otro lado, @ControllerAdvice centraliza el manejo de excepciones, permitiendo devolver errores JSON consistentes, evitando duplicación de lógica en cada controlador y mejorando la mantenibilidad del sistema.

Tarea 1:
Definir DTOs con anotaciones de validación (por ejemplo: @NotNull, @Size, @Email, @Positive, etc.).

Tarea 2:
Habilitar validación automática usando @Valid o @Validated en controladores.

Tarea 3:
Crear una clase con @ControllerAdvice y métodos @ExceptionHandler que capturen excepciones comunes y devuelvan respuestas JSON uniformes para errores de validación, negocio o sistema.

Códigos HTTP Estándar Utilizados

400 – Bad Request:
Se devuelve cuando la petición contiene datos inválidos o no cumple las reglas de validación (por ejemplo, errores detectados por Bean Validation).

404 – Not Found:
Se usa cuando el recurso solicitado no existe en el sistema (ID no encontrado, entidad inexistente, etc.).

500 – Internal Server Error:
Indica un error inesperado en el servidor, normalmente excepciones no controladas o fallos internos que no dependen de la entrada del usuario.

Alternativas:
Validación manual mediante código imperativo, JSON Schema Validation, Vavr Validation o Apache Commons Validator.

Consecuencia:
La API obtiene un sistema de validación estándar, fácil de mantener y completamente integrado con Spring.
El manejo de errores queda centralizado, reduciendo duplicación de código y garantizando respuestas JSON consistentes y legibles para el cliente.

ADR-006: Separación entre DTOs y Modelo de Dominio

Decisión arquitectónica:
Estableceremos una separación clara entre DTOs y entidades de dominio.
Los DTOs (Data Transfer Objects) representan el contrato público de la API: lo que entra y sale a través de los endpoints REST.
Las entidades de dominio representan la estructura interna de datos tal como se almacena en la base de datos.

Contexto:
La aplicación necesita exponer datos a clientes a través de endpoints REST sin comprometer la seguridad ni acoplar la API al modelo interno de la base de datos.
Se requiere un enfoque que permita evolucionar la API y la base de datos de forma independiente, ocultar campos sensibles y agregar campos calculados o combinados.

Razones:

Control total sobre la información expuesta a la API.
Protección de campos sensibles y control de entrada de datos.
Posibilidad de evolucionar la API sin tocar la base de datos.ç
Código explícito y fácil de mantener al realizar conversiones manuales entre DTOs y entidades.

Tarea 1:
Definir DTOs en el paquete dto/ (por ejemplo: ProductoDto, UsuarioDto) con los campos públicos que la API debe exponer.

Tarea 2:
Implementar métodos de conversión manual en la capa de servicio:

public ProductoDto toDto(Producto entidad) { ... }
public Producto fromDto(ProductoDto dto) { ... }

Tarea 3:
Mantener las entidades de dominio en el paquete model/ separadas de los DTOs, evitando exponerlas directamente en los controladores.

Alternativas:
Exponer directamente las entidades en la API (más simple, pero inseguro y acoplado).
Usar librerías de mapeo automático como ModelMapper o MapStruct (reduce código repetitivo, pero añade dependencia y algo de “magia”).
Usar records de Java como DTOs ligeros (menos flexible si se requiere lógica interna).

Consecuencia:
Se logra un control seguro y explícito sobre los datos expuestos y recibidos por la API.
El código es más mantenible y claro, aunque requiere escribir métodos de conversión adicionales.
Además, mejora la seguridad al prevenir exposición accidental de campos sensibles y limita modificaciones indebidas desde el cliente.

ADR-007: Versionado de API y Estructura de Rutas

Decisión arquitectónica:
Todos nuestros endpoints REST utilizarán el prefijo /api/v1/ seguido de nombres de recursos claros y descriptivos como /productos, /workouts, /categorias.
Este esquema de versionado permite evolucionar la API sin romper clientes existentes y mantener compatibilidad hacia el futuro.
Los recursos seguirán convenciones RESTful estándar: GET para listar y obtener, POST para crear, PUT para actualizar y DELETE para eliminar.
Los filtros y parámetros de búsqueda se pasarán como query strings en la URL, por ejemplo:

/api/v1/productos?categoria=fitness&min=100&max=500

Esto mantiene las URLs limpias y permite combinaciones flexibles de filtros.

Contexto:
La aplicación necesita exponer datos a clientes (Flutter, web o desktop) de forma clara y consistente.
Se requiere un mecanismo de versionado para que futuras modificaciones de la API no rompan clientes antiguos, y una estructura de rutas RESTful que sea intuitiva y fácil de usar.

Razones:

Mantener URLs limpias, intuitivas y fáciles de recordar.

Permitir evolución futura de la API sin afectar clientes existentes.

Seguir estándares REST ampliamente reconocidos en la industria.

Facilitar filtros y búsquedas de manera flexible mediante query strings.

Tarea 1:
Agregar prefijo de versionado /api/v1/ a todos los controladores de la API.

Tarea 2:
Definir rutas claras y consistentes para todos los recursos: /productos, /workouts, /categorias, etc.

Tarea 3:
Implementar convención RESTful usando verbos HTTP (GET, POST, PUT, DELETE) para cada operación sobre recursos.

Tarea 4:
Definir filtros y parámetros de búsqueda usando query strings para permitir búsquedas flexibles:

/api/v1/productos?categoria=fitness&min=100&max=500

Alternativas:

Versionado mediante headers HTTP (Accept: application/vnd.miapp.v1+json)

Versionado mediante query string (/api/productos?version=1)

No versionar la API y mantener un único endpoint (/api/productos) haciendo cambios backwards-compatible internos.

Consecuencia:

Las URLs son claras, fáciles de usar y siguen estándares RESTful.

Permite evolucionar la API sin romper clientes existentes.

Facilita la implementación de filtros y búsquedas.

Cualquier cambio incompatible requerirá crear una nueva versión (/api/v2/), lo que añade mantenimiento adicional pero mantiene la estabilidad para clientes antiguos.

ADR-008: Perfiles de Configuración y Variables de Entorno

Decisión arquitectónica:
Implementaremos dos perfiles de Spring Boot: dev para desarrollo local y prod para producción.
Cada perfil tendrá su propio archivo de configuración (application-dev.yml y application-prod.yml) con ajustes específicos para ese entorno, como URLs de base de datos, niveles de log y timeouts.
Las credenciales sensibles (contraseñas de base de datos, claves de API, secrets) nunca se incluirán directamente en los archivos de configuración. En su lugar, se usarán variables de entorno que Spring Boot resolverá automáticamente (DB_HOST, DB_USER, DB_PASSWORD, API_KEY, etc.).

Contexto:
La aplicación necesita diferentes configuraciones para desarrollo local y producción, garantizando que los secretos no se filtren al repositorio y que cada desarrollador pueda configurar su entorno de manera independiente.
Además, se requiere un mecanismo claro y documentado para que todos los miembros del equipo conozcan qué variables de entorno deben definir.

Razones:

Separar configuración de desarrollo y producción facilita pruebas y despliegues.
Evitar hardcodear credenciales sensibles protege la seguridad de la aplicación.
Documentar variables de entorno permite que cualquier desarrollador configure su entorno correctamente.
Spring Boot resuelve automáticamente estas variables, reduciendo errores de configuración.

Tarea 1:
Crear application-dev.yml con configuración local:

URL a localhost

Logs detallados

Sin SSL

Tarea 2:
Crear application-prod.yml con configuración de producción:

URL a servidor real

Logs optimizados

SSL habilitado

Tarea 3:
Definir y usar variables de entorno para credenciales sensibles:

DB_HOST

DB_USER

DB_PASSWORD

API_KEY

Tarea 4:
Documentar todas las variables requeridas en un archivo .env.example.
Cada desarrollador copiará este archivo a .env (que estará en .gitignore) y lo completará con sus valores locales.

Alternativas:

Hardcodear credenciales en archivos .yml (no seguro, no recomendado).
Usar servicios de gestión de secretos externos como Vault o AWS Secrets Manager (más seguro pero más complejo).
Usar solo un archivo .properties para todos los entornos (menos flexible y arriesgado en producción).

Consecuencia:

Configuración segura y flexible según el entorno (dev o prod).
Credenciales sensibles nunca se suben al repositorio.
Los desarrolladores pueden trabajar en sus entornos locales de forma independiente.
Posibilidad de desplegar en producción de manera confiable sin modificar archivos de configuración.

ADR-009: Estrategia de Logging Simplificada

Decisión arquitectónica:
Configuraremos el sistema de logs con nivel INFO como estándar, evitando frameworks de logging complejos o demasiado verbosos.
Los logs proporcionarán información útil para diagnóstico sin generar ruido innecesario, permitiendo identificar problemas reales de forma rápida.
Registraremos eventos clave como inicio de endpoints, número de filas devueltas en consultas, tiempos de ejecución y errores con stack traces completos.

Contexto:
La aplicación necesita un mecanismo de logging que permita monitorear y diagnosticar problemas durante el desarrollo y producción, sin exponer datos sensibles ni saturar los registros con información irrelevante.

Razones:

Detectar errores y problemas de rendimiento sin depender de depuración manual.

Mantener la seguridad evitando registrar contraseñas, tokens o claves.

Balancear la cantidad de información registrada usando nivel INFO.

Proveer suficiente detalle para diagnosticar la mayoría de los problemas (≈95%) sin abrumar con detalles.

Tarea 1:
Configurar Spring Boot para usar nivel INFO en application.yml:

logging.level.root=INFO

Tarea 2:
Registrar eventos clave:

Inicio de cada endpoint con parámetros principales.

Cantidad de registros devueltos en consultas.

Tiempos de ejecución de operaciones costosas (ms).

Excepciones con contexto completo (stack trace).

Tarea 3:
Asegurar que nunca se registren:

Contraseñas o tokens de autenticación

Datos personales sensibles

Claves de API o secrets

Alternativas:

Nivel DEBUG para logging más detallado (demasiado verboso para producción).

Usar frameworks complejos de logging como Log4j o SLF4J con configuraciones avanzadas (mayor complejidad).

No registrar logs (no recomendable, dificulta diagnóstico).

Consecuencia:

Logs claros y útiles para diagnóstico de problemas.

Información suficiente para monitorizar rendimiento y errores.

Seguridad reforzada al no registrar datos sensibles.

Balance entre detalle y simplicidad, evitando saturación de registros.

ADR-010: Despliegue Local con Docker Compose

Decisión arquitectónica:
Crearemos un archivo docker-compose.yml que orqueste todos los servicios necesarios para el entorno de desarrollo: MySQL, la API Spring Boot y, opcionalmente, Adminer para la administración de la base de datos.
Este enfoque garantiza que todos los desarrolladores trabajen con exactamente el mismo entorno y elimina problemas de “funciona en mi máquina”.

Contexto:
Se requiere un entorno de desarrollo reproducible y consistente que permita a cualquier desarrollador levantar la aplicación completa con un solo comando, sin configuraciones adicionales ni conflictos de dependencias.

Razones:
Docker Compose simplifica el despliegue de múltiples servicios, gestionando redes, volúmenes, variables de entorno y dependencias entre contenedores automáticamente.
Permite levantar todo el stack con docker-compose up y limpiar el entorno con docker-compose down, reduciendo errores y tiempo de configuración.
Además, se asegura persistencia de datos en MySQL mediante volúmenes y se aplican principios de mínimo privilegio con un usuario específico (app_user) para la base de datos.

Tarea 1:
Crear un archivo docker-compose.yml incluyendo los servicios: MySQL, API Spring Boot y opcionalmente Adminer.

Tarea 2:
Configurar un volumen persistente para MySQL y definir las credenciales de la base de datos mediante variables de entorno en un archivo .env.

Tarea 3:

Verificar que al ejecutar docker-compose up se levanten correctamente todos los servicios, que la API se conecte a la base de datos y que docker-compose down limpie el entorno sin dejar rastros.

Alternativas:

Despliegue manual de cada servicio en cada máquina del desarrollador.

Uso de máquinas virtuales o entornos locales independientes (Vagrant, VirtualBox).

Consecuencia:
Todos los desarrolladores cuentan con un entorno idéntico, reduciendo errores de configuración y tiempo de puesta en marcha.
El entorno es fácil de levantar y limpiar, pero depende de Docker y Docker Compose, por lo que se requiere que todos los desarrolladores tengan estas herramientas instaladas y actualizadas.

5.3. Integraciones, datos y dependencias

La aplicación de finanzas personales se apoyará en varias integraciones y componentes para funcionar correctamente y mantener un flujo de datos consistente:

Base de datos

Se utilizará MySQL como sistema de gestión de base de datos principal.

MyBatis se encargará del acceso a datos, permitiendo consultas SQL explícitas y mapeo automático a objetos Java.

Flyway gestionará las migraciones y el versionado del esquema, asegurando que todos los entornos (desarrollo, pruebas y producción) estén sincronizados.

API y servicios internos

La aplicación expondrá endpoints REST para que el cliente (Flutter, web o desktop) pueda consumir los datos.

Todos los endpoints estarán versionados (/api/v1/) para facilitar futuras actualizaciones sin romper clientes existentes.

DTOs se utilizarán para transferir datos entre la API y el cliente, manteniendo separada la capa de dominio de la exposición pública.

Seguridad y autenticación

Se integrará con la API de Google para login de usuarios, gestionando credenciales de forma segura mediante OAuth2.

Las credenciales y secretos sensibles se almacenarán únicamente en variables de entorno, nunca en el código ni en archivos de configuración.

Configuración y perfiles

Spring Boot utilizará perfiles dev y prod con configuraciones específicas para cada entorno (URLs, timeouts, logs, SSL).

Esto permite mantener entornos de desarrollo independientes y configuraciones de producción optimizadas y seguras.

Dependencias externas

Spring Boot con dependencias: Web, Validation, MyBatis y MyBatis migrations.

Hibernate Validator (Bean Validation) para validar datos de entrada.

Controladores y servicios internos se comunicarán usando HTTP/REST, y todas las dependencias estarán gestionadas vía Maven para asegurar versiones consistentes y reproducibles.

Manejo de datos y consistencia

Se implementarán validaciones en la capa de servicio para asegurar la integridad de los datos.

Los errores se devolverán al cliente en formato JSON estandarizado, incluyendo códigos HTTP adecuados (400, 404, 500).

Logs en nivel INFO registrarán eventos clave como llamadas a endpoints, tiempos de ejecución y excepciones, sin almacenar información sensible.

5.4. Riesgos técnicos y mitigación

Durante el desarrollo de la aplicación de finanzas personales, se han identificado varios riesgos técnicos que podrían afectar la estabilidad, seguridad o mantenibilidad del sistema. A continuación se detallan los riesgos y las estrategias de mitigación:

Riesgo: Errores en las consultas SQL

Descripción: Al usar MyBatis con SQL explícito, es posible cometer errores en consultas que afecten la integridad de los datos o generen fallos en la aplicación.

Mitigación: Implementar pruebas unitarias y de integración para todas las consultas críticas; usar mapeos automáticos de MyBatis para reducir errores de asignación de columnas; revisar y versionar migraciones de esquema cuidadosamente.

Riesgo: Problemas de seguridad y exposición de datos sensibles

Descripción: El manejo incorrecto de credenciales o datos de usuario podría provocar vulnerabilidades.

Mitigación: Almacenar todas las credenciales en variables de entorno y no en el código; utilizar OAuth2 para login con Google; validar y sanitizar toda entrada de usuario; aplicar HTTPS y cifrado donde sea necesario.

Riesgo: Inconsistencias entre entornos (dev/prod)

Descripción: Diferencias en configuración o esquema de base de datos pueden generar errores en producción.

Mitigación: Usar perfiles de Spring Boot (dev y prod) con configuraciones separadas; mantener migraciones de esquema con MyBatis Migrations para asegurar consistencia; pruebas automáticas en entornos de staging antes de producción.

Riesgo: Problemas de rendimiento en consultas o endpoints

Descripción: Consultas pesadas o endpoints muy concurridos podrían afectar la experiencia del usuario.

Mitigación: Monitorizar tiempos de ejecución mediante logging; optimizar consultas SQL; paginar resultados; revisar índices en la base de datos; implementar caché si es necesario en el futuro.

Riesgo: Dependencias externas y compatibilidad de versiones

Descripción: Cambios o incompatibilidades en librerías externas podrían romper la aplicación.

Mitigación: Gestionar dependencias con Maven y fijar versiones estables; revisar actualizaciones de librerías antes de integrarlas; pruebas automatizadas tras cualquier actualización de dependencias.

6. Requisitos no funcionales (NFR)

La convención de numeración será NFR-001, NFR-002, NFR-003…, sucesivamente.

1. RNF-01:Rendimiento y latencia(Carga de datos)

- **Definición** :
    - Dado el RNF de rendimiento de carga de datos,
    - Cuando se lee el requisito,
    - Entonces indica:
        - Métrica: Tiempo de latencia desde la solicitud del usuario hasta el renderizado visual completo (Time-to-Interactive).
        - Umbral: Menos de 3 segundos para la carga de gráficos y datos.
        - Método de verificación: Pruebas de carga (Load Testing) con herramientas como JMeter o las DevTools del navegador en un entorno de red 4G/Wifi estándar.
- **Trazabilidad**:
    - Vinculado a HU-006 (Carga rápida): Esta historia establece explícitamente que el usuario quiere ver sus datos en menos de 3 segundos.

2. RNF-02: Usabilidad y Accesibilidad (Contraste Visual)

Este requisito garantiza que la aplicación sea cómoda de usar y accesible, alineándose con el objetivo de "elementos armoniosos".

- **Definición** :
    - Dado el RNF de accesibilidad visual,
    - Cuando se lee el requisito,
    - Entonces indica:
        - Métrica: Ratio de contraste de color entre el texto/iconos y el fondo.
        - Método de verificación: Auditoría automática con herramientas como Google Lighthouse o Accessibility Scanner en Figma.
- **Trazabilidad**:
    - Vinculado a HU-005 (Elementos armoniosos): El usuario solicita una apariencia armoniosa y legible, especialmente aquellos con dificultades visuales.

3. RNF-03: Fiabilidad de Datos (Exactitud Visual)

Este requisito es general pero fundamental para una app de finanzas: lo que veo en la pantalla debe ser matemáticamente real.

- **Definición** :
    - Dado el RNF de exactitud en la representación de datos,

    - Cuando se lee el requisito,
    - Entonces indica:
        - Métrica: Discrepancia entre el valor almacenado en la base de datos y el valor mostrado en la lista o gráfico.
        - Umbral: 0% de error. El valor numérico en la pantalla debe ser idéntico al registro guardado en MySQL.
        - Método de verificación: Prueba de "Caja Negra": ingresar un gasto conocido (ej. 50€), consultar la base de datos para confirmar el guardado, y verificar que en la lista de la app aparece exactamente "50€".
- **Trazabilidad**:
    - Vinculado a HU-003 (Ver lista de gastos): En sus criterios de aceptación se exige que "los valores mostrados son correctos y legibles".
    - Vinculado a HU-001 (Visualizar gráfico general): Requiere que el gráfico muestre gastos e ingresos reales y se actualice con los "datos correspondientes".

4. RNF-04: Portabilidad (Entorno de Desarrollo)

Garantiza que cualquier nuevo desarrollador del equipo pueda ejecutar el proyecto inmediatamente, eliminando el problema de "en mi máquina funciona".

- **Definición** :
    - Dado el RNF de portabilidad del entorno,
    - Cuando se lee el requisito,
    - Entonces indica:
        - Métrica: Tasa de éxito en el despliegue local usando un solo comando.
        - Umbral: El comando `docker-compose up` debe levantar todos los servicios (API, BD) sin errores manuales de configuración en una instalación limpia.
        - Método de verificación**:** Ejecución del script de despliegue en una máquina "limpia" (sin dependencias previas de Java/MySQL instaladas localmente, solo Docker).
- **Trazabilidad:**
    - Vinculado a ADR-010 (Despliegue Local con Docker): Define el uso de Docker Compose para orquestar servicios y garantizar entornos idénticos.

5. RNF-05: Adaptabilidad de Interfaz (Tema Claro/Oscuro)

Este es un requisito de diseño muy visual y fácil de entender: si el usuario cambia la configuración de su móvil, la app debe adaptarse.

- **Definición :**
  - Dado el RNF de adaptabilidad visual de la interfaz,
  - Cuando se lee el requisito,
  - Entonces indica:
    - Métrica**:** Porcentaje de elementos de la interfaz (fondos, textos, botones) que cambian de paleta de color correctamente.
    - Umbral: 100% de los elementos visibles deben adaptarse al tema seleccionado sin perder legibilidad.
    - Método de verificación: Inspección visual manual cambiando la configuración del sistema operativo (o un botón en la app) de "Modo Claro" a "Modo Oscuro" y viceversa.
- **Trazabilidad:**
  - Vinculado a HU-005 (Elementos armoniosos): Esta historia especifica explícitamente: "Dado que el usuario cambia el tema de la aplicación (modo claro/oscuro)... entonces todos los componentes deben adaptarse correctamente"

6. RNF-06: Integridad de Capas MVC y Estándares de Código

Este requisito asegura que el código no se convierta en un "espagueti", manteniendo separadas las responsabilidades tal como definisteis en vuestra arquitectura (Controlador/ Servicio/ DAO).

- **Definición** :
  - Dado el RNF de calidad y adherencia al patrón MVC,
  - Cuando se analiza el código fuente del backend,
  - Entonces indica:
    - Métrica: Número de violaciones de reglas de arquitectura (ej: un Controlador accediendo directamente al DAO) y de estilo (naming conventions).
    - Umbral: 0 violaciones arquitectónicas (estricto cumplimiento de capas) y 0 errores de severidad alta en el linter.
    - Método de verificación:
      1. Estático: Ejecución automática de Checkstyle o SonarQube en el pipeline de compilación.
      2. Arquitectónico: Uso de una prueba unitaria con ArchUnit (librería Java) que verifique que el paquete controller solo accede a service, y service a dao.
- **Trazabilidad**:
  - Vinculado a ADR-001 (Arquitectura Monolítica con Spring Boot): Este ADR define explícitamente la estructura de paquetes requerida: controller/, service/, dao/, model/ y dto/. El RNF verifica que esta estructura se respete.

- Vinculado a ADR-006 (Separación entre DTOs y Modelo de Dominio): Este ADR exige que las entidades de dominio (model/) no se expongan en los controladores, sino que se usen DTOs. El análisis estático debe confirmar que los controladores no devuelven objetos de entidad directamente.

7.RNF-07: Usabilidad de Inicio de Sesión (Google Auth)

Este requisito garantiza que el proceso de inicio de sesión con Google sea rápido y que la aplicación maneje correctamente las credenciales del usuario.

- **Definición:**
  - Dado el RNF de usabilidad del proceso de autenticación con Google,
  - Cuando el usuario pulsa el botón "Iniciar sesión con Google" por primera vez,
  - Entonces indica:
    - Métrica**:** Tiempo transcurrido desde que se pulsa el botón hasta que la pantalla principal de la aplicación es visible (incluyendo la comunicación con el backend para crear/verificar el usuario).
    - Umbral**:** Menos de 5 segundos.
    - Método de verificación**:** Pruebas de usabilidad (Usability Testing) con un cronómetro en la mano, o registro automático de eventos (Analytics/Logs de Firebase) en el dispositivo del usuario para medir el evento **login_start** hasta **home_screen_loaded**.
- **Trazabilidad:**
  - Vinculado a ADR-02 (Inicio de sesión rápido): Aunque el usuario no mencionó Google específicamente, esta historia cubre el requisito general de un inicio de sesión "rápido" para no frustrarlo.
  - Vinculado a la Mitigación de Riesgos (Sección 6.3 - Seguridad): Se establece que se utilizará OAuth2 para login con Google, por lo que este RNF valida la calidad de esa decisión arquitectónica.

7. Prototipo Figma y Anexos

7.1. Prototipo navegable (lista de pantallas/flows y estados)

7.2. Guía de diseño (tokens: color, tipografía, espaciado; componentes y variantes)

8. Anexos

8.3.1. Plan de pruebas y evidencias (capturas, vídeos, enlaces)

8.3.2. KPIs iniciales (aceptación HU, defectos, lead time)

8.3.3. Enlaces y referencia de artefactos (repo, tablero, ADR, NFR)

8.3.4. Changelog de E1 (tabla de cambios relevantes)