Universidad de Zaragoza

Análisis Técnico para un Sistema de Monitorización Remota de la Salud de Ancianos Utilizando Datos de Fitbit

Pablo Moreno Muñoz

Trabajo Fin de Grado

Grado en Ingeniería Informática

Tutor: Nombre del Tutor

23 de abril de 2025

Resumen

Este trabajo se enmarca en la creciente necesidad de soluciones tecnológicas para la atención sanitaria de la población anciana. Presentamos el diseño y la implementación de un sistema de monitorización remota de salud, apoyado en dispositivos Fitbit, que permite el seguimiento continuo de indicadores como la frecuencia cardíaca, los niveles de actividad física y la calidad del sueño.

El sistema desarrollado emplea un backend modular basado en microservicios, diseñado con criterios de escalabilidad y resiliencia. Se integra de manera segura con la API de Fitbit utilizando el protocolo OAuth 2.0, y permite la adquisición y almacenamiento eficiente de datos en bases de datos especializadas en series temporales. Además, se incorpora un pipeline que procesa los datos de forma automatizada tras su recepción, permitiendo la detección de eventos críticos, y paneles visuales accesibles e intuitivos que permiten a cuidadores o profesionales de la salud visualizar la evolución del estado de los usuarios de forma clara.

La tesis expone los requerimientos técnicos, las decisiones arquitectónicas adoptadas, los desafíos enfrentados y las pruebas realizadas para validar el sistema. Asimismo, se analizan las implicaciones éticas y legales relacionadas con el tratamiento de datos personales de salud, cumpliendo con el Reglamento General de Protección de Datos (RGPD). Finalmente, se discuten las oportunidades de mejora y expansión del sistema, incluyendo la integración con nuevas plataformas de dispositivos vestibles y el uso de técnicas de inteligencia artificial para predicción de eventos adversos.

Palabras clave: Monitorización remota, Fitbit, Salud ancianos, OAuth 2.0.

Índice general

Capítulo 1

Introducción

1.1. Contexto y Motivación

El progresivo envejecimiento de la población es una realidad demográfica global, y particularly acentuada en países como España, donde las proyecciones indican un aumento sostenido de la población mayor de 65 años en las próximas décadas Instituto Nacional de Estadística (INE) [2022]. Este fenómeno conlleva un aumento en la prevalencia de enfermedades crónicas y una creciente demanda de servicios sanitarios y de cuidados de larga duración, generando una presión significativa sobre los sistemas de salud y las familias Organización Mundial de la Salud (OMS) [2022]. En este escenario, la tecnología emerge como un aliado fundamental para desarrollar soluciones innovadoras que mejoren la calidad de vida de las personas mayores, promuevan su autonomía y faciliten su cuidado.

La monitorización remota de la salud, apoyada en dispositivos electrónicos y sensores, ofrece un enorme potencial para el seguimiento continuo y no invasivo de indicadores fisiológicos y de actividad Majumder et al. [2017]. Los dispositivos vestibles o *wearables*, como las pulseras de actividad y los relojes inteligentes, se han popularizado enormemente en los últimos años. Marcas como Fitbit® ofrecen dispositivos accesibles y ampliamente adoptados que recopilan datos valiosos sobre el ritmo cardíaco, los patrones de sueño, los niveles de actividad física, entre otros. Estos datos, si se gestionan y presentan adecuadamente, pueden proporcionar información muy útil sobre el bienestar general y las tendencias de salud de una persona.

La motivación principal de este Trabajo Fin de Grado (TFG) radica en explorar y aprovechar el potencial de estos dispositivos comerciales, específicamente los de Fitbit®, para construir un sistema que facilite la monitorización remota de personas mayores por parte de sus cuidadores o personal sanitario. Se busca ofrecer una herramienta tecnológica que contribuya a una atención más proactiva, permitiendo detectar posibles cambios en el estado de salud de forma temprana y mejorando la tranquilidad de los familiares y responsables del cuidado, al tiempo que se respeta la autonomía y privacidad del usuario monitorizado.

1.2. Definición del Problema

A pesar de la disponibilidad de datos generados por dispositivos como Fitbit®, a menudo existe una brecha entre la simple recolección de datos y su utilización efectiva para el seguimiento de la salud, especialmente en el contexto de personas mayores y sus cuidadores. Los problemas específicos que este trabajo busca abordar son:

- La falta de sistemas integrados que recopilen automáticamente datos relevantes de Fitbit® y los presenten de forma clara, contextualizada y comprensible para cuidadores no necesariamente expertos en tecnología.

- La dificultad para realizar un seguimiento longitudinal de los indicadores clave de salud obtenidos a través de estos dispositivos, identificando tendencias o desviaciones significativas de los patrones habituales del usuario.
- La necesidad de implementar soluciones técnicas robustas que gestionen de forma segura la autenticación con servicios de terceros (Fitbit API) y que respeten escrupulosamente la privacidad y la normativa de protección de datos (como el RGPD) al manejar información personal y sensible de salud.
- La ausencia frecuente de arquitecturas flexibles y escalables en prototipos de este tipo, que permitan una futura expansión para incluir más usuarios, más tipos de datos o integración con otros sistemas.

Este TFG se enfoca en el diseño e implementación de un prototipo que dé respuesta a estos desafíos, proporcionando una solución técnica robusta, funcional y bien documentada.

1.3. Objetivos

Para abordar el problema definido, se establecen los siguientes objetivos:

1.3.1. Objetivo General

Diseñar e implementar un prototipo de sistema software para la monitorización remota de indicadores de salud de personas mayores, utilizando datos obtenidos de pulseras de actividad Fitbit® y presentando la información de forma útil y accesible para cuidadores o personal autorizado, con un enfoque en la seguridad, la privacidad y la escalabilidad.

1.3.2. Objetivos Específicos

1. Investigar en profundidad la API web de Fitbit®, su modelo de datos, las políticas de acceso, la granularidad de los datos disponibles y el proceso de autorización seguro mediante OAuth 2.0.
2. Diseñar una arquitectura software para el backend del sistema, basada en microservicios (ej. utilizando Python con Flask/Django y contenedores Docker), que sea modular, escalable y resiliente.
3. Implementar un microservicio responsable de gestionar la autenticación de los usuarios del sistema (cuidadores, administradores) y la obtención segura de tokens de acceso para la API de Fitbit® mediante el flujo OAuth 2.0.
4. Desarrollar un microservicio encargado de la adquisición periódica y automatizada de los datos de interés de los usuarios desde la API de Fitbit® (ej. frecuencia cardíaca diaria/intradía si disponible, resumen del sueño, pasos y niveles de actividad).
5. Seleccionar, configurar e implementar una base de datos especializada en series temporales (ej. InfluxDB o TimescaleDB), adecuada para el almacenamiento eficiente y la consulta de los datos biométricos y de actividad a lo largo del tiempo.

6. Implementar la lógica necesaria para el procesamiento básico de los datos adquiridos, incluyendo la validación, limpieza y transformación de los datos para su almacenamiento y posterior visualización (ej. cálculo de promedios, identificación de periodos de inactividad/actividad).

7. Desarrollar un panel de visualización web (dashboard) sencillo e intuitivo (ej. utilizando Dash/Plotly o un framework JS como React/Vue) que permita a los cuidadores consultar el histórico de datos mediante gráficos interactivos y visualizar indicadores clave.

8. Analizar y aplicar las medidas técnicas y organizativas necesarias para garantizar la seguridad (autenticación, autorización, protección contra ataques comunes) y privacidad de los datos (minimización, seudonimización si aplica, gestión de consentimientos), en cumplimiento con los principios del Reglamento General de Protección de Datos (RGPD).

9. Validar el correcto funcionamiento del prototipo desarrollado mediante un conjunto definido de pruebas funcionales (cobertura de requisitos), de integración (comunicación entre microservicios y con API externa) y del sistema (flujos de usuario principales).

1.4. Alcance y Limitaciones

El sistema desarrollado en este TFG es un prototipo funcional centrado en demostrar la viabilidad técnica de la solución propuesta. El alcance del trabajo cubre:

- La integración con la API de Fitbit® para obtener datos de frecuencia cardíaca (resumen diario y/o intradía según disponibilidad de la API), patrones de sueño (fases, duración) y actividad física (pasos, minutos activos).
- El desarrollo de un backend basado en microservicios utilizando Python (ej. Flask) y desplegado mediante Docker.
- El almacenamiento de los datos temporales en una base de datos InfluxDB
- La implementación completa del flujo de autenticación OAuth 2.0 con Fitbit® para la autorización segura por parte del usuario.
- El desarrollo de una interfaz web básica para cuidadores que permite visualizar gráficos históricos de los datos mencionados y gestionar usuarios monitorizados.
- La aplicación de principios de diseño orientados a la seguridad y al cumplimiento del RGPD (ej. cifrado de datos sensibles en reposo/tránsito donde aplique, gestión de tokens segura).

Es importante destacar las siguientes limitaciones:

- El sistema **no es un dispositivo médico certificado** y la información proporcionada no debe utilizarse para autodiagnóstico ni para sustituir la consulta con un profesional sanitario cualificado. Su propósito es informativo y de apoyo al cuidado.

- La funcionalidad está limitada a los datos y la granularidad que Fitbit® expone a través de su API web estándar (ej. la frecuencia cardíaca intradía puede estar limitada a intervalos de 1 minuto o más, dependiendo del nivel de acceso a la API).
- El prototipo ha sido validado funcionalmente en un entorno de desarrollo y pruebas, pero no ha sido sometido a pruebas de carga extensivas ni a una evaluación de usabilidad formal con usuarios finales (ancianos y cuidadores).
- La interfaz de usuario, aunque funcional, representa un diseño básico y podría beneficiarse de mejoras significativas en términos de experiencia de usuario (UX) y diseño de interfaz (UI) para una adopción real.
- El análisis de cumplimiento del RGPD se basa en los principios de diseño y buenas prácticas, pero no constituye una auditoría legal completa ni garantiza la conformidad total en un entorno de producción sin revisiones adicionales.
- El sistema no incluye mecanismos avanzados de detección de anomalías o predicción basados en inteligencia artificial, aunque la arquitectura sentaría las bases para su futura integración.

1.5. Estructura del Documento

La presente memoria se organiza en los siguientes capítulos, siguiendo la estructura propuesta para facilitar la lectura y comprensión del trabajo realizado:

- **Capítulo 1: Introducción.** (Este capítulo) Presenta el contexto, la motivación, el problema a resolver, los objetivos, el alcance y la estructura del documento.
- **Capítulo 2: Estado del Arte y Marco Tecnológico.** Revisa soluciones existentes en el ámbito de la monitorización remota de salud con wearables y describe en detalle las tecnologías clave seleccionadas y empleadas en el proyecto (Fitbit API, OAuth 2.0, microservicios, bases de datos de series temporales, etc.).
- **Capítulo 3: Requisitos y Metodología.** Detalla los requisitos funcionales (lo que el sistema debe hacer) y no funcionales (atributos de calidad como rendimiento, seguridad, usabilidad) identificados para el sistema, y describe brevemente la metodología de desarrollo seguida (ej. iterativa, basada en prototipos).
- **Capítulo 4: Diseño y Arquitectura del Sistema.** Expone las decisiones de diseño tomadas, presentando la arquitectura general del sistema, el diseño detallado de los microservicios del backend, el esquema de la base de datos, el flujo de datos y la integración con la API externa.
- **Capítulo 5: Implementación.** Describe los detalles concretos de la implementación de los componentes más relevantes del sistema, incluyendo el entorno de desarrollo, las librerías principales utilizadas, fragmentos de código ilustrativos y los desafíos técnicos encontrados y cómo fueron resueltos.
- **Capítulo 6: Pruebas y Validación.** Explica la estrategia de pruebas definida y llevada a cabo (pruebas unitarias, de integración, del sistema) para asegurar la calidad del software y validar que el prototipo cumple con los requisitos especificados.

- **Capítulo 7: Resultados y Discusión.** Presenta el prototipo funcional resultante, mostrando ejemplos de su operación (ej. capturas de pantalla del dashboard) y discute los resultados obtenidos en términos de cumplimiento de objetivos, rendimiento observado y las limitaciones inherentes al sistema desarrollado.
- **Capítulo 8: Conclusiones y Trabajo Futuro.** Resume las principales conclusiones extraídas del desarrollo del TFG, destacando las contribuciones del trabajo y proponiendo posibles líneas de mejora, expansión y trabajo futuro sobre el sistema desarrollado.

Capítulo 2

Estado del Arte y Marco Tecnológico

Este capítulo tiene un doble propósito. Primero, revisar brevemente el estado actual de la monitorización remota de salud en personas mayores y el papel de los dispositivos vestibles como Fitbit®. Segundo, describir en detalle las tecnologías clave que habilitan el sistema desarrollado en este TFG, justificando su elección y explicando sus conceptos fundamentales.

2.1. Monitorización Remota de Salud en Personas Mayores

La monitorización remota de pacientes (RPM, por sus siglas en inglés, **Remote Patient Monitoring**) ha ganado una tracción significativa en los últimos años, impulsada por los avances tecnológicos y la necesidad de modelos de atención sanitaria más eficientes y centrados en el paciente Noah et al. [2022]. Especialmente en el contexto del cuidado de personas mayores, la RPM ofrece beneficios potenciales considerables, como la detección temprana de deterioros en la salud, la reducción de hospitalizaciones y visitas a urgencias, la mejora de la adherencia al tratamiento y el fomento de la independencia y la tranquilidad tanto para los usuarios como para sus cuidadores Bashshur et al. [2020].

Existen diversas aproximaciones a la RPM, desde sistemas basados en sensores ambientales instalados en el hogar hasta el uso de dispositivos médicos específicos o, cada vez más, el aprovechamiento de dispositivos de consumo como smartphones y wearables Majumder et al. [2017]. Sin embargo, la implementación exitosa de sistemas RPM para la población mayor también enfrenta desafíos importantes. Entre ellos destacan la usabilidad y aceptación de la tecnología por parte de los usuarios mayores, la gestión de la gran cantidad de datos generados, la necesidad de garantizar la fiabilidad y precisión de los datos, la interoperabilidad entre diferentes dispositivos y sistemas, y la gestión de la privacidad y seguridad de datos de salud altamente sensibles Lee et al. [2021].

2.2. Dispositivos Wearables: El Caso de Fitbit

El mercado de dispositivos wearables ha experimentado un crecimiento exponencial, ofreciendo una amplia gama de productos capaces de monitorizar diversos parámetros fisiológicos y de actividad Fortune Business Insights [2024] . Fitbit® (ahora parte de Google) se ha consolidado como una de las marcas líderes en el segmento de pulseras y relojes de actividad física y bienestar. Sus dispositivos suelen incluir sensores como acelerómetros (para contar pasos y detectar movimiento/sueño) y fotopletismógrafos (PPG) para medir la frecuencia cardíaca Fitbit.

Los datos típicamente accesibles a través de la API de Fitbit® incluyen resúmenes diarios y, en algunos casos, datos intradía (con granularidad variable) de pasos, distancia, calorías quemadas, minutos de actividad, fases y duración del sueño, y frecuencia

cardíaca Fitbit Developer Portal. Si bien estos dispositivos no son instrumentos médicos certificados, diversos estudios han evaluado su precisión. Por ejemplo, la medición de la frecuencia cardíaca en reposo suele considerarse razonablemente precisa, aunque puede disminuir durante actividad física intensa. La detección de fases del sueño y el conteo de pasos también muestran una correlación aceptable con métodos de referencia en muchos estudios, aunque existen limitaciones y variabilidad entre dispositivos y condiciones de uso Haghayegh et al. [2019], Nelson et al. [2016]. Es crucial tener en cuenta estas consideraciones al interpretar los datos y diseñar el sistema. La adquisición de Fitbit por parte de Google también puede tener implicaciones futuras en la disponibilidad y políticas de acceso a la API Google.

2.3. Tecnologías Habilitadoras

El desarrollo del sistema de monitorización propuesto se apoya en un conjunto de tecnologías clave que se describen a continuación.

2.3.1. API de Fitbit y OAuth 2.0

El acceso a los datos de los usuarios de Fitbit® se realiza exclusivamente a través de su API web oficial. Se trata de una API RESTful que utiliza el formato JSON para el intercambio de datos Fitbit Developer Portal. Proporciona diversos *endpoints* para obtener información del perfil del usuario, resúmenes de actividad diaria, datos de series temporales (como frecuencia cardíaca o pasos a lo largo del día con cierta granularidad), información sobre el sueño, etc. Para poder acceder a los datos de un usuario, es imprescindible obtener su consentimiento explícito a través del protocolo de autorización estándar **OAuth 2.0** Hardt [2012].

En este proyecto, se implementa el flujo *Authorization Code Grant* de OAuth 2.0, considerado el más seguro para aplicaciones web con backend. Dado que la aplicación web desarrollada está pensada para ser operada por personal autorizado (ej. un recepcionista o cuidador en una residencia) y no directamente por el usuario final (la persona mayor), el flujo de vinculación se adapta ligeramente:

1. El personal autorizado inicia sesión en la aplicación web del sistema de monitorización.

2. Dentro de la aplicación, selecciona al residente o usuario final cuya cuenta de Fitbit® desea vincular (identificado por su nombre y correo electrónico asociado a Fitbit®).

3. La aplicación web redirige el navegador del personal autorizado a la página de inicio de sesión y autorización de Fitbit®, indicando los permisos (scopes) específicos que la aplicación necesita (ej. leer datos de frecuencia cardíaca, leer datos de actividad).

4. El personal autorizado (o, idealmente, el propio residente si está presente y puede hacerlo) introduce las credenciales de la cuenta de Fitbit® del residente y autoriza explícitamente a la aplicación a acceder a los datos solicitados en nombre de ese residente.

5. Fitbit® redirige el navegador de vuelta a la aplicación web del sistema, incluyendo un código de autorización temporal en la URL de redirección.

6. El backend de la aplicación web recibe este código de autorización. De forma segura y sin exponerlo al navegador, intercambia este código (junto con las credenciales de la aplicación cliente registrada en Fitbit®) directamente con el servidor de autorización de Fitbit® para obtener un token de acceso (Access Token) y un token de refresco (Refresh Token) asociados a la cuenta del residente.

7. El token de acceso se almacena de forma segura asociado al residente y se utiliza para realizar las llamadas posteriores a la API para obtener sus datos. Estos tokens tienen una vida útil limitada (ej. 8 horas).

8. Cuando el token de acceso expira, el backend utiliza el token de refresco almacenado (que suele tener una vida útil mucho más larga o no expirar si se usa periódicamente) para obtener un nuevo par de tokens de acceso y refresco de forma automática, sin necesidad de que el personal autorizado o el residente vuelvan a intervenir.

La correcta y segura gestión de estos tokens (almacenamiento cifrado o seguro, uso exclusivo en el backend, uso de HTTPS en todas las comunicaciones) es fundamental para la seguridad y privacidad del sistema Lodderstedt et al. [2017].

2.3.2. Arquitecturas de Microservicios

Frente a las arquitecturas monolíticas tradicionales, donde toda la funcionalidad de la aplicación reside en un único proceso desplegable, la arquitectura de microservicios estructura la aplicación como una colección de servicios pequeños, autónomos y débilmente acoplados Fowler and Lewis [2014]. Cada servicio se centra en una capacidad de negocio específica, se comunica con otros servicios a través de APIs bien definidas (normalmente sobre HTTP/REST o colas de mensajes) y puede ser desarrollado, desplegado y escalado de forma independiente Newman [2021].

Las ventajas clave de este enfoque, relevantes para nuestro sistema, incluyen:

- **Escalabilidad Independiente:** Cada servicio puede escalarse horizontalmente según sus necesidades específicas (ej. escalar más instancias del servicio de adquisición de datos si hay muchos usuarios).

- **Resiliencia:** Un fallo en un servicio no tiene por qué detener todo el sistema; otros servicios pueden seguir funcionando (con mecanismos de tolerancia a fallos como *circuit breakers*).

- **Flexibilidad Tecnológica:** Cada servicio puede desarrollarse con la tecnología más adecuada para su tarea específica (diferentes lenguajes, bases de datos).

- **Despliegue Independiente:** Los cambios en un servicio pueden desplegarse sin necesidad de redesplegar todo el sistema, agilizando las actualizaciones.

Sin embargo, los microservicios también introducen complejidad en áreas como la gestión distribuida de datos, la monitorización de múltiples servicios, el despliegue orquestado (donde herramientas como Docker y Kubernetes son muy útiles) y la necesidad de una cultura DevOps madura Newman [2021]. Para este TFG, se adopta un enfoque pragmático, diseñando un número limitado de microservicios bien definidos para aprovechar las ventajas de modularidad y escalabilidad potencial.

2.3.3. Bases de Datos de Series Temporales

Los datos generados por dispositivos wearables como Fitbit® son inherentemente datos de series temporales: secuencias de mediciones indexadas por tiempo (timestamp). Si bien es posible almacenar estos datos en bases de datos relacionales tradicionales (como PostgreSQL o MySQL), las bases de datos especializadas en series temporales (TSDB - Time Series Databases) están optimizadas para este tipo de carga de trabajo DB-Engines.

Las TSDB suelen ofrecer ventajas significativas para datos de series temporales, como:

- **Alto Rendimiento en Ingesta:** Optimizadas para escribir grandes volúmenes de datos nuevos secuencialmente en el tiempo.
- **Consultas Eficientes Basadas en Tiempo:** Indexación y funciones específicas para agregar, muestrear o filtrar datos por rangos de tiempo de forma muy rápida.
- **Compresión de Datos:** Técnicas específicas para comprimir datos temporales, que suelen tener cierta redundancia o patrones, ahorrando espacio de almacenamiento.
- **Políticas de Retención de Datos:** Facilidades para descartar automáticamente datos antiguos que ya no son necesarios (ej. mantener datos con granularidad de minutos por 1 mes, pero solo resúmenes diarios después de eso).

Ejemplos populares de TSDB incluyen InfluxDB y TimescaleDB (una extensión para PostgreSQL) InfluxData, Timescale. Para este proyecto, se optó por **TimescaleDB** debido a su integración nativa con PostgreSQL, lo que permite combinar las ventajas de una TSDB con las capacidades de una base de datos relacional robusta, su uso de SQL estándar para las consultas y su madurez como proyecto Timescale.

2.3.4. Herramientas de Backend y Procesamiento

El backend del sistema, responsable de orquestar la autenticación, la adquisición de datos, el procesamiento y la exposición de APIs internas o para el frontend, se ha desarrollado utilizando **Python**. Python es una elección popular para el desarrollo web y el procesamiento de datos debido a su sintaxis clara, su amplio ecosistema de librerías y su gran comunidad Python Software Foundation.

Como framework web ligero para construir las APIs de los microservicios, se ha empleado **Flask** Pallets Projects. Flask es un microframework que proporciona las herramientas básicas para el enrutamiento de peticiones HTTP y la gestión de respuestas, permitiendo una gran flexibilidad para elegir y añadir otras librerías según sea necesario (ej. para la interacción con la base de datos, serialización de datos, etc.). Esto se alinea con la filosofía de microservicios, manteniendo cada componente lo más ligero posible.

Para la adquisición periódica de datos desde la API de Fitbit® (una tarea que debe ejecutarse de forma programada en segundo plano para cada usuario vinculado), se utiliza la librería **APScheduler** Grönholm. APScheduler permite definir trabajos (jobs) que se ejecutan en intervalos fijos, en fechas específicas o según expresiones cron, siendo adecuada para tareas de planificación dentro de una aplicación Python, como la consulta periódica a la API de Fitbit para cada usuario registrado.

2.3.5. Tecnologías de Frontend/Visualización

Para presentar la información monitorizada de forma clara y útil a los cuidadores, se ha desarrollado un panel de visualización web (dashboard) interactivo. Dada la base tecnológica del backend en Python/Flask y la naturaleza de los datos a visualizar (series temporales, gráficos estadísticos), se optó por construir la interfaz utilizando **Dash** Plotly.

Dash es un framework de Python construido sobre Flask, React y Plotly.js. Su principal ventaja en este contexto es que permite desarrollar aplicaciones web analíticas completamente interactivas escribiendo únicamente código Python. Esto facilitó enormemente la integración con el backend existente, el acceso a los datos almacenados en TimescaleDB y el uso de librerías de análisis y visualización de Python (como Pandas y Plotly) para generar los gráficos interactivos (líneas de tiempo, resúmenes diarios/semanales, histogramas, etc.) que componen el dashboard del cuidador. Se utilizan diversos componentes gráficos de Plotly para renderizar las visualizaciones de forma clara y eficiente. Esta aproximación permitió concentrar el esfuerzo de desarrollo dentro del ecosistema Python, minimizando la necesidad de mantener un stack tecnológico separado (HTML, CSS, JavaScript y un framework como React/Vue/Angular) exclusivamente para el frontend.

2.4. Consideraciones Éticas y Legales (RGPD)

El tratamiento de datos personales, y muy especialmente los datos relativos a la salud, está estrictamente regulado en la Unión Europea por el **Reglamento General de Protección de Datos (RGPD)** - Reglamento (UE) 2016/679 Unión Europea [2016]. Los datos recogidos por dispositivos como Fitbit® (frecuencia cardíaca, sueño, actividad) son considerados datos relativos a la salud y, por tanto, pertenecen a las categorías especiales de datos según el Artículo 9 del RGPD, cuyo tratamiento está prohibido por defecto salvo excepciones específicas, siendo la más relevante en este contexto el **consentimiento explícito** del interesado (el usuario del dispositivo Fitbit).

El diseño e implementación de este sistema ha tenido en cuenta los principios fundamentales del RGPD Agencia Española de Protección de Datos (AEPD):

- **Licitud, lealtad y transparencia:** Obtener el consentimiento explícito e informado del usuario final (dueño de la cuenta Fitbit) antes de acceder a sus datos, explicando claramente qué datos se recogerán, para qué finalidad (monitorización de apoyo al cuidado) y quién tendrá acceso a ellos (personal autorizado de la residencia/cuidadores designados). El flujo OAuth 2.0, donde el usuario aprueba los permisos en la plataforma de Fitbit, ayuda a instrumentalizar parte de este consentimiento.

- **Limitación de la finalidad:** Los datos recogidos solo deben usarse para el propósito específico informado al usuario y no para otros fines incompatibles.

- **Minimización de datos:** Recoger y procesar únicamente los datos que sean estrictamente necesarios para cumplir con la finalidad declarada (ej. no recoger datos de localización si no son necesarios para el servicio).

- **Exactitud:** Poner medios razonables para asegurar que los datos sean exactos (reflejando lo que proporciona Fitbit) y ofrecer mecanismos para que el usuario o su representante puedan solicitar su rectificación si detectan errores.

- **Limitación del plazo de conservación:** Establecer plazos de conservación claros para los datos recogidos (ej. mantener datos detallados por X meses, luego solo resúmenes) y borrarlos o anonimizarlos de forma segura cuando ya no sean necesarios para la finalidad.
- **Integridad y confidencialidad (Seguridad):** Aplicar medidas técnicas y organizativas apropiadas para garantizar la seguridad de los datos, protegiéndolos contra el acceso no autorizado (controles de acceso basados en roles para el personal), la pérdida o la destrucción (ej. uso obligatorio de HTTPS, almacenamiento seguro de credenciales de API y tokens OAuth, cifrado de datos sensibles en la base de datos si aplica, copias de seguridad).
- **Responsabilidad proactiva (Accountability):** Poder demostrar el cumplimiento de los principios anteriores (ej. manteniendo registros de auditoría de accesos y consentimientos, teniendo documentadas las políticas de privacidad y seguridad).

Además, se deben respetar los derechos de los interesados recogidos en el RGPD, como el derecho de acceso, rectificación, supresión ("derecho al olvido"), limitación del tratamiento, portabilidad de los datos y oposición. El diseño del sistema (actual y futuro) debe contemplar mecanismos para facilitar el ejercicio de estos derechos por parte de los usuarios o sus representantes legales.

Capítulo 3

Requisitos y Metodología

Este capítulo detalla los requisitos que debe cumplir el sistema desarrollado y la metodología seguida durante su construcción. Los requisitos se dividen en funcionales, que describen las capacidades del sistema, y no funcionales, que especifican sus atributos de calidad.

3.1. Requisitos Funcionales

Los requisitos funcionales (RF) definen las tareas y servicios específicos que el sistema de monitorización debe ser capaz de realizar. Han sido identificados a partir de los objetivos del proyecto definidos en el Capítulo 1 y las necesidades del escenario de uso previsto (personal autorizado monitorizando a residentes/ancianos). A continuación, se enumeran los requisitos funcionales clave implementados en el prototipo:

RF-01: Autenticación Compartida de Personal El sistema debe proveer un mecanismo de inicio de sesión único para el personal autorizado, utilizando credenciales compartidas (usuario/contraseña) gestionadas a través de la configuración del sistema (variables de entorno). Debe permitir también cerrar la sesión. *(Nota: Se reconoce que un sistema multiusuario individual sería preferible en producción, ver Capítulos 7 y* **??***).*

RF-02: Gestión de Vinculaciones Fitbit®-Nombre El sistema debe permitir al personal autorizado:

- Visualizar la lista de cuentas de email de Fitbit® disponibles para vincular (obtenidas de la configuración o base de datos inicial).
- Asociar un nombre identificativo (proporcionado por el personal) a una cuenta de email de Fitbit® durante el proceso de vinculación inicial o al reasignar un dispositivo/email ya existente.
- Visualizar la lista de cuentas actualmente vinculadas, mostrando el nombre asociado y el email de Fitbit®.

(Nota: No se implementa una gestión completa de perfiles de residentes independiente de la vinculación, ver Capítulos **??** *y* **??***).*

RF-03: Vinculación de Cuentas Fitbit® (OAuth 2.0) Tras seleccionar un email y asociarle un nombre (nuevo o reasignado), el sistema debe gestionar de forma transparente y segura el flujo de autorización OAuth 2.0 (*Authorization Code Grant*) con Fitbit®, redirigiendo al usuario a Fitbit para la autenticación y autorización de permisos, y manejando el callback para obtener los tokens.

RF-04: Gestión de Reasignación El sistema debe permitir reasignar una cuenta de email de Fitbit® (que ya podría estar asociada a un nombre anterior) a un nuevo nombre identificativo, gestionando la posible necesidad de reautorización si los tokens no son válidos. *(Nota: La funcionalidad explícita para "desvincular.º revocar tokens de una asociación sin reasignarla no está implementada en esta versión).*

RF-05: Adquisición Automática de Datos El sistema debe ser capaz de obtener periódicamente (con una frecuencia determinada por la configuración del planificador APScheduler) los datos de salud y actividad disponibles (frecuencia cardíaca, patrones de sueño, pasos) de todas las cuentas Fitbit® activamente vinculadas y con tokens válidos.

RF-06: Almacenamiento de Datos Temporales El sistema debe persistir de forma estructurada los datos adquiridos de Fitbit® en la base de datos de series temporales (TimescaleDB), asegurando que cada dato quede asociado al email y nombre correspondientes y conserve su información temporal (timestamp).

RF-07: Procesamiento Básico de Datos El sistema debe realizar un procesamiento mínimo sobre los datos crudos recibidos de la API antes de su almacenamiento o visualización, como la validación de formato y cálculo del tiempo total de sueño, extracción de pasos totales diarios, etc.

RF-08: Visualización de Datos Históricos (Pendiente) El sistema debe ofrecer un panel de visualización (dashboard) accesible vía web para el personal autorizado. Este panel permitirá seleccionar un residente (por su nombre/email asociado) y mostrar de forma clara e intuitiva sus datos históricos (frecuencia cardíaca, sueño, actividad) mediante representaciones gráficas interactivas y/o tablas resumen. *(Nota: La implementación de esta interfaz de visualización está planificada pero no completada en la fase actual descrita en este documento).*

RF-09: Gestión Segura de Tokens El sistema debe implementar mecanismos seguros para el almacenamiento y la gestión del ciclo de vida (obtención, uso, refresco, manejo de errores en la revocación) de los tokens de acceso y refresco de OAuth 2.0 obtenidos de Fitbit®. *(Nota: Los detalles específicos sobre el cifrado en reposo de los tokens se describen en el Capítulo 5).*

Estos requisitos funcionales constituyen la base sobre la cual se ha diseñado e implementado la funcionalidad del prototipo actual.

3.2. Requisitos No Funcionales

Además de las funciones que debe realizar, el sistema debe cumplir ciertos atributos de calidad y restricciones operativas, conocidos como Requisitos No Funcionales (RNF). Estos requisitos definen *cómo* debe operar el sistema. Para este proyecto, se han considerado los siguientes RNF clave:

RNF-01: Usabilidad (Interfaz de Personal) La interfaz web destinada al personal autorizado debe ser intuitiva y fácil de usar, especialmente en las tareas críticas como la vinculación/reasignación de dispositivos y la visualización de datos (cuando esté implementada). Los mensajes de error deben ser claros y orientativos.

RNF-02: Rendimiento (Adquisición y Almacenamiento) El proceso de adquisición de datos (tanto diario como intradía) debe ejecutarse eficientemente para respetar los límites de la API de Fitbit® y procesar los datos de múltiples usuarios sin demoras excesivas. La escritura en la base de datos PostgreSQL/TimescaleDB debe ser eficiente. El rendimiento de lectura para el dashboard (cuando se implemente) debe ser adecuado para una experiencia de usuario fluida.

RNF-03: Seguridad

- Autenticación: El acceso a la aplicación web debe estar protegido mediante autenticación (actualmente compartida, ver RF-01).
- Autorización: Asegurar que solo el personal autenticado pueda realizar acciones o ver datos.
- Gestión de Tokens: Los tokens OAuth 2.0 deben almacenarse cifrados en la base de datos y transmitirse de forma segura (ver RF-09 y Sección 4.4).
- Comunicaciones: Toda la comunicación sensible (login, callbacks OAuth, llamadas API backend) debe realizarse obligatoriamente sobre HTTPS (especialmente en un entorno de producción).
- Protección Web: El sistema debe tener en cuenta las buenas prácticas de seguridad web para prevenir vulnerabilidades comunes (ej. validación de entradas, uso de sesiones seguras, configuración de cabeceras HTTP apropiadas), siguiendo recomendaciones como las del OWASP Top 10 OWASP Foundation [2021].

RNF-04: Fiabilidad y Disponibilidad El sistema debe ser razonablemente fiable. La ejecución programada de los scripts de adquisición mediante ‘cron‘ debe ser robusta. Los scripts y la aplicación web deben manejar correctamente errores esperables (ej. fallos de red, errores de la API Fitbit, errores de BD) registrando la información relevante para diagnóstico sin detener por completo el servicio.

RNF-05: Mantenibilidad El código fuente está organizado en módulos Python (‘app.py‘, ‘auth.py‘, ‘db.py‘, ‘fitbit.py‘, etc.), es legible y está comentado. Se utiliza el sistema de control de versiones Git, con el repositorio alojado en GitHub Moreno Muñoz [2025], para gestionar los cambios y facilitar la colaboración o futuras revisiones.

RNF-06: Escalabilidad (Diseño) La arquitectura (aplicación Flask modular, scripts independientes de adquisición, base de datos PostgreSQL/TimescaleDB) proporciona una base que podría escalarse (ej. ejecutando más instancias de los scripts de adquisición, escalando la base de datos) si fuera necesario manejar un mayor volumen de usuarios o datos en el futuro.

RNF-07: Privacidad (Cumplimiento RGPD) El sistema se ha diseñado e implementado siguiendo los principios del RGPD, como se detalló en la Sección 2.4, incluyendo el cifrado de tokens y la gestión del consentimiento implícita en el flujo OAuth.

(Nota: La sección 3.3 sobre Casos de Uso se omite en esta versión para mayor brevedad).

3.3. Metodología de Desarrollo

El desarrollo de este Trabajo Fin de Grado se ha abordado siguiendo un enfoque principalmente **iterativo e incremental**, adaptado a la naturaleza exploratoria y de creación de prototipos propia de un proyecto académico de este tipo. No se siguió estrictamente una metodología ágil formal como Scrum, pero se adoptaron algunos de sus principios, como la flexibilidad ante cambios y la entrega de valor funcional en ciclos cortos.

Las fases principales del desarrollo se pueden resumir en:

1. **Investigación y Definición (Fase Inicial):**
   - Revisión bibliográfica sobre monitorización remota, wearables y tecnologías relevantes.
   - Estudio detallado de la documentación de la API de Fitbit® y el protocolo OAuth 2.0 con PKCE.
   - Definición inicial de los objetivos y alcance del proyecto en colaboración con el tutor.
   - Identificación de los requisitos funcionales y no funcionales preliminares.
2. **Diseño de la Arquitectura y Tecnologías:**
   - Toma de decisiones sobre la arquitectura general (aplicación web Flask, scripts Python independientes para adquisición, base de datos PostgreSQL con posible uso de TimescaleDB para métricas, scheduler externo ‘cron‘).
   - Selección de las tecnologías principales (Python, Flask, psycopg2, cryptography, etc.).
   - Diseño del esquema de la base de datos (tabla ‘users‘ y estructura pensada para tablas de métricas) y las interfaces entre componentes (rutas Flask, funciones en módulos Python).
3. **Implementación Iterativa (Ciclos de Desarrollo):**
   - Desarrollo incremental de la funcionalidad principal, priorizando los módulos clave:
   - Implementación del flujo de autenticación OAuth 2.0 con Fitbit® (‘auth.py‘).
   - Desarrollo del módulo de base de datos (‘db.py‘) incluyendo cifrado de tokens (‘encryption.py‘).
   - Creación de la aplicación web Flask (‘app.py‘) con las rutas para la gestión de vinculaciones y autenticación del personal.
   - Desarrollo de los scripts independientes para la adquisición de datos diarios (‘fitbit.py‘) e intradía (‘fitbit_intraday.py‘).
   - Configuración de la ejecución programada mediante ‘cron‘ y los scripts ‘.sh‘.
   - *(Pendiente/Realizado)* Implementación de la interfaz de visualización (Dashboard con Dash).
   - Realización de pruebas funcionales manuales y depuración durante el desarrollo.

4. **Pruebas y Validación:**
    - Ejecución de pruebas sobre el prototipo desplegado (en VM) para verificar el cumplimiento de los requisitos implementados (vinculación, adquisición, almacenamiento básico).
    - Pruebas del flujo completo de vinculación y adquisición programada.
    - Depuración y corrección de errores encontrados.
5. **Documentación:**
    - Redacción de la memoria del TFG (este documento).
    - Comentarios en el código fuente.
    - Elaboración de diagramas y esquemas necesarios.

Para la gestión del código fuente y el control de versiones se utilizó **Git**, alojando el repositorio centralizado en la plataforma **GitHub** Moreno Muñoz [2025], lo que permitió un seguimiento detallado de los cambios y la posibilidad de colaboración. La gestión de tareas se realizó mediante seguimiento personal y comunicación con el tutor.

Capítulo 4

Diseño y Arquitectura del Sistema

En este capítulo se describe la arquitectura software global del sistema de monitorización y se detallan las decisiones de diseño clave tomadas para cada uno de sus componentes principales, basándose en la implementación realizada y disponible en el repositorio del proyecto Moreno Muñoz [2025]. El diseño busca satisfacer los requisitos funcionales y no funcionales definidos en el capítulo anterior, con especial énfasis en la modularidad, la seguridad y la gestión eficiente de los datos.

4.1. Arquitectura General

El sistema se ha diseñado como una aplicación web con un backend que interactúa con una base de datos PostgreSQL (donde se habilita la extensión TimescaleDB para las tablas de métricas) y la API externa de Fitbit®. La adquisición de datos se realiza mediante scripts Python independientes ejecutados periódicamente por el planificador del sistema operativo ('cron'). La figura 4.1 ilustra esta arquitectura.

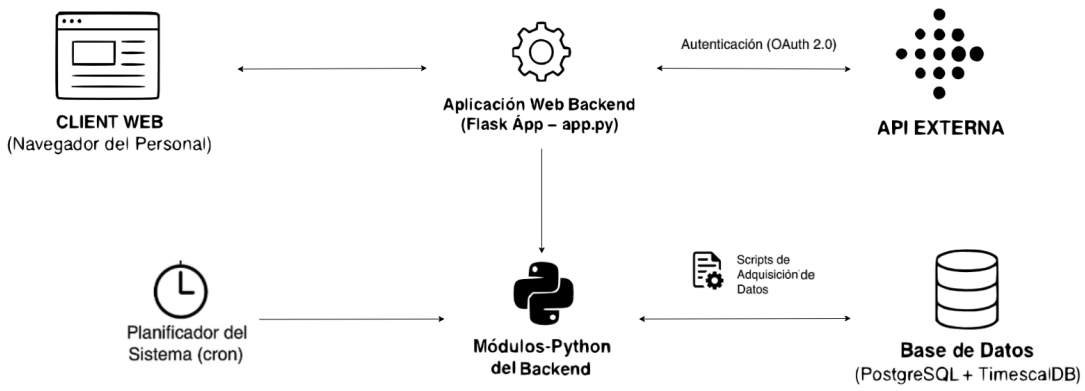

Figura 4.1: Arquitectura General del Sistema de Monitorización.

(Descripción conceptual de la Figura 4.1):

- **Cliente Web (Navegador del Personal):** Interfaz de usuario HTML/CSS/JavaScript renderizada por Flask (para login/vinculación) y por Dash (para el dashboard), accesible vía navegador.
- **Aplicación Web Backend (Flask App - `app.py`):** Punto de entrada principal para las interacciones del personal. Gestiona autenticación, interfaz/lógica de vinculación, y sirve la aplicación Dash/API para el dashboard.
- **Módulos Python del Backend:** Componentes lógicos como `auth.py`, `db.py`, `encryption.py` utilizados por la app Flask y los scripts.

- **Scripts de Adquisición de Datos (`fitbit.py`, `fitbit_intraday.py`):** Procesos Python independientes para obtener datos diarios e intradía de la API Fitbit, manejando tokens y almacenamiento.
- **Planificador del Sistema ('cron'):** Utilidad del sistema operativo que ejecuta periódicamente los scripts de adquisición (`.sh`).
- **Base de Datos (PostgreSQL + TimescaleDB):** Instancia PostgreSQL con tabla relacional `users` (configuración, tokens cifrados) y hipertables TimescaleDB para datos de series temporales.
- **API Externa de Fitbit®:** Servicio externo que provee los datos y maneja la autorización OAuth 2.0.

Esta arquitectura desacopla la interacción del usuario (gestionada por Flask) de la adquisición de datos (gestionada por scripts y 'cron').

4.2. Diseño del Backend (Aplicación Flask y Scripts)

El backend se compone de dos partes principales que interactúan a través de la base de datos:

1. **Aplicación Web Flask (`app.py`):** Actúa como el servidor web principal y gestiona las interacciones síncronas del personal. Utiliza Flask-Login para la autenticación (compartida). Sus responsabilidades clave son:
   - Servir las páginas HTML para el login, la selección de email, la asignación de nombre y las confirmaciones (usando plantillas Jinja2).
   - Gestionar el flujo OAuth 2.0: generar parámetros ('state', 'code_challenge'), construir la URL de autorización, manejar la redirección del usuario a Fitbit® y procesar el 'callback'.
   - Interactuar con `auth.py` para obtener los tokens a partir del código de autorización.
   - Interactuar con `db.py` y `encryption.py` para guardar/actualizar la información del usuario y los tokens cifrados en la tabla `users`.
   - Servir la aplicación Dash que constituye el dashboard.
2. **Scripts de Adquisición de Datos (`fitbit.py`, `fitbit_intraday.py`):** Son procesos Python independientes ejecutados por 'cron'. Cada script típicamente realiza un bucle sobre los usuarios activos recuperados de la base de datos (`db.py`) y para cada uno:
   - Obtiene y descifra los tokens (`encryption.py`, `db.py`).
   - Verifica la validez del token de acceso (`expires_at`). Si es necesario, intenta refrescarlo usando el token de refresco (`auth.py`, `fitbit.py`) y actualiza los tokens cifrados y la expiración en la BD (`db.py`).
   - Si los tokens son válidos, realiza las llamadas correspondientes a la API de Fitbit® (`fitbit.py`, `fitbit_intraday.py`).

- Procesa la respuesta JSON.
- Se conecta a la BD (`db.py`) para insertar los datos procesados en las tablas/hipertablas de TimescaleDB apropiadas.
- Maneja errores durante el proceso (ej. logueando el error y continuando con el siguiente usuario).

Esta separación permite que la adquisición de datos no bloquee la aplicación web y pueda ejecutarse en segundo plano de forma fiable.

4.3. Diseño de la Base de Datos (PostgreSQL + TimescaleDB)

Se utiliza una única base de datos **PostgreSQL**, aprovechando la extensión **TimescaleDB** para optimizar el manejo de datos de series temporales. El diseño se basa en una tabla relacional para la gestión de usuarios y vinculaciones, y un conjunto de hipertables para los datos temporales.

- **Tabla `users`:** Almacena la información de las vinculaciones Fitbit® y las credenciales de acceso a la API. Se trata de una tabla relacional estándar en PostgreSQL. Contiene información como el nombre asociado, el email de la cuenta Fitbit®, los tokens cifrados, su fecha de expiración y los permisos concedidos. El esquema detallado con las sentencias SQL `CREATE TABLE` e `CREATE INDEX` se puede encontrar en el **Anexo A**.
- **Hipertablas TimescaleDB:** Para almacenar eficientemente los grandes volúmenes de datos de series temporales (intradía, resúmenes diarios, sueño), se utiliza la extensión TimescaleDB sobre PostgreSQL. Se definen estructuras específicas para cada tipo principal de dato, optimizadas para consultas basadas en tiempo. Se propone la creación de hipertables como `intraday_metrics`, `daily_summaries` y `sleep_logs`, particionadas por la dimensión temporal (`time`, `date` o `start_time`). Las definiciones SQL detalladas (`CREATE TABLE`, `create_hypertable()`, índices recomendados) para estas hipertables se encuentran también en el **Anexo A**. El uso de hipertables es fundamental para el rendimiento de la ingesta y consulta de estos datos.
- **Cifrado:** Conforme a los requisitos de seguridad y privacidad, los tokens de acceso y refresco en la tabla `users` se almacenan cifrados. Se utiliza cifrado simétrico mediante la librería `cryptography.fernet` (`encryption.py`), empleando una clave secreta (`ENCRYPTION_KEY`) gestionada externamente (variable de entorno). Esto proporciona una capa esencial de seguridad para proteger estas credenciales en reposo.

4.4. Diseño de la Integración con Fitbit

La integración con la API de Fitbit® se ha diseñado priorizando la seguridad y la robustez:

- **OAuth 2.0 con PKCE:** Se implementa el flujo Authorization Code Grant con PKCE (`auth.py`, `app.py`) para obtener los tokens iniciales de forma segura, utilizando 'state' contra CSRF y 'code_verifier'/`code_challenge` contra interceptación del código de autorización.
- **Gestión de Tokens Segura:**
  - Los tokens (acceso y refresco) se cifran inmediatamente después de obtenerlos y antes de almacenarlos en la base de datos (`db.py`, `encryption.py`).
  - Solo se descifran en memoria en el momento necesario para realizar una llamada a la API o para el proceso de refresco (`fitbit.py`, `fitbit_intraday.py`).
  - La clave de cifrado (`ENCRYPTION_KEY`) se gestiona de forma segura fuera del código fuente (variable de entorno).
- **Lógica de Refresco Robusta (`fitbit.py`):**
  - Antes de cada intento de acceso a datos protegidos, se comprueba la fecha de expiración (`expires_at`) del token de acceso.
  - Si está caducado o próximo a caducar, se utiliza el token de refresco (descifrado) para solicitar un nuevo par de tokens al endpoint de refresco de Fitbit.
  - Si el refresco tiene éxito, los nuevos tokens se cifran y se actualizan inmediatamente en la base de datos junto con la nueva `expires_at`.
  - Si el refresco falla (ej. refresh token inválido o revocado por el usuario), se registra el error y no se intenta acceder a los datos. La vinculación podría requerir intervención manual.
- **Llamadas a la API Controladas:**
  - Se utilizan los tokens de acceso descifrados y vigentes en la cabecera `Authorization`.
  - Se gestionan los códigos de estado HTTP de respuesta (200, 401, 403, 429, 5xx) para reintentar (si aplica), registrar errores o marcar problemas con la vinculación.
  - Se tiene en cuenta el versionado de la API de Fitbit (ej. `/1/user/...` vs `/1.2/user/...` para sueño).
  - Se configuran timeouts adecuados en las peticiones HTTP.

4.5. Diseño del Pipeline de Datos

El flujo de datos principal, desde la fuente hasta la visualización, sigue los siguientes pasos diseñados e implementados:

1. **Se ejecuta ('cron'):** El planificador del sistema operativo ejecuta los scripts `.sh` (`run_fitbit.sh`, `run_fitbit_intraday.sh`) según la frecuencia definida.
2. **Ejecución del Script Python:** El script (`fitbit.py` o `fitbit_intraday.py`) se inicia.

3. **Obtención de Usuarios y Credenciales:** El script consulta la tabla `users` para obtener la lista de emails activos y sus tokens cifrados correspondientes.
4. **Procesamiento por Usuario (Bucle):** Para cada usuario:
   a) Descifra los tokens necesarios (`encryption.py`).
   b) Valida/Refresca el token de acceso llamando a la lógica en `fitbit.py` (que actualiza la BD si es necesario). Si hay error irrecuperable, registra y salta al siguiente usuario.
   c) Realiza las llamadas a la API de Fitbit® para obtener los datos (diarios o intradía) desde la última recogida registrada (requiere almacenar este punto de referencia).
   d) Valida y procesa la respuesta JSON (RF-07).
   e) Inserta los datos procesados en las hipertables TimescaleDB (`db.py`). Registra éxito o error.
5. **Consulta para Visualización (Flask/Dash):**
   a) El personal interactúa con el Dashboard (aplicación Dash).
   b) Las callbacks de Dash (en Python) ejecutan consultas SQL directamente sobre las hipertables TimescaleDB (filtrando por `user_email`, `time`, `type`).
   c) Los datos recuperados se procesan (ej. con Pandas) para adaptarlos al formato requerido por Plotly.
   d) Los gráficos de Plotly se generan y se envían al navegador para su renderización.

Este diseño utiliza 'cron' para la orquestación de la ingesta y PostgreSQL/TimescaleDB como punto central de almacenamiento e intercambio de información entre los componentes de adquisición y visualización.

4.6. Diseño de la Interfaz de Usuario

La interfaz de usuario (UI) se compone de dos partes diferenciadas:

- **Interfaz de Gestión y Vinculación (Flask/HTML):**
  - Implementada mediante plantillas Jinja2 renderizadas por Flask (`app.py`).
  - Incluye las páginas: `login.html`, `link_device.html`, `assign_user.html`, `reassign_confirm` `confirmation.html`, `link_auth.html`.
  - Utiliza formularios HTML para la interacción.
  - El estilo visual es básico, basado en HTML estándar con mínimo CSS.
- **Panel de Visualización (Dashboard - Dash/Plotly):**
  - Implementado como una aplicación Dash, servida a través de la aplicación Flask principal.

- **Layout:** Definido en Python (`app.py` o un módulo `dashboard.py`) usando `dash_html_components` y `dash_core_components`. Estructura clara con título, controles de selección y área de gráficos.
- **Controles Interactivos:**
  - `dcc.Dropdown` para seleccionar el residente (poblado con nombres/emails de la tabla `users`).
  - `dcc.DatePickerRange` para seleccionar el intervalo de fechas a visualizar.
- **Visualizaciones (`dcc.Graph` con Plotly)**:
  - Gráfico de línea temporal para frecuencia cardíaca (mostrando datos intradía si están disponibles para el rango seleccionado, o promedio diario para rangos largos).
  - Gráfico de barras para pasos diarios.
  - Gráfico de barras apiladas (o similar) para mostrar la duración en las diferentes fases del sueño (REM, ligero, profundo, despierto) por noche.
  - *(Opcional)* Otros gráficos relevantes (minutos activos, calorías, etc.).
  - *(Opcional)* `dash_table.DataTable` para resúmenes numéricos.
- **Interactividad (Callbacks):** Funciones Python decoradas con `@app.callback` que se disparan al cambiar la selección del residente o las fechas. Estas funciones consultan los datos necesarios de TimescaleDB, los procesan si es necesario (ej. con Pandas), y actualizan las propiedades `figure` de los gráficos correspondientes. *'**********Esto revisarlo cuando lo hagooooo**********'*

El diseño busca ofrecer una herramienta funcional y clara, separando la gestión administrativa de la visualización de datos de monitorización.

Capítulo 5

Implementación

Este capítulo detalla el proceso de construcción del sistema de monitorización remota, materializando el diseño arquitectónico expuesto en el Capítulo 4 en componentes software funcionales. Se aborda la configuración del entorno, la implementación de los componentes del backend (aplicación web y scripts de adquisición), la configuración y manejo de la base de datos, la lógica de procesamiento de datos, las medidas de seguridad aplicadas y la interfaz de visualización. Finalmente, se comentan algunos de los desafíos técnicos encontrados. El código fuente completo está disponible en el repositorio del proyecto [Moreno Muñoz, 2025].

5.1. Entorno de Desarrollo y Tecnologías

La implementación del prototipo se ha realizado utilizando un conjunto de tecnologías seleccionadas por su adecuación a los requisitos del proyecto. El lenguaje de programación principal es **Python** (versión 3.x) [Python Software Foundation], aprovechando su flexibilidad y la riqueza de sus librerías para desarrollo web y procesamiento de datos.

Las herramientas y librerías clave empleadas son:

- **Control de Versiones:** Se utilizó **Git** para el control de versiones, alojando el repositorio en **GitHub** [Moreno Muñoz, 2025].
- **Backend Web: Flask** [Pallets Projects] como microframework para construir la aplicación web principal (`app.py`) y gestionar las rutas, peticiones y respuestas HTTP. Se complementa con **Flask-Login** para la gestión de sesiones y autenticación del personal.
- **Base de Datos: PostgreSQL** [PostgreSQL Global Development Group] como sistema gestor de base de datos relacional, extendido con **TimescaleDB** [Timescale] para la gestión optimizada de datos de series temporales. La interacción desde Python se realiza mediante la librería **psycopg2** (o `psycopg2-binary`) [Psycopg Team].
- **Seguridad:** La librería **cryptography** [Python Cryptographic Authority] se emplea para el cifrado simétrico (Fernet) de los tokens OAuth 2.0 almacenados en la base de datos.
- **Comunicaciones API:** La librería **requests** [Reitz] se utiliza para realizar las llamadas HTTP a la API web de Fitbit.
- **Visualización: Dash**: Construido sobre Flask, Plotly.js y React.js, permite crear el dashboard interactivo utilizando únicamente Python. **Plotly** [Plotly] es la librería subyacente para la generación de los gráficos.

- **Manipulación de Datos:** (*********Opcional pero frecuente con Dash**********) **Pandas** [The Pandas Development Team] para la manipulación eficiente de los datos recuperados de la base de datos antes de visualizarlos.
- **Planificación de Tareas:** El sistema **cron** del sistema operativo se utiliza para lanzar los scripts de adquisición de datos (`fitbit.py`, `fitbit_intraday.py`) a intervalos regulares, mediante los scripts wrapper `run_fitbit.sh` y `run_fitbit_intraday.sh`.

5.2. Implementación del Backend

El backend comprende la aplicación web Flask, que gestiona la interacción con el personal, y los scripts independientes que realizan la adquisición de datos de forma asíncrona. Aunque no se implementó una arquitectura de microservicios estricta (servicios independientes comunicados por red [Fowler and Lewis, 2014, Newman, 2021]), la separación en una aplicación web y scripts autónomos para tareas de fondo proporciona una modularidad similar.

5.2.1. Aplicación Web Flask (`app.py`)

La aplicación Flask centraliza las siguientes funcionalidades:

- **Autenticación del Personal:** Gestiona el inicio y cierre de sesión del personal autorizado utilizando Flask-Login y comparando las credenciales proporcionadas con las configuradas de forma segura (ej. variables de entorno).
- **Gestión de Vinculaciones:** Proporciona las rutas y plantillas HTML (renderizadas con Jinja2) para que el personal visualice las cuentas de Fitbit disponibles, asocie un nombre a un email de Fitbit, y vea las vinculaciones activas.
- **Orquestación del Flujo OAuth 2.0:** Inicia el proceso de autorización redirigiendo al usuario a Fitbit (construyendo la URL con parámetros como `client_id`, `scope`, `response_type`, `redirect_uri`, `state`, y `code_challenge` para PKCE, según especificación OAuth 2.0 [Hardt, 2012]). Procesa la respuesta de Fitbit en la ruta de callback, extrayendo el código de autorización y el estado. Llama a las funciones del módulo `auth.py` para intercambiar el código por los tokens. Almacena los tokens cifrados y la información del usuario asociado mediante el módulo `db.py`.
- **Servir el Dashboard Dash:** Actúa como servidor para la aplicación Dash, montándola en una ruta específica (ej. `/dashboard`) y protegiendo el acceso mediante el decorador `@login_required` de Flask-Login.

Referencia de Código: véase Anexo B.1

5.2.2. Scripts de Adquisición (`fitbit.py`, `fitbit_intraday.py`)

Estos scripts operan de forma independiente, ejecutados por `cron`:

- **Iteración sobre Usuarios:** Recuperan la lista de usuarios vinculados y sus credenciales cifradas de la base de datos consultando la tabla `users` a través de `db.py`.

- **Gestión de Tokens:** Para cada usuario, descifran los tokens y gestionan su ciclo de vida: comprueban la expiración del token de acceso y, si es necesario, intentan refrescarlo utilizando el token de refresco y el endpoint correspondiente de la API de Fitbit. Los nuevos tokens obtenidos se cifran y actualizan en la base de datos. Se manejan los fallos en el refresco.
- **Llamadas a la API de Fitbit:** Utilizan el token de acceso válido para solicitar los datos de actividad, sueño o frecuencia cardíaca a los endpoints RESTful de la API de Fitbit, especificando el usuario, el periodo de tiempo y el formato deseado (JSON), utilizando la librería `requests`. Se implementa manejo básico de códigos de estado HTTP (200, 401, 403, 429).
- **Persistencia de Datos:** Una vez obtenidos y procesados los datos (ver Sección 5.4), los scripts llaman a las funciones de inserción del módulo `db.py` (ej. `insert_intraday_metrics`, `insert_daily_summary`, `insert_sleep_log`) para almacenarlos en las hipertablas TimescaleDB correspondientes.

La ejecución mediante `cron` asegura la recogida periódica y automatizada de datos sin intervención manual. La configuración de `cron` se muestra en el Anexo C.1. *Referencia de Código: véase Anexos B.5 y B.6*

5.3. Implementación de la Persistencia (Base de Datos)

La capa de persistencia se basa en PostgreSQL con la extensión TimescaleDB, gestionada a través del módulo `db.py` utilizando `psycopg2`.

5.3.1. Configuración Inicial

Como se describió en la Sección 5.1, el primer paso es habilitar la extensión TimescaleDB en la base de datos PostgreSQL (‘CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;‘). Posteriormente, se crean las tablas según el esquema del Anexo **??** (asumiendo etiqueta ‘anexo:db_schema‘), convirtiendo `intraday_metrics`, `daily_summaries` y `sleep_logs` en hipertablas mediante ‘SELECT create_hypertable(...)‘. Esta conversión es transparente para las consultas SQL estándar pero optimiza internamente el almacenamiento y las consultas basadas en tiempo [DB-Engines].

5.3.2. Módulo `db.py`

Este módulo abstrae las interacciones con la base de datos:

- **Conexión:** La función `connect_to_db()` establece la conexión usando `psycopg2`, obteniendo las credenciales de forma segura.
- **Gestión de Usuarios y Tokens:** Incluye funciones para crear/actualizar usuarios en la tabla `users`, almacenar los tokens cifrados (obtenidos de `encryption.py`, basado en `cryptography`), recuperar usuarios y sus tokens, etc. Es fundamental asegurar que los tokens se manejen siempre en formato cifrado excepto en el momento preciso de su uso.

- **Funciones de Inserción:** Se implementaron funciones como `insert_intraday_metrics` (ver Anexo C.2), `insert_daily_summary`, y `insert_sleep_log`. Estas funciones reciben los datos procesados por los scripts de adquisición y utilizan preferentemente métodos eficientes como `execute_values` de `psycopg2.extras` para inserciones masivas. Se define una estrategia para manejar conflictos (ej. `ON CONFLICT DO NOTHING`).
- **Funciones de Consulta:** Funciones como `get_hr_data` (ver Anexo C.3), `get_steps_data`, `get_sleep_data`, y `get_linked_users` recuperan los datos necesarios para el dashboard o para la lógica de los scripts. Estas funciones construyen consultas SQL optimizadas para TimescaleDB (filtrando por tiempo y usuario) y devuelven los resultados en un formato útil (listas, diccionarios o DataFrames de Pandas).

El uso de TimescaleDB requirió asegurar que las columnas de tiempo (`time`, `date`, `start_time`) tuvieran el tipo adecuado (`TIMESTAMPTZ` o `DATE`) y que se crearan los índices apropiados para acelerar las consultas. *Referencia de Código: véase Anexo B.3*

5.4. Implementación de la Lógica de Procesamiento

Dentro de los scripts de adquisición (`fitbit.py`, `fitbit_intraday.py`), se implementa la lógica necesaria para transformar los datos crudos de la API de Fitbit en un formato adecuado para su almacenamiento y posterior visualización.

Las tareas clave de procesamiento incluyen:

- **Parsing de JSON:** Extraer los datos relevantes de las complejas estructuras JSON devueltas por la API de Fitbit. Esto implica navegar por diccionarios y listas anidadas.
- **Validación de Datos:** Comprobaciones básicas para asegurar que los datos recibidos tienen el formato esperado antes de intentar insertarlos.
- ************Formateo de Timestamps************:** Una de las tareas más críticas. La API de Fitbit puede devolver fechas y horas en diferentes formatos y, a menudo, sin información explícita de zona horaria o relativa a la zona horaria del usuario. Es fundamental convertir estas representaciones a objetos `datetime` de Python que sean "timezone-aware"(conscientes de la zona horaria), preferiblemente en UTC, antes de insertarlos en columnas `TIMESTAMPTZ` de PostgreSQL. Esto puede requerir obtener la zona horaria del perfil del usuario de Fitbit (si la API lo permite) o hacer suposiciones documentadas. Se utilizaron librerías como `datetime` y potencialmente `pytz` para este manejo.
- **Cálculos Derivados:** Realizar cálculos simples si es necesario, como sumar los minutos en diferentes fases del sueño para obtener el tiempo total dormido, o convertir unidades (ej. distancia).
- **Estructuración para Inserción:** Agrupar los datos procesados en listas de tuplas o diccionarios que coincidan exactamente con el orden y tipo de las columnas de las tablas/hipertablas destino en la base de datos.

Este procesamiento asegura la calidad y consistencia de los datos almacenados.

5.5. Implementación de la Seguridad (OAuth, RGPD)

La seguridad y el cumplimiento normativo (RGPD [Unión Europea, 2016]) fueron consideraciones centrales durante la implementación.

5.5.1. Autenticación y Autorización (OAuth 2.0)

La integración con Fitbit se implementó siguiendo las mejores prácticas de OAuth 2.0 [Hardt, 2012]:

- **Flujo Authorization Code con PKCE:** Se implementó este flujo, considerado el más seguro para aplicaciones web con backend [Lodderstedt et al., 2017]. El módulo `auth.py` y las rutas de Flask en `app.py` gestionan la generación del `code_verifier` y `code_challenge`, el parámetro `state` para prevenir CSRF, el intercambio del código de autorización por tokens, y el manejo seguro de `client_id` y `client_secret`.
- **Gestión Segura de Tokens:** Los tokens de acceso y refresco se consideran información altamente sensible. Se cifran inmediatamente después de su obtención utilizando cifrado simétrico (AES mediante Fernet en la librería `cryptography`) con una clave secreta gestionada externamente (variable de entorno, no en el código fuente). Solo se descifran en memoria en el momento exacto de su uso.
- **Refresco de Tokens:** La lógica para refrescar tokens caducados se implementó de forma robusta, manejando posibles errores y actualizando los tokens en la base de datos de forma atómica.
- **HTTPS:** Aunque la configuración de HTTPS es a nivel de despliegue (servidor web/proxy inverso), el diseño asume y requiere que toda la comunicación (frontend-backend, backend-Fitbit API) se realice sobre HTTPS para proteger los datos en tránsito. Se recomienda seguir buenas prácticas de seguridad web como las delineadas por OWASP [OWASP Foundation, 2021].

Referencia de Código: véase Anexo B.2

5.5.2. Consideraciones RGPD

Se implementaron medidas técnicas y organizativas básicas alineadas con los principios del RGPD [Agencia Española de Protección de Datos (AEPD)]:

- **Consentimiento:** El flujo OAuth 2.0 actúa como mecanismo para obtener el consentimiento explícito del usuario (o su representante autorizado) para acceder a los datos de Fitbit. Los permisos (`scopes`) solicitados se limitan a los necesarios para la funcionalidad (minimización de datos).
- **Seguridad de Datos:** El cifrado de tokens en reposo y el uso de HTTPS en tránsito contribuyen a la integridad y confidencialidad. El control de acceso a la aplicación mediante login protege contra accesos no autorizados.
- **Minimización:** Solo se solicitan y almacenan los datos definidos en los requisitos (FC, pasos, sueño).

Es importante destacar que un cumplimiento completo del RGPD requeriría políticas de privacidad detalladas, mecanismos para ejercer los derechos ARSOPOL+ (Acceso, Rectificación, Supresión, Oposición, Portabilidad, Limitación), y posiblemente una Evaluación de Impacto relativa a la Protección de Datos (EIPD) en un entorno de producción real.

5.6. Implementación de la Visualización

La interfaz de visualización se desarrolló como un dashboard interactivo utilizando Dash, integrado en la aplicación Flask.

5.6.1. Layout del Dashboard

Se definió una estructura clara utilizando `dash_html_components` y `dash_core_components` (ahora `html` y `dcc` en versiones recientes de Dash), incluyendo:

- Título principal y secciones para controles y gráficos.
- Componentes interactivos: `dcc.Dropdown` para seleccionar el residente y `dcc.DatePickerRange` para el intervalo temporal.
- Contenedores `dcc.Graph` para mostrar las visualizaciones generadas con Plotly (frecuencia cardíaca, pasos, sueño).

Un ejemplo de la estructura del layout se encuentra en el Anexo C.4.

5.6.2. Callbacks para Interactividad

La funcionalidad dinámica se implementó mediante callbacks de Dash:

- **Población del Dropdown:** Un callback inicial consulta la base de datos (usando `db.get_linked_users`) para obtener la lista de residentes vinculados y actualizar las opciones del `dcc.Dropdown`.
- **Actualización de Gráficos:** El callback principal se activa con cambios en el dropdown de residente o en el selector de fechas. Este callback:
  1. Obtiene el email y las fechas seleccionadas.
  2. Llama a las funciones de consulta en `db.py` (`get_hr_data`, `get_steps_data`, etc.) para recuperar los datos de TimescaleDB.
  3. Procesa los datos recuperados (posiblemente con Pandas [The Pandas Development Team]).
  4. Genera las figuras de Plotly (`go.Figure`) para cada gráfico.
  5. Devuelve las figuras para actualizar los componentes `dcc.Graph` correspondientes.

Un ejemplo de este callback se muestra en el Anexo C.5. La implementación asegura que la visualización sea reactiva a las selecciones del personal.

5.7. Desafíos y Soluciones Técnicas

Durante la implementación surgieron diversos desafíos técnicos que requirieron soluciones específicas:

- **Manejo de Zonas Horarias:** Coordinar las zonas horarias entre la API de Fitbit (que puede usar la hora local del usuario), Python y PostgreSQL (que almacena `TIMESTAMPTZ` típicamente en UTC) fue complejo. La solución implicó intentar obtener la zona horaria del usuario desde Fitbit (si es posible) o asumir una por defecto, y convertir consistentemente todos los timestamps a UTC antes de almacenarlos en la base de datos, utilizando librerías como `datetime` y potencialmente `pytz` o el módulo `zoneinfo`.
- **Gestión de Límites de Tasa de la API (Rate Limiting):** La API de Fitbit [Fitbit Developer Portal] impone límites en el número de peticiones que una aplicación puede realizar en un periodo determinado. Aunque no se implementó un sistema sofisticado de gestión de caché o colas, la ejecución espaciada mediante `cron` y el procesamiento de usuarios de forma secuencial ayudaron a mitigar el riesgo de exceder los límites básicos. En un sistema con muchos usuarios, serían necesarias estrategias más avanzadas (ej. esperar y reintentar con backoff exponencial, caché de respuestas).
- **Complejidad de los Callbacks de Dash:** El callback principal que actualiza todos los gráficos puede volverse complejo y potencialmente lento si las consultas a la base de datos o el procesamiento de datos son costosos. Se intentó mantener las consultas eficientes (aprovechando TimescaleDB) y el procesamiento directo. Para dashboards más complejos, podrían explorarse técnicas como callbacks en paralelo (si aplica), almacenamiento en caché de resultados intermedios (`dcc.Store`), o incluso dividir en múltiples callbacks más pequeños.
- **Gestión Segura de Claves:** Asegurar que la clave de cifrado para los tokens (`ENCRYPTION_KEY`) y las credenciales de la API de Fitbit (`client_id`, `client_secret`) no se almacenen directamente en el código fuente fue crucial. La solución adoptada fue gestionarlas a través de variables de entorno, cargadas por la aplicación al inicio. En entornos de producción, se podrían usar sistemas de gestión de secretos más robustos.
- **Robustez de los Scripts de Adquisición:** Asegurar que un error al procesar un usuario (ej. token inválido, error inesperado de la API) no detuviera la adquisición para los demás usuarios. Esto se logró implementando bloques `try...except` alrededor del procesamiento de cada usuario individual y registrando los errores adecuadamente (utilizando el módulo `logging` de Python) para su posterior revisión.

Abordar estos desafíos fue esencial para lograr un prototipo funcional y razonablemente robusto.

Capítulo 6

Pruebas y Validación

6.1. Pruebas de Rendimiento

6.2. Estudio de Usabilidad con Usuarios Mayores

Capítulo 7

Resultados y Discusión

Capítulo 8

Conclusiones y Trabajo Futuro

Bibliografía

Bibliografía

Agencia Española de Protección de Datos (AEPD). Principios de protección de datos. URL https://www.aepd.es/es/derechos-y-deberes/conoce-los-principios-del-rgpd.

Rashid L. Bashshur, Charles R. Doarn, Julio M. Frenk, Joseph C. Kvedar, and James O. Woolliscroft. Telemedicine and the covid-19 pandemic, lessons for the future. *Telemedicine and e-Health*, 26(5):571–573, 2020. doi: 10.1089/tmj.2020.29040.rb.

DB-Engines. Db-engines ranking of time series dbms. URL https://db-engines.com/en/ranking/time+series+dbms. Ranking actualizado periódicamente. Accedido el 20-04-2025.

Fitbit. How does my fitbit device track my heart rate? URL https://help.fitbit.com/articles/en_US/Help_article/1565.htm. Accedido el 20-04-2025.

Fitbit Developer Portal. Fitbit web api reference. URL https://dev.fitbit.com/build/reference/web-api/. Accedido el 20-04-2025.

Fortune Business Insights. Wearable technology market size, share & covid-19 impact analysis ..., 2024. URL https://www.fortunebusinessinsights.com/wearable-technology-market-106000.

Martin Fowler and James Lewis. Microservices: a definition of this new architectural term. Mar 2014. URL https://martinfowler.com/articles/microservices.html. Artículo fundamental sobre microservicios. Accedido el 20-04-2025.

Google. Google completes fitbit acquisition.

Alex Grönholm. Apscheduler documentation. URL https://apscheduler.readthedocs.io/. Accedido el 20-04-2025.

Shadab Haghayegh, Soroush Khoshnevis, Michael H. Smolensky, Kenneth R. Diller, and Richard J. Castriotta. Accuracy of wristband fitbit models in assessing sleep: A comparison with polysomnography. *Journal of Clinical Sleep Medicine*, 15(10):1475–1481, 2019. doi: 10.5664/jcsm.7997.

D. (Editor) Hardt. The oauth 2.0 authorization framework, Oct 2012. URL https://www.rfc-editor.org/info/rfc6749.

InfluxData. Influxdb documentation. URL https://docs.influxdata.com/influxdb/. Accedido el 20-04-2025.

Instituto Nacional de Estadística (INE). Proyecciones de población de españa 2022-2072, 2022. URL https://www.ine.es/dyngs/INEbase/es/operacion.htm?c=Estadistica_C&cid=1254736176951&menu=ultiDatos&idp=1254735572981. Accedido el 20-04-2025.

Sang Lang Lee, Donghoon Lee, and Sungwan Kim. Challenges and opportunities of remote patient monitoring: A systematic review. *Healthcare*, 9(7):767, 2021. doi: 10.3390/healthcare9070767.

T. Lodderstedt, M. McGloin, and P. Hunt. Oauth 2.0 for native apps. RFC 8252, Oct 2017. URL https://www.rfc-editor.org/info/rfc8252.

Sumit Majumder, Tirthankar Mondal, and M. Jamal Deen. Wearable sensors for remote health monitoring. *Sensors*, 17(1):130, 2017. ISSN 1424-8220. doi: 10.3390/s17010130.

Pablo Moreno Muñoz. fitbit_project: Repositorio del proyecto tfg, 2025. URL https://github.com/morenopablo16/fitbit_project. Accedido el 20-04-2025.

M. Brennan Nelson, Leonard A. Kaminsky, D. Clark Dickin, and Alexander H. K. Montoye. Validity of consumer-based physical activity monitors for specific activity types. *Medicine & Science in Sports Exercise*, 48(8):1619–1628, 2016. doi: 10.1249/MSS.0000000000000941.

Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2nd edition, 2021.

Bordin Noah, Andrea Guidi, Giacomo Tenti, Alessandro Poli, Andrea Cenci, Marco Avvenuti, and Gigliola Vaglini. Mobile health and remote patient monitoring: A systematic review. *Sensors*, 22(24):9729, 2022. doi: 10.3390/s22249729.

Organización Mundial de la Salud (OMS). Envejecimiento y salud, Oct 2022. URL https://www.who.int/es/news-room/fact-sheets/detail/ageing-and-health. Accedido el 20-04-2025.

OWASP Foundation. Owasp top 10, 2021. URL https://owasp.org/www-project-top-ten/. Referencia estándar sobre riesgos de seguridad en aplicaciones web. Accedido el 21-04-2025.

Pallets Projects. Flask documentation. URL https://flask.palletsprojects.com/. Accedido el 20-04-2025.

Plotly. Dash documentation user guide. URL https://dash.plotly.com/. Accedido el 20-04-2025.

PostgreSQL Global Development Group. Postgresql documentation. URL https://www.postgresql.org/docs/. Consultar versión específica utilizada. Accedido el 22-04-2025.

Psycopg Team. Psycopg 2 documentation. URL https://www.psycopg.org/docs/. Accedido el 22-04-2025.

Python Cryptographic Authority. Cryptography documentation. URL https://cryptography.io/en/latest/. Accedido el 22-04-2025.

Python Software Foundation. Python.org. URL https://www.python.org/. Accedido el 20-04-2025.

Kenneth Reitz. Requests: Http for humans™. URL https://requests.readthedocs.io/en/latest/. Accedido el 22-04-2025.

The Pandas Development Team. pandas documentation. URL https://pandas.pydata.org/docs/. Accedido el 22-04-2025.

Timescale. Timescaledb documentation. URL https://docs.timescale.com/. Accedido el 20-04-2025.

Unión Europea. Reglamento (ue) 2016/679 del parlamento europeo y del consejo (reglamento general de protección de datos), Abr 2016. URL https://eur-lex.europa.eu/eli/reg/2016/679/oj?locale=es. Texto consolidado.

Apéndice A

Esquema Detallado de la Base de Datos

Este anexo contiene las definiciones SQL (`CREATE TABLE`) detalladas para la tabla relacional `users` y las hipertables propuestas para almacenar los datos de series temporales en PostgreSQL con la extensión TimescaleDB, tal como se describen en la Sección 4.3.

Tabla `users`

Tabla relacional estándar para almacenar la información de vinculación y tokens cifrados.

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    access_token_enc BYTEA,
    refresh_token_enc BYTEA,
    expires_at TIMESTAMPTZ,
    linked_at TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
    scopes TEXT[] -- Almacena los permisos como un array de texto
);
CREATE INDEX idx_users_email ON users(email); -- Índice para búsquedas
```

Hipertabla `intraday_metrics`

Hipertabla propuesta para datos intradía (frecuencia cardíaca, pasos, calorías, distancia).

```
CREATE TABLE intraday_metrics (
    time TIMESTAMPTZ NOT NULL,
    user_email VARCHAR(255) NOT NULL,
    type VARCHAR(50) NOT NULL, -- 'heart_rate', 'steps', etc.
    value DOUBLE PRECISION NOT NULL,
    PRIMARY KEY (time, user_email, type)
);
-- Convertir a Hipertabla (Requiere extensión TimescaleDB instalada)
SELECT create_hypertable('intraday_metrics', 'time');
```

Tabla/Hipertabla `daily_summaries`

Tabla propuesta para resúmenes diarios de actividad. Puede ser una tabla relacional normal o una hipertabla particionada por ‘date‘.

```
CREATE TABLE daily_summaries (
    date DATE NOT NULL,
    user_email VARCHAR(255) NOT NULL,
    steps INTEGER,
    distance_km DOUBLE PRECISION,
    calories_out INTEGER,
    active_minutes_total INTEGER,
    -- Otras métricas de resumen diario...
    PRIMARY KEY (date, user_email)
);
-- Opcional: Convertir a Hipertabla
-- SELECT create_hypertable('daily_summaries', 'date');
```

Hipertabla `sleep_logs`

Hipertabla propuesta para los registros detallados de sueño.

```
CREATE TABLE sleep_logs (
   log_id BIGINT PRIMARY KEY, -- ID único del log de Fitbit
   user_email VARCHAR(255) NOT NULL,
   start_time TIMESTAMPTZ NOT NULL,
   end_time TIMESTAMPTZ NOT NULL,
   duration_ms BIGINT,
   efficiency INTEGER,
   minutes_asleep INTEGER,
   minutes_awake INTEGER,
   minutes_in_rem INTEGER,
   minutes_in_light INTEGER,
   minutes_in_deep INTEGER
);
-- Convertir a Hipertabla
SELECT create_hypertable('sleep_logs', 'start_time');
-- Índice útil para buscar logs de un usuario en un rango de tiempo
CREATE INDEX idx_sleep_logs_user_email_start ON sleep_logs(user_email, start_time DES
```

Apéndice B

Anexos de Ficheros de Código

B.1. Aplicación Principal: `app.py`

```
% Aquí pegarías el contenido adaptado de app.py
```

Listing B.1: Archivo app.py

B.2. Módulo de Autenticación: `auth.py`

```
% Aquí pegarías el contenido adaptado de auth.py
```

Listing B.2: Archivo auth.py

B.3. Módulo de Base de Datos: `db.py`

```
% Aquí pegarías el contenido adaptado de db.py
```

Listing B.3: Archivo db.py

B.4. Módulo de Cifrado: `encryption.py`

```
% Aquí pegarías el contenido adaptado de encryption.py
```

Listing B.4: Archivo encryption.py

B.5. Script de Resumen Diario: `fitbit.py`

```
texoto wfds jfb
% Aquí pegarías el contenido adaptado de fitbit.py
```

Listing B.5: Archivo fitbit.py

B.6. Script de Datos Intradía: `fitbit_intraday.py`

```
% Aquí pegarías el contenido adaptado de fitbit_intraday.py
```

Listing B.6: Archivo fitbit$_intraday.py$

B.7. Script de Ejecución: `run_fitbit.sh`

```
% Aquí pegarías el contenido adaptado de run_fitbit.sh
```

Listing B.7: Script $\text{run}_f itbit.sh$

B.8. Script de Ejecución Intradía: `run_fitbit_intraday.sh`

```
% Aquí pegarías el contenido adaptado de run_fitbit_intraday.sh
```

Listing B.8: Script $\text{run}_f itbit_i ntraday.sh$

Apéndice C

Anexos de Código de Implementación Especifico

C.1. Configuración de `cron`

Ejemplo de la configuración utilizada en `crontab` para la ejecución periódica de los scripts de adquisición de datos, asegurando que se ejecuten dentro del entorno virtual correcto y redirigiendo la salida a ficheros de log.

```
# Activar entorno virtual y ejecutar script de datos diarios dos veces al dia
# Ejecutar a las 9:05 y 22:05 cada día
5 9,22 * * * cd /ruta/completa/al/proyecto && /ruta/completa/al/venv/bin/python
    fitbit.py >> /ruta/completa/al/proyecto/logs/cron_daily.log 2>&1

# Activar entorno virtual y ejecutar script de datos intradia cada 15 minutos
# Ejecutar en los minutos 0, 15, 30, 45 de cada hora
*/15 * * * * cd /ruta/completa/al/proyecto && /ruta/completa/al/venv/bin/python
    fitbit_intraday.py >> /ruta/completa/al/proyecto/logs/cron_intraday.log 2>&1
```

Listing C.1: Ejemplo de configuración de crontab para scripts de adquisición.

Nota: Las rutas (`/ruta/completa/al/...`) deben reemplazarse por las rutas absolutas correctas en el sistema de despliegue.

C.2. Función de Inserción en TimescaleDB (`insert_intraday_met`

Ejemplo de función en `db.py` para insertar eficientemente datos en la hipertabla `intraday_metrics` utilizando `psycopg2.extras.execute_values`. Incluye manejo básico de errores y conflictos.

```
import psycopg2
from psycopg2.extras import execute_values # Para inserción eficiente
import logging # Mejor usar logging que print para mensajes

# Configurar logging (preferiblemente al inicio de db.py o app.py)
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
    %(message)s')

# Asumiendo que 'conn' es una conexión psycopg2 válida

def insert_intraday_metrics(conn, data_list):
    """
    Inserta una lista de métricas intradía en la hipertabla.
    data_list: lista de tuplas [(time, user_email, type, value), ...]
```

```
                time debe ser un objeto datetime con timezone (idealmente UTC).
    """
    if not data_list:
        logging.info("No hay datos intradía para insertar.")
        return True # Nada que insertar

    sql = """
        INSERT INTO intraday_metrics (time, user_email, type, value)
        VALUES %s
        ON CONFLICT (time, user_email, type) DO NOTHING;
        -- Estrategia de conflicto: Ignorar duplicados.
        -- Alternativa: ON CONFLICT (time, user_email, type)
        -- DO UPDATE SET value = EXCLUDED.value; (Actualizar si existe)
    """
    cursor = None # Inicializar cursor fuera del try para el finally
    try:
        cursor = conn.cursor()
        # execute_values es eficiente para inserciones múltiples
        execute_values(cursor, sql, data_list, page_size=100) # page_size
    ajustable
        conn.commit()
        logging.info(f"Insertadas/Ignoradas {len(data_list)} métricas
    intradía.")
        return True
    except (Exception, psycopg2.DatabaseError) as error:
        logging.error(f"Error insertando métricas intradía: {error}")
        if conn:
            conn.rollback() # Deshacer transacción en caso de error
        return False
    finally:
        if cursor:
            cursor.close() # Siempre cerrar el cursor

# --- Ejemplo de preparación de datos en fitbit_intraday.py ---
# (Incluir ejemplo de parsing de tiempo y construcción de data_list
#  como en la respuesta anterior, asegurando manejo de timezone)
# from datetime import datetime, timezone
# import pytz
# ... (código de parse_fitbit_time y bucle de procesamiento) ...
```

Listing C.2: Ejemplo de función de inserción masiva en TimescaleDB (`db.py`).

Nota: La implementación real debe incluir un manejo robusto de zonas horarias, errores de parsing y podría beneficiarse de logging más detallado.

C.3. Función de Consulta en TimescaleDB (`get_hr_data`)

Ejemplo esquemático de una función en `db.py` para recuperar datos de frecuencia cardíaca para el dashboard, filtrando por usuario y rango de tiempo.

```
import psycopg2
from datetime import datetime
```

```
import logging
# import pandas as pd # Opcional: devolver como DataFrame

def get_hr_data(conn, user_email: str, start_date: datetime, end_date: datetime):
    """
    Recupera datos de frecuencia cardíaca para un usuario en un rango de fechas.
    start_date y end_date deben ser objetos datetime (preferiblemente aware, UTC).
    """
    # Asegurarse que las fechas de entrada son conscientes de zona horaria si es necesario
    # O convertir a UTC si la columna 'time' está en UTC
    # Ejemplo: Asumiendo que start/end_date son naive, y BD está en UTC
    # start_date_utc = start_date.replace(tzinfo=timezone.utc)
    # end_date_utc = end_date.replace(tzinfo=timezone.utc)

    sql = """
        SELECT time, value
        FROM intraday_metrics
        WHERE user_email = %s
          AND type = 'heart_rate'
          AND time >= %s -- Fecha/hora de inicio inclusiva
          AND time < %s  -- Fecha/hora de fin exclusiva
        ORDER BY time ASC;
    """
    results = []
    cursor = None
    try:
        cursor = conn.cursor()
        # Pasar las fechas como parámetros
        cursor.execute(sql, (user_email, start_date, end_date))
        results = cursor.fetchall() # Lista de tuplas [(time, value), ...]
        logging.info(f"Recuperados {len(results)} puntos de FC para {user_email}.")

        # --- Opcional: Convertir a DataFrame de Pandas ---
        # if results:
        #     df = pd.DataFrame(results, columns=['time', 'value'])
        #     # Asegurar que la columna de tiempo sea datetime y tenga timezone
        #     df['time'] = pd.to_datetime(df['time'], utc=True)
        #     return df
        # else:
        #     # Devolver DataFrame vacío con columnas definidas
        #     return pd.DataFrame(columns=['time', 'value'])
        # --- Fin Opcional Pandas ---

        return results # Devuelve lista de tuplas por defecto
    except (Exception, psycopg2.DatabaseError) as error:
        logging.error(f"Error recuperando datos de FC para {user_email}: {error}")
```

```
        # return pd.DataFrame(columns=['time', 'value']) # O DataFrame vacío
        return [] # Lista vacía en caso de error
    finally:
        if cursor:
            cursor.close()

# --- Ejemplo de uso en el callback del dashboard ---
# db_conn = db.connect_to_db()
# if db_conn:
#     hr_data_tuples = db.get_hr_data(db_conn, selected_email, start_dt_obj, end_dt_obj)
#     db_conn.close()
#     # Procesar hr_data_tuples para generar el gráfico Plotly...
```

Listing C.3: Ejemplo de función de consulta de FC (`db.py`).

Nota: El manejo preciso de las fechas y zonas horarias (`start_date`, `end_date`) al pasarlas a la consulta SQL es crucial y depende de cómo se almacenen en la BD (con o sin zona horaria) y cómo se reciban del DatePicker.

C.4. Layout Básico del Dashboard (`app.py` o `dashboard.py`)

Estructura básica del layout de la aplicación Dash definida en Python, utilizando los componentes de `dash` para crear la interfaz interactiva.

```
import dash
# En versiones nuevas de Dash (>=2.0):
from dash import dcc, html
# En versiones antiguas:
# import dash_core_components as dcc
# import dash_html_components as html
from dash.dependencies import Input, Output, State
# from flask_login import login_required # Para proteger ruta Flask

# --- Asumiendo 'server' es la instancia de Flask ---
# app_dash = dash.Dash(__name__, server=server, url_base_pathname='/dashboard/')
# # Configurar Dash para servir assets locales si los tienes (CSS, JS)
# # app_dash.config.suppress_callback_exceptions = True # Si callbacks están en otro fichero

# --- Layout de Dash ---
# (Puede estar en app.py o importado de dashboard_layout.py)
# layout = html.Div([ ... ]) # Definición del layout como antes...

def create_dashboard_layout():
    """Función que devuelve el layout para permitir actualizaciones si es necesario."""
    return html.Div([
        html.H1("Panel de Monitorización de Residentes"),
        # Dropdown para seleccionar residente
        html.Div([
            html.Label("Seleccionar Residente:"),
```

```
            dcc.Dropdown(
                id='resident-dropdown',
                options=[
                    # Las opciones se cargan dinámicamente con un callback
                ],
                placeholder="Seleccione un residente...",
                clearable=False, # Evitar que quede vacío si solo hay 1 opción
                style={'width': '50%'} # Ajustar estilo si es necesario
            )
        ], style={'padding': 10}),

        # Selector de rango de fechas
        html.Div([
            html.Label("Seleccionar Rango de Fechas:"),
            dcc.DatePickerRange(
                id='date-picker-range',
                start_date_placeholder_text="Fecha Inicio",
                end_date_placeholder_text="Fecha Fin",
                display_format='YYYY-MM-DD',
                # Podrías establecer fechas iniciales por defecto
                # initial_visible_month=datetime.date.today(),
                # start_date=datetime.date.today() - datetime.timedelta(days=7),
                # end_date=datetime.date.today(),
                clearable=True,
                style={'marginLeft': '10px'}
            )
        ], style={'padding': 10, 'display': 'flex', 'alignItems': 'center'}),

        html.Hr(), # Separador visual

        # Contenedor para los gráficos (se actualizan con callbacks)
        html.Div(id='graphs-container', children=[
            html.Div([
                html.H3("Frecuencia Cardíaca"),
                dcc.Loading( # Añadir indicador de carga
                    type="default",
                    children=dcc.Graph(id='hr-graph')
                )
            ]),
            html.Div([
                html.H3("Pasos Diarios"),
                 dcc.Loading(
                    type="default",
                    children=dcc.Graph(id='steps-graph')
                 )
            ]),
            html.Div([
                html.H3("Resumen del Sueño"),
                 dcc.Loading(
                    type="default",
                    children=dcc.Graph(id='sleep-graph')
```

```
                )
            ]),
            # Añadir más gráficos aquí si es necesario
        ])
        # dcc.Store(id='intermediate-data-store') # Para almacenar datos intermedios
    ])

# Asignar el layout a la app Dash
# app_dash.layout = create_dashboard_layout

# --- Ruta Flask para servir el dashboard ---
# @server.route('/dashboard/')
# @login_required # Proteger la ruta
# def dashboard_page():
#     # Renderizar la plantilla base de Flask que contiene el layout de Dash
#     # O directamente servir app_dash.index() si no necesitas plantilla Flask
#     return app_dash.index() # Método estándar para servir Dash standalone/integrado
```

Listing C.4: Ejemplo de layout de Dash integrado en Flask.

Nota: Se han añadido componentes `dcc.Loading` para mejorar la experiencia de usuario mientras se cargan los datos.

C.5. Callback Principal del Dashboard (`app.py` o `dashboard_callb`

Ejemplo esquemático del callback principal que actualiza los gráficos del dashboard. Muestra la estructura de inputs, outputs y la lógica de consulta y generación de figuras Plotly.

```
from dash.dependencies import Input, Output, State
import plotly.graph_objs as go
import plotly.express as px # Alternativa para crear figuras más rápido
from datetime import datetime, date, timedelta
import logging
# import db # Asumiendo funciones de db.py
# import pandas as pd # Si se usa Pandas para procesar

# Asumiendo 'app_dash' es la instancia de la app Dash

# --- Callback para poblar el dropdown de residentes (ejecutar al inicio) ---
@app_dash.callback(
    Output('resident-dropdown', 'options'),
    Input('resident-dropdown', 'id') # Input dummy para disparar al cargar
)
def update_resident_options(_):
    options = []
    conn = None
    try:
        conn = db.connect_to_db()
        if conn:
```

```
22              # Necesitas una función en db.py que devuelva {'name': ...,
    'email': ...}
23              users = db.get_linked_users(conn)
24              options = [{'label': f"{user['name']} ({user['email']})", 'value':
    user['email']}
25                         for user in sorted(users, key=lambda u: u.get('name',
    ''))] # Ordenar por nombre
26          else:
27              logging.error("No se pudo conectar a BD para cargar residentes.")
28      except Exception as e:
29          logging.error(f"Error cargando lista de residentes: {e}")
30      finally:
31          if conn:
32              conn.close()
33      return options
34
35  # --- Callback principal para actualizar gráficos ---
36  @app_dash.callback(
37      [Output('hr-graph', 'figure'),
38       Output('steps-graph', 'figure'),
39       Output('sleep-graph', 'figure')],
40      [Input('resident-dropdown', 'value'),
41       Input('date-picker-range', 'start_date'),
42       Input('date-picker-range', 'end_date')],
43      # prevent_initial_call=True # Evitar ejecución inicial si no hay valores
    por defecto
44  )
45  def update_graphs(selected_email, start_date_str, end_date_str):
46      """
47      Callback para actualizar todos los gráficos basado en residente y fechas.
48      """
49      # --- Crear figuras vacías por defecto ---
50      def create_empty_figure(title="Seleccione residente y rango de fechas"):
51          fig = go.Figure()
52          fig.update_layout(
53              title=title,
54              xaxis = {"visible": False},
55              yaxis = {"visible": False},
56              annotations = [{
57                  "text": "No hay datos para mostrar.",
58                  "xref": "paper",
59                  "yref": "paper",
60                  "showarrow": False,
61                  "font": {"size": 16}
62              }]
63          )
64          return fig
65
66      hr_fig = create_empty_figure("Frecuencia Cardíaca")
67      steps_fig = create_empty_figure("Pasos Diarios")
68      sleep_fig = create_empty_figure("Resumen del Sueño")
```

```
69
70      # --- Validar Inputs ---
71      if not selected_email or not start_date_str or not end_date_str:
72          # Si falta algún input, devolver figuras vacías
73          return hr_fig, steps_fig, sleep_fig
74
75      # --- Procesar Fechas ---
76      try:
77          # Convertir string a objeto date (DatePickerRange devuelve date)
78          start_date_obj = date.fromisoformat(start_date_str)
79          end_date_obj = date.fromisoformat(end_date_str)
80
81          # Convertir a datetime para consultas (inicio del día y fin del día+1)
82          # !Ajustar según cómo esperen las funciones de BD y la zona horaria!
83          start_dt = datetime.combine(start_date_obj, datetime.min.time())
84          end_dt = datetime.combine(end_date_obj + timedelta(days=1),
    datetime.min.time())
85
86      except (ValueError, TypeError) as e:
87          logging.error(f"Error procesando fechas: {e}")
88          # Devolver figuras vacías si las fechas son inválidas
89          return hr_fig, steps_fig, sleep_fig
90
91      # --- Conexión y Consultas a BD ---
92      conn = None
93      try:
94          conn = db.connect_to_db()
95          if not conn:
96              logging.error("No se pudo conectar a BD para actualizar gráficos.")
97              # Actualizar figuras para mostrar error de conexión
98              error_title = "Error de Conexión a Base de Datos"
99              hr_fig.update_layout(title=error_title)
100             steps_fig.update_layout(title=error_title)
101             sleep_fig.update_layout(title=error_title)
102             return hr_fig, steps_fig, sleep_fig
103
104         # --- Generar Gráfico de Frecuencia Cardíaca ---
105         hr_data = db.get_hr_data(conn, selected_email, start_dt, end_dt)
106         if hr_data: # Asume lista de tuplas (time, value)
107             times, values = zip(*hr_data)
108             # Usar Plotly Express para simplificar
109             hr_fig = px.line(x=list(times), y=list(values), labels={'x':'Hora',
    'y':'Pulsaciones/min'})
110             hr_fig.update_layout(title=f'Frecuencia Cardíaca
    ({selected_email})')
111             hr_fig.update_traces(mode='lines+markers') # Añadir marcadores si
    se desea
112         else:
113             hr_fig = create_empty_figure(f'Sin datos de FC ({selected_email})')
114
115         # --- Generar Gráfico de Pasos Diarios ---
```

```
116         steps_data = db.get_steps_data(conn, selected_email, start_date_obj,
    end_date_obj) # Pasar date
117         if steps_data: # Asume lista de tuplas (date, steps)
118             dates, steps = zip(*steps_data)
119             steps_fig = px.bar(x=list(dates), y=list(steps),
    labels={'x':'Fecha', 'y':'Número de Pasos'})
120             steps_fig.update_layout(title=f'Pasos Diarios ({selected_email})')
121         else:
122             steps_fig = create_empty_figure(f'Sin datos de Pasos
    ({selected_email})')
123
124         # --- Generar Gráfico de Sueño ---
125         sleep_data = db.get_sleep_data(conn, selected_email, start_dt, end_dt)
    # Pasar datetime
126         if sleep_data: # Asume lista de dicts con fases
127             # Procesar para formato apilado (ej. usando Pandas o bucle)
128             # df_sleep = pd.DataFrame(sleep_data)
129             # df_sleep['date'] = pd.to_datetime(df_sleep['start_time']).dt.date
130             # df_melted = df_sleep.melt(id_vars='date',
131             #                           value_vars=['minutes_in_rem',
    'minutes_in_light',
132             #                                       'minutes_in_deep',
    'minutes_awake'],
133             #                           var_name='Fase', value_name='Minutos')
134             # sleep_fig = px.bar(df_melted, x='date', y='Minutos', color='Fase',
135             #                    labels={'date':'Noche del', 'Minutos':'Minutos
    en Fase'})
136             # sleep_fig.update_layout(title=f'Fases del Sueño
    ({selected_email})')
137
138             # Alternativa sin Pandas (más verboso):
139             dates_sleep = [log['start_time'].date() for log in sleep_data]
140             fig_data = [
141                 go.Bar(name='REM', x=dates_sleep, y=[log.get('minutes_in_rem',
    0) for log in sleep_data]),
142                 go.Bar(name='Ligero', x=dates_sleep,
    y=[log.get('minutes_in_light', 0) for log in sleep_data]),
143                 go.Bar(name='Profundo', x=dates_sleep,
    y=[log.get('minutes_in_deep', 0) for log in sleep_data]),
144                 go.Bar(name='Despierto', x=dates_sleep,
    y=[log.get('minutes_awake', 0) for log in sleep_data])
145             ]
146             sleep_fig = go.Figure(data=fig_data)
147             sleep_fig.update_layout(barmode='stack', title=f'Fases del Sueño
    ({selected_email})',
148                                     xaxis_title='Noche del',
    yaxis_title='Minutos')
149
150         else:
151             sleep_fig = create_empty_figure(f'Sin datos de Sueño
    ({selected_email})')
```

```
152
153     except Exception as e:
154         logging.error(f"Error general en callback update_graphs: {e}",
    exc_info=True)
155         # Mostrar error genérico en los gráficos
156         error_title_gen = "Error al generar gráficos"
157         hr_fig.update_layout(title=error_title_gen)
158         steps_fig.update_layout(title=error_title_gen)
159         sleep_fig.update_layout(title=error_title_gen)
160     finally:
161         if conn:
162             conn.close() # Asegurarse siempre de cerrar la conexión
163
164     return hr_fig, steps_fig, sleep_fig
```

Listing C.5: Ejemplo de callback principal en Dash para actualizar gráficos.

Nota: Este callback es complejo. Requiere funciones de base de datos robustas, un manejo cuidadoso de los tipos de datos (especialmente fechas/horas) y errores. El uso de Plotly Express puede simplificar la creación de figuras. Se ha añadido logging básico.