PlanifyMe - Documentación Técnica del Proyecto

# 1. Introducción

**PlanifyMe** es una aplicación web desarrollada en el marco de la asignatura Integración de Sistemas de Información (ISI), concebida como una solución práctica para la planificación personal inteligente. Su objetivo fundamental es integrar fuentes de información externas mediante microservicios y permitir al usuario organizar su tiempo de forma eficiente y contextualizada, teniendo en cuenta variables como el estado meteorológico, la oferta cultural (eventos) y el entretenimiento cinematográfico en su ciudad.

En la sociedad actual, donde la gestión del tiempo y la productividad personal son prioridades para muchas personas, contar con herramientas tecnológicas que faciliten la toma de decisiones basadas en información en tiempo real se ha vuelto indispensable. Desde planificar un paseo o una actividad al aire libre, hasta decidir asistir a un evento o ver una película cuando las condiciones climáticas no son favorables, las decisiones cotidianas pueden verse optimizadas gracias a la integración de datos dinámicos. PlanifyMe responde a esa necesidad, combinando funcionalidad práctica con simplicidad de uso.

El sistema permite a cada usuario registrarse, iniciar sesión y gestionar una lista de tareas personalizada. Estas tareas no son meramente notas estáticas, sino que pueden nutrirse de las recomendaciones que el sistema brinda basadas en la información externa obtenida en tiempo real. Por ejemplo, si se detecta que hará buen tiempo en Madrid el fin de semana, el usuario podrá planear actividades al aire libre. Si por el contrario se prevé lluvia, podrá consultar automáticamente qué películas hay en cartelera o qué eventos cubiertos están disponibles en su ciudad.

Desde el punto de vista técnico, el proyecto ha sido diseñado siguiendo una arquitectura de microservicios, permitiendo una separación clara de responsabilidades y una escalabilidad natural. Se han implementado tres microservicios independientes, cada uno encargado de consumir y ofrecer una API específica:

* **Microservicio de Clima**: Consulta el tiempo actual por ciudad a través de una API meteorológica.
* **Microservicio de Eventos**: Proporciona eventos culturales disponibles en una determinada ciudad.
* **Microservicio de Películas**: Ofrece una lista de películas en cartelera o disponibles en plataformas.

Cada uno de estos servicios ha sido desarrollado utilizando Flask, ejecutado de forma independiente en contenedores Docker, y orquestado mediante Docker Compose. Esta arquitectura permite una modularidad que no solo mejora el mantenimiento, sino que facilita pruebas, despliegues y futuras ampliaciones del sistema.

La aplicación principal se comunica con los microservicios a través de peticiones HTTP y está encargada de la lógica de usuario, la interfaz y la persistencia de datos en una base de datos relacional MySQL. Todo el sistema ha sido encapsulado y automatizado para facilitar su ejecución con un simple comando make, siguiendo prácticas DevOps básicas.

En este documento se recoge el desarrollo integral de **PlanifyMe**, incluyendo la motivación del proyecto, los requisitos funcionales, la arquitectura técnica adoptada, la implementación de cada componente, las pruebas realizadas y una evaluación crítica del resultado final. Asimismo, se incluye una guía de despliegue y uso para facilitar su comprensión y utilización por parte de cualquier evaluador o desarrollador interesado.

# 2. Objetivos del Proyecto

El desarrollo de PlanifyMe responde a una serie de objetivos definidos en el marco de la asignatura **Integración de Sistemas de Información (ISI)**, orientados a aplicar de forma práctica los conocimientos adquiridos sobre arquitectura de software, orquestación de contenedores y consumo de servicios externos. A continuación, se detallan los objetivos principales del proyecto:

### 1. ****Diseñar una arquitectura basada en microservicios usando Docker****

Uno de los pilares del proyecto ha sido la creación de una arquitectura modular, compuesta por servicios desacoplados que se comunican entre sí mediante APIs REST. Cada microservicio ha sido desplegado de forma independiente utilizando **Docker**, lo que permite una gestión eficiente de los recursos, mejora la escalabilidad y simplifica el mantenimiento. La contenedorización facilita además la portabilidad del sistema entre entornos de desarrollo, testeo y producción.

### 2. ****Integrar servicios externos mediante APIs para obtener información en tiempo real****

Se ha llevado a cabo la **conexión con APIs públicas** para acceder a datos actualizados sobre:

* **Clima actual** según la ciudad del usuario.
* **Eventos locales** relevantes para la planificación.
* **Películas en cartelera**, con el objetivo de sugerir ocio audiovisual.  
  Estos microservicios están desarrollados en Flask y funcionan de forma autónoma, sirviendo datos en formato JSON a la aplicación principal. Este objetivo refuerza la capacidad de interactuar con fuentes de información externas y transformarlas en servicios consumibles por otras aplicaciones.

### 3. ****Implementar un backend funcional con Flask para la gestión de usuarios y tareas****

La aplicación principal integra un backend desarrollado en **Flask** que permite:

* Registrar e iniciar sesión con cuentas de usuario.
* Gestionar tareas personales o “planes”.
* Consultar datos ofrecidos por los microservicios para decidir cuándo y cómo organizar dichos planes.  
  Además, se ha implementado la **persistencia de los datos** utilizando **MySQL**, asegurando la consistencia y disponibilidad de la información a través del uso de modelos y servicios internos.

### 4. ****Diseñar una interfaz sencilla y eficaz basada en HTML5 y Bootstrap****

Se ha desarrollado un **frontend minimalista pero funcional**, que permite al usuario interactuar fácilmente con el sistema. Utilizando **HTML5**, **CSS** y el framework **Bootstrap**, se ha conseguido una experiencia de usuario clara, responsiva y adecuada tanto para escritorio como para dispositivos móviles. La navegación incluye elementos como formularios, botones de acción, visualización de tareas y menús para consultar información adicional.

### 5. ****Orquestar y desplegar los servicios localmente, con visión de despliegue en cloud****

Todos los servicios han sido definidos y orquestados a través de un archivo docker-compose.yml, permitiendo levantar la infraestructura completa con un solo comando. Se ha diseñado un **Makefile** que automatiza tareas clave como compilación, arranque, parada y testeo de los servicios. Si bien el despliegue final se realiza en local, la estructura está preparada para una futura migración a entornos cloud como AWS, GCP o Azure mediante herramientas como Kubernetes o Docker Swarm.

# 3. Arquitectura del Sistema

La arquitectura de PlanifyMe ha sido diseñada siguiendo un enfoque de **microservicios**, una de las tendencias actuales más relevantes en el desarrollo de software distribuido. Esta elección arquitectónica permite desarrollar, desplegar y escalar cada componente de forma independiente, favoreciendo la mantenibilidad del sistema y facilitando la integración de nuevas funcionalidades en el futuro.

### 3.1. Componentes principales

El sistema está compuesto por los siguientes elementos:

#### ✅ ****Aplicación principal (Flask App)****

Ubicada en el directorio sprint4/app/, esta aplicación representa el núcleo del sistema. Es responsable de:

* Gestionar la autenticación y los datos de los usuarios.
* Gestionar la creación, listado y eliminación de tareas personales.
* Coordinar las llamadas a los microservicios externos (clima, eventos y películas).
* Renderizar las vistas HTML mediante Jinja2.

Esta aplicación se conecta a una base de datos MySQL para almacenar persistentemente la información del usuario y sus tareas.

#### ✅ ****Microservicios****

Cada microservicio es una aplicación Flask ligera e independiente, ubicada en su respectivo subdirectorio:

* weather\_service/: consulta la API del clima para una ciudad determinada.
* events\_service/: devuelve eventos culturales o locales disponibles.
* movies\_service/: obtiene películas actuales en cartelera.

Cada uno escucha en un puerto distinto y expone endpoints RESTful. Por ejemplo:

make up # Levanta todos los contenedores

make down # Detiene y elimina todos los contenedores

make logs # Muestra los logs de ejecución

make test # Realiza pruebas de integración o funcionamiento

3.3. Comunicación entre servicios

La aplicación principal realiza llamadas HTTP a los microservicios usando sus nombres como hosts dentro de la red Docker. Por ejemplo, para obtener el clima:

requests.get("<http://weather:5000/weather?city=Madrid>")

De esta forma, se evita la necesidad de usar direcciones IP fijas, lo que facilita el despliegue en diferentes entornos.

# 4. Estructura del Repositorio

El desarrollo de PlanifyMe se ha estructurado por etapas, siguiendo una estrategia iterativa de construcción por **sprints**, que permiten abordar progresivamente los distintos componentes del sistema. A continuación, se describen los aspectos clave del proceso de implementación.

### 4.1. Desarrollo del Backend Principal

La lógica de negocio principal se encuentra en la aplicación Flask ubicada en sprint4/app/. Su estructura modular está dividida en carpetas como routes, services y models, lo cual favorece la mantenibilidad y separación de responsabilidades.

#### Funcionalidades principales:

* **Registro de usuario**: los usuarios pueden registrarse con nombre, email y contraseña.
* **Inicio de sesión**: autenticación básica contra los datos almacenados en MySQL.
* **Gestión de tareas**: permite añadir, visualizar y eliminar tareas personalizadas.
* **Integración de APIs**: el usuario recibe recomendaciones de planes según el clima, eventos o películas disponibles.

Todos los datos de usuario y tareas se almacenan persistentemente en la base de datos mediante consultas SQL usando mysql-connector-python.

### 4.2. Implementación de Microservicios

Cada microservicio se ha diseñado como una aplicación Flask independiente, con un único archivo app.py y lógica sencilla que se basa en consumir APIs públicas (o responder con datos simulados cuando no se tiene acceso externo).

#### Características:

* **Autónomos**: no dependen del backend principal ni de la base de datos.
* **Intercambio JSON**: exponen endpoints que devuelven respuestas en formato JSON.
* **Escalables**: pueden ser reemplazados por versiones más complejas (FastAPI, con caché, etc.) sin afectar el sistema.

Por ejemplo, el microservicio de clima responde con:

{

"clima": "Soleado",

"temperatura": "28°C"

}

### 4.3. Interfaz de Usuario

La interfaz está implementada en HTML y CSS con **Bootstrap 5**, con páginas simples pero funcionales. El objetivo es facilitar la experiencia del usuario y presentar la información de forma clara.

#### Vistas principales:

* **Login / Registro**
* **Dashboard del usuario**
* **Listado de tareas**
* **Panel de recomendaciones (clima, eventos, películas)**

El uso de Jinja2 permite renderizar dinámicamente los datos obtenidos desde la base de datos y los microservicios.

### 4.4. Contenedorización con Docker

Todos los servicios (app principal, microservicios y base de datos) han sido contenedorizados mediante Docker. Se han creado imágenes personalizadas a partir de python:3.11-slim, lo que permite entornos ligeros, portables y coherentes entre equipos.

#### Ventajas de esta aproximación:

* No es necesario instalar manualmente dependencias.
* Asegura que el entorno de desarrollo es idéntico al de despliegue.
* Facilita la orquestación y escalado con herramientas como Docker Compose y Kubernetes.

# 5. Detalles Técnicos

### 5.1. Despliegue Local con Docker Compose

Para facilitar el despliegue y la ejecución del sistema completo, se ha configurado un entorno multi-contenedor mediante Docker Compose. El archivo docker-compose.yml, ubicado en sprint4/docker/, orquesta la construcción y ejecución de todos los servicios:

* main\_app: backend principal Flask
* mysql\_db: base de datos relacional MySQL
* weather\_service, movies\_service, events\_service: microservicios independientes
* Red interna por defecto para la comunicación entre contenedores

make up

Este comando lanza el sistema completo en segundo plano, permitiendo a los desarrolladores iniciar el entorno con facilidad y sin conflictos de puertos (se usaron 3307, 5001, 5002, 5003).

### 5.2. Estructura del Makefile

Se ha incluido un Makefile para automatizar las tareas más comunes del proyecto:

up:

docker-compose -f docker/docker-compose.yml up -d

down:

docker-compose -f docker/docker-compose.yml down

logs:

docker-compose -f docker/docker-compose.yml logs -f

build:

docker-compose -f docker/docker-compose.yml up –build

Esto permite simplificar la ejecución de comandos repetitivos y evitar errores humanos durante el desarrollo y pruebas.

### 5.3. Pruebas Funcionales

Durante el desarrollo se han realizado pruebas manuales para verificar:

* Registro e inicio de sesión de usuarios
* Creación y eliminación de tareas
* Visualización correcta de datos provenientes de los microservicios
* Persistencia de los datos en MySQL
* Accesibilidad y funcionamiento de los endpoints

Además, se realizaron llamadas internas desde el contenedor main\_app a los servicios weather, movies y events mediante requests.get(), comprobando que la resolución DNS entre contenedores funcionaba correctamente dentro del entorno Docker.

### 5.4. Preparación para el Despliegue en la Nube

Aunque el sistema ha sido desplegado localmente, su estructura está preparada para un futuro despliegue en la nube:

* Docker y Docker Compose permiten integrarlo con plataformas como Heroku, AWS ECS, Azure o Railway.
* Cada servicio puede exponerse con NGINX o un reverse proxy.
* La base de datos puede migrarse a un servicio gestionado como PlanetScale o RDS.

En futuras versiones, también se podrían añadir herramientas como:

* **CI/CD con GitHub Actions**
* **Certificados SSL**
* **Monitorización con Prometheus o Grafana**
* **Orquestación con Kubernetes**

# 6. Conclusiones

El desarrollo de **PlanifyMe** ha permitido integrar conocimientos clave sobre microservicios, orquestación con Docker y diseño de sistemas distribuidos, aplicados en un caso práctico funcional. A continuación se exponen las principales conclusiones extraídas del proyecto:

### 6.1. Integración de Sistemas Heterogéneos

Uno de los principales logros del proyecto ha sido la integración de múltiples servicios independientes en una única aplicación coherente. Cada microservicio se comunica con el backend mediante llamadas HTTP, y todos comparten una red virtual interna gracias a Docker. Esta separación de responsabilidades promueve el mantenimiento, escalabilidad y reutilización del código.

### 6.2. Automatización y Contenerización

El uso de **Docker** y **Docker Compose** ha facilitado el despliegue reproducible y aislado de todos los componentes. Gracias al Makefile, se puede levantar o reiniciar el sistema completo con un solo comando, simplificando las tareas de desarrollo, pruebas y mantenimiento.

### 6.3. Arquitectura preparada para producción

Aunque la aplicación ha sido desplegada en entorno local, la estructura está pensada para crecer:

* Se pueden externalizar los microservicios a distintos servidores o plataformas cloud.
* El backend puede escalar horizontalmente, ya que es stateless.
* La base de datos puede migrarse a un servicio administrado con backups automáticos.
* La interfaz web puede integrarse fácilmente con un frontend moderno basado en React o Vue.

### 6.4. Mejora de la Experiencia del Usuario

Gracias a la integración de datos en tiempo real (clima, cartelera, eventos), el usuario puede tomar mejores decisiones al planificar sus actividades. Esta capa de inteligencia contextual es un valor añadido clave respecto a gestores de tareas convencionales.

## 7. Resumen Final del Proyecto

**PlanifyMe** ha demostrado cómo los principios de la integración de sistemas pueden aplicarse a una solución funcional, modular y escalable. Mediante una arquitectura de microservicios, una base de datos bien estructurada y una interfaz clara, se ha construido un entorno en el que los usuarios pueden:

* Crear y gestionar sus tareas
* Acceder a información actualizada sobre el entorno (clima, ocio, eventos)
* Guardar y recuperar sus datos desde una base relacional
* Ejecutar el sistema de forma automatizada mediante contenedores

El proyecto refleja fielmente los objetivos de la asignatura ISI, abordando tanto los aspectos técnicos como organizativos del desarrollo de software distribuido y orientado a servicios.