



Arquitectura de Software II

Trabajo Práctico Nº3 - 2025s1

Maria Laura Medici
Natalia Stefania Rodriguez

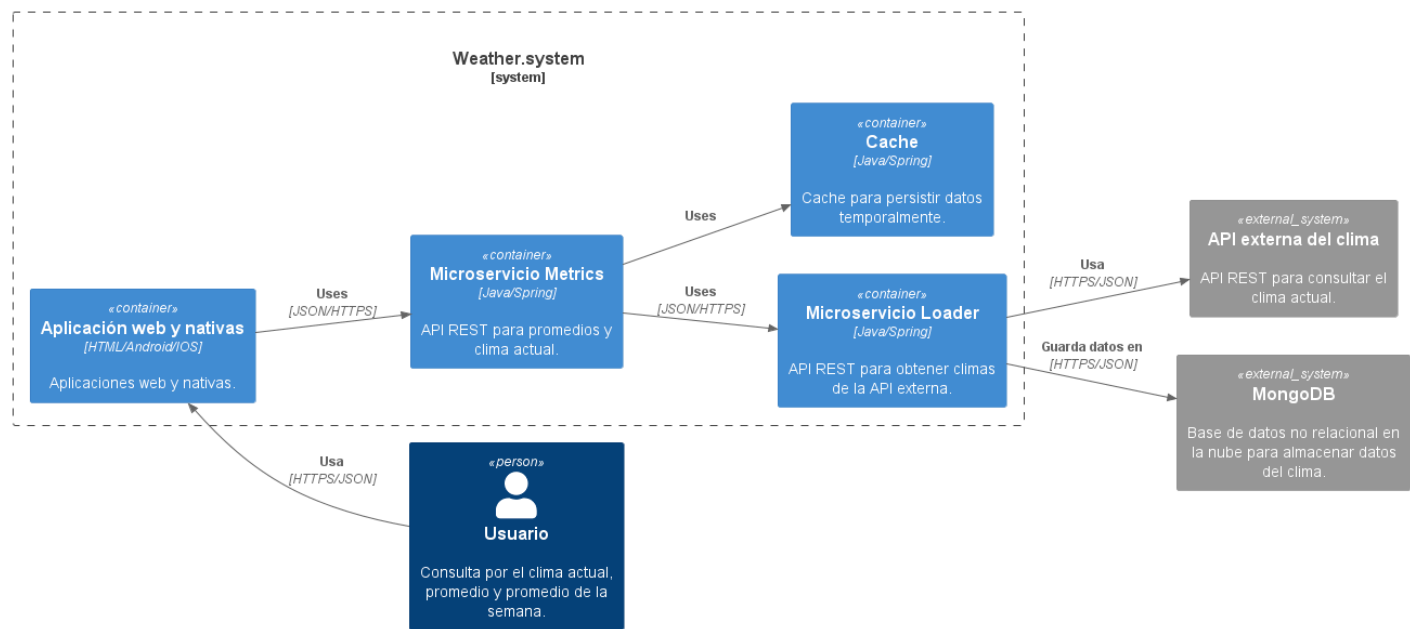
Índice:

Introducción	3
Requerimientos del sistema	3
Casos de uso	3
Atributos de calidad	4
1. Escalabilidad	5
2. Desplegabilidad	5
3. Performance	5
4. Usabilidad	5
5. Interoperabilidad	6
6. Seguridad	6
Tipo de arquitectura: Microservicios	6
Modelo de datos	7
Estrategias de Tolerancia a Fallos	7
Time-out	7
Implementación en Java	8
Circuit breaker	9
Implementación en Java	10

Request Caché	12
Implementación en Java	13
Proceso de Consulta a la Base de Datos	13
Estrategias de Observabilidad	13
¿Qué es la Agregación de Métricas?	13
Beneficios de la Agregación de Métricas	13
Implementación de métricas	14
Dashboards	14
Métricas de hardware	14
Métricas de cada endpoint	15
Temperatura diaria mínima y máxima	15
Temperatura semanal mínima y máxima	16
Alerting	16
Ejemplo: Uso de CPU del Sistema	17
Especificaciones de las APIs	19
Listado de endpoints	19
Pruebas de Carga	20
Implementación	21
Primera prueba: Requests ejecutados de forma simultánea	21
Segunda prueba: Requests ejecutados de forma distribuida	21
Tercera prueba: Apuntamos a una base de datos inaccesible con timeout de 3 segundos	22
Cuarta prueba: Sin timeouts entre servicios ni hacia la base de datos	22
Quinta prueba: Carga realista a gran escala (1000 usuarios)	23
Conclusiones	23
Diagrama de secuencia	24
Weather.loader	24
Proceso batch interno para obtener datos de la api externa y persistirlos en una base	
Mongo	24
Obtener clima de hoy	25
Obtener clima de hoy - error al acceder a la base de datos	25
Obtener temperaturas de hoy	25
Obtener temperaturas de hoy - error al acceder a la base de datos	26
Obtener temperaturas de la última semana	26
Obtener temperaturas de la última semana - error al acceder a la base de datos	26
Weather.metrics	27
Obtener clima de hoy - llamando al api externo	27
Obtener clima de hoy - Llama a la caché	27
Obtener clima de hoy - Error al llamar al api externa	28
Obtener clima promedio de hoy	28
Obtener clima promedio de hoy - Error al llamar al api externa	28
Obtener clima promedio semanal	29
Obtener clima promedio semanal - Error al llamar al api externa	29

Introducción

El objetivo de la solución es brindar un servicio al cual se le pueda consultar información sobre el clima para la ciudad de Quilmes de forma rápida, eficiente y con baja tasa de errores.



Requerimientos del sistema

- WeatherLoaderComponent:
 - Debe consumir datos del clima de la API de OpenWeatherMap (<https://openweathermap.org/>) de forma periódica.
 - Debe persistir los datos del clima obtenidos en una base de datos.
- WeatherMetricsComponent:
 - Debe consumir los datos del clima persistidos por el WeatherLoaderComponent.
 - Contener una API para consultar los siguientes reportes:
 - Reporte de la temperatura actual.
 - Promedio de la temperatura del último día.
 - Promedio de la temperatura de la última semana.

Casos de uso

Nombre	Obtener Datos del Clima
Descripción	Permite consumir datos del clima de la API de OpenWeatherMap y persistirlos en una base de datos.
Precondición	<ul style="list-style-type: none">• La API de OpenWeatherMap debe estar accesible.• La base de datos debe estar configurada y en

	funcionamiento.
Flujo Principal	<ol style="list-style-type: none"> 1. El WeatherLoaderComponent consume datos del clima de la API de OpenWeatherMap de forma periódica. 2. Los datos del clima obtenidos se persisten en la base de datos. 3. El WeatherLoaderComponent tiene un endpoint para enviar los datos a métricas.
Flujo Alternativo	<ul style="list-style-type: none"> • Si la API de OpenWeatherMap no responde o devuelve un error, el sistema registra el error y no persiste datos. • Si hay un problema al guardar los datos en la base de datos, el sistema registra el error.
Postcondición	Los datos del clima se han guardado en la base de datos y están disponibles para su consulta.

Nombre	Consultar Reportes de Clima
Descripción	Permite consultar reportes sobre la temperatura actual y promedios de temperatura.
Precondición	Los datos del clima deben estar disponibles en la base de datos.
Flujo Principal	<ol style="list-style-type: none"> 1. El WeatherMetricsComponent consume los datos del clima persistidos por el WeatherLoaderComponent. 2. El sistema proporciona una API para consultar: <ul style="list-style-type: none"> • Reporte de la temperatura actual (<i>weather/current</i>). • Promedio de la temperatura del último día (<i>weather/average/today</i>). • Promedio de la temperatura de la última semana (<i>weather/average/week</i>).
Flujo Alternativo	<ul style="list-style-type: none"> • Si no hay datos disponibles, el sistema devuelve un mensaje de error indicando que no se pueden obtener los reportes.
Postcondición	Se devuelven los reportes solicitados al cliente.

Atributos de calidad

Se enumeran algunos de los atributos y sus escenarios, importantes para esta aplicación, con una breve descripción:

1. Escalabilidad

- **Fuente del estímulo (Source):** Administrador del sistema
- **Estímulo (Stimulus):** Agregar nuevas funcionalidades relacionadas al clima
- **Artefacto (Artifact):** Código e interfaz de usuario
- **Entorno (Environment):** Desarrollo
- **Respuesta (Response):** La nueva funcionalidad se agrega, se testea y se despliega
- **Medición de la respuesta (Response Measure):** El esfuerzo de no menos de 2 personas durante un mes
- **Justificación:** La escalabilidad es crucial para permitir el crecimiento del sistema, ya que anticipa la necesidad de agregar nuevas funcionalidades y características a medida que el negocio crece.

2. Desplegabilidad

- **Fuente del estímulo (Source):** Administrador del sistema
- **Estímulo (Stimulus):** Nueva versión del sistema a ser desplegada
- **Artefacto (Artifact):** Actualización de funcionalidades existentes y nuevas
- **Entorno (Environment):** Despliegue completo
- **Respuesta (Response):** Incorporar la funcionalidad de reportes sin interrumpir el servicio
- **Medición de la respuesta (Response Measure):** Tiempo total del despliegue y efectividad de los reportes
- **Justificación:** Un sistema de CI/CD bien implementado es esencial para facilitar el despliegue de nuevas versiones y actualizaciones, minimizando el tiempo de inactividad y asegurando una transición fluida.

3. Performance

- **Fuente del estímulo (Source):** Usuario final
- **Estímulo (Stimulus):** Consulta de reportes de clima
- **Artefacto (Artifact):** Sistema completo
- **Entorno (Environment):** Normal
- **Respuesta (Response):** El sistema retorna resultados de la consulta relacionadas al clima
- **Medición de la respuesta (Response Measure):** Latencia
- **Justificación:** La performance es crítica, ya que los usuarios esperan respuestas rápidas al consultar la información del clima

4. Usabilidad

- **Fuente del estímulo (Source):** Usuario final
- **Estímulo (Stimulus):** El usuario entiende cómo usar el sistema
- **Artefacto (Artifact):** Sistema completo

- **Entorno (Environment):** Tiempo de ejecución
- **Respuesta (Response):** El sistema es fácil de usar, con ayudas y tutoriales cuando sea necesario
- **Medición de la respuesta (Response Measure):** Cantidad de errores, satisfacción del usuario, tasa de retención de usuarios
- **Justificación:** La usabilidad es fundamental para garantizar que los usuarios puedan interactuar con el sistema de manera efectiva

5. Interoperabilidad

- **Fuente del estímulo (Source):** Usuario final
- **Estímulo (Stimulus):** Integración con otras aplicaciones que consumen datos del clima
- **Artefacto (Artifact):** API del sistema de clima, junto con la infraestructura de conexión entre servicios.
- **Entorno (Environment):** Tiempo de ejecución
- **Respuesta (Response):** Permitir que otros sistemas consuman datos de clima correctamente y de manera eficiente
- **Medición de la respuesta (Response Measure):** Cantidad de integraciones exitosas y latencia en la conexión entre servicios.
- **Justificación:** La interoperabilidad es esencial para asegurar que el sistema funcione bien con otras aplicaciones y servicios.

6. Seguridad

- **Fuente del estímulo (Source):** Usuario final
- **Estímulo (Stimulus):** Acceso a datos sensibles
- **Artefacto (Artifact):** Base de datos y sistema de autenticación
- **Entorno (Environment):** Tiempo de ejecución
- **Respuesta (Response):** Proteger la información sensible y garantizar que solo los usuarios autorizados tengan acceso
- **Medición de la respuesta (Response Measure):** Número de brechas de seguridad y accesos no autorizados
- **Justificación:** En cualquier sistema, la seguridad es crítica para proteger la información personal y financiera de los usuarios (en este caso compradores o vendedores), lo que a su vez genera confianza en la plataforma.

Tipo de arquitectura: Microservicios

Se tomó la decisión de utilizar una arquitectura de microservicios debido a la necesidad de desarrollar un sistema flexible, escalable y robusto, capaz de adaptarse a un entorno tecnológico con cambios constantes. En este contexto, la evolución continua nos permite adaptarnos rápidamente a las necesidades del mercado, mejorando la calidad de nuestro software e innovando constantemente.

En particular, se han definido dos componentes clave en esta arquitectura:

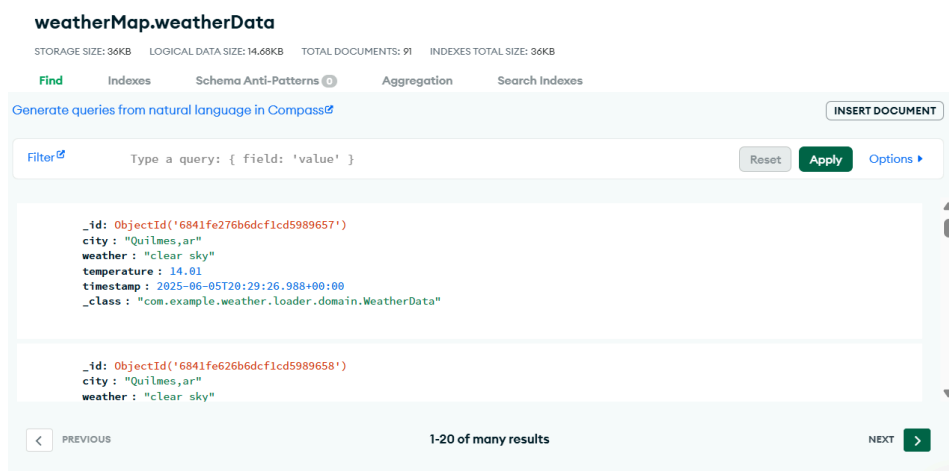
- **WeatherLoaderComponent:** Este componente es responsable de consumir datos del clima de la API de *OpenWeatherMap* (<https://openweathermap.org/>) de forma periódica. Además, debe persistir los datos del clima obtenidos en una base de datos, asegurando que la información esté disponible para su posterior análisis.
- **WeatherMetricsComponent:** Este componente consume los datos del clima que han sido persistidos por el *WeatherLoaderComponent*. Además, proporciona una API que permite consultar diversos reportes, tales como el reporte de la temperatura actual, el promedio de la temperatura del último día y el promedio de la temperatura de la última semana.

Los microservicios nos ofrecen una escalabilidad granular, mayor resiliencia y una independencia total para los equipos, acelerando los ciclos de desarrollo y despliegue. Este nuevo enfoque introduce nuevas complejidades en la gestión y el monitoreo, pero los beneficios a largo plazo relacionados a la agilidad y la escalabilidad del negocio son innegables.

Modelo de datos

La elección de utilizar bases de datos no relacionales en nuestro microservicio de loader, nos ofrece la flexibilidad, escalabilidad y rendimiento necesarios para adaptarnos a los cambios en los requisitos del negocio y a las necesidades de nuestros usuarios. MongoDB tiene la capacidad de escalar y manejar grandes volúmenes de datos, lo que la convierte en una solución ideal para soportar el crecimiento y la evolución de nuestra plataforma.

Forma de visualización de los datos creados en MongoDB:



Estrategias de Tolerancia a Fallos

Time-out

El time-out es un límite de tiempo predefinido que se establece para la duración máxima de una operación o solicitud. Si la operación no se completa (es decir, no devuelve una respuesta, ya sea exitosa o con error) dentro de este período de tiempo especificado, se considera que ha fallado, y la operación se cancela o se interrumpe automáticamente. El objetivo principal es:

- **Evitar esperas indefinidas:** Previene que un servicio se quede "colgado" o bloqueado indefinidamente esperando una respuesta de otro servicio que podría estar lento, inactivo o inaccesible.
- **Liberar recursos:** Al interrumpir una operación que excede su tiempo, se liberan los recursos (hilos, conexiones, memoria) que estaban siendo utilizados, permitiendo que el sistema los reutilice para otras tareas.
- **Mejorar la experiencia del usuario:** Evita que el usuario final tenga que esperar mucho tiempo por una respuesta que nunca llegará, lo que podría llevar a frustración o a que la aplicación parezca no responder.

Implementación en Java

Proyecto weather.loader:

Para consumir la url `api.openweathermap.org` se utilizó `RestTemplate`, una clase de Spring que actúa como un cliente HTTP sincrónico. Sirve para hacer llamadas a servicios REST desde tu aplicación Java, de forma muy simple.

Por defecto, cuando se crea un `RestTemplate` Spring no aplica ningún timeout, y para poder personalizar el comportamiento de esta clase, tuvimos que crear una clase de configuración llamada `RestTemplateConfig` para poder configurar el timeout. Al iniciar la aplicación, Spring detecta esta clase porque tiene la anotación `@Configuration`, y se da cuenta de que provee una instancia única de `RestTemplate` para utilizarla globalmente en todo el proyecto y la inyecta como `@Bean`. Esta instancia de `RestTemplate` viene configurada con `HttpComponentsClientHttpRequestFactory` para proveer más control y funcionalidades avanzadas en las llamadas HTTP, en vez de la implementación por defecto del JDK. Nosotras lo utilizamos para configurar los timeouts, pero también sirve para configurar autenticación, conexiones https más avanzadas, etc.

```
@Configuration
public class RestTemplateConfig {

    @Bean
    public RestTemplate restTemplate() {
        HttpComponentsClientHttpRequestFactory factory = new HttpComponentsClientHttpRequestFactory();
        factory.setConnectTimeout(5000);
        factory.setReadTimeout(5000);
        return new RestTemplate(factory);
    }
}
```

Para consumir la base de datos no relacional, utilizamos la interfaz `MongoRepository`, una interfaz de Spring Data que te permite trabajar con MongoDB de forma declarativa, sin escribir queries. `MongoRepository` no se encarga directamente de la conexión, sino que usa una instancia de `MongoClient` que por defecto puede tener timeouts muy largos. Entonces, al igual que con el `RestTemplate`, debemos crear una clase de configuración para otorgar una única instancia de `MongoClient` que tenga los timeouts configurados.

También se le podría haber agregado a la url de conexión de la base el timeout, igual a un query param pero preferimos hacerlo de este modo, por si quierámos cambiar de url y que el timeout quede centralizado para todas las próximas conexiones a base de datos.

Gracias a esta configuración, pasamos de esperar 30 segundos de no tener conexión a 5 segundos.

```
@Configuration
public class MongoConfig {

    @Value("${mongodb.uri}")
    private String mongoUri;

    @Value("${mongodb.database}")
    private String databaseName;

    @Bean
    public MongoClient mongoClient() {
        ConnectionString connectionString = new ConnectionString(mongoUri);

        MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .applyToClusterSettings(builder ->
                builder.serverSelectionTimeout(3, TimeUnit.SECONDS))
            .applyToSocketSettings(builder -> builder
                .connectTimeout(3, TimeUnit.SECONDS)
                .readTimeout(3, TimeUnit.SECONDS))
            .build();

        return MongoClient.create(settings);
    }
}
```

Proyecto weather.metrics:

Para consumir los métodos del microservicio de Weather.loader, se creó un cliente Feign, la interfaz LoaderClient que lleva la anotación @FeignClient. Feign es un cliente de Spring Cloud que permite declarar interfaces Java para consumir APIs REST externas.

Spring utiliza Feign y crea un proxy para hacer la llamada HTTP.

Con Feign se pueden configurar los timeouts en el archivo properties application.properties. Se los puede configurar por cliente o de forma global.

```
#Timeout esperando la conexion contra el Loader
spring.cloud.openfeign.client.config.loader-client.connectTimeout=1000
#Timeout esperando una respuesta contra el Loader
spring.cloud.openfeign.client.config.loader-client.readTimeout=2000
```

Circuit breaker

Circuit Breaker es una estrategia de tolerancia a fallos inspirada en los disyuntores eléctricos en una casa. Cuando un disyuntor eléctrico detecta una sobrecarga o un cortocircuito, se "abre" para proteger el circuito y evitar daños mayores. De manera similar, un Circuit Breaker en software "abre" un circuito cuando un servicio remoto comienza a fallar repetidamente, impidiendo que se sigan enviando solicitudes a ese servicio. Los objetivos de este patrón son:

- **Prevenir el "efecto dominó" (cascading failures):** Evita que la falla de un microservicio sobrecargue y cause la falla de otros microservicios que dependen de él.
- **Dar tiempo para la recuperación:** Permite que un servicio que está experimentando problemas se recupere al detener el flujo de solicitudes entrantes.
- **Reducir la latencia:** Cuando el circuito está abierto, las solicitudes fallan instantáneamente en lugar de esperar un time-out, mejorando la respuesta del sistema.

Un Circuit Breaker típicamente opera en tres estados:

1. **Cerrado (Closed):**

- Este es el estado inicial y normal.
- Las solicitudes al servicio remoto pasan a través del Circuit Breaker.
- El Circuit Breaker monitorea el número de fallos (errores, time-outs, etc.).
- Si el número de fallos excede un umbral predefinido dentro de un cierto período de tiempo, el Circuit Breaker "se abre" y pasa al estado Abierto.

2. **Abierto (Open):**

- En este estado, el Circuit Breaker bloquea **inmediatamente** todas las nuevas solicitudes al servicio remoto.
- Las solicitudes no llegan al servicio remoto; en su lugar, el Circuit Breaker devuelve una excepción o una respuesta de "fallback" de forma instantánea.
- Después de un período de tiempo predefinido, el Circuit Breaker pasa al estado Semi-abierto.

3. **Semi-abierto (Half-Open):**

- Este estado es un "sondeo" para ver si el servicio remoto se ha recuperado.
- El Circuit Breaker permite que un **número limitado** de solicitudes pasen al servicio remoto.
- Si estas solicitudes de prueba tienen éxito, el Circuit Breaker asume que el servicio se ha recuperado y vuelve al estado Cerrado.
- Si estas solicitudes de prueba fallan, el Circuit Breaker asume que el servicio sigue fallando y vuelve al estado Abierto por otro período de tiempo.

Implementación en Java

Proyecto weather.loader:

El patrón Circuit Breaker es una técnica esencial para hacer la aplicación más resiliente frente a fallos externos, como caídas de servicios, APIs lentas o errores en red. Es propio de Resilience4j, y funciona gracias a AOP de Spring

Es un patrón de diseño que detecta fallos repetidos en una operación (como una llamada HTTP o a una base de datos) y "abre" el circuito para cortar temporalmente el tráfico a ese recurso y así:

- Evitar seguir fallando innecesariamente
- Proteger el sistema de sobrecargas
- Dar tiempo a que el servicio externo se recupere

Se decidió implementar Circuit Breaker en este proyecto ya que aquí es donde se hace el llamado a la api externa `api.openweathermap.org` para obtener los datos del clima. Nuestro servicio tiene configurado un proceso interno llamado `WeatherLoaderComponent` que se ejecuta cada 1 hora, consulta el clima en ese instante y guarda la información en base de datos. Al no ser una acción realizada por un usuario, decidimos que era el mejor lugar para implementarlo ya que las llamadas al api son programadas y podríamos establecer un fallback en caso de error.

Para poder implementar Circuit Breaker, en nuestra clase `Application` primero tuvimos que agregar la anotación `@EnableAspectJAutoProxy`, que habilita el procesamiento de aspectos AOP (Programación Orientada a Aspectos) para poder aplicar lógica antes/después/en caso de excepción (lo que hace el Circuit Breaker). Luego agregar la anotación `@CircuitBreaker(name = "myDataCircuitBreaker", fallbackMethod = "fallbackRandomActivity")` sobre el método que ejecuta el proceso. Esta anotación le da el nombre `myDataCircuitBreaker` al Circuit Breaker y en caso de que el código que hay dentro del proceso diera error, se llamaría automáticamente al método `fallbackRandomActivity` para reemplazar su ejecución.

```
@Scheduled(fixedRate = 3600000) // 1 hora
@CircuitBreaker(name = "myDataCircuitBreaker", fallbackMethod = "fallbackRandomActivity")
public void fetchAndSendWeather() throws Exception {

    Logger.info("Ejecutando proceso...");

    String url = String.format(weatherUrl, city, apiKey);

    String response = getUrl(url);

    com.fasterxml.jackson.databind.JsonNode node = new com.fasterxml.jackson.databind.ObjectMapper().readTree(response);

    double temp = node.get("main").get("temp").asDouble();
    String weatherDescription = node.get("weather").get(0).get("description").asText();

    // Crear objeto WeatherData
    WeatherDataDTO weatherData = new WeatherDataDTO(city, temp, weatherDescription, LocalDateTime.now());

    // Guardar en MongoDB
    weatherLoaderService.createWeatherData(weatherData);

    Logger.info("Temperatura actual guardada");
    System.out.println("Se guardo la temperatura actual para el día " + weatherData.getTimestamp());

}
```

El método de fallback tiene que tener el mismo tipo de respuesta que el método `fetchAndSendWeather()`, que ejecuta el proceso. En nuestro caso no hacemos nada y solamente imprimimos el error por consola y en log4j para analizarlo en el momento, y así evitar que el resto del código del proceso se siga ejecutando.

```
/**
 * Metodo para fallback
 * En caso de que el metodo fetchAndSendWeather de algun error, el circuitbreaker llamara a este metodo
 * Y el Scheduled seguira funcionando
 */
public void fallbackRandomActivity(Throwable t) {

    Logger.error("Fallback activado, error: ", t);
    System.err.println("getFallback "+new Date().toString());

}
```

Para configurar el circuito, en las properties `application.properties` agregamos:

resilience4j.circuitbreaker.configs.default.sliding-window-size=5

Se consideran las últimas 5 llamadas para calcular la tasa de error.

resilience4j.circuitbreaker.configs.default.minimum-number-of-calls=2

Mínimo de 2 llamadas para empezar a evaluar fallos.

resilience4j.circuitbreaker.configs.default.failure-rate-threshold=50

Porcentaje de fallos permitido antes de abrir el circuito. Si más del 50% de las llamadas fallan en la ventana definida, el Circuit Breaker se abre.

resilience4j.circuitbreaker.configs.default.automatic-transition-from-open-to-half-open-enabled=true

Habilita la transición automática de OPEN a HALF_OPEN después del tiempo de espera definido por wait-duration-in-open-state.

resilience4j.circuitbreaker.configs.default.wait-duration-in-open-state=20s

El tiempo que el Circuit Breaker permanece en estado OPEN antes de intentar pasar a HALF_OPEN. Durante ese tiempo, todas las llamadas fallan automáticamente (van al método de fallback).

resilience4j.circuitbreaker.configs.default.permitted-number-of-calls-in-half-open-state=3

Cuando el circuito pasa a HALF_OPEN, esta es la cantidad de llamadas de prueba permitidas para ver si se cierra o vuelve a abrir.

Request Caché

La caché es un mecanismo que permite almacenar temporalmente datos, con el objetivo de mejorar el rendimiento y la eficiencia de las aplicaciones. Dentro de las estrategias de caché existentes, se implementó la estrategia **Server-side Cache** mediante la biblioteca Caffeine y la anotación `@Cacheable`, mencionadas a continuación.

El caching del lado del servidor implica almacenar datos en la memoria del servidor para que puedan ser recuperados rápidamente en futuras solicitudes. Esto es útil para reducir la carga en los recursos del servidor y mejorar el rendimiento de la aplicación. Esto se debe a que:

- **Mejora del Rendimiento:** Almacenar datos frecuentemente solicitados en caché reduce la latencia en las respuestas, ya que evita la necesidad de realizar consultas repetidas a la bases de datos o servicios externos. Esto es especialmente importante en sistemas que requieren respuestas rápidas, como el WeatherMetricsComponent, que necesita acceder a los datos del clima actual de eficiente, dado que el estado del clima actual se actualiza cada hora.
- **Reducción de Carga en el Sistema:** La caché ayuda a disminuir la carga en las bases de datos y otros servicios, permitiendo que el sistema maneje un mayor volumen de solicitudes sin comprometer el rendimiento. Esto es fundamental en un entorno de microservicios donde múltiples componentes pueden estar accediendo a los mismos datos. En nuestra implementación, al almacenar los resultados de las llamadas a `obtenerTemperaturaActual()`, se evita la necesidad de realizar la misma operación repetidamente, lo que puede ser costoso en términos de tiempo y recursos.

- **Escalabilidad:** Al utilizar caché, los microservicios pueden escalar de manera más efectiva, ya que se reduce la necesidad de recursos para manejar solicitudes repetitivas. Esto permite que los equipos se enfoquen en mejorar la funcionalidad y la calidad del software.

Implementación en Java

Se implementó una caché que almacena la temperatura actual durante 1 hora, para poder hacerlo Java cuenta con la anotación `@Cacheable` de Spring, y con *Caffeine*, una librería de alta performance de caché que almacena datos de acceso frecuente en memoria, permitiendo establecer políticas de expiración, como el tiempo de vida (TTL), para que los datos en caché se eliminen automáticamente después de un período específico.

Juntos proporcionan un mecanismo eficiente para gestionar la caché en aplicaciones Java. Permitiendo mejorar el rendimiento al reducir la carga en la base de datos y optimizar el tiempo de respuesta al almacenar datos frecuentemente solicitados en memoria.

Proceso de Consulta a la Base de Datos

- **Consulta a la Caché:** Cuando se invoca un método anotado con `@Cacheable`, Caffeine primero verifica si el resultado ya está en caché.
- **Cache Hit:** Si el resultado está presente, se devuelve directamente desde la caché, evitando la consulta a la base de datos.
- **Cache Miss:** Si el resultado no está en caché, se realiza la consulta a la base de datos para obtener los datos necesarios. Una vez recuperados, se almacenan en la caché para futuras solicitudes.

Estrategias de Observabilidad

La observabilidad es un aspecto crucial en el desarrollo y mantenimiento de sistemas complejos, ya que permite a los equipos de desarrollo y operaciones comprender el estado y el comportamiento de sus aplicaciones en tiempo real. Una de las estrategias más efectivas para lograr una buena observabilidad es la *agregación de métricas*.

¿Qué es la Agregación de Métricas?

La agregación de métricas se refiere al proceso de recopilar, procesar y resumir datos de rendimiento y comportamiento de un sistema en un formato que sea fácil de analizar y visualizar. Esto implica la recopilación de datos de diversas fuentes, como servidores, aplicaciones y servicios, y su consolidación en un único sistema de monitoreo.

Beneficios de la Agregación de Métricas

- **Visibilidad Centralizada:** Al agregar métricas de diferentes componentes del sistema, se obtiene una vista unificada del rendimiento general. Esto facilita la identificación de problemas y la toma de decisiones informadas.

- **Detección de anomalías:** La agregación permite establecer umbrales y patrones de comportamiento esperados, lo que ayuda a detectar anomalías y problemas antes de que afecten a los usuarios finales.
- **Análisis de Tendencias:** Con datos agregados, es posible realizar análisis de tendencias a lo largo del tiempo, lo que permite a los equipos anticipar problemas y planificar mejoras en la infraestructura.
- **Optimización de Recursos:** La observación de métricas agregadas puede revelar ineficiencias en el uso de recursos, permitiendo a los equipos optimizar el rendimiento y reducir costos.

Implementación de métricas

Para la implementación en Java, es necesario configurar en nuestro proyecto *MeterRegistry* (eso lo hacemos a través de la dependencias de *Micrometer*), que es el componente central para la recopilación de métricas. En nuestro proyecto, lo utilizamos para crear contadores que registren el número de solicitudes a diferentes endpoints, servicios, métricas del hardware, además de la temperatura más baja y alta del último día y la última semana.

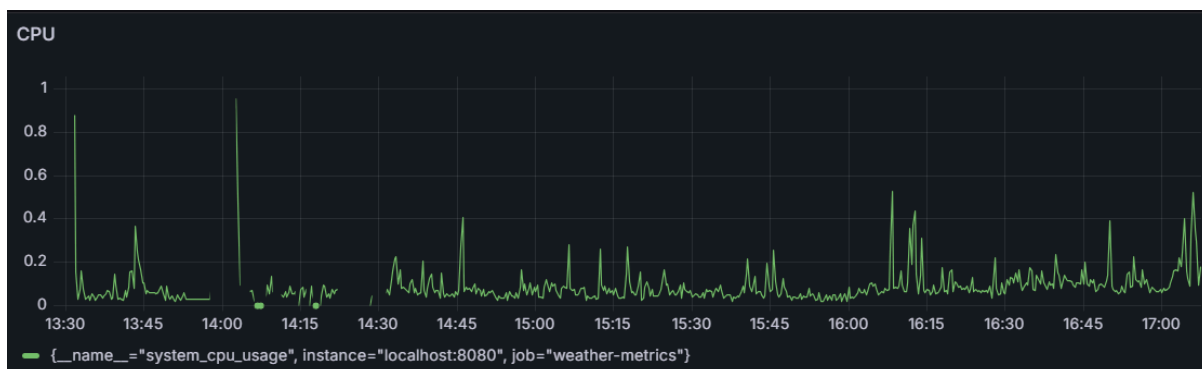
Una vez que las métricas están configuradas y recopiladas, se exponen a *Prometheus*. Para ello, nos proporciona el siguiente endpoint que expone las métricas en un formato que prometheus puede entender:

<http://localhost:8080/actuator/prometheus>

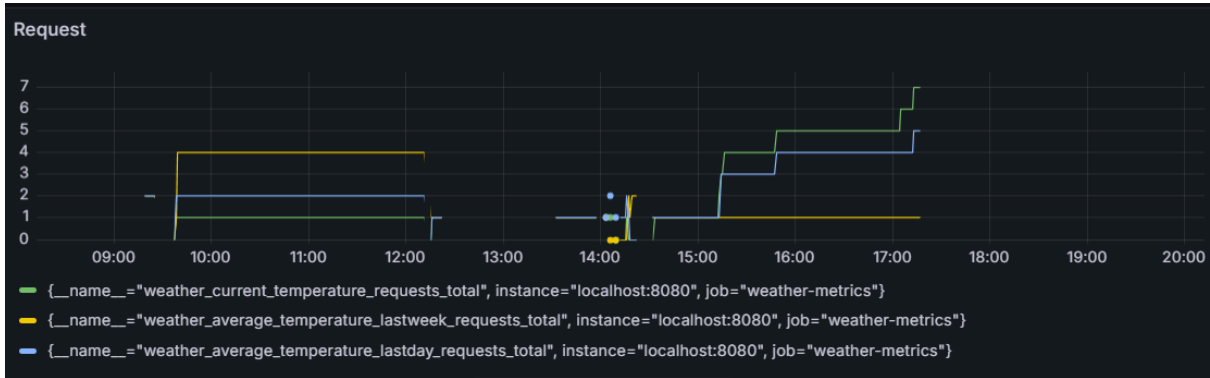
Por último, para visualizar las métricas desde *Grafana*, y cuando el mismo se encuentre corriendo se debe de agregar *Prometheus* como una fuente de datos. Proporcionando la URL donde Prometheus está corriendo (en nuestro caso, <http://localhost:9090>). Cuando eso ya está listo, se crean los dashboards correspondientes pasándole los counters.

Dashboards

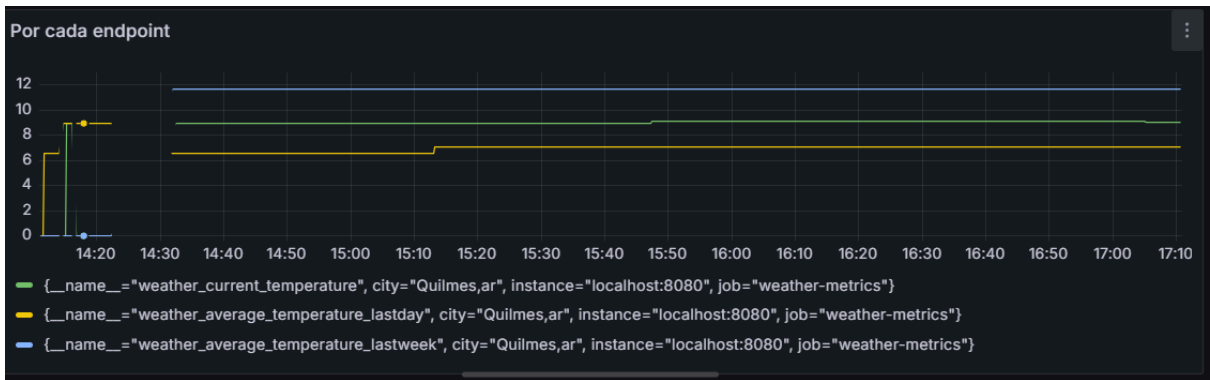
Métricas de hardware



Métricas de cada endpoint

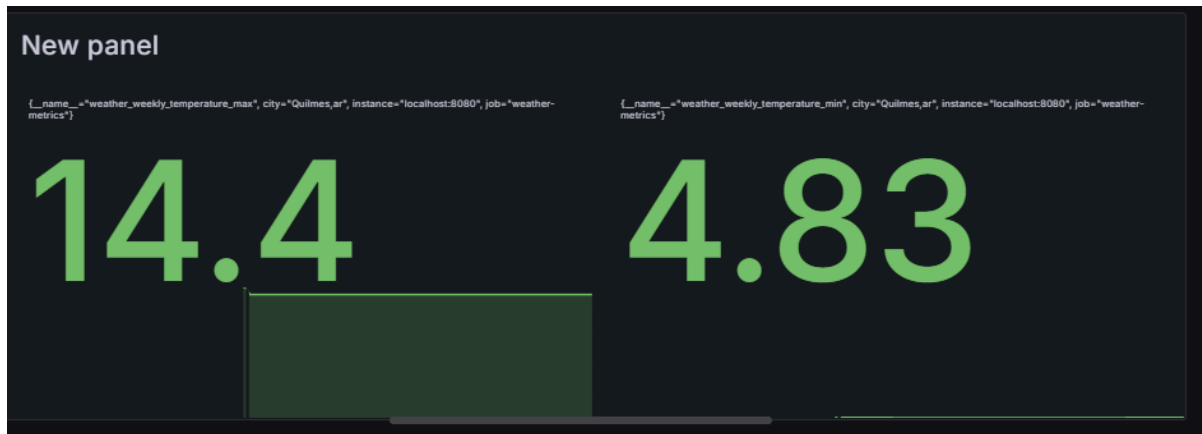


Métricas de negocio

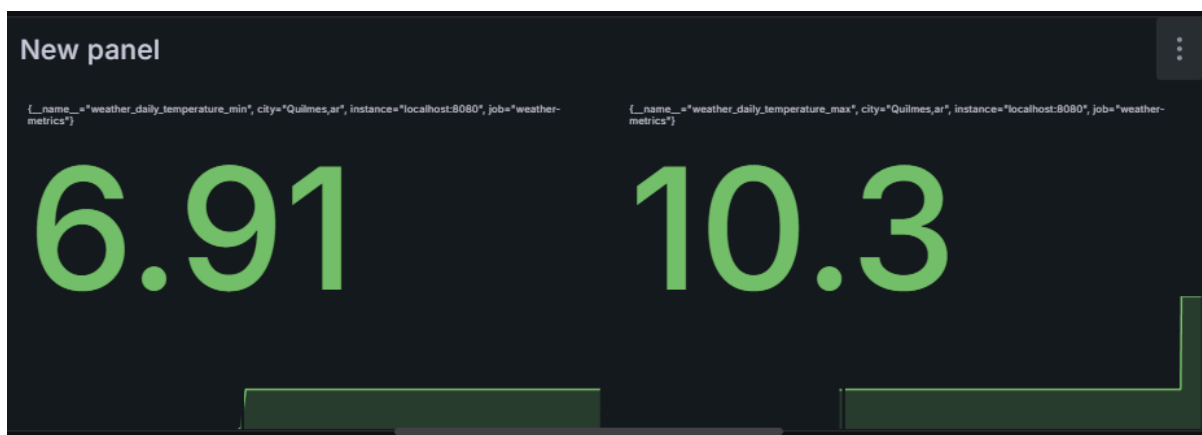
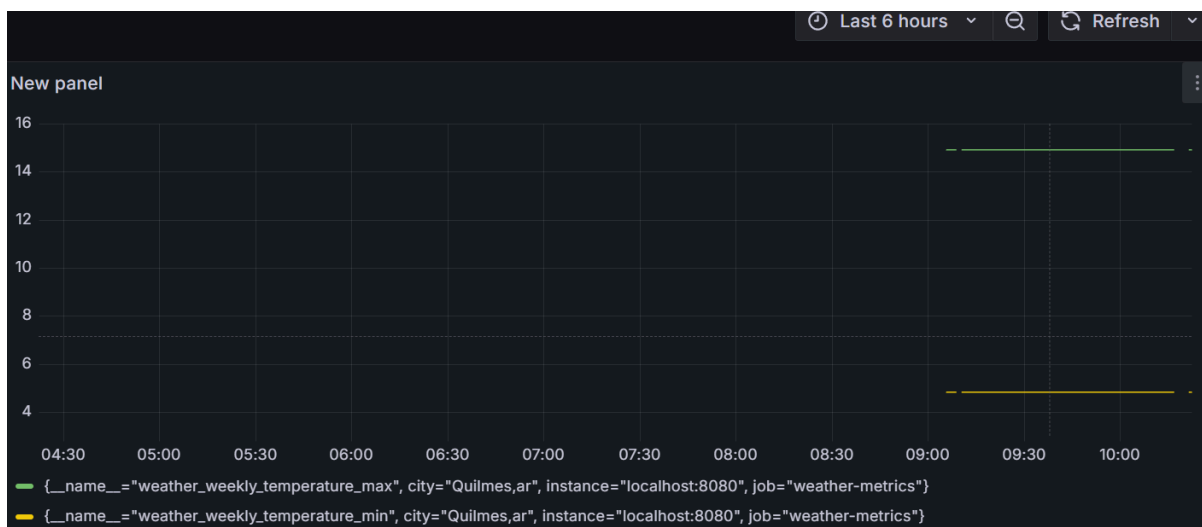


Temperatura diaria mínima y máxima





Temperatura semanal mínima y máxima



Alerting

Nuestra estrategia de alertamiento se basa en el principio de que las alertas deben ser relevantes, únicas, a tiempo, priorizadas, entendibles, claras, informativas y enfocadas. Para ello, definimos umbrales específicos para métricas críticas, permitiéndonos actuar en consecuencia.

El ciclo de vida de un alert es el siguiente:

Piensa en el ciclo de vida de una alerta:

- **Normal/Resolved:** La condición de la alerta no se cumple. Todo está bien.
- **Pending:** La condición de la alerta se ha cumplido por primera vez, pero aún no ha pasado el tiempo mínimo configurado.
- **Firing:** La condición de la alerta se ha cumplido y se ha mantenido verdadera durante el período FOR configurado. En este punto, la alerta está completamente activa y el sistema de notificación de Grafana procederá a enviar las notificaciones.
- **Resolved/Normal:** La condición de la alerta deja de cumplirse. Después de un tiempo de gracia, la alerta volverá al estado "Normal" y Alertmanager enviará una notificación de resolución (si está configurado).

Ejemplo: Uso de CPU del Sistema

Para monitorear el uso de CPU de todo el sistema donde reside nuestra aplicación, hemos configurado una alerta en Grafana.

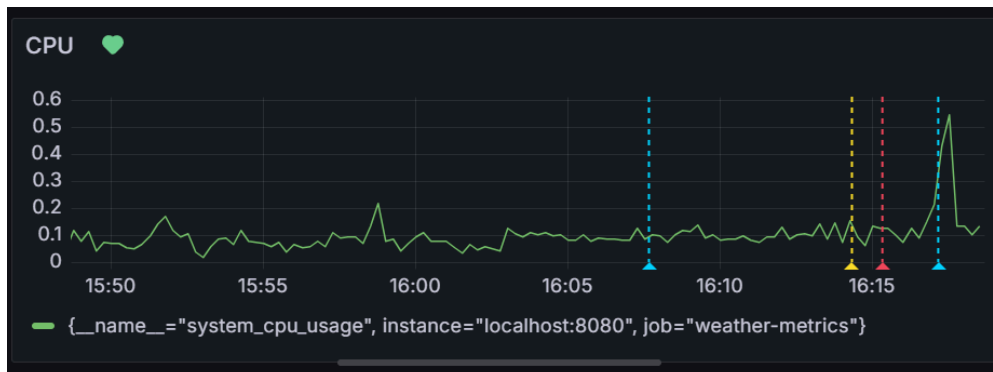
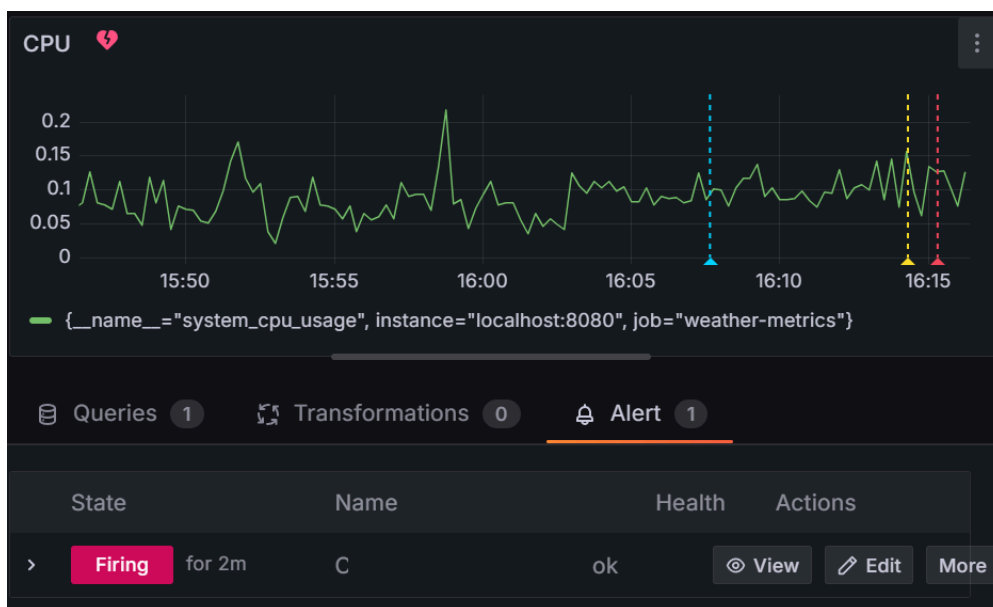
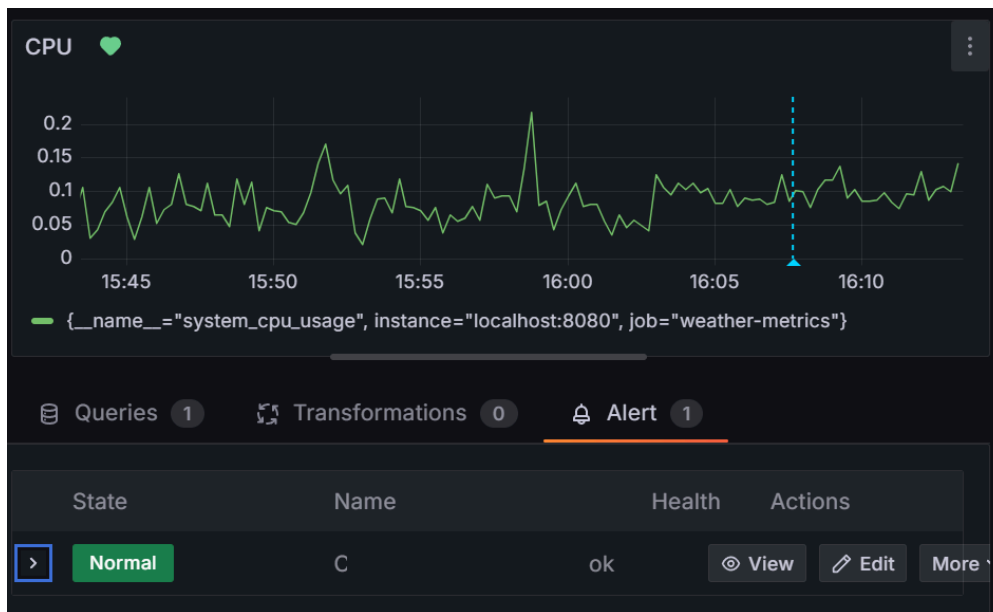
- **Métrica Monitoreada:** `system_cpu_usage` (Uso de CPU de todo el sistema en el que se ejecuta la aplicación, en el rango de 0.0 a 1.0).
- **Instancia Monitoreada:** `instance="localhost:8080"` (identificación del servidor).
- **Alerta de Advertencia (Warning - 80%):** Si el valor instantáneo de `system_cpu_usage` es superior a `0.8`.
- **Propósito:** Nos notifica inmediatamente sobre un alto consumo de CPU en la máquina, lo que puede indicar una sobrecarga general del servidor o un comportamiento inesperado de la aplicación u otros procesos.
- **Comentarios:** Sin embargo, para un entorno de producción más exigente y para reducir la 'fatiga de alertas' por fluctuaciones temporales, se debería de implementar alertas para que se basen en un **promedio sostenido del uso de CPU durante un período de tiempo**.

The screenshot shows the Grafana Alerting configuration page for a CPU usage alert. The alert is currently in a 'Normal' state. The configuration includes a query for 'system_cpu_usage' from 'prometheus' and a threshold condition set to 'Is above 0'. The alert is currently firing, as indicated by the 'Firing' status at the bottom.

Alert Configuration Details:

- Alert Name:** CPU
- Status:** Normal
- Query and conditions:**
 - Query:** `system_cpu_usage{instance="localhost:8080"}`
 - Condition:** Threshold (Alert condition)
 - Input:** A Is above 0
- Table:**

Label	Value
<code>{__name__="system_cpu_usage", instance="localhost:8080", job="weather-metrics"}</code>	0.15698
- Alert Status:** 1 Firing



En el gráfico de consumo de CPU, podemos observar la visualización de los diferentes estados de las alertas:

- La línea verde continua representa el comportamiento normal de la métrica *system_cpu_usage*.

- Cuando la métrica cruza un umbral y la alerta entra en estado 'Pending', se indica mediante una línea vertical amarilla punteada. Esto significa que la condición de la alerta se ha cumplido, pero aún no ha pasado el tiempo configurado para su disparo definitivo.
- Si la condición se mantiene durante el tiempo establecido y la alerta pasa al estado 'Firing', se visualiza con una línea vertical roja punteada, indicando que la alerta está activa y se están enviando las notificaciones correspondientes.
- Las líneas verticales azules punteadas marcan los momentos en que la condición de la alerta se resuelve, y el sistema vuelve a un estado de normalidad. Esto puede ocurrir después de un estado 'Pending' o 'Firing'.

Especificaciones de las APIs

Para asegurarnos que nuestros microservicios sean interoperables, fáciles de consumir y consistentes, hemos adoptado los principios de diseño RESTful (*Representational State Transfer*) para la exposición de nuestras APIs. Estos principios ayudan a crear APIs escalables, eficientes y fáciles de utilizar.

- **Cliente-servidor:** El cliente y el servidor están completamente separados y son independientes. En nuestro sistema la interfaz de usuario (cliente) y el backend (servidor) están desacoplados, por lo que ambos se comunican mediante solicitudes y respuestas.
- **Stateless:** La comunicación entre el cliente y el servidor debe de ser sin estado, lo que implica que no se almacenan ni se comparte información entre peticiones. Siendo las mismas independientes y contienen sólo la información que es necesaria para procesarlas. En nuestra aplicación, en las solicitudes mandamos solamente la información necesaria, y no dependemos de datos almacenados en el servidor.
- **Cacheable:** cuando sea necesario, es posible almacenar los datos de nuestras entidades en caché, ya sea del lado cliente o del lado del servidor, para mejorar el rendimiento de nuestra aplicación.
- **Sistema en capas:** nuestra arquitectura presenta múltiples capas, donde cada capa tiene una función específica y puede interactuar con otras capas sin que el cliente conozca la estructura interna. En la siguiente imagen se puede observar como es la división en capas de cada uno de nuestros servicios.
- **Identificador Único:** cada recurso tiene un identificador único e irrepetible, no pueden existir más de un recurso con el mismo identificador en la red.
- **Uso correcto de HTTP:** Rest debe de respetar los verbos y códigos de estado para cada operación. En nuestra aplicación se podrán encontrar los siguientes métodos:
 - **GET:** Se utiliza para recuperar los datos de un recurso en específico.

Listado de endpoints

1. Servicio: Loader
 - <http://localhost:8085/current>

```
{
  "id": "685d484062916235fd55aa8f",
  "city": "Quilmes,ar",
  "weather": "clear sky",
  "temperature": 6.89,
  "timestamp": "2025-06-26T10:16:48.221"
}
```

2. Servicio: Metrics

- <http://localhost:8080/weather/current>

```
{
  "id": "685d484062916235fd55aa8f",
  "city": "Quilmes,ar",
  "weather": "clear sky",
  "temperature": 6.89,
  "timestamp": "2025-06-26T10:16:48.221"
}
```

- <http://localhost:8080/weather/average/today>

```
{
  "Quilmes,ar": 6.316666666666666
}
```

- <http://localhost:8080/weather/average/week>

```
{
  "Quilmes,ar": 6.910303030303031
}
```

Pruebas de Carga

En el mundo de la tecnología, donde cada día surgen nuevas aplicaciones y sistemas innovadores, el rendimiento y la estabilidad se han convertido en factores críticos para el éxito de cualquier proyecto. Nada puede desalentar más a los usuarios que una aplicación lenta, inestable o que no responde cuando más se necesita.

Para abordar estos desafíos, las **pruebas de rendimiento** —y específicamente las **pruebas de carga y de estrés**— han permitido a los equipos evaluar el desempeño y la resistencia de sus aplicaciones y sistemas bajo condiciones extremas.

Las pruebas de carga, por ejemplo, son un tipo de prueba de rendimiento que se realiza para entender cómo se comporta una aplicación (un sitio web, una API, un software, etc.) bajo una carga de trabajo esperada o simulada. Cuyo objetivo principal es determinar la estabilidad, el tiempo de respuesta, la escalabilidad y el uso de recursos del sistema cuando un número específico de usuarios o transacciones interactúan con él.

Implementación

Para realizar las pruebas de carga, decidimos utilizar el método que nos devuelve el clima semanal promedio ya que es el que con más datos tiene que trabajar.

GET <http://localhost:8080/weather/average/week>

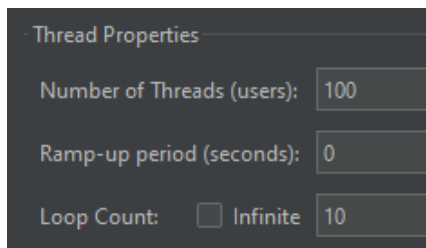
Este llamado tiene un timeout de respuesta de 2 segundos para brindarle al usuario una respuesta inmediata sobre el promedio del clima y también porque es un método de recuperación de datos, y no un método transaccional que quizás necesita más tiempo para terminar la operación.

Como herramienta para lograr nuestro objetivo elegimos Apache Jmeter, una herramienta de testing de rendimiento y carga 100% gratuita y open-source, desarrollada por la Fundación Apache.

La instalamos y desde la consola lo ejecutamos y se abrirá una ventana donde podemos configurar nuestros tests.

Primera prueba: Requests ejecutados de forma simultánea

En esta primera prueba de carga, simulamos 100 usuarios concurrentes que invocan el servicio 10 veces cada uno, generando un total de 1000 peticiones. La configuración se hizo con 0 segundos de ramp-up, lo que implica que todos los usuarios comienzan al mismo tiempo.



Thread Properties

Number of Threads (users): 100

Ramp-up period (seconds): 0

Loop Count: ☐ Infinite 10

Ejecutada la prueba, estos son los resultados:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Get promedio ...	1000	692	52	2052	429.71	0.20%	119.3/sec	23.77	15.84	204.1
TOTAL	1000	692	52	2052	429.71	0.20%	119.3/sec	23.77	15.84	204.1

En la primera prueba de carga, el servicio procesó 1000 requests con un tiempo promedio de respuesta de 692 ms. El throughput fue de 119.3 requests/segundo, con una tasa de error mínima (0.2%). Sin embargo, se observaron algunas respuestas lentas, alcanzando hasta 2052 ms, lo que sugiere posibles cuellos de botella bajo alta concurrencia.

Segunda prueba: Requests ejecutados de forma distribuida

Como segunda prueba de carga, configuramos la ejecución para que los requests no se disparen de forma simultánea, sino que se distribuyan a lo largo del tiempo. Esto permite simular un comportamiento más realista, en el que los usuarios acceden al sistema de manera progresiva.

Thread Properties

Number of Threads (users): 100

Ramp-up period (seconds): 10

Loop Count: ☐ Infinite 10

De esta prueba, obtuvimos los siguientes resultados:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Get promedio ...	1000	55	44	105	6.91	0.00%	95.7/sec	19.07	12.72	204.0
TOTAL	1000	55	44	105	6.91	0.00%	95.7/sec	19.07	12.72	204.0

En esta segunda prueba, al simular una carga progresiva (ramp-up), el servicio mostró un rendimiento mucho más estable y veloz, con un tiempo de respuesta promedio de apenas 55 ms, sin errores, y con una desviación estándar mínima. Si bien el throughput bajó levemente a 95.7 req/s, la calidad de las respuestas y la ausencia total de fallos lo hacen un resultado excelente.

Tercera prueba: Apuntamos a una base de datos inaccesible con timeout de 3 segundos

Como tercera prueba, configuramos el sistema para que apunte a una base de datos inaccesible, simulando una caída o pérdida de conectividad. El objetivo es evaluar el comportamiento ante fallos en servicios críticos. El timeout de conexión a la base está configurado en 3 segundos, y se reutiliza la misma configuración de carga que en la prueba anterior.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Get promedio ...	1000	2015	2003	2220	16.57	100.00%	33.3/sec	8.62	4.43	265.0
TOTAL	1000	2015	2003	2220	16.57	100.00%	33.3/sec	8.62	4.43	265.0

En esta tercera prueba, se configuró el sistema para apuntar a una base de datos inaccesible, simulando una caída del servicio loader, que es el componente encargado de acceder a la base. El servicio metrics, que depende de loader, tiene configurado un timeout de 2 segundos para dicha comunicación.

Durante la ejecución se enviaron 1000 solicitudes, de las cuales el 100% falló. Todas las peticiones presentaron un tiempo promedio de respuesta de 2015 ms, y un mínimo de 2003 ms, lo que confirma que las respuestas se cortaron cerca del límite de timeout configurado.

Además, el throughput disminuyó drásticamente a 33.3 solicitudes por segundo, lo que representa un cuello de botella significativo en contextos de alta concurrencia, dado que las conexiones permanecen abiertas durante los intentos fallidos.

Cuarta prueba: Sin timeouts entre servicios ni hacia la base de datos

En esta prueba se configuró el sistema para eliminar los timeouts tanto en la comunicación del servicio metrics hacia loader, como en el acceso de loader a la base de datos, la cual se encuentra intencionalmente inaccesible.

El objetivo fue analizar el comportamiento del sistema en un escenario extremo donde no existen límites de espera ante fallos en servicios críticos.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Get promedio t...	1000	30089	30005	33678	410.44	100.00%	3.2/sec	0.83	0.43	265.0
TOTAL	1000	30089	30005	33678	410.44	100.00%	3.2/sec	0.83	0.43	265.0

Como resultado, cada request quedó completamente bloqueado, generando un tiempo promedio de respuesta de 30 segundos, con un throughput de apenas 3.2 solicitudes por segundo, y una tasa de errores del 100%.

Este comportamiento pone en evidencia el riesgo de no establecer límites de tiempo en servicios interdependientes.

El sistema quedó virtualmente inutilizable y vulnerable a saturación, demostrando la necesidad crítica de establecer timeouts controlados y mecanismos de resiliencia.

Quinta prueba: Carga realista a gran escala (1000 usuarios)

En esta prueba se mantuvo la misma configuración que en el segundo escenario, con timeouts correctamente definidos y una carga escalonada gracias al ramp-up, pero se aumentó significativamente la cantidad de usuarios concurrentes: de 100 a 1000 usuarios.

El objetivo fue evaluar cómo se comporta el sistema en condiciones de uso masivo, pero bajo un patrón de acceso realista (no simultáneo), para verificar su escalabilidad.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Get promedio t...	10000	6548	442	9745	1782.44	16.50%	130.2/sec	27.31	17.29	214.9
TOTAL	10000	6548	442	9745	1782.44	16.50%	130.2/sec	27.31	17.29	214.9

El sistema logró un throughput estable y alto de 130.2 req/seg, lo cual demuestra una buena capacidad de procesamiento. Sin embargo, este volumen generó un incremento significativo en los tiempos de respuesta (6.5 segundos en promedio) y una tasa de errores del 16.5%.

Esto indica que el sistema comienza a presentar síntomas de saturación cuando se expone a una concurrencia masiva, con una degradación evidente de la experiencia del usuario.

Esta prueba es fundamental para delimitar la capacidad operativa máxima antes de que sea necesario escalar o reforzar la arquitectura.

Conclusiones

Las cinco pruebas realizadas demostraron que el uso (o ausencia) de timeouts tiene un impacto directo y significativo en la estabilidad, rendimiento y resiliencia de la aplicación.

En los escenarios donde se definieron timeouts claros entre servicios y hacia la base de datos (Pruebas 2 y 5), el sistema logró mantenerse funcional, incluso bajo alta carga, mostrando buenos niveles de throughput y latencias controladas, aunque con algunas fallas esperables por saturación.

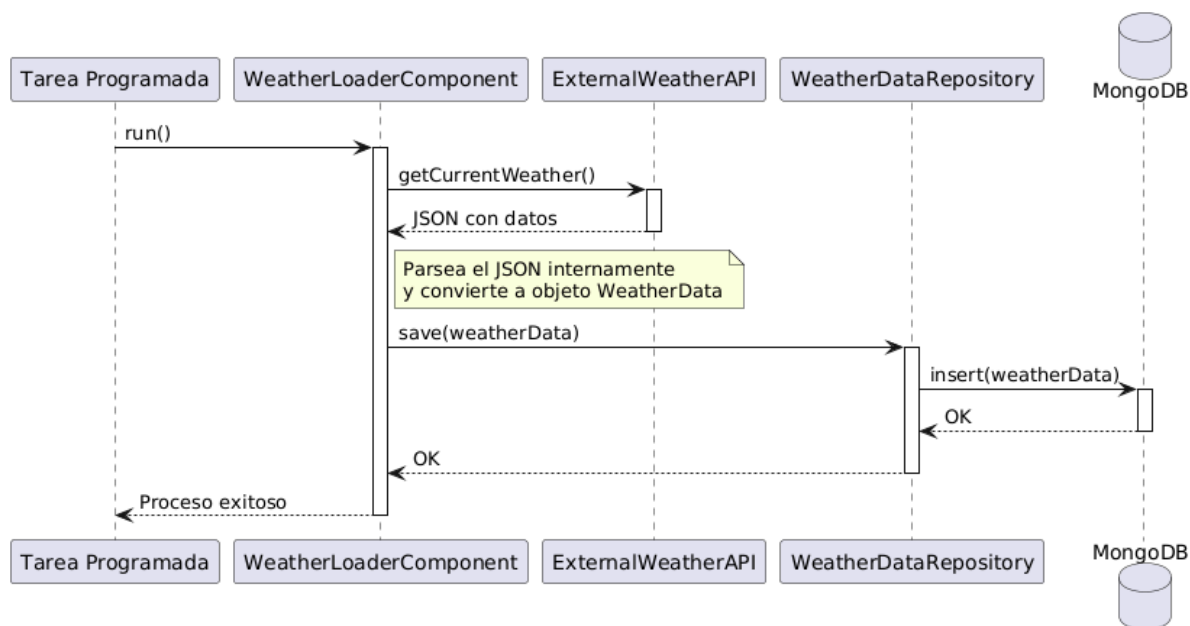
Por el contrario, en los escenarios sin timeouts (Pruebas 3 y 4), el sistema quedó completamente bloqueado. Esto evidencia que los timeouts no son un detalle técnico menor, sino un mecanismo esencial de defensa para:

- Evitar bloqueos prolongados
- Liberar recursos ante fallas externas
- Proteger al sistema de la propagación de errores
- Mantener la experiencia del usuario en condiciones aceptables

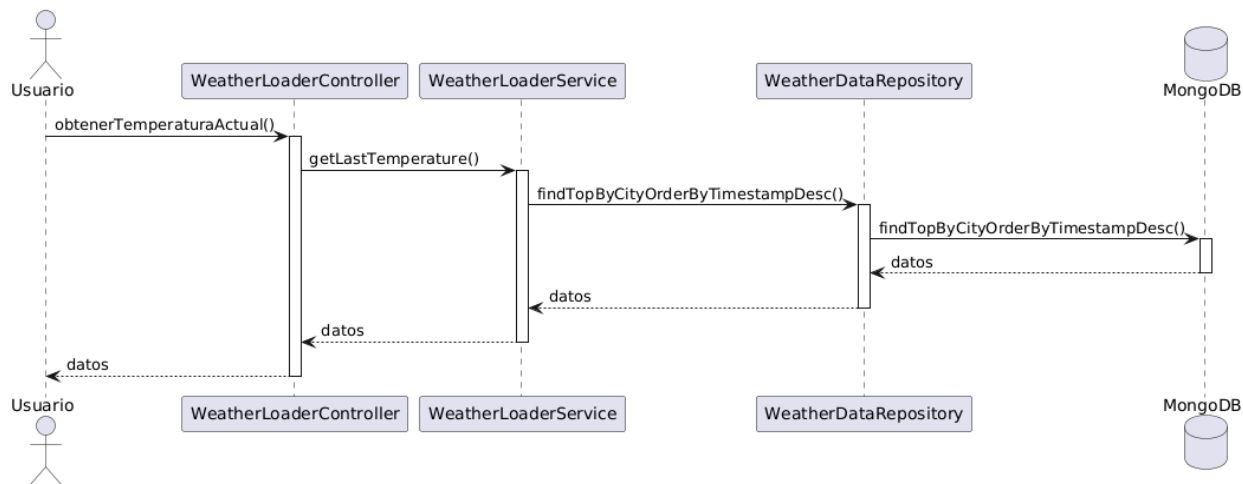
Diagrama de secuencia

Weather.loader

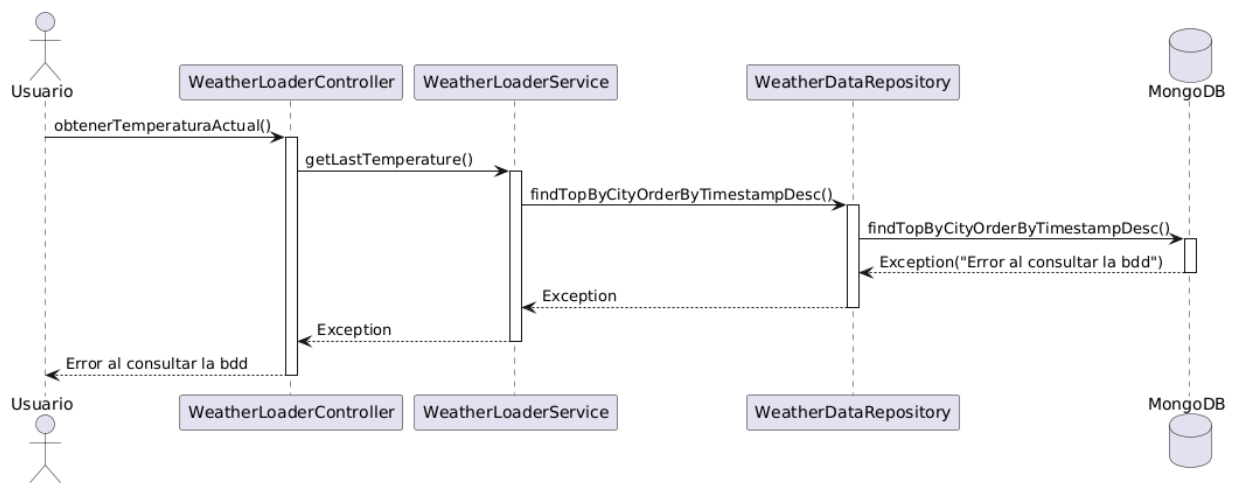
Proceso batch interno para obtener datos de la api externa y persistirlos en una base Mongo



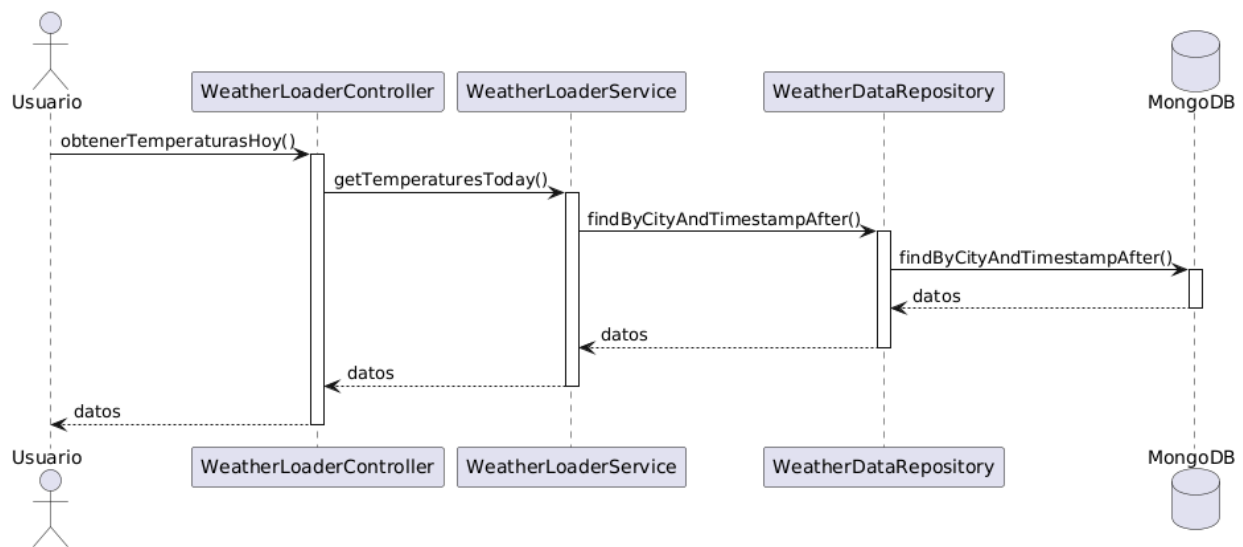
Obtener clima de hoy



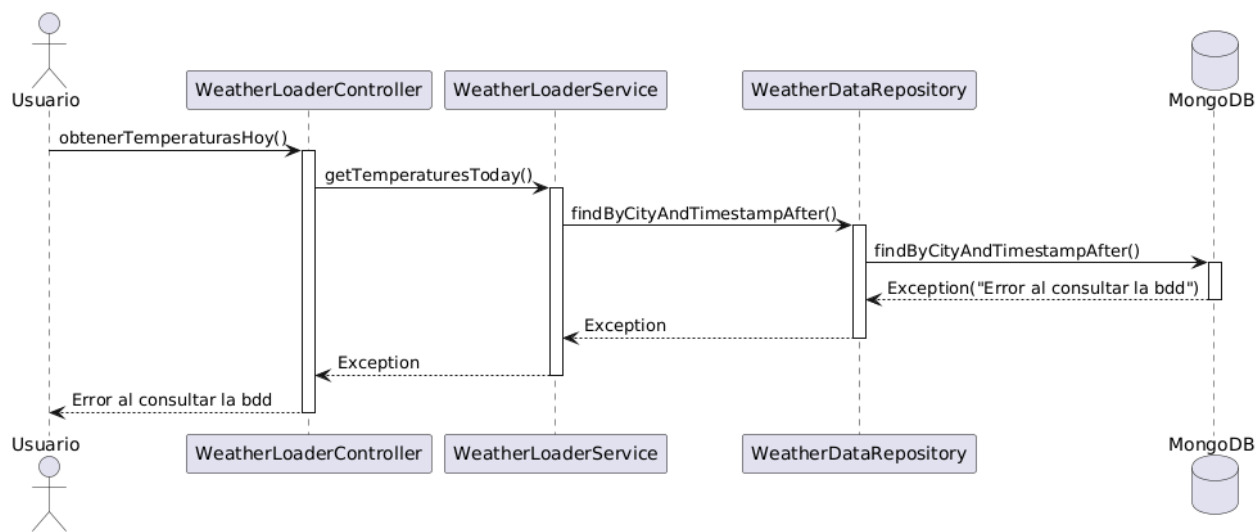
Obtener clima de hoy - error al acceder a la base de datos



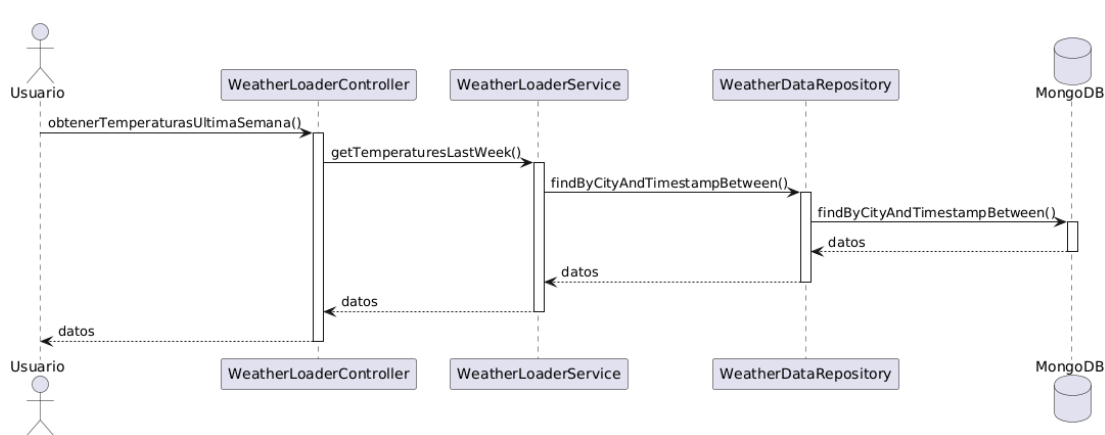
Obtener temperaturas de hoy



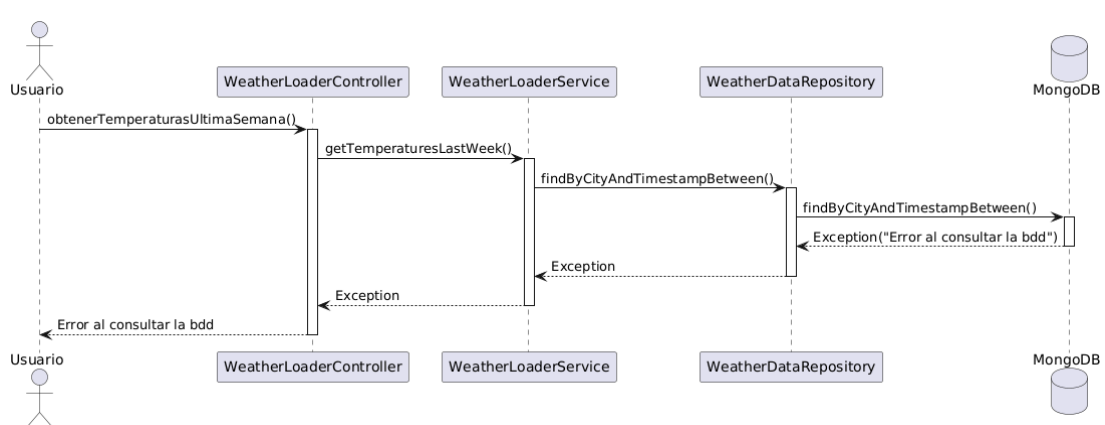
Obtener temperaturas de hoy - error al acceder a la base de datos



Obtener temperaturas de la última semana

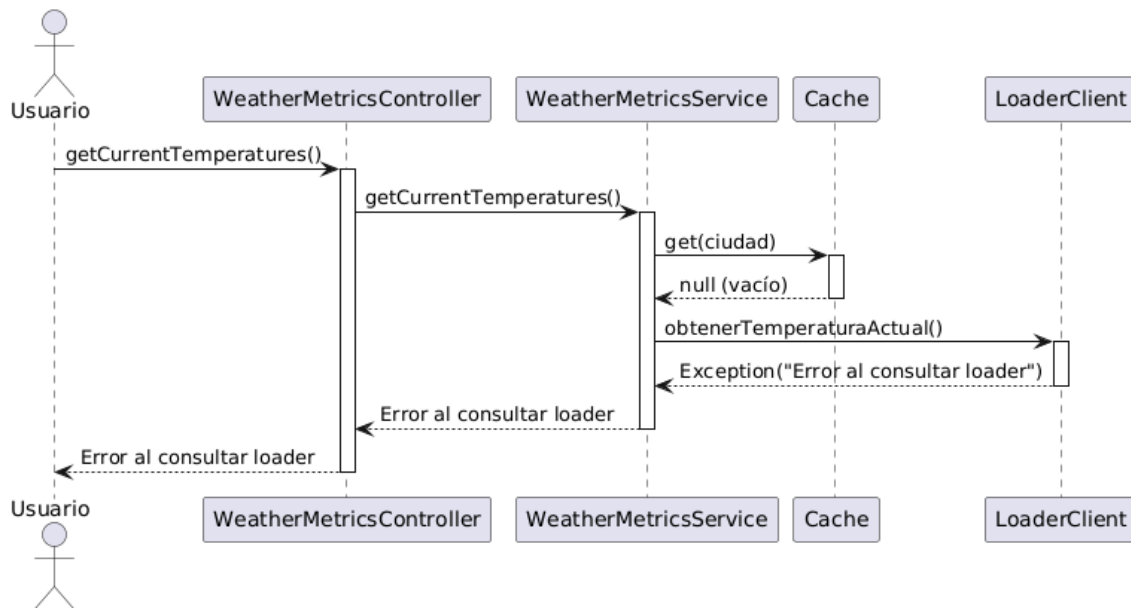


Obtener temperaturas de la última semana - error al acceder a la base de datos

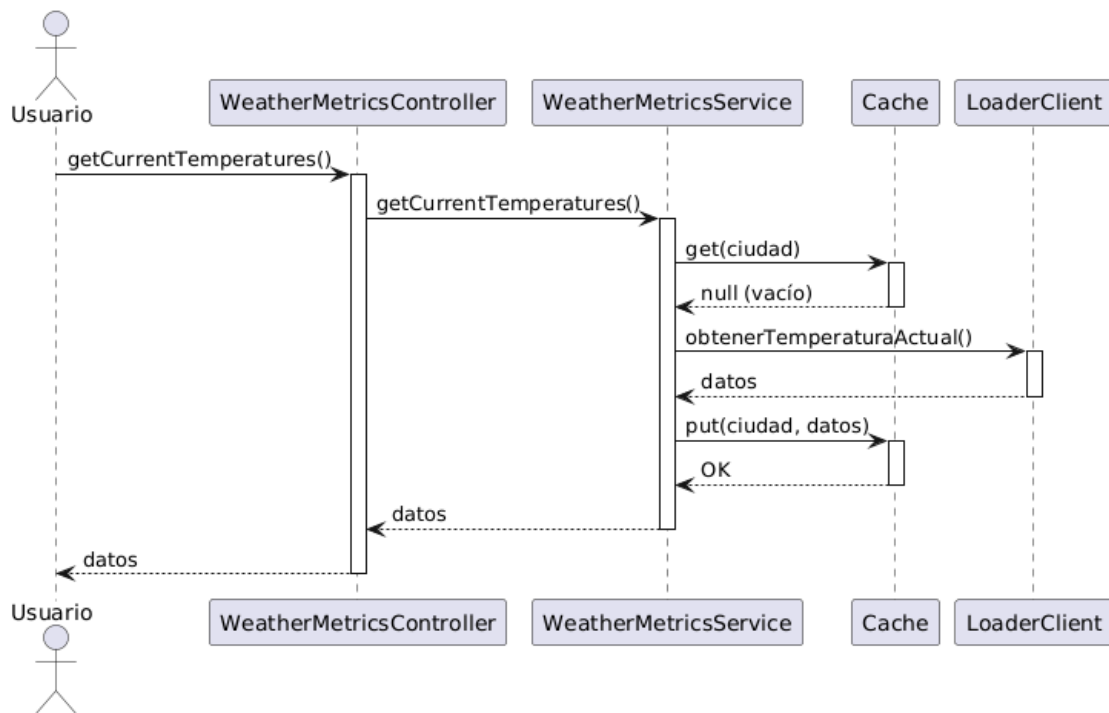


Weather.metrics

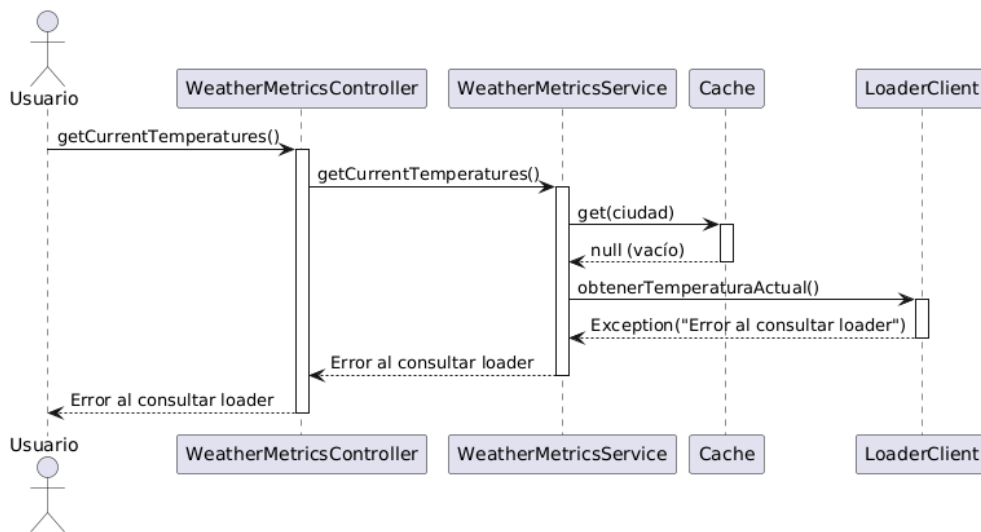
Obtener clima de hoy - llamando al api externo



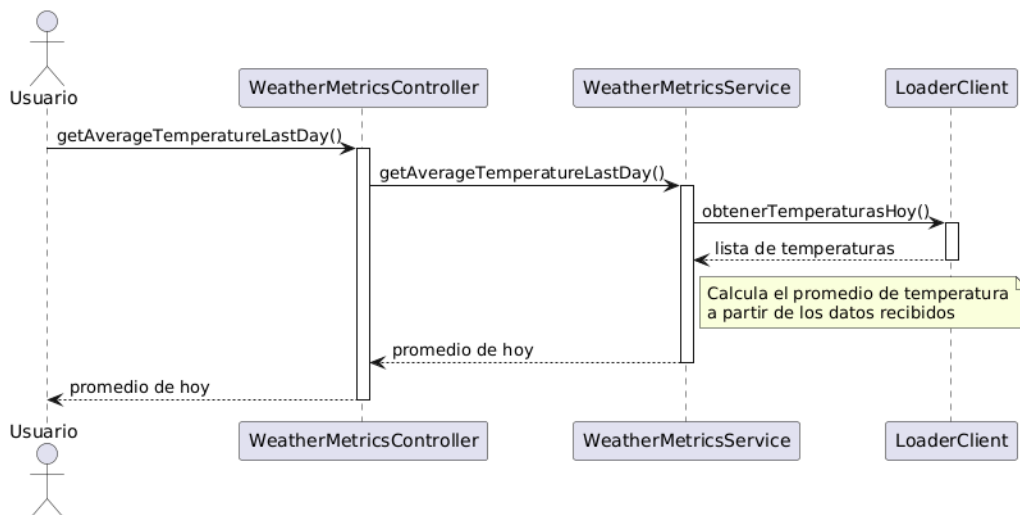
Obtener clima de hoy - Llama a la caché



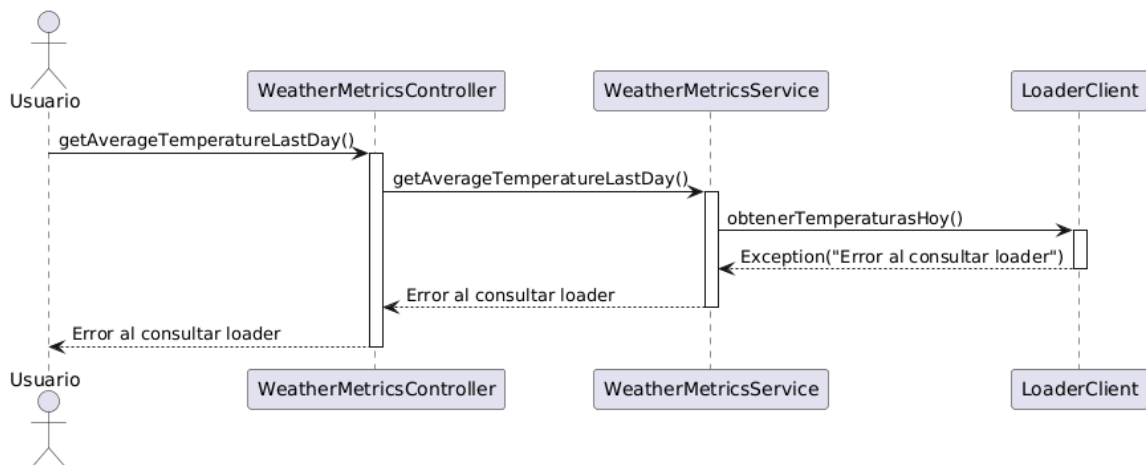
Obtener clima de hoy - Error al llamar al api externa



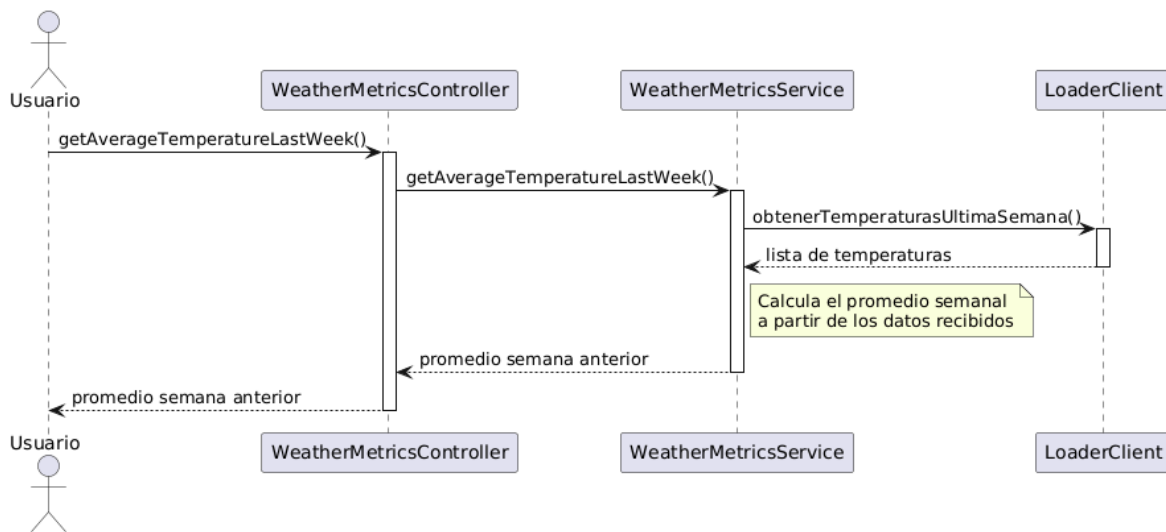
Obtener clima promedio de hoy



Obtener clima promedio de hoy - Error al llamar al api externa



Obtener clima promedio semanal



Obtener clima promedio semanal - Error al llamar al api externa

