



Máster en Cloud Apps

Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2021/2022

Trabajo de Fin de Máster

Your Race

Autores: Rafael Gómez Olmedo,
Raquel Toscano Marchena
Tutor: Micael Gallego



Universidad
Rey Juan Carlos

Contenido

[Resumen](#)

[Agradecimientos](#)

[Introducción](#)

[Problema a resolver](#)

[Objetivos](#)

[Modelo de Gestión](#)

[Alcance](#)

[Modelo del dominio](#)

[Requisitos](#)

[Requisitos Funcionales](#)

[Requisitos No Funcionales](#)

[Asunciones](#)

[Fuera de Alcance](#)

[Casos de Uso](#)

[Caso de Uso Race Registration](#)

[Arquitectura](#)

[Arquitectura por capas para Api Rest](#)

[Patrón DTO](#)

[Implementación](#)

[Metodología de implementación](#)

[Monolito](#)

[Diseño: Bootify](#)

[TDD: de los requisitos funcionales al desarrollo basado en pruebas.](#)

[TBD: Trunk Based Development](#)

[Versionado semántico SemVer](#)

[VCS GIT, GitHub](#)

[Integración Continua \(CI\)](#)

[Desarrollo de la aplicación](#)

[Java 17](#)
[Spring Boot 2.7.3](#)
[Maven](#)
[JPA/Hibernate](#)
[OpenApi](#)
[Lombok](#)
[Micrometer](#)
[Resilience4j](#)
[Togglz](#)

[Testing](#)

[Unitary Junit](#)
[Integration](#)
[Mockito](#)
[Faker](#)
[E2E](#)
[Testcontainer](#)
[Architecture ArchUnit](#)
[Jacoco](#)
[Sonarcloud](#)
[Zally](#)

[Despliegue](#)

[Recursos adicionales necesarios](#)
[PostgreSQL 14.5](#)
[RabbitMQ 3.9](#)
[Prometheus](#)
[Grafana](#)
[Postgres exporter](#)
[Docker](#)
[Kubernetes](#)
[Minikube](#)
[Helm](#)
[Istio](#)

[Observabilidad y Monitorización](#)

[Prometheus](#)
[Grafana](#)
[Lens](#)
[Minikube dashboard](#)

[Performance Testing](#)

Escenarios

Escenario 1. (Raquel)

Escenario 2. (Rafa)

Situación de partida versus situación final

Objetivos

Tipos de errores producidos

Testing Exploratorio.

Mejoras capa de negocio

Uso de BBDD no relacionales

Mejoras datos de entrada

Muerte de BBDD en peticiones concurrentes

Circuit breaker

Riesgo de escalado impostor

Determinación del punto óptimo

CronJob

El game changer: RabbitMQ

Arranque de clúster (tras creación o parada)

Resumen Flujo Desarrollo y CI/CD

Conclusiones

Trabajos futuros

Bibliografía

Repositorio GitHub

Casos de prueba de performance

Resumen

Your Race es una PoC (Prueba de Concepto) para la creación de una plataforma de registro de atletas en carreras deportivas de alta demanda.

La particularidad de este tipo de carreras es que tienen una capacidad limitada que resulta ser muy baja en relación a la cantidad de personas interesadas en ellas.

Cuando el sistema se abre para la recepción de solicitudes de registro, se debe garantizar la continuidad del servicio, además de la gestión de la avalancha de peticiones recibidas en un instante determinado y durante un corto espacio de tiempo.



Este estudio es extrapolable a muchos otros casos de uso donde el escalado de la aplicación juega un papel importante, como por ejemplo, la compra de tickets para un concierto con mucho público interesado.

En el desarrollo de nuestro trabajo hemos estudiado cómo resolver esta situación real, analizando el modelo del dominio y modelando y diseñando una solución mínima que nos ha permitido llevar a cabo pruebas de performance.

Con ellas hemos podido analizar el comportamiento del sistema, elaborar hipótesis y extraer conclusiones.

Para ello hemos aplicado muchas de las técnicas estudiadas en los diferentes módulos del Máster CloudApps, pudiendo aprovechar las sinergias existentes entre ellas, a la vez que afianzar y profundizar en los conocimientos adquiridos.

Agradecimientos

Muchas gracias a Mica por guiarnos en este trabajo. A Luis, Patxi, Michel, Óscar, Marta, Chema, Felipe y en general, a todos los profesores y personas que han colaborado con esta edición del máster, por transmitirnos su experiencia y conocimientos. Consideramos que nuestro perfil profesional es ahora más completo.

Introducción

Your Race está inspirada en carreras populares con alta demanda tales como:

- 101 Km de Ronda (<https://www.lalegion101.com/>)
- Maratón de Nueva York (<https://www.nyrr.org/tcsnymarathon>)

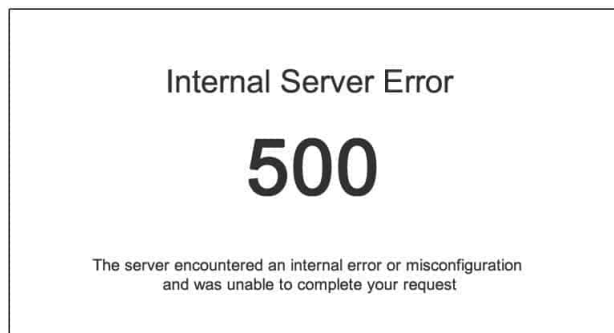
El problema a resolver se inspira en lo expuesto aquí (ver URL en bibliografía):

- [Cómo conseguir una plaza en los 101 kilómetros de Ronda.](#)
- [Así funciona la lotería](#)

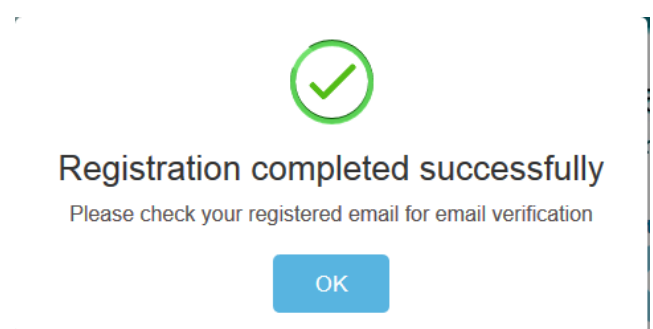
Se trata de carreras populares, con muy alta demanda y que ofrecen un número de plazas limitado de muchísima menor proporción.

Problema a resolver

En el caso de la 101 de Ronda, tal y como se expone en el artículo anterior, las personas con interés suelen recurrir a trucos tales como pedir colaboración a familiares y amigos para que soliciten el registro a la carrera además de ellos y aumentar así las posibilidades, y abrir varias pestañas en el navegador con antelación suficiente para “garantizar que <<si se cae>> podrás saltar a la siguiente” .



vs



Objetivos

El objetivo de Your Race es ofrecer una plataforma escalable para gestionar el registro de atletas en carreras populares de alta demanda.

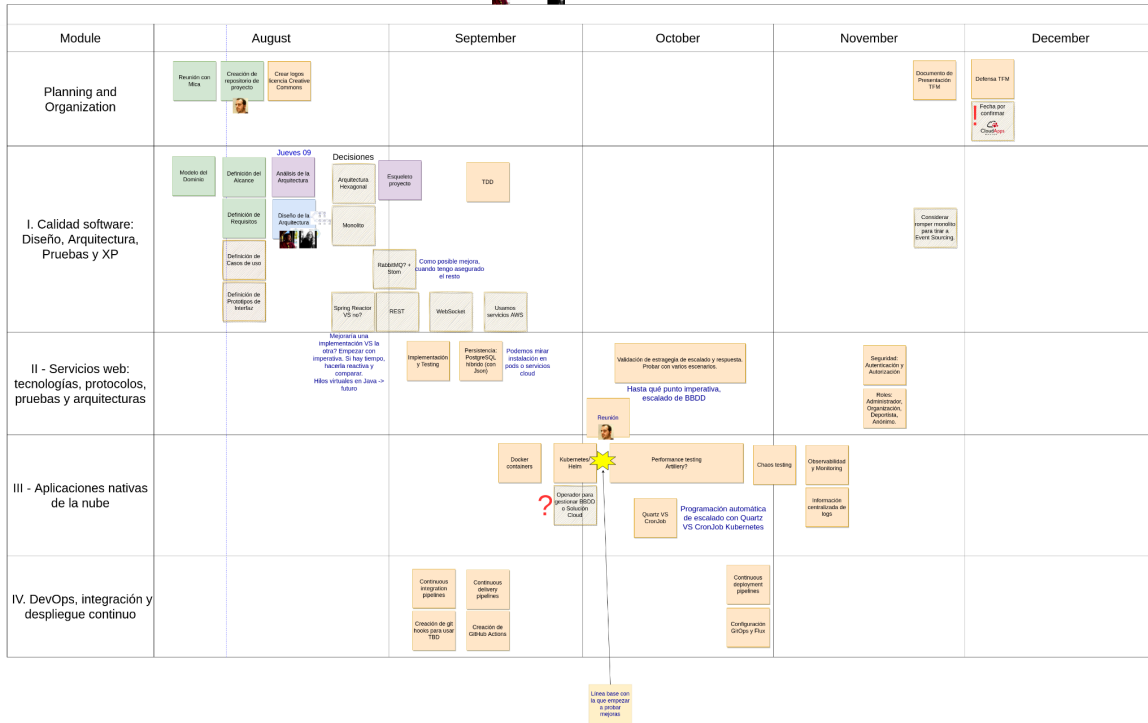
Si bien cada carrera tiene su propia normativa, suele haber un elemento común, y es que en un momento determinado en el que se abre el periodo de registro de la carrera o se publica la asignación de dorsales, se espera recibir una avalancha de peticiones.

El principal objetivo es asegurar la continuidad del servicio en todo momento, especialmente en aquellos casos en los que la avalancha de peticiones tiene lugar en un mismo instante.

Modelo de Gestión

En la gestión de este trabajo hemos seguido un modelo ágil, iterativo e incremental, con un alcance que se ha ido adaptando a lo largo del proceso y a la dimensión del equipo (2 personas). Un aspecto muy importante a tener en cuenta son las limitaciones de comunicación debido al trabajo asíncrono por horarios incompatibles con los de una jornada laboral.

En la planificación hemos tenido en cuenta el aseguramiento de la incorporación de líneas de trabajo correspondientes a las temáticas de cada uno de los módulos del máster:



Para la gestión de tickets de trabajo nos hemos basado en GitHub issues:

github.com/MasterCloudApps-Projects/your-race/issues?page=2&q=is%3Aissue+is%3Aclosed

News Bookmarks Contenidos Slack | Master... nca | Microsof... CloudAppsTFM CloudAppsDo... CloudAppsAWS CloudApps CloudAppsWeb CloudAppsMic... CloudAppsGit CloudApps-re...

Search or jump to... Pull requests Issues Codespaces Marketplace Explore

MasterCloudApps-Projects / your-race Public Edit Pins Unwatch 3 Fork 0 Starred 4

Code Issues 14 Pull requests 1 Discussions Actions Projects Wiki Security Insights Settings

Label issues and pull requests for new contributors
Now, GitHub will help potential first-time contributors discover issues labeled with good first issue

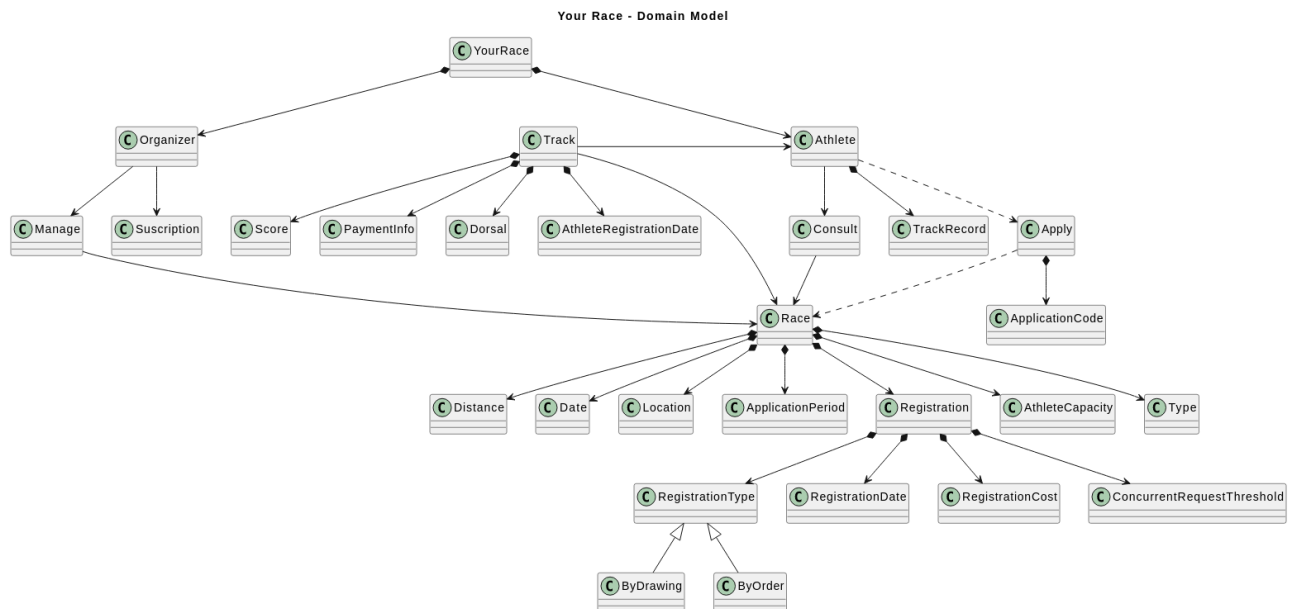
Filters Q is:issue is:closed Labels 9 Milestones 0 New issue

Clear current search query, filters, and sorts

	14 Open	37 Closed	Author	Label	Projects	Milestones	Assignee	Sort
☐	☒							
☐	☒							
☐	☒							
☐	☒							
☐	☒							

Alcance

Modelo del dominio



Requisitos

Requisitos Funcionales

Ofrecer un sistema para gestionar el registro de atletas en carreras de alta demanda. Existen 2 tipos de roles: atletas y organizadores.

Organizadores:

- Deben crear una suscripción a nuestra plataforma.
- Gestionan las carreras.

Atletas:

- Pueden preinscribirse a una carrera determinada.
- Como resultado de la preinscripción, obtienen un ApplicationCode.
- Cuentan con un palmarés de marcas personales (TrackRecord).
- Consultan la información de carreras disponibles (hayan participado o no).

Carreras:

- Cuentan con la siguiente información: Distancia, Fecha, Localización, Score, Periodo de preinscripción (ApplicationPeriod), Tipo (carrera, bicicleta)
- Cuentan también con un Registro.

Registro (Track):

- Es el mecanismo por el que un atleta preinscrito a una carrera obtiene dorsal (y entrada) a la misma.
- Tiene información de pago (tasas, estado, etc), dorsal (número).
- Se produce en una fecha determinada.
- Existen dos tipos de registro:
 - Por sorteo.
 - Por orden de llegada.

Requisitos No Funcionales

- Alto rendimiento y escalabilidad. Se debe asegurar en todo momento la continuidad del servicio. Aquí está nuestro negocio y no podemos fallar en esto.
- Debe soportar una carga de demanda de 25.000 peticiones concurrentes en un momento determinado, ya que en el peor de los casos se estima que todos los atletas con preinscripción van a acceder a la misma hora.
- Es posible prever en qué momento se va a producir esta avalancha de demanda. Y debemos implementar mecanismos programados de escalado automático de recursos en base a las fechas de carga críticas (fecha de comienzo de inscripción a carreras)
- Debe existir tener unas tablas de escalabilidad y facturación dimensionadas con los resultados de performance tests. Que de ahí salga tanto el nº de pods como la facturación del servicio (tarifas,...).
- Es posible que aunque tengamos previsto el pico de 25.000, finalmente sea algo mayor (30.000) ya que los atletas pueden haber solicitado ayuda a familiares y amigos.
- Cada vez que haya una nueva feature o de **forma automática diariamente por las noches**, se deberá comprobar que los tiempos no se ven perjudicados a través de los correspondientes pruebas de performance. En caso contrario hay que generar una alarma.
- Un factor a determinar será el decidir cuándo se va a producir el desescalado de la aplicación, debiendo ajustarse lo máximo posible.
- La aplicación debe ser segura e incorporar las principales directrices de seguridad.
- Autorización y Autenticación.

Asunciones

Consideramos que los resultados del análisis realizado son extrapolables en función de la capacidad del hardware utilizado.

Es decir, aunque los valores han sido obtenidos utilizando nuestros propios ordenadores, se pueden utilizar para elaborar afirmaciones basadas en ellos. Esperamos que en entornos Cloud o a mayor capacidad de infraestructura, los resultados deberían ser mejorados.

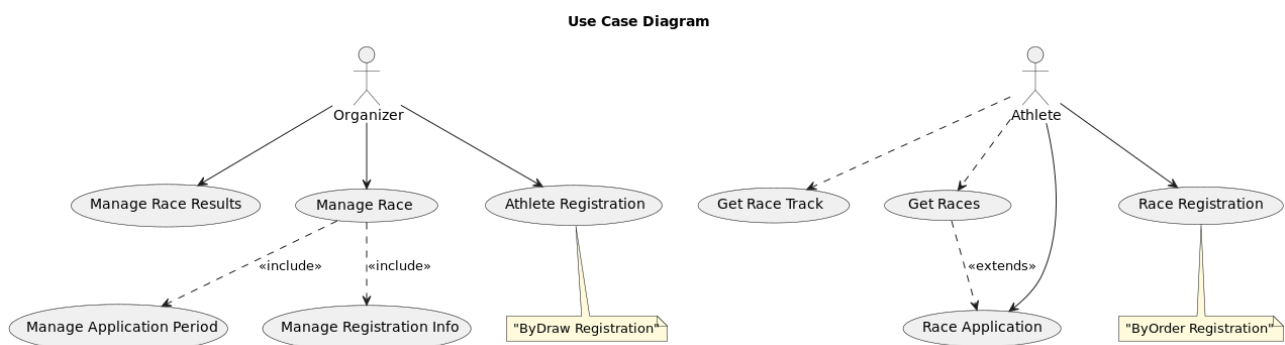
Fuera de Alcance

No está en el objetivo del proyecto realizar un servicio completo que cubra todos los requisitos identificados.

El objetivo es alcanzar un producto mínimo viable que nos permita realizar el estudio de alto rendimiento y escalabilidad para cumplir con el objetivo, a modo de prueba de concepto.

Por tanto, solo se proporcionan los servicios únicamente necesarios y a través de una API REST. No hay implementación de interfaces con el usuario final.

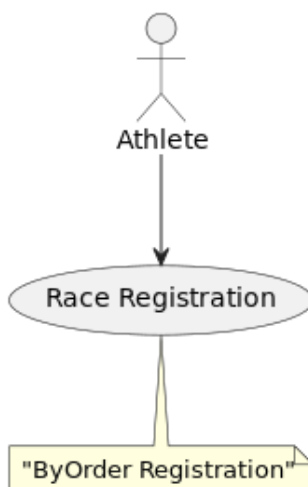
Casos de Uso



Caso de Uso Race Registration

El caso de uso bajo análisis objeto del estudio de performance es el de la inscripción a la carrera por orden de llegada: "ByOrder Registration".

Use Case Race Registration



Arquitectura

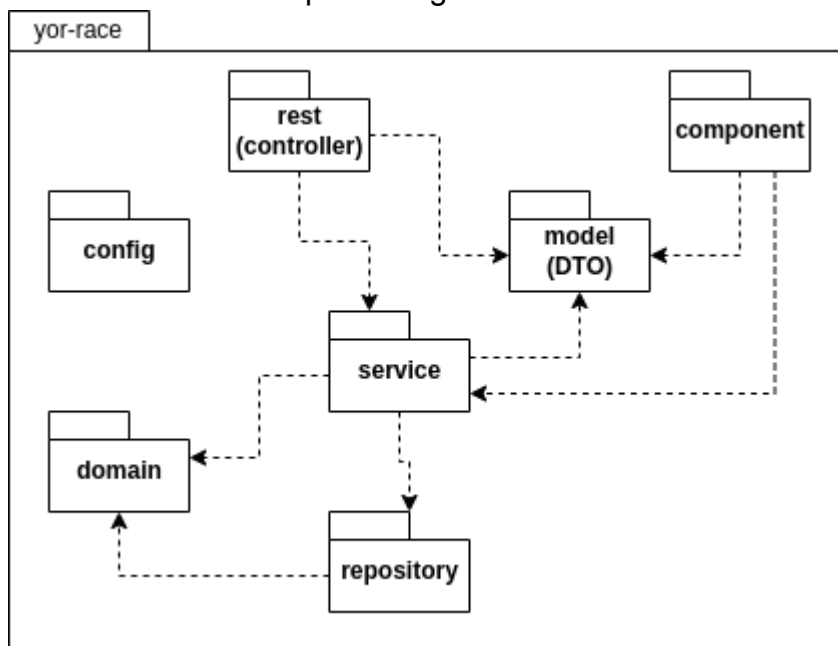
Arquitectura por capas para Api Rest

La aplicación se descompone en capas

- Cada capa da servicio a la capa de nivel superior
- Cada capa usa los servicios de la capa inmediatamente por debajo
- Una capa sólo puede usar los servicios de la capa inmediatamente superior o inferior

Se consideran 3 capas

- Presentación → controladores. Las peticiones se reciben en la capa de presentación
- Lógica de negocio → servicios y modelo. La lógica de negocio es invocada desde la capa de presentación
- Persistencia o infraestructura → repositorios. La capa de infraestructura es invocada desde la capa de negocio



Paquetes:

- config: para beans de configuración
- rest: donde se alojan los controladores rest.
- service: servicios que tratan la información de entrada y salida con los controladores
- domain: objetos del dominio
- repository: interfaces de interacción con DB.
- model: modelado de objetos de transferencia DTO.
- component: componente del consumer de broker de mensajería.

Las peticiones se reciben en la capa superior o inferior

- Se van pasando hacia arriba o abajo capa a capa
- Hasta llegar a la última capa o hasta que pueda atenderse en alguna capa intermedia

Patrón DTO

- Se desea poder acceder a los datos desde diferentes componentes situados en otras capas.
- Se quiere reducir el número de peticiones a la capa de persistencia.
- Se desea ocultar detalles de implementación de la persistencia de los datos.

Implementación

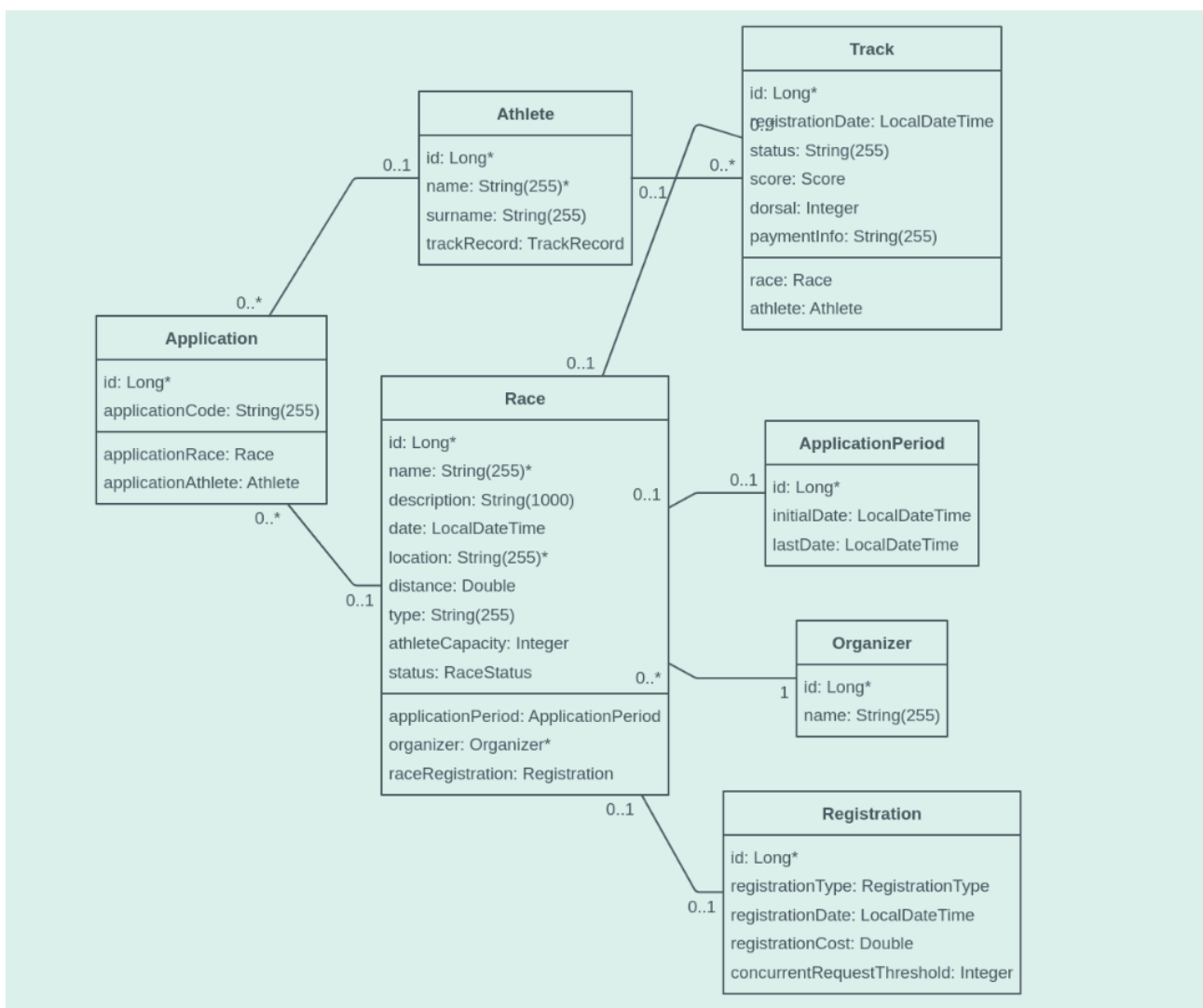
Metodología de implementación

Monolito

Se trata de una aplicación de arquitectura monolítica modularizada, que podría ser objeto de división en microservicios conforme evolucione y en función de las necesidades y uso de la misma.

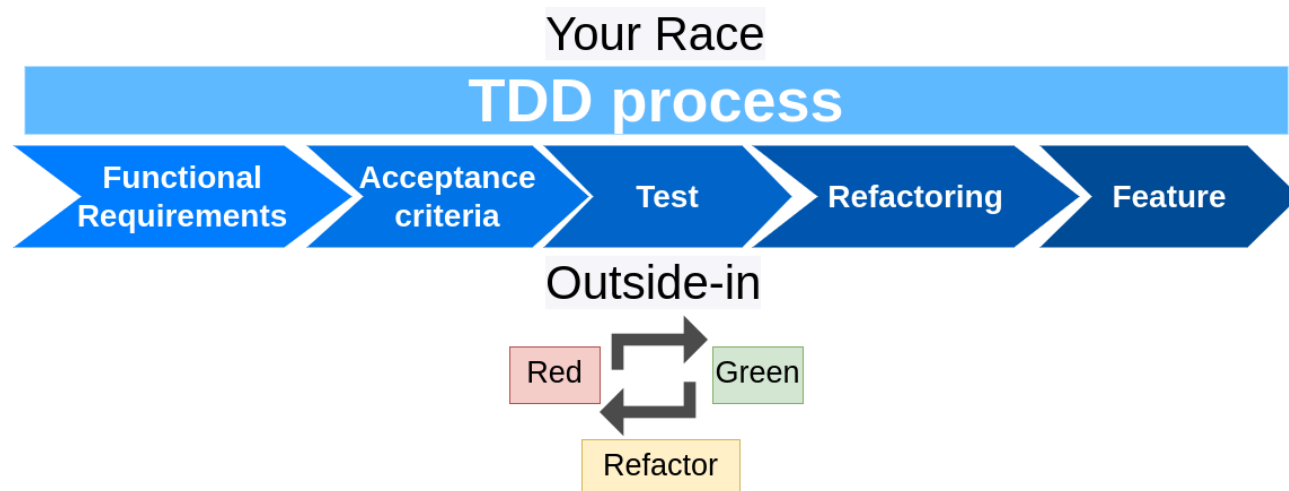
Diseño: Bootify

Para agilizar la implementación hemos utilizado Bootify (<https://bootify.io/>) para elaborar un Esquema de Entidad-Relación que es exportado de forma automática a un proyecto de Spring Boot estándar.



TDD: de los requisitos funcionales al desarrollo basado en pruebas.

Partiendo del diseño generado con Bootify, para implementar la lógica de negocio se ha utilizado TDD con enfoque Outside-in.



TBD: Trunk Based Development

Práctica de TBD en la medida de lo posible, durante las sesiones de Pair Programming. Uso de hooks de git para la automatización de los tests y validaciones de Sonar Cloud. Nuevas funcionalidades encapsuladas con feature toggles. Disponemos de una sola rama trunk lista para desplegar en todo momento. Modificaciones en funcionalidades aplicando Branch by abstraction.

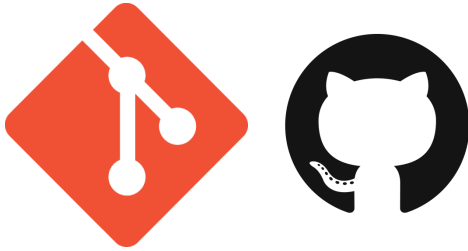
Versionado semántico SemVer

Tanto para la aplicación como la generación de artefactos Java y Docker.

```
<groupId>es.codeurjc.mastercloudapps</groupId>
  <artifactId>your-race</artifactId>
  <version>1.3.1</version>
  <packaging>jar</packaging>
  <name>your-race</name>
```

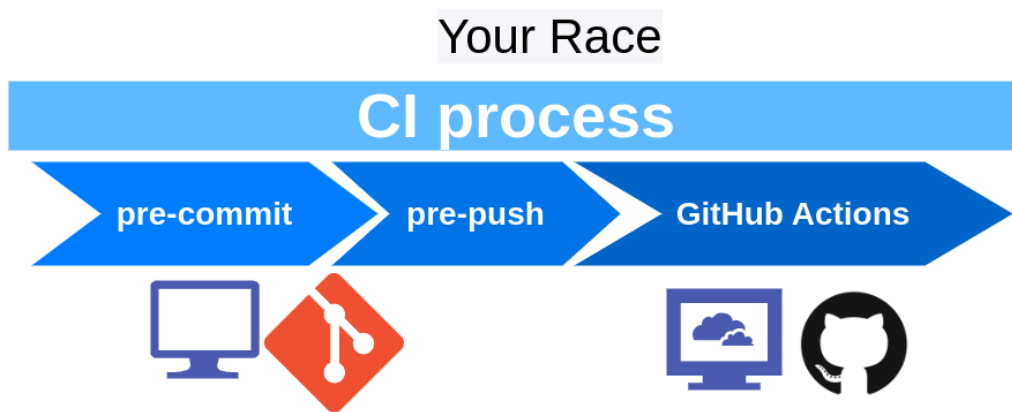
VCS GIT, GitHub

Sistema de control de versiones Git y GitHub para repositorio remoto. Utilizamos Git en local para gestionar nuestro repositorio, el cual tiene su copia principal en GitHub, desde donde gestionamos PRs y Workflows para el control de la versión y su ciclo de vida.

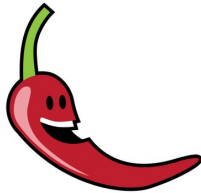


Integración Continua (CI)

Apoyada en hooks de git en las fases de pre-commit y pre-push, y en Github actions en las fases de integración a trunk y deploy.



Desarrollo de la aplicación

Java 17 	Spring Boot 2.7.3 	Maven 
JPA/Hibernate 	OpenApi 3 	Lombok 
Micrometer 	Resilience4j 	Togglz 

Testing

Unitary Junit

Como nuestra metodología de desarrollo se basa en TDD, es estrictamente necesario disponer de test ya que se crean y modifican en la propia evolución del desarrollo. Realizamos testing unitario de todas las funcionalidades de cada uno de las capas utilizando Junit.



Integration

Se implementan test de integración contra la base de datos principalmente para garantizar la interacción entre las capas.

Mockito

Para excluir factores externos al SUT y agilizar la ejecución de los test, utilizamos Mockito y Hamcrest para mockear valores y funcionalidades y así permitir realizar pruebas sobre el SUT a evaluar.



Faker

Empleamos Faker para tener testing dinámico con valores no predefinidos, esto en un proceso de evolución iterativa como TDD permite detectar errores tempranos basada en la aleatoriedad del dato de entrada.

E2E

A través de MockMvc o RestAssured implementamos test end to end para comprobar la funcionalidad de los diferentes endpoints, desde el exterior de la aplicación.

Testcontainer

Para garantizar la ejecución de los test en un entorno idéntico al de producción y sin la necesidad de disponer de una base de datos real implementamos testcontainer. Con esto conseguimos que nuestros test se puedan ejecutar con la única dependencia de Docker.



Architecture ArchUnit

Para garantizar el cumplimiento de la arquitectura por capas, la nomenclatura correcta de las clases o la detección de ciclos utilizamos ArchUnit. Definiendo diferentes test a nivel de proyecto que evalúa qué se está cumpliendo con la definición de arquitectura durante todo el desarrollo.



Jacoco

Agregamos la dependencia de jacoco para determinar el grado de cobertura de nuestro código ya que será necesario para evaluar el cumplimiento de las quality gates de sonar.



Sonarcloud

Disponemos de sonar como linter para detectar la calidad de código, así como detección de problemas de seguridad y comprobación de cumplimiento de Quality Gates establecidas, estas QG son restrictivas para permitir la persistencia del código en los repositorios comunes.



Zally

También agregamos el plugin maven del linter zally a nuestro proyecto para determinar el grado de cumplimiento en la estandarización de la definición de nuestra Api Rest, no utilizándolo por el momento con una QG restrictiva.



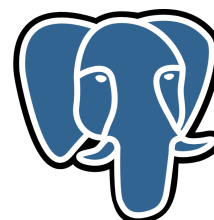
Despliegue

Recursos adicionales necesarios

Junto con nuestra aplicación, necesitamos los siguientes recursos adicionales para hacerla funcionar:

PostgreSQL 14.5

Usamos la bbdd relacional PostgreSQL dado que es OpenSource, su gran adopción por la comunidad y su estabilidad. Dispone de compatibilidad con la mayoría de lenguajes y soporte para los mismos incluso en Cloud. Además sus métricas son exportables a prometheus mediante Postgres exporter.



RabbitMQ 3.9

Es un broker de mensajería muy extendido y OpenSource. Es adecuado para nuestro entorno ya que necesitamos una solución ligera y fácil de implementar en un entorno cloud. Soporta varios protocolos de comunicación, además es escalable y está diseñado para soportar alta disponibilidad.



Y para poder realizar la monitorización también necesitaremos:

Prometheus 	Grafana  Grafana	Postgres exporter 
--	---	---

Docker

Es necesario disponer de **docker** para el entorno de desarrollo y despliegue.

Para el despliegue de la aplicación, esta se empaqueta en una imagen Docker, la cual se puede crear a través del Dockerfile creado o con el paquete JIB, que también se integra en el proyecto. Las imágenes se publican en DockerHub.



Para el entorno de desarrollo existe también una definición docker-compose, que permite probar todo el conjunto de aplicaciones necesarias para la aplicación.



Kubernetes

Será necesario disponer de **kubectl** de kubernetes en el entorno de desarrollo y despliegue.

Se han creado los manifiestos de kubernetes para el despliegue de todos los recursos necesarios, que incluyen tales como:

- Deployments
- Services
- PersistentVolumeClaims
- HorizontalPodAutoscaler



kubernetes



Minikube

Utilizamos el cluster de k8s **minikube** para realizar las pruebas en el entorno de desarrollo.

La configuración mínima que establecemos es de 4 cpus y 12G de memoria.

Para ello además habilitamos los siguientes Addons necesarios:

- metrics-server
- istio-provisioner
- istio



Helm

También será necesario disponer de **helm** para instalar en nuestro cluster, el chart de prometheus.



Istio

En nuestro cluster, es necesario disponer de **istio** como service mesh, para actuar como gateway y mejorar la tolerancia a fallos:

- Reintentar peticiones fallidas
- Proteger servicios Circuit breaker, políticas...)
- Controlar los timeouts



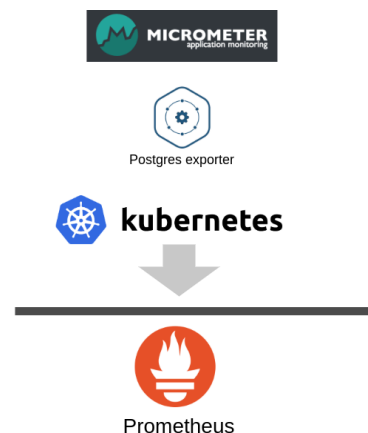
Observabilidad y Monitorización

Prometheus

Utilizamos este software especializado como sistema de monitorización y alertas. Todos los datos y métricas se almacenan en la base de datos como series temporales (junto al instante de tiempo en el que el valor se ha registrado).

Las métricas que monitorizamos tienen como fuente de datos:

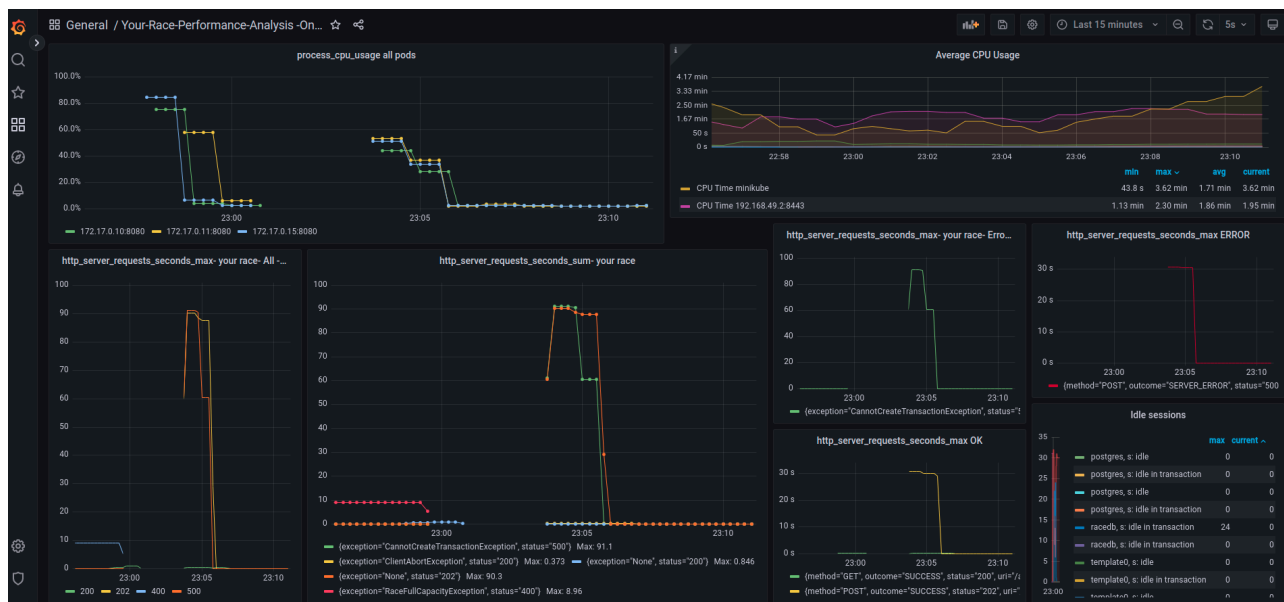
- Micrometer, para obtener métricas de nuestra aplicación Java.
- Kubernetes, del cual obtenemos datos de monitorización del cluster
- Postgres exporter, del cual obtenemos métricas del estado de la base de datos Postgres.



Grafana

Incorporamos grafana como interface de visualización de métricas con fuente de datos Prometheus, en el que importamos varios dashboards estándar de los servicios de los que tenemos métricas, y también creamos uno propio donde podemos monitorizar los parámetros de necesarios para evaluar las pruebas de performance.

Este dashboard personalizado puede exportarse a cualquier Grafana, ya que lo hemos importado en nuestro proyecto.

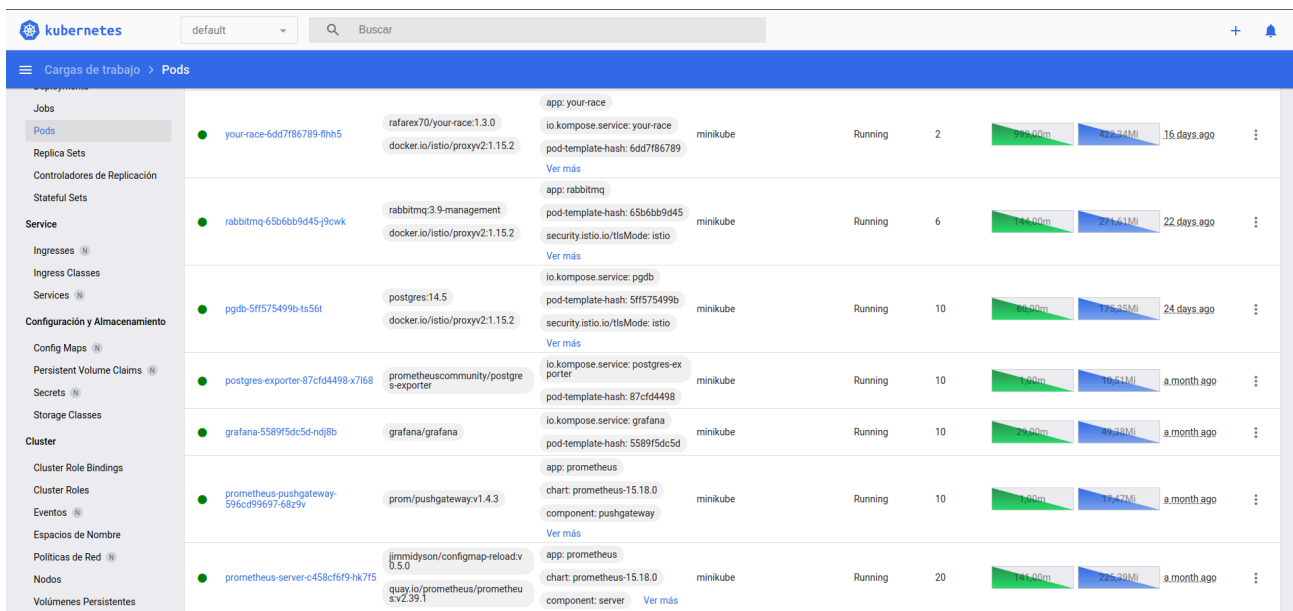


Lens

Utilizamos Lens para la monitorización del cluster de k8s, ya que nos permite con una interfaz de usuario más amigable realizar acciones sobre el cluster (escalado, reinicios, consulta de logs,...)

Minikube dashboard

También hemos empleado para monitorizar el estado de los pods el dashboard que incorpora minikube ya que combina algunas métricas gráficas con datos de los pods.



Con el siguiente gráfico de flujo, se resume todo el desarrollo descrito anteriormente.



Performance Testing

Escenarios

Las pruebas han sido realizadas basadas en diferentes escenarios:

Escenario 1. (Raquel)

Ordenador portátil Toshiba.

Memoria de 16 Gb.

Procesador Intel® Core™ i7-6500U CPU @ 2.50GHz x 4

Ubuntu 20.04.

minikube v1.26.0.

Escenario 2. (Rafa)

Ordenador portátil HP.

Memoria de 32 Gb.

Procesador Intel® Core™ i7-10750H CPU @ 2.60GHz × 12

Ubuntu 22.04.

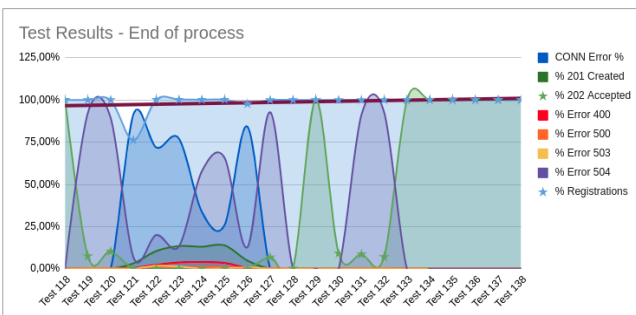
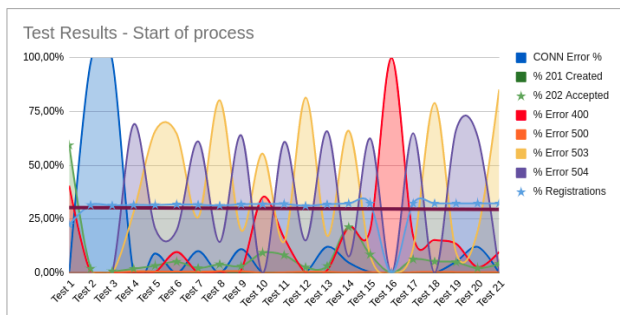
minikube v1.27.0.

Aunque en algunos casos los resultados de las pruebas han sido diferentes, hemos llegado a las mismas conclusiones sobre el rendimiento óptimo final.

Situación de partida versus situación final

Comenzamos realizando tests para recopilar datos e ir introduciendo cambios progresivos. Algunos cambios contribuyeron a una mejora del rendimiento y otros, sin embargo, se han descartado sirviendo de aprendizaje.

Como se aprecia en este gráfico, a lo largo del proceso de prueba se ha ido evolucionando hacia una imagen más estable, consiguiendo el procesamiento de todas las peticiones en el tiempo deseado.



Se puede consultar la ejecución detallada de los tests realizados en el anexo "[Casos de prueba de performance](#)".

Objetivos

Nuestro caso de uso a estudiar, es el proceso de registro en una carrera de alta demanda, donde una cantidad grande de personas compiten por conseguir un dorsal en un mismo instante en el que se abre el proceso de registro.

Por tanto, hemos considerado que el valor esperado para recibir toda la **avalancha de peticiones es de unos 30 segundos**.

El proceso llevado a cabo persigue tres grandes objetivos:

1. Averiguar el punto óptimo de procesamiento de peticiones dadas las restricciones del hardware utilizado.
2. Conseguir la mayor eficiencia posible: registrar el mayor número de peticiones en el menor tiempo posible.
3. Ser resilientes en momentos de alta demanda y evitar la rotura del servicio.

A partir de una versión estable del servicio, hemos comenzado por una fase de testing exploratorio y después hemos ido introduciendo cambios y registrando resultados hasta llegar a la situación final.

A continuación se describe cada una de las fases del testeo y los hechos observados.

Tipos de errores producidos

Errores de comunicación TCP/IP:

- ECONNREFUSED - Connection Refused.
- ETIMEDOUT - Connection Timeout.

Errores HTTP Rest:

- 200 - ClientAbortException Error.
- Error 400 - AthleteAlreadyRegisteredToRace.
- Error 400 - RaceFullCapacityException.
- Error 500 - CannotCreateTransactionException (Server).
- Error 500 - Server - DataIntegrityViolationException (Server).
- Error 503 - Service Unavailable (Client).

- Error 504 - Gateway Timeout (Client).

Testing Exploratorio.

El objetivo de esta fase ha sido recopilar datos y averiguar patrones de comportamiento.

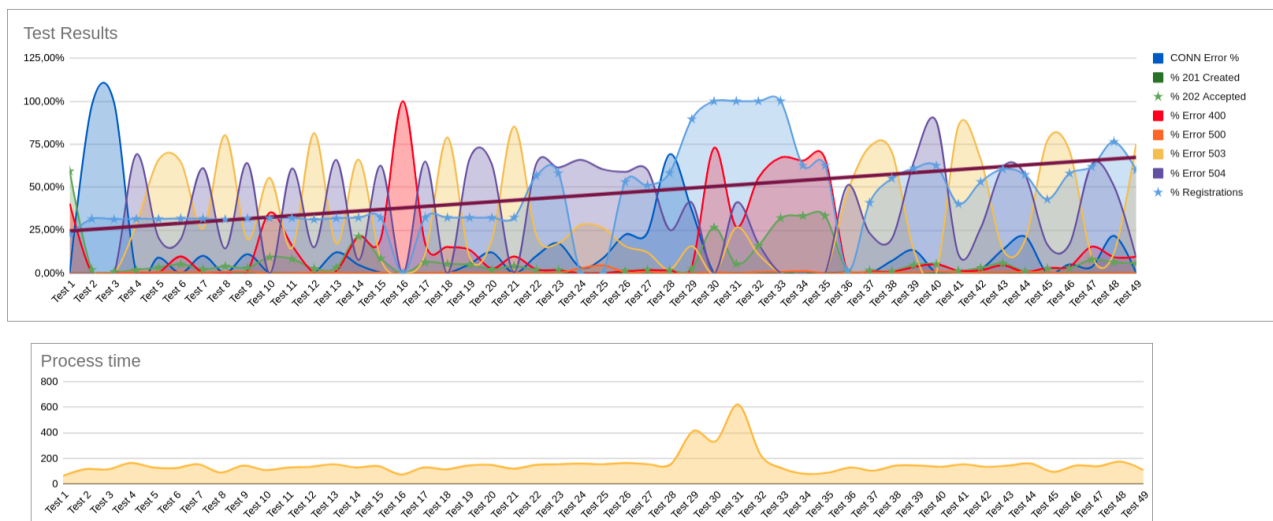
Uso de JMeter

El testing exploratorio lo comenzamos con JMeter con el objetivo de medir rendimiento a alto nivel para detectar los primeros puntos ineficientes y errores de aplicación. También nos aportó un dato muy importante: el límite de nuestra máquina para generar peticiones concurrentes, la cual en el escenario 2 llegaba aproximadamente a las 1000 rps y en el escenario 1 unas 400 rps.

Uso de Artillery.

Para la ejecución de artillery creamos un script bastante sencillo que genera peticiones POST sobre el endpoint a estudiar, al cual ahora le vinculamos un juego de datos (en CSV) con códigos de aplicación a carreras, incluyendo valores tanto válidos como erróneos, para realizar una prueba de rendimiento que emula un escenario real.

Estos son algunos de los resultados de forma gráfica:



Parámetros favorables

Los parámetros objetos de estudio que han tenido impacto en el resultado de forma positiva han sido:

Carga de las pruebas con Artillery:

- Timeout.
- Duration
- ArrivalRate

- Max Users.

Un elemento a destacar que debe ser estudiado es la cantidad de errores 400 por la excepción "AthleteAlreadyRegisteredException". Este error se produce cuando se intenta registrar un atleta que ya está registrado. Sin embargo el juego de datos proporcionado no tiene ningún elemento duplicado.

Tampoco se obtiene en esta etapa la excepción "RaceFullCapacityException", que se devuelve cuando la capacidad de registro de la carrera se ha completado.

Parámetros "neutros"

Los siguientes parámetros se han ido variando a lo largo de las pruebas sin ningún efecto diferenciador:

- Parámetro SPRING_DATASOURCE_HIKARI_MAXIMUM_POOL_SIZE.

Errores a resolver en siguientes fases

- Timeouts. El incremento del límite de timeout en el script de Artillery ayuda a reducir el número de errores por timeout. Sin embargo, se siguen produciendo durante los diferentes tests. Además los procesos siguen corriendo principalmente en bbdd aunque devolvamos timeout.
- Errores 400 "AthleteAlreadyRegisteredException".
- 500 errors:
 - 503
 - 504

Conclusiones

La base de datos otorga tiempos de respuesta muy altos en todas las peticiones, lo que provoca que no soporte muy bien la concurrencia. Es el primer punto de mejora que detectamos. El límite de concurrencia que soportamos no supera las 10 rps.

Solo se ha conseguido completar el registro de todos los atletas incrementando el timeout del registro, lo cual no es aceptable en nuestro caso de uso.

En los resultados se van obteniendo errores 503 y 504 de forma casi alterna y aleatoria, provocados por el timeout configurado.

La cantidad de errores 400 por "AthleteAlreadyRegisteredException" debe ser objeto de estudio. Parece que los parámetros de entrada que utiliza Artillery producen llamadas que se realizan más de una vez, pero no lo podemos confirmar.

En el caso del Horizontal Pod Autoscaler, obtenemos diferentes resultados según el escenario, por lo que el hardware utilizado tiene gran impacto en el resultado:

- Escenario 1: Los pods no escalan probablemente porque, en el caso de haber rebasado el límite de un 60% de CPU establecido en el Horizontal Pod Autoescaler, en un tiempo de 30 segundos no da tiempo para ello.
- Escenario 2: Si se produce un escalado eficiente a medida que aumenta la carga concurrente, pero el autoescalado al centrarse solo en your-race, según se va otorgando más cuota a la aplicación, más peticiones concurrentes tiene que ser manejadas por el único pod de bbdd, lo que como veremos más adelante, ocasiona un problema.

Pensamos que los errores de comunicación TCP/IP (ECONNREFUSED y ETIMEDOUT) se deben a las limitaciones propias de Artillery, ya que llega un momento en que no es capaz de superar el límite de rps que puede generar.

Mejoras capa de persistencia y negocio

El primer punto de mejora ocurre con las mediciones de tiempos de las queries que realizamos para determinar si el código de aplicación es válido, si la carrera a la que aplica tiene plazas libres y si no ha sido registrado previamente.

Detectamos que algunas consultas se realizaba un findAll y luego se realizaba el filtrado sobre el resultado de la query en la aplicación, cuando sería más óptimo filtrar en bbdd por su identificador único en los casos que disponemos de él. La situación inicial cuando no había datos en la bbdd no arrojaba un deterioro en el tiempo, pero según la bbdd estaba más poblada, más costosos se volvían los findAll de manera exponencial. Este cambio fue muy significativo, reduciendo el tiempo de las queries hasta el 99%.

Otro punto que detectamos de mejora se veía motivado por queries que no disponían de identificador único, si no de otro campo o combinación de campos para obtener un registro. Una muy significativa es la consulta por el código de aplicación, ya que se realiza siempre y no es identificador de tabla. Como en el caso anterior, cuantos más datos se agregan a las tablas de bbdd, más costoso es obtener resultados filtrados por campos que no son índice. Para estos casos lo que hacemos es determinar ciertos campos que debemos convertir en índices de las tablas. Este cambio también aporta un gran impacto en los tiempos de respuesta.

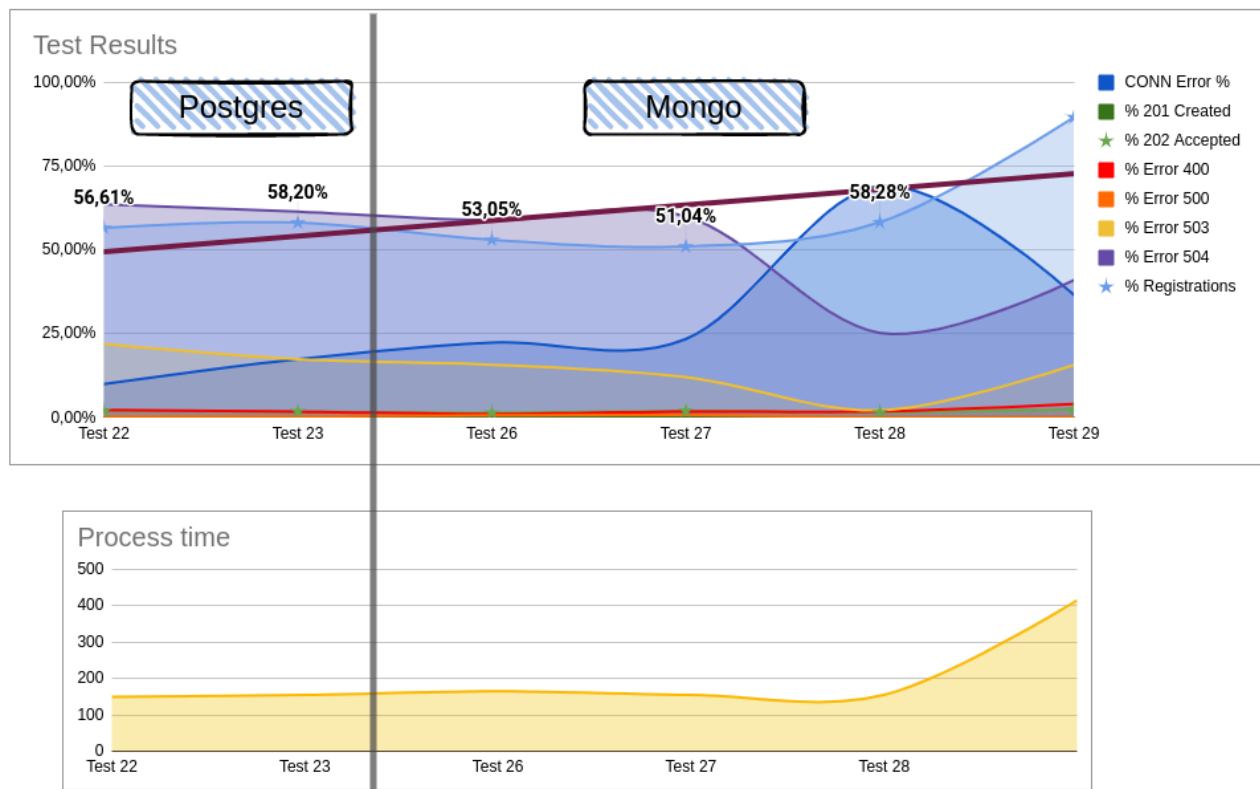
Aparte de mejorar los tiempos de respuesta de la base de datos, también trabajamos en la optimización de llamadas que realizamos. Para ello introducimos mejoras en la capa de negocio con el objetivo de evitar queries redundantes y apoyándonos en campos de cacheo de valores. Algunos objetos del dominio incorporan datos de "Status" (RaceStatus) que surgen como requisito para llevar a cabo esta optimización.

Evaluamos la posibilidad de cambiar el acceso a la base de datos con un modelo reactivo, pero encontramos que R2DBC tiene ciertas limitaciones que no podíamos salvar, como que no existan relaciones entre entidades porque son bloqueantes, a parte de no soportar caché. Por ello no realizamos su implementación.

Uso de BBDD no relacionales

Haciendo uso de Feature Flags y la técnica de Branch by abstraction, introdujimos una línea de investigación basada en pasar de una base de datos relacional como PostgreSQL a otra no relacional como MongoDB.

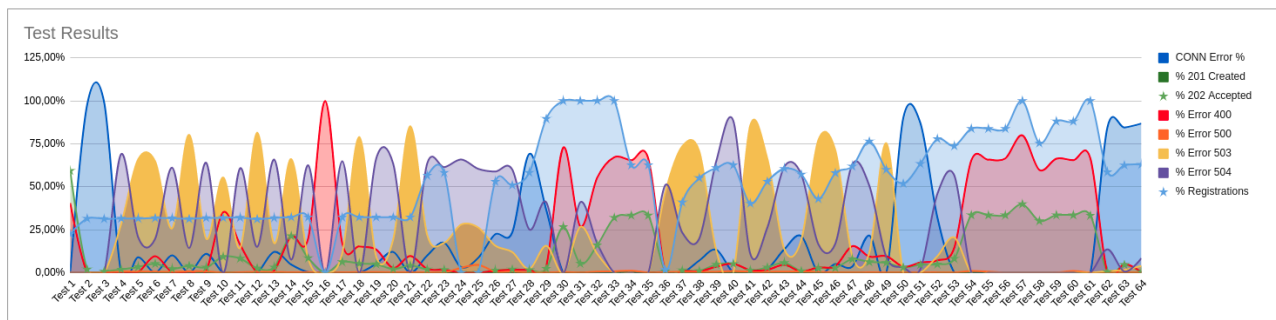
Sin embargo, esta opción quedó descartada, ya que el resultado de los tests eran muy parecidos y no proporcionaban ninguna mejora.



Mejoras datos de entrada

En el test 50 generamos una nueva batería de datos de prueba donde en el conjunto de datos posibles se eliminan caracteres especiales, utilizando solo letras.

En el escenario 1 de pruebas observamos que los errores 503, presentes en casi todos los tests anteriores, prácticamente desaparecen. Esta medida, sin embargo, no tiene ningún efecto en el escenario 2.



Muerte de BBDD en peticiones concurrentes

En el escenario 2 donde si se produce un escalado horizontal automatizado de forma efectiva pudimos comprobar como el cuello de botella, en cuanto al procesamiento de peticiones concurrentes, se trasladaba a la base de datos.

En un momento inicial con 1 solo pod y 500 rps la base de datos encolaba las peticiones e iba devolviendo respuestas incluso 8 min después del timeout, el pod de la aplicación se cargaba y requería de otro para procesar, aunque la carga estaba a la espera de bbdd.

Al tener el autoescalado, la segunda iteración, se levanta un segundo pod con un escenario es parecido al anterior, pero la bbdd y el cluster empiezan a notar más la falta de recursos para procesar la carga, en este caso después de 15 minutos habiendo fallado todas las peticiones por timeout, la base de datos termina de persistir las peticiones recibidas.

En el siguiente escenario duplicamos los pods, 4 pods para procesar 500 rps, aquí la base de datos recibe tantas peticiones concurrentes desde varios pods, que es incapaz de gestionar hasta el rechazo de nuevas conexiones y se bloquea y empieza a tomar recursos del cluster hasta que deja sin recursos al propio cluster y solo podemos reiniciar el cluster entero. La base de datos ha muerto y con ella arrastró el cluster.

Como comprobación de que cuantos más pods de la aplicación tengamos levantados, peor va a ser el resultado, elevamos el número de pods hasta los 8, y efectivamente, en la ejecución tardamos esta vez segundos en matar la bbdd y el cluster.

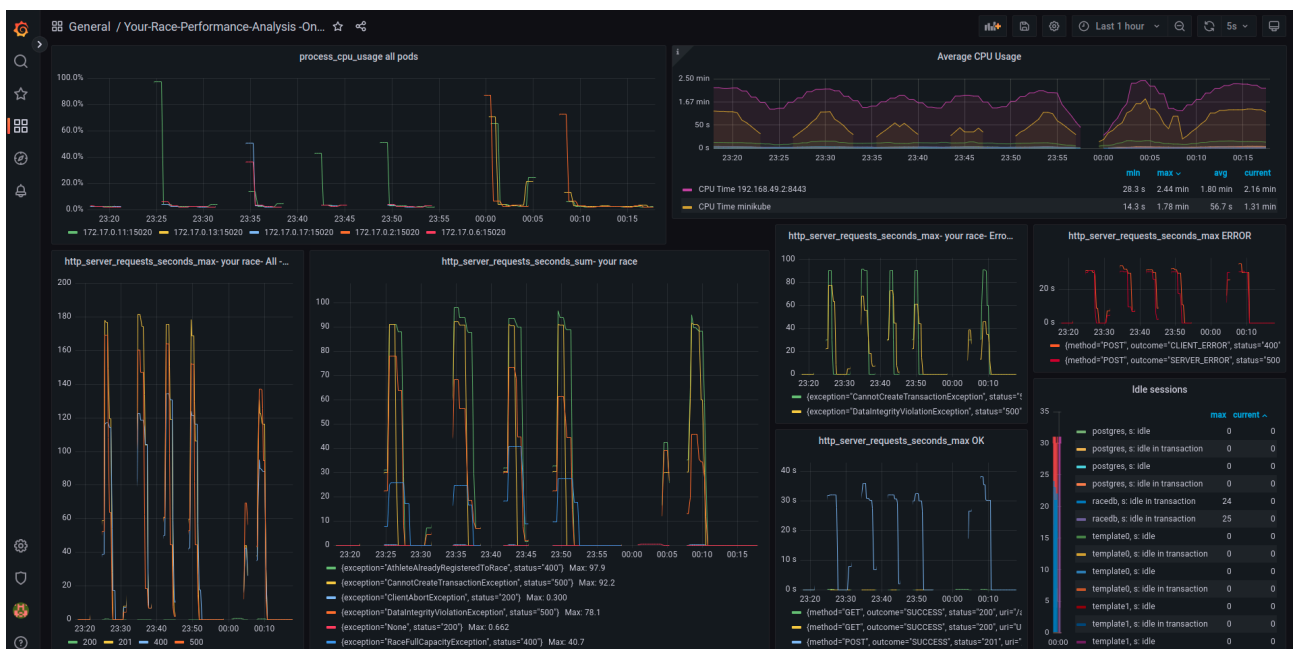
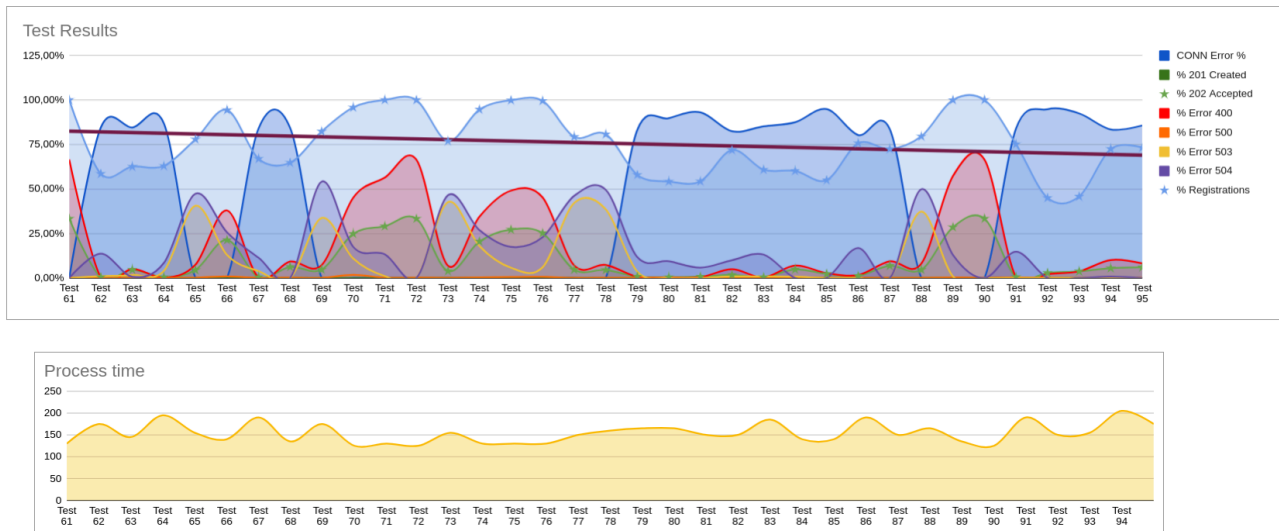
Durante el desarrollo de las pruebas observamos que el clúster queda afectado por lo que hemos denominado muerte de BBDD en peticiones concurrentes. Este estado solo se estabiliza con una parada y arranque de clúster.

A nivel cliente se generan errores TCP/IP (ECONNREFUSED, ETIMEDOUT) y 504 principalmente.

A nivel servidor:

- 400 - AthleteAlreadyRegisteredToRace
- 500 - CannotCreateTransactionException

- 200 - ClientAbortException
- 500 - DataIntegrityViolationException



Circuit breaker

En el escenario anterior, se determina que con el escalado de pods cada vez somos capaces de obtener más peticiones concurrentes y por lo tanto menos errores por petición, pero el error se traslada al nuevo cuello de botella que son las conexiones a la base de datos.

Para evitar una saturación en las peticiones a la base de datos y que esta termine por colapsar y morir, decidimos implementar mecanismos de resiliencia CircuitBreaker, con el objetivo de crear un escenario de circuito abierto o semiabierto cuando la base de datos empiece a rechazar nuevas conexiones (Errores).

Este mecanismo fue implementado bajo el mecanismo de feature toggles, para comprobar diferencias en tiempo de ejecución, logramos que nuestro CB proteja la base de datos y cluster y ya no se caiga, en cambio el método de fallback, hace que en este caso se aumenten las respuestas de error, la diferencia es que en esta ocasión es de forma controlada.

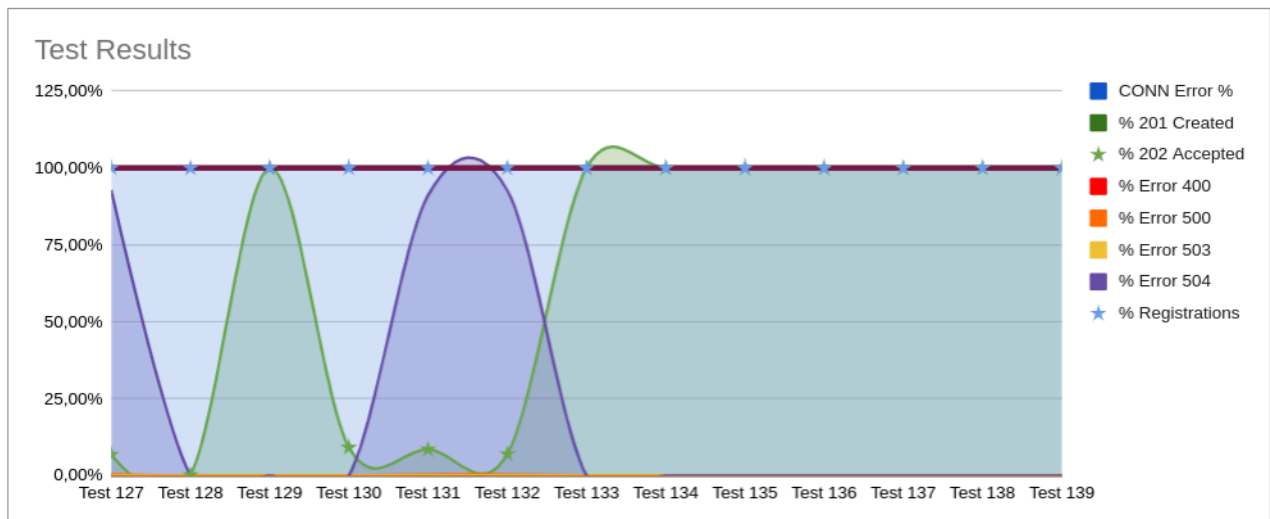
Riesgo de escalado impostor

El arranque de un pod durante la inicialización del servicio consume más del límite del 60% de CPU establecido durante unos minutos. Este hecho puede dar lugar a un escalado falso por no ser necesario, ya que la carga del pod se estabiliza una vez finalizado el arranque.

El “game changer”: RabbitMQ

Con la incorporación de RabbitMQ y su concepto de encolado de peticiones, conseguimos finalmente el objetivo deseado:

1. Procesar 30.000 solicitudes a razón de 1.000 peticiones por segundo, durante 30 segundos.
2. Sin errores.



Con RabbitMQ desaparecen por completo los errores 400 de `AthleteAlreadyRegisteredToRace`, aunque no podemos confirmar si estos errores se producían por errores en Artillery al tomar los datos de entrada o bien por errores en los envíos.

No obstante, se sigue produciendo un exceso de registros en asignación de dorsales de 1-2 normalmente, esto se solucionaría con un solo pod de consumer y creando un índice de dorsal con race_id.

De forma síncrona ahora solo se obtendrá respuesta de denegación de registro si la carrera ya está llena (RaceFullCapacityException).

Como contrapartida, se pasa a un modelo asíncrono, donde el atleta solo podrá conocer que su petición se ha recibido correctamente (estado 202 Accepted).

Necesitará acceder al endpoint de registros para conocer si está registrado y cuál se su número de dorsal.

Desde el punto de vista de experiencia de usuario y teniendo en cuenta que el tiempo total de los test realizados, debería conocerse el resultado del registro en menos de 5 minutos, mediante consulta del endpoint de registros.

Actualmente el productor y consumidor de la cola están en la misma aplicación por lo que está vinculado directamente su escalado. Determinamos que sería un punto conveniente de ruptura, en 2 microservicios diferentes para que puedan escalar en 2 niveles, según la necesidad real de producción o de consumo.

Determinación del punto óptimo

Tras una serie razonable de tests y teniendo en cuenta las restricciones del caso de uso, la dimensión óptima de peticiones y pods varía según cada escenario.

Escenario 1 (Raquel)

- 316 rps.
- 95 secs.
- 3 pods.

Dado que en 30 segundos no da tiempo de conseguir el escalado de pods, los pods deben ser arrancados previamente, mediante la actualización del manifiesto del deployment.

Escenario 2 (Rafa)

- 800 rps.
- 50 secs.
- 5 pods.

Estas limitaciones determinadas en este punto atienden a la capacidad de la máquina de generar peticiones concurrentes, como determinamos en el primer momento con Jmeter, nuestras máquinas tienen una limitación para generar peticiones concurrentes que ataquen al servicio. Como se ve en el siguiente gráfico, la ejecución de los test de performance con 1000 peticiones concurrentes toman toda la capacidad de

procesamiento de CPU. Provocando que no se pueda llegar a alcanzar la generación de todas las peticiones concurrentes.



CronJob

Adicionalmente se han realizado pruebas con CronJob y su uso se ha descartado por los siguientes motivos:

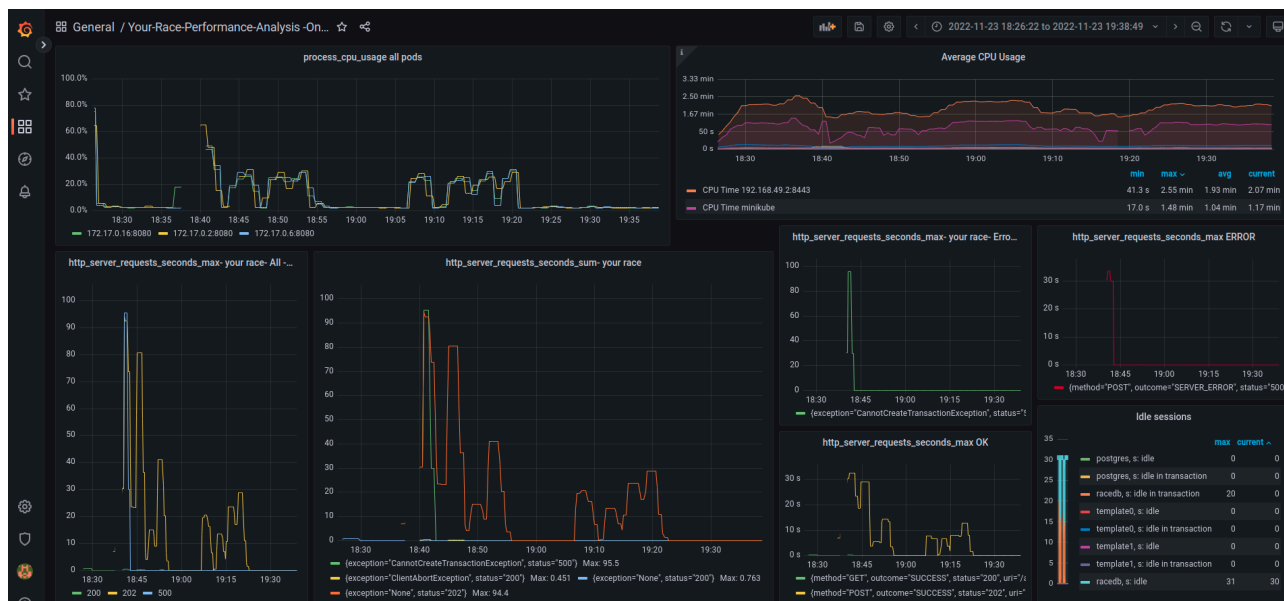
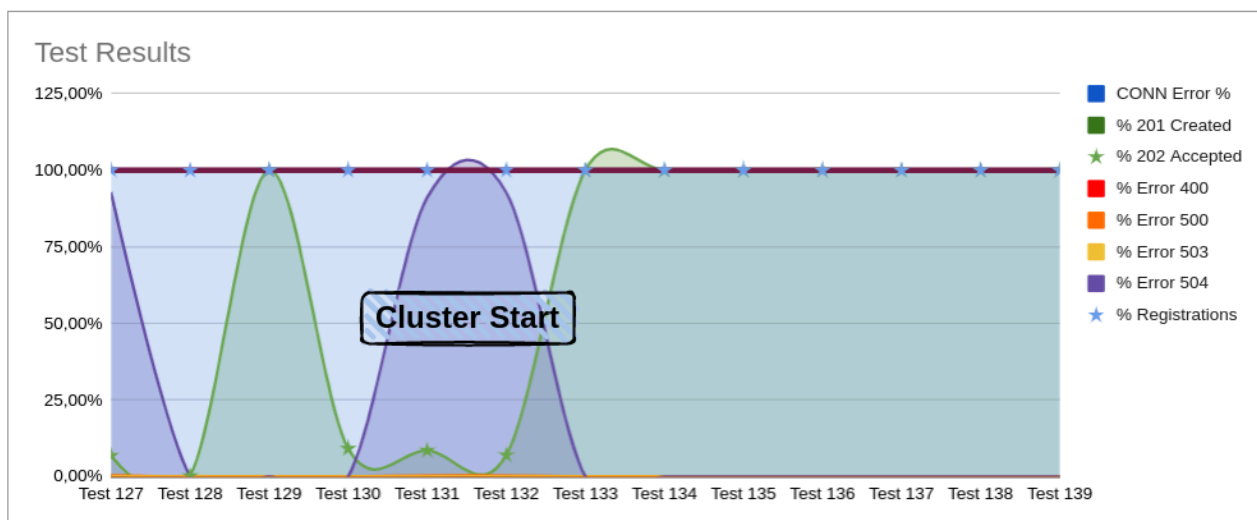
Los CronJob no sirven para escalar un pod, sino para ejecutar una tarea determinada de forma programada.

Incluso si el CronJob se utiliza para crear pods dentro del clúster, el rendimiento es mucho mejor cuando se modifica el manifiesto del deployment para establecer un mayor número de réplicas.

Sin embargo, sería posible investigar si para un escenario como el 2 donde los pods consiguen escalar en 30 segundos, podrían ser utilizados para activar un Horizontal Pod Autoscaler justo unos minutos antes de cada carrera.

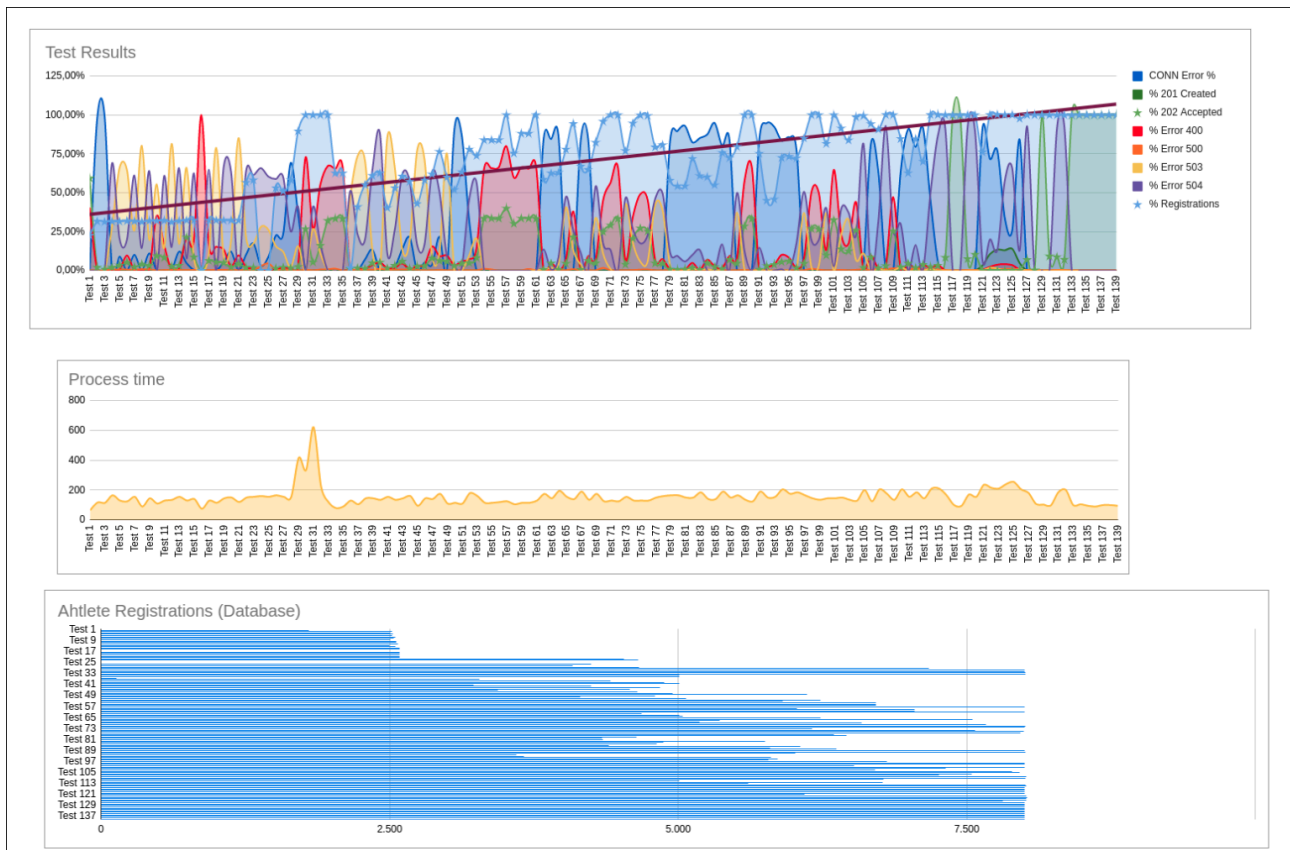
El efecto clúster “veterano”: arranque de clúster tras creación o parada

Se observa un aspecto muy importante: Los resultados de las pruebas no son óptimos en la 1ª prueba tras levantar un clúster (nuevo o parado), sino que van mejorando a partir de la 2ª prueba hasta estabilizarse.



Conclusiones

Evolución gráfica del resultado de todos los tests realizados:



El protocolo HTTP REST parece no ser la mejor forma de solucionar este caso de uso, ya que es muy difícil atender un pico tan elevado de demanda concentrada en un momento determinado.

Sin embargo, es posible introducir medidas que mitiguen esta desventaja obteniendo una experiencia de usuario aceptable.

Podemos concluir, por tanto:

1. **Es posible realizar el registro en una carrera con capacidad de 8.000 atletas recibiendo 30.000 peticiones a razón de 800 rps (en menos de 40 segundos).**
2. En nuestro caso de estudio el hardware ha sido determinante, ya que el último cuello de botella está en el número de peticiones concurrentes capaces de crear por artillery.
3. Desde el punto de vista del usuario, éste recibirá una confirmación de que su petición se ha recibido y se está procesando (respuesta 202) de forma inmediata, mientras exista capacidad en la carrera.
4. En caso de estar la carrera llena, recibirá como respuesta un error 400 de `RaceFullCapacityException`.

5. El proceso completo se cierra en unos 2-3 minutos, donde el usuario puede consultar su estado de registro (o no) en la carrera mediante consulta del endpoint de registros:

```
GET {{baseUrl}}/api/tracks?open=true  
Body: {"raceId": {raceId}, "athleteId": {athleteId}}
```

```
curl -X GET Http://localhost:8080/api/races?open=true -H  
"content-type: application/json" -d '{"raceId": 10003,  
"athleteId": 10010}'
```

Medidas que han tenido efecto:

- Uso de RabbitMQ para absorber el pico de registros y encolándolos para poder ser procesados secuencialmente.
- Optimización en queries SQL, añadiendo índices en consultas críticas.
- Mejoras en la lógica de negocio, principalmente minimizar al máximo el número de accesos a la BBDD.
- Incorporación de Circuit Breaker para proteger la caída de la BBDD.
- Escalado de pods con antelación al momento esperado del pico de demanda.
- Mejoras en el conjunto de datos de entrada.
- Ejecución de las pruebas en un clúster previamente utilizado.
- Uso de escalado horizontal (HorizontalPodAutoScaler). Aunque con el hardware utilizado en 30 segundos no da tiempo de completar el escalado, aporta estabilidad en un momento de carga altas. Si no escalara, podría morir y caería la aplicación.

Medidas que no han tenido efecto o han sido contraproducentes:

- Uso de BBDD noSQL (MongoDB) en lugar de BBDD relacional (Postgres).
- Cambio del número de pool de conexiones de BBDD en Spring Boot.
- “Calentado” de clúster con el lanzamiento de peticiones en los momentos previos al pico de demanda esperado.

Trabajos futuros

- Securización del servicio.
- Despliegue continuo en cloud con canary release.
- Pruebas de rendimiento en cloud con pods dedicados.
- Implementar WebSockets.
- Aplicar arquitectura Hexagonal.
- Extraer a un nuevo servicio el consumer para optimizar el escalado de recursos.
- Crear un chart de helm con toda la configuración necesaria.
- Loki para monitorización de logs.
- Healthchecks.
- Modelado de pricing de la organización de carreras.
- Crear script para escalado automático del deployment antes de cada carrera. Igualmente hay que crear otro para el desescalado cuando el pico del registro haya tenido lugar.

Bibliografía

<https://refactorizando.com/autoescalado-prometheus-spring-boot-kubernetes/>
https://jschmitz.dev/posts/testcontainers_how_to_use_them_in_your_spring_boot_integration_tests/

<https://www.rfc-editor.org/rfc/rfc7231#section-6.3.3>

<https://andaluciarunning.com/como-conseguir-una-plaza-en-los-101-kilometros-de-ronda/#~:text=%2D%20Abre%20varias%20pesta%C3%B1as%20en%20el,el%20servidor%20del%20sitio%20web>

<https://maratondeny.com/asi-funciona-la-loteria/>

<https://www.lalegion101.com/>

<https://www.nyrr.org/tcsnymarathon>

<https://www.npr.org/2021/11/07/1053354892/new-york-city-marathon-winner-albert-korir-peres-jepchirchir>

https://jschmitz.dev/posts/testcontainers_how_to_use_them_in_your_spring_boot_integration_tests/

<https://www.baeldung.com/java-faker>

<https://www.mongodb.com/docs/manual/tutorial/getting-started/>

<https://use-the-index-luke.com/sql/example-schema/postgresql/where-clause>

<https://www.artillery.io/docs/guides/guides/test-script-reference>

<https://github.com/features/actions>

Anexos

Repositorio GitHub

<https://github.com/MasterCloudApps-Projects/your-race>

Casos de prueba de performance

YourRaceTestingScenarios:

https://docs.google.com/spreadsheets/d/1K2KCRoR6Kmkq3UN-WFXWZY6JZzejWN9V1HZ-6EVJA_Q/edit?usp=sharing