UNIVERSIDAD
NACIONAL
DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Programa de Apoyo al Egreso para Alumnos con Práctica Profesional Supervisada

TÍTULO: Comparador de Performance en Sitios WEB

AUTOR/A: Andrés Rodríguez

DIRECTOR/A ACADÉMICO: Armando De Giusti

DIRECTOR/A PROFESIONAL: Fernando Gomez

CODIRECTOR/A ACADÉMICO:

CARRERA: Licenciatura en Sistemas

RESUMEN

La acelerada evolución de la web ha generado una creciente demanda por sitios web que no solo sean funcionales, sino también rápidos, accesibles y optimizados para los motores de búsqueda
El objetivo de este proyecto es facilitar el análisis continuo del rendimiento web, proporcionando una herramienta flexible y extensible que nos permita hacer un seguimiento periódico de estas métricas y ver su evolución a lo largo del tiempo.

Para esto se desarrolló una aplicación la cual debe obtener los valores sobre performance web de los sitios que se quiera medir, basando su desempeño utilizando los parámetros que evalúa la suite de Google Lighthouse

Palabras Claves

- Performance Web
- Métricas
- Google Lighthouse
- Presets
- Receivers
- Pages
- Comparador
- Tableros BI
- FCP
- LCP
- Score

Conclusiones

Encontramos con una manera de medir y monitorear a lo largo del tiempo como se comportan los flujos más importantes de nuestro sitio.

Identificamos fluctuaciones, encontramos las causas y las corregimos. Nos dimos cuenta hacer una comparación con los sitios de la competencia, no era tan relevante como creímos, sirviendo como comparador de nosotros mismos en el pasado

Trabajos Realizados

Se creo un servidor web capaz de ejecutar continuamente pruebas de Lighthouse sobre las páginas objetivo, reportar los resultados a un lugar centralizado y se generaron tableros para el seguimiento y mejora de las métricas

Trabajos Futuros

1. Actualización de Angular a su versión 16
2. Utilización de imágenes por defecto para los primeros resultados
3. Cambiar la arquitectura en la solicitud de resultados de hoteles, mejorando la experiencia de usuario

Fecha de la presentación: AGOSTO 2024

Índice General

Introducción **3**
- Motivación 3
- Objetivo 3

Proyecto realizado **4**
- Servicios expuestos por el servidor 4
- Especificación de las pruebas 7
- Componentes de la especificación 8
  - Presets 8
  - Receivers 9
  - Pages 11
    - Construcción del “Target” 12
      - Construcción de los parámetros 14
  - Tags 17
- Las métricas que se trackean 17
  - INP (Interaction to Next Paint) 18
    - Valores de Referencia 18
  - TTFB (Time To First Byte) 18
    - Valores de Referencia 19
  - FCP (First Contentful Paint) 19
    - Valores de referencia 19
  - LCP (Largest Contentful Paint) 19
    - Valores de referencia 20
  - TBT (Total Blocking Time) 20
    - Valores de referencia 21
  - CLS (Cumulative Layout Shift) 21
    - Valores de referencia 22
  - Speed Index 22
    - Valores de referencia 22
  - Score 22
  - Dom Size 23
    - Valores de referencia 23
- Cómo es una ejecución 23
- Tableros 28
  - Evolución en el tiempo 28
  - Acumulado en el tiempo 29
- Infraestructura y deploy 30
  - Dockerfile 31
  - Recursos del contenedor 32
  - Pipeline de Integración Continua (CI) / Deployment Continuo (CD) 33

Resultados obtenidos **34**
  - ¿Cómo medimos? 34

Líneas de trabajo futuros **37**
  - Llamado condicional a la Full 37

Conclusiones..39
Glosario alfabético..40
Referencias bibliográficas..42
Web..42
Libros...42

Introducción

Motivación

La acelerada evolución de la web ha generado una creciente demanda por sitios web que no solo sean funcionales, sino también rápidos, accesibles y optimizados para los motores de búsqueda. El rendimiento web es un factor crucial que influye directamente en la experiencia del usuario y en el éxito de una página. En este contexto, surge la necesidad de herramientas que permitan evaluar y mejorar continuamente estos aspectos.

Objetivo

El objetivo principal de este proyecto es facilitar el análisis continuo del rendimiento web, proporcionando una herramienta flexible y extensible que nos permita hacer un seguimiento periódico de estas métricas y ver su evolución a lo largo del tiempo.

Para esto se desarrolló una aplicación llamada **“Performance Comparator”** la cual debe obtener los valores sobre performance web de los sitios que se quiera medir, basando su desempeño utilizando los parámetros que evalúa la suite de Google Lighthouse

Definimos que estas métricas deben tener las siguientes características:

- **Accesibles**: Deben poder ser consultadas por cualquier equipo en un lugar centralizado.
- **Representativas**: Deben tener el volumen y la diversidad de contextos suficientes para representar el universo de casuísticas, para generar confiabilidad
- **Periódicas**: Deben basarse en muestras obtenidas de manera periódica.
- **Trackeables**: Se debe poder ver su evolución en el tiempo.

Este desarrollo señala un norte para que todos los equipos de desarrollo las puedan consultar, en pos de evaluar mejoras y velar por la estabilidad/correctitud de las mismas.

También implica disponibilizar estos valores para ser presentados mensualmente ante inversores, dando visibilidad tanto de mejoras sobre el sitio como también poder informar sobre la degradación en el mismo y encontrar qué lo está causando.

Luego en un segundo plano es poder obtener estas mismas métricas para los sitios de la **“competencia”** y de esta forma tener un indicador, que en consecuencia nos permita definir objetivos para trabajar.

Proyecto realizado

La aplicación desarrollada, consta de un servidor implementado en **TypeScript** utilizando **Node.js** y **Express.js** la cual utiliza como librerías principales **Puppeteer** para correr instancias de **Chromium** automatizadas y **Google Lighthouse**. Para una página dada, ejecute las pruebas de performance y arroje los resultados, el cual es el foco principal de este proyecto.

El servidor tiene la finalidad actuar como un **"supervisor"** de la ejecución de las pruebas automatizadas. Es el encargado de, a partir de una configuración, establecer un cron para mantener de manera continua la ejecución de las pruebas.
Además es el encargado de gestionar el estado actual de la ejecución y de su resultado previo. Dentro de esta gestión se encuentra poder informar si se está ejecutando una prueba en un momento dado, si la última ejecución fue exitosa o no y lo más importante es implementar un mecanismo de bloqueo para evitar más de una corrida simultánea de las pruebas, dado que es una tarea demandante a nivel de uso de CPU y red, donde la multiplicidad de instancias llevaría a resultados transgiversados.

Servicios expuestos por el servidor

- **/health-check:** Servicio GET, retorna una respuesta HTTP 200, indicando que la aplicación está funcionando, esto es utilizado por la infraestructura para el monitoreo de aplicaciones
- **/run-status:** Servicio GET, retorna una respuesta HTTP 200 y en su body un JSON con el estado actual de la ejecución, lo que incluye si está corriendo o no, su periodicidad y el estado de la última ejecución.

```json
{
    "status": 200,
    "running": true,
    "last_run_status": "SUCCESS",
    "scheduled": "10m"
}
```

*(**Respuesta)***

- **/stop-schedule:** Servicio PUT, retorna una respuesta HTTP 200 con un mensaje del resultado de la operación, este servicio se encarga de cancelar el cron, de forma que al terminar (si la hay) una ejecución en curso no siga ejecutando.

```json
{
    "status": 200,
    "message": "Schedule stopped"
}
```

(***Respuesta)***

- **/schedule-run:** Servicio PUT, retorna una respuesta HTTP 200 o 400, con un mensaje del resultado de la operación, este servicio recibe un body con el tiempo y la unidad para crear el cron.

```json
// unit es un enumerativo con los valores ["ms", "s", "m", "h"]
{
    "timer": 10,
    "unit": "m"
}
```

(***Cuerpo de la petición)***

```json
// Caso exitoso
{
    "status": 200,
    "message": "Run scheduled at every 10m"
},
// Casos de error
{
    "status": 400,
    "message": "The body is mandatory"
},
{
    "status": 400,
    "message": "Invalid timer, needs to be greather than 0"
},
{
    "status": 400,
    "message": "Invalid unit, it has to be one of [ ms, s, m, h ]"
}
```

(***Respuesta)***

- **/run-evaluation:** Servicio PUT, Retorna una respuesta HTTP 200 o 400 con el resultado de la operación, su función es poder ejecutar una prueba utilizando la especificación base, haciendo una sobreescritura de las propiedades indicadas en el body, el mismo no es mandatorio, en caso de la ausencia de alguna propiedad se utiliza el default.

```
// Body del servicio
{
    "players": ["DESPEGAR"], // Puede utilizarse ALL para indicar todos los players
    "products": ["HOTELS"],  // Puede utilizarse ALL para indicar todos los products
    "steps": ["RESULTS"],    // Puede utilizarse ALL para indicar todos los steps
    "presets": ["mobile"]    // Puede utilizarse ALL para indicar todos los presets
},
```

(Cuerpo de la petición)

```
// Caso exitoso
{
    "status": 200,
    "message": "Run started successfully, check on /run-status to see the results",
    "overrided_config": {
        "players": ["DESPEGAR"],
        "products": ["HOTELS"],
        "steps": ["RESULTS"],
        "presets": ["mobile"]
    }
},
```

```
// Casos de error
{
    "status": 400,
    "message": "Run in progress, cannot run right now"
},
{
    "status": 400,
    "message": "Invalid players, got: ['AN_INVALID_VALUE'] expected: ['DESPEGAR', 'EXPEDIA', 'ALMUNDO', 'BOOKING','GOL', 'LATAM']"
},
{
    "status": 400,
    "message": "Invalid products, got: ['AN_INVALID_VALUE'] expected: ['home', 'hotels','flights','landing']"
},
{
    "status": 400,
    "message": "Invalid steps, got: ['AN_INVALID_VALUE'] expected: ['RESULTS', 'DETAIL']"
},
{
    "status": 400,
    "message": "Invalid presets, got: ['AN_INVALID_VALUE'] expected: ['desktop-default', 'mobile-default', 'mobile-fast']"
}
```

(Respuestas posibles)

- **/unlock:** Servicio PUT, Retorna una respuesta HTTP 200, con el resultado de la operación, su función es en caso de que por algún error desconocido, no se esté ejecutando ninguna prueba pero haya quedado el bloqueo activo.

```
// Caso exitoso
{
    "status": 200,
    "message": "Lock cleared successfully"
}
```

(Respuesta)

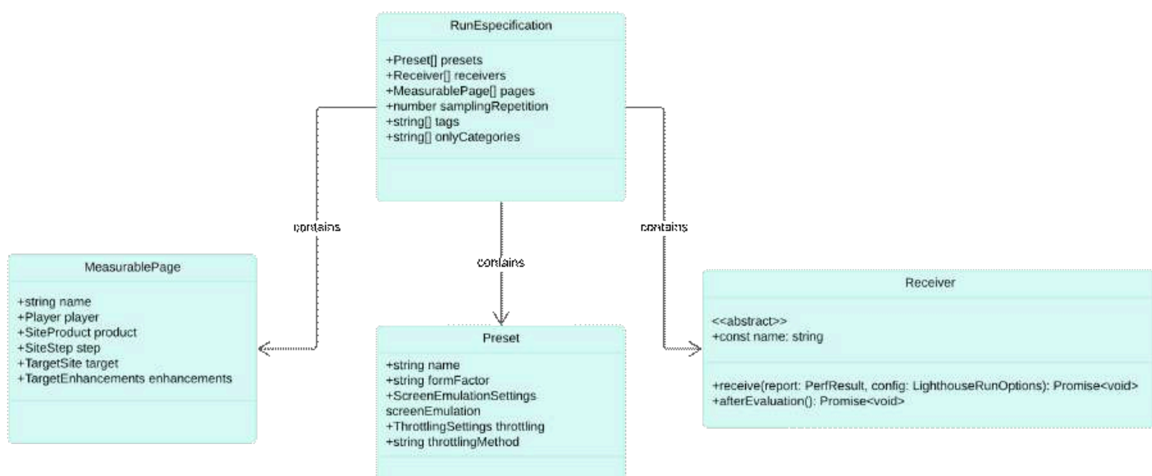
Especificación de las pruebas

Se creó una especificación para indicarle a la herramienta cómo es la estructura de las pruebas a ejecutar y la misma incluye.

- **Dispositivos**: Se le especifican las resoluciones a utilizar, perfil de dispositivo, calidad de red.
- **Frecuencia:** Cada cuánto tiempo debe ejecutar la suite
- **Envío de datos**: A donde debe enviar los resultados obtenidos
- **Páginas:** En qué sitios y con qué configuraciones debe hacerlo.

Esta especificación se construye al momento de iniciar la aplicación, permitiendo luego en tiempo de ejecución si es necesario por una **API REST** ajustar algunos de sus parámetros.

Este es el diagrama de clases, que luego, se va a ir desarrollando cada componente individualmente a lo largo de esta tesis.



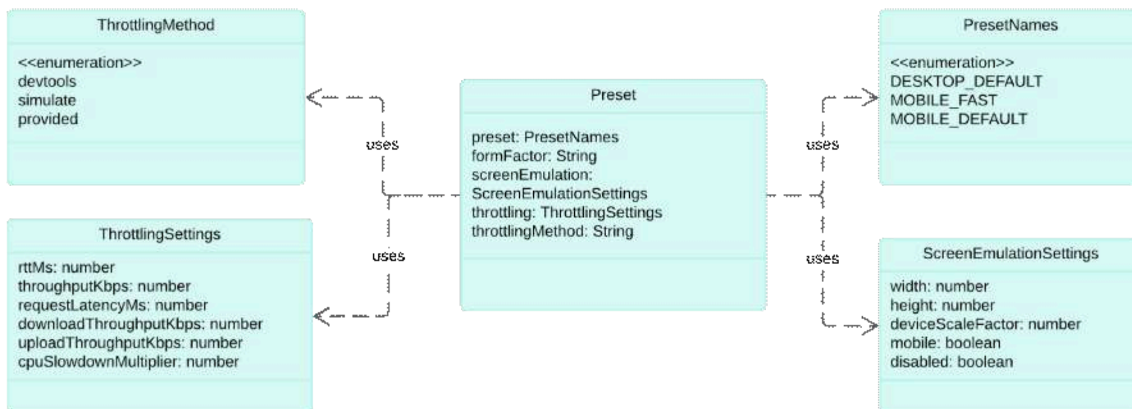
Los componentes más importantes son

- Presets
- Receivers
- Pages
- Tags

Componentes de la especificación

Presets

Los presets definen el **cómo** ejecutar la prueba, este es su diagrama UML y sus atributos principales



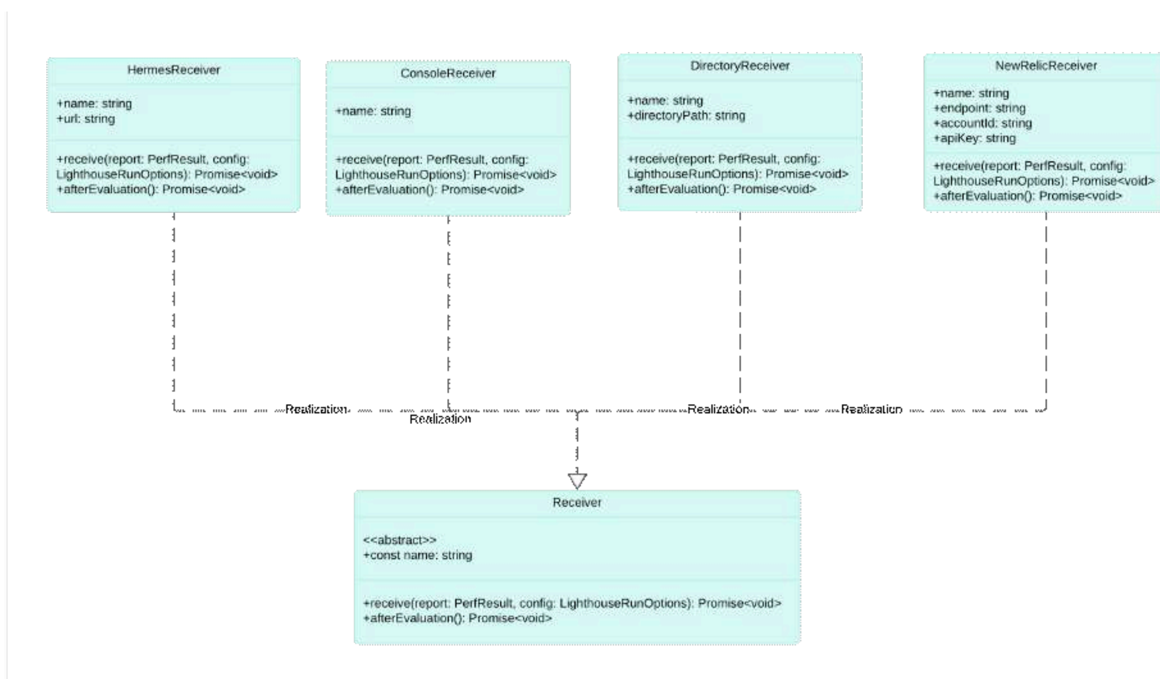
- **name**: El nombre para identificarlo
- **formFactor**: Indica si se va a correr como un dispositivo móvil o de escritorio
- **screenEmulation**: Permite especificar la resolución y escalado de la pantalla
- **throttling**: Especifica la forma en la que se va a hacer la simulación de las capacidades del dispositivo, al estar corriendo sobre máquinas no comparables con lo que utiliza un usuario final, se puede simular rendimientos inferiores y redes más lentas
- **throttlingMethod:** Permite especificar la herramienta que se va a utilizar para aplicar las **throttilingSettings**, los valores pueden ser
    - **devtools**: Utiliza las “DevTools” de Chromium para aplicar la limitación de red y CPU.
        - Ventajas
            - Proporciona una simulación precisa y confiable ya que utiliza las herramientas internas del navegador.
            - Amplio soporte y documentación.
        - Desventajas:
            - Puede ser más intensivo en recursos en comparación con otros métodos.
    - **provided:** Utiliza las capacidades nativas de la API proporcionada por el entorno de ejecución particular que se esté utilizando
        - Ventajas
            - Puede ser más rápido y menos costoso en términos de recursos si no se utiliza Chromium como entorno
        - Desventajas:
            - La precisión puede variar dependiendo del entorno y la implementación.
    - **simulate**: Simular condiciones de red y CPU sin utilizar las capacidades internas de DevTools. Para la red, puede interceptar solicitudes y aplicar

retrasos artificiales, mientras que la CPU, puede ejecutar ciclos de procesamiento intensivo para imitar una CPU lenta.
  - Ventajas:
    - Útil en entornos donde DevTools no está disponible o es limitado.
    - Puede ser más flexible en términos de personalización y ajuste fino.
  - Desventajas:
    - La precisión y confiabilidad pueden no ser tan altas como con el método **devtools**.

Receivers

Los **receivers** son los encargados de recibir el resultado de la ejecución de la prueba e implementar qué hacer con los mismos.

Este es el diagrama UML del receiver y los que la herramienta implementa.



Como se vé en el diagrama deben implementar 3 miembros

- **name**: Un nombre para identificar el receiver
- **receive**: Es una función que recibe como parámetro el resultado de la ejecución y la configuración para esa ejecución en particular.
- **afterEvaluation**: Es una función para el caso que se necesite hacer alguna limpieza luego del envío de datos, como por ejemplo cerrar una conexión a una base de datos.

A su vez, la herramienta implementa 4 receivers básicos con diferentes utilidades

- **ConsoleReceiver:** Imprime en consola el resultado, útil en ambientes locales y productivos para ver a simple vista las métricas más relevantes.

- **DirectoryReceiver**: Se encarga de guardar en disco el resultado de la ejecución, junto con el reporte que genera Lighthouse para ser visualizado luego

- **NewRelicReceiver: NewRelic** es una herramienta de monitoreo en tiempo real para aplicaciones web, muy utilizada dentro de la organización, este receiver nos permite enviar los resultados a esta plataforma para luego mediante una query **NRQL** (**N**ew **R**elic **Q**uery **L**anguage) visualizar los resultados, fue nuestra primera implementación para poder ver resultados promediados y/o porcentuales en una plataforma externa.

- **HermesReceiver:** Hermes un servicio REST, para el cual se le da de alta un **tópico** y nos permite postear la información resultante de la prueba y alojarla en un datastore, para luego poder crear tableros y gráficos con herramienta de **BI** (Business Intelligence) llamada **Metabase.** Este es el más importante,es el encargado de persistir los resultados productivos.

```
export class HermesReciever extends Receiver {

    readonly name: string = 'HERMES'
    private readonly url: string

    constructor(url: string) {
        super()
        this.url = url
    }

    async receive(report: PerfResult, config: LighthouseRunOptions): Promise<void> {
        const hermesReport = new HermesReport(report, config)
        const options = {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                'X-Client': 'perf-comp',
                'X-Uow': 'perf-comp'
            },
            body: JSON.stringify(hermesReport.getTopicBody()),
        }

        try {
            await fetch(this.url, options)
        } catch(error: any) {
            const errorBody = await error.response.text()
            LOGGER.error('Could not send report to Hermes - error was:', errorBody)
        }
    }

    async afterEvaluation(): Promise<void> {
        return Promise.resolve()
    }
```

(Ejemplo: **HermesReceiver**)

Pages

La especificación para **pages**, está compuesta por una colección de objetos donde cada uno tiene la información correspondiente a una página que se quiere medir, se lo nombró **MesurablePage,** cada una puede descomponerse en dos partes, una parte de **identificación** y una parte **datos**

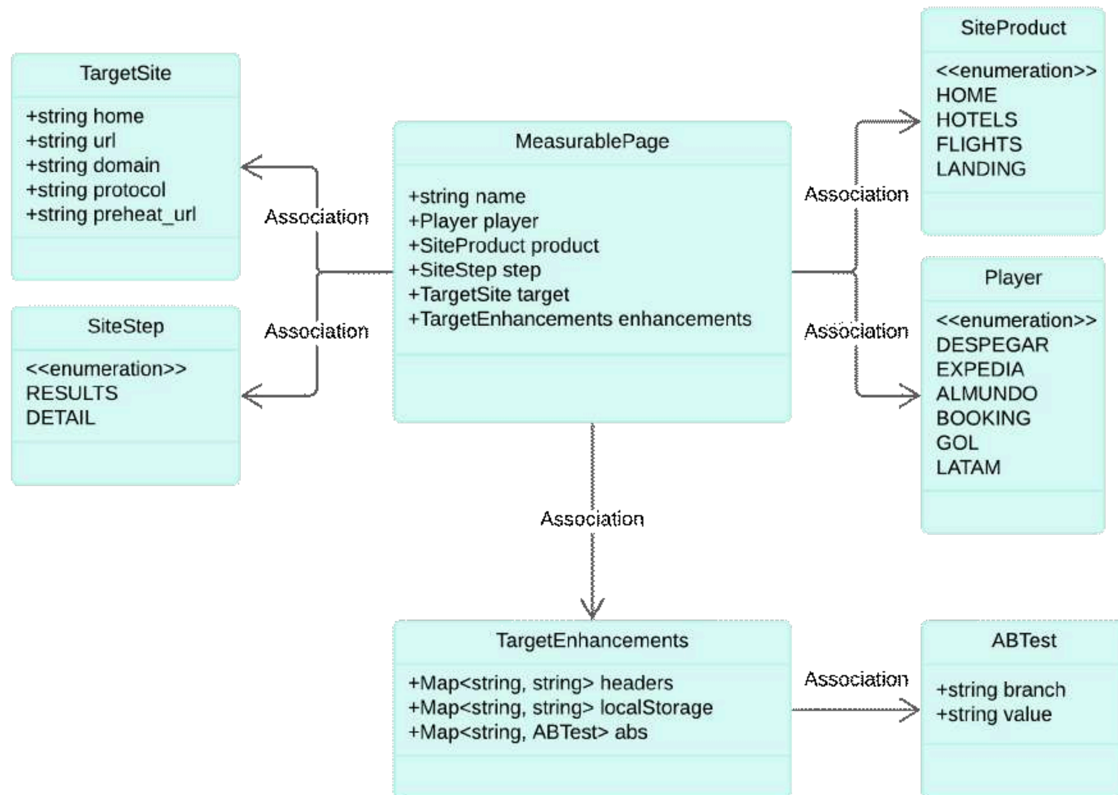
- **Identificación:**
    - **name:** Nombre asignado la página a medir
    - **player:** Representa la compañía/dueño de la página a medir, por ejemplo "Despegar"
    - **product:** Representa el producto asociado a esa página, por ejemplo "Hoteles"
    - **step:** Representa el paso en el flujo de la página, por ejemplo "Resultados"

A partir de la combinación de **player, product y step** podemos establecer un punto de comparación, donde para mismo **product y step** podemos comparar el resultado entre los distintos **players.**

- **Datos:**
    - **target:** Dentro de este objeto tenemos lo más importante, la url que en definitiva queremos medir, el target está compuesto principalmente por:
        - **home**: La url de la home del sitio, es útil para poder setear valores en la sesión antes de iniciar a medir.
        - **url**: La URL completa que se va a medir, con todos sus parámetros
        - **preheat_url:** Es una URL adicional, parecida a la que se va a medir, con algunos parámetros alterados para evitar caches de los servicios, es útil dado que al correr cada vez se instancia un navegador nuevo, se debe descargar todos los recursos estáticos del sitio, generando aumentos en los tiempos que los usuarios que frecuentan el sitio no sufren.
    - **enhancements:** La función de este objeto es albergar configuraciones adicionales para enriquecer la prueba, contiene la posibilidad de:
        - **Agregar headers**: Muchas veces, para hacer pruebas internas, una serie de headers nos permiten acceder a funcionalidades de debug o funcionalidades que se encuentran en ambientes de testing.
          Esto nos permite hacer pruebas prematuras sobre estos ambientes inestables.
        - **Agregar entradas al local storage:** Como se mencionó anteriormente, al iniciar un navegador limpio nos encontramos con distintas comunicaciones que se le muestran a los usuarios la primera vez que navegan, como por ejemplo el banner de consentimiento de cookies, promociones, etc.
          Esto afecta al resultado de las pruebas, incluso no permitiendo medir adecuadamente lo que nos interesa. Por eso mediante entradas en el localstorage podemos evitar la renderización de ciertos elementos.
        - **Agregar A/B Testings:** Esto es una funcionalidad más interna, nos permite de forma fácil agregar headers necesarios para forzar

distintas ramas de un A/B Testing y así poder medir la performance entre la rama control y test.

Este es el diagrama UML correspondiente a una **“MesurablePage”**



Construcción del **“Target”**

La parte central de esta especificación son los **Targets**, dado que son los sitios que se quieren medir, para eso se definió una serie de **“Builders”** que encapsulan la lógica para armar los componentes del target, dándole dinamismo entre cada ejecución

Por ejemplo, para una búsqueda de Hoteles, los atributos clave son

- **Destino**
- **Fecha de check in**
- **Fecha de check out**
- **Cantidad de personas**

Si bien no podemos asegurar que para esta combinación siempre tengamos resultados en todos los sitios, se utilizan destinos populares y fechas promedio en las que los usuarios buscan para el producto, para este ejemplo se utilizan fechas al azar, dentro de 4 meses hacia adelante y con una estadía no mayor a 5 noches.

```
const DESPEGAR_HOTELS_RESULTS = (): MeasurablePage => {
    const translator = new DespegarHotelsParameterTranslator()
    const site = SiteRandomizer.get()
    const destination = translator.translateCity(CityRandomizer.get())
    const travelers = translator.translateTravelers(TravelersRandomizer.get())
    return {
        name: "DESPEGAR_HOTELS_RESULTS",
        player: Player.DESPEGAR,
        product: SiteProduct.HOTELS,
        step: SiteStep.RESULTS,
        enhancements: { ...DEFAULT_ENHANCEMENTS },
        target: TargetSiteBuilder.build(
            DespegarUrlBuilder.buildHotelsUrl(
                site,
                destination,
                translator.translateDateRange(DateRangeRandomizer.get(5)), // 5 dias MAXIMO
                travelers.distribution
            ),
            DespegarUrlBuilder.buildHotelsUrl(
                site,
                destination,
                translator.translateDateRange(DateRangeRandomizer.get(5)), // 5 dias MAXIMO
                travelers.distribution
            )
        )
    }
}
```

(Ejemplo de del **“Builder”** para resultados de **“Hoteles”** y player **“Despegar”)**

Construcción de los parámetros

Cuando hablamos de generar dinamismo en las búsquedas, necesitamos un método para obtener valores al azar para cada una de las ejecuciones, para esto se hicieron una series de clases utilitarias que se las llamó **Randomizers.**

```
export class CityRandomizer {
    static get(exclude?: CityParameter): CityParameter {
        let allCities = ALL_CITIES
        if(exclude) {
            allCities = ALL_CITIES.filter(city => city !== exclude)
        }
        const randomCityIndex = Math.floor(Math.random() * allCities.length)
        return allCities[randomCityIndex]
    }
}

export enum CityParameter {
    MIAMI = 'miami',
    BUENOS_AIRES = 'buenos-aires',
    NEW_YORK = 'new-york',
    RIO_DE_JANEIRO = 'rio-de-janeiro',
    CANCUN = 'cancun',
    MADRID = 'madrid',
    BARCELONA = 'barcelona',
    PARIS = 'paris',
}

const ALL_CITIES: CityParameter[] = Object.values(CityParameter)
```

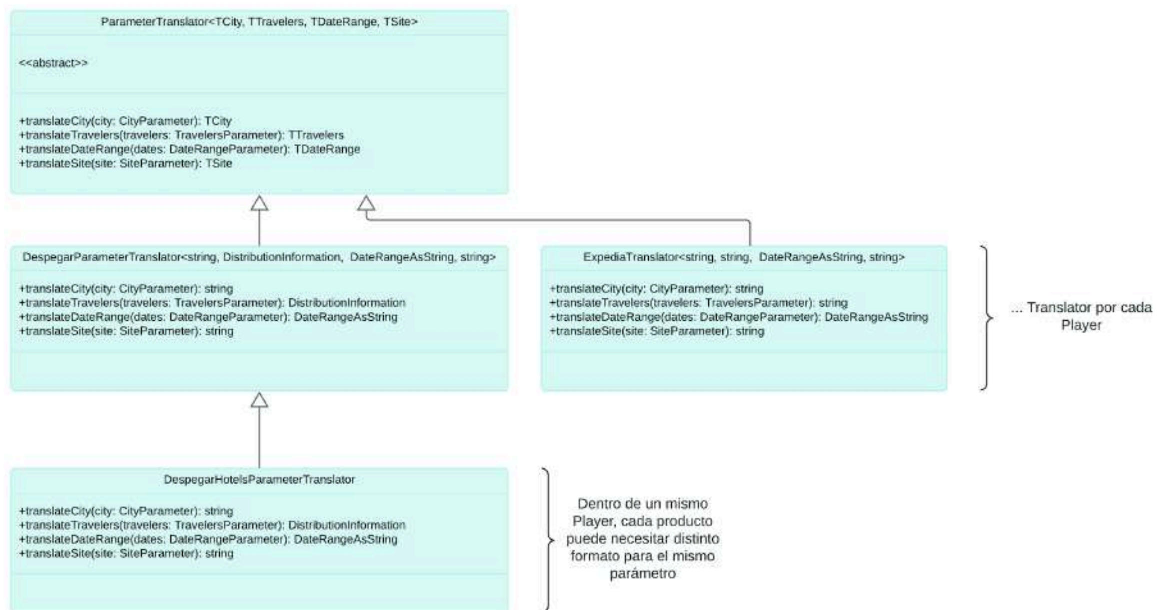
(Ejemplo de **CityRandomizer**)

Mediante el método ***get(exclude?: CityParameter): CityParameter*** obtiene una ciudad al azar permitiendo, opcionalmente, excluir la que se le pasa por parámetro, es útil para los productos como Vuelos que tienen un origen y un destino que no pueden ser el mismo.

Este esquema nos lleva al concepto de **parámetro canónico**. Estos nombres de ciudades (y todos los parámetros que provee un randomizer) son canónicos.
Un **parámetro canónico** refiere a un formato de datos genérico y estandarizado que se utiliza como base común para la generación de datos.
Actúan como una representación intermedia que no depende de casos específicos, lo que permite una mayor flexibilidad y reusabilidad del código. Posteriormente, estos parámetros son transformados a formatos específicos según las necesidades particulares de cada caso de uso.

Este concepto nos facilita la abstracción y simplificación de los procesos de generación de datos, promoviendo una arquitectura modular y adaptable.
Además, permite que las clases utilitarias operen de manera uniforme, asegurando consistencia, coherencia en la manipulación y procesamiento de datos en la aplicación.

Como se puede ver en la imagen de ejemplo existen los que llamamos **ParameterTranslators**, estos componentes tienen como responsabilidad traducir del parámetro canónico a un parámetro específico, el cual es el que se utiliza para construir la URL final a la que se va a testear.
Estos **Translators** tienen el siguiente diagrama UML



ParameterTranslator es una clase abstracta que define 1 método por cada parámetro posible y para su construcción necesita de el tipo de retorno de cada método. De esta forma se implementa un translator por cada Player y podemos llevar nuestros parámetros canónicos a cualquier formato que necesiten los sitios particulares.
Incluso como se ve en el diagrama, podemos subclasificar para hacer una implementación específica para un producto, como es el caso de Hoteles donde el destino utiliza un formato distinto a Vuelos.

```
export abstract class ParameterTranslator <
    TCity = any,
    TTravelers = any,
    TDateRange = any,
    TSite = any
> {
    abstract translateCity(city: CityParameter): TCity
    abstract translateTravelers(travelers: TravelersParameter): TTravelers
    abstract translateDateRange(dates: DateRangeParameter): TDateRange
    abstract translateSite(site: SiteParameter): TSite
}
```

*(Definición de **ParameterTranslator**)*

```
    translateTravelers(travelers: TravelersParameter): DistributionInformation {
        switch(travelers) {
            case TravelersParameter.COUPLE:
                return {
                    adults: 2,
                    children: 0,
                    infants: 0,
                    distribution: "2"
                }
            case TravelersParameter.FRIENDS:
                return {
                    adults: 4,
                    children: 0,
                    infants: 0,
                    distribution: "4"
                }
            case TravelersParameter.FAMILY:
                return {
                    adults: 2,
                    children: 2,
                    infants: 0,
                    distribution: "2-2:6-11"
                }
            default:
                return {
                    adults: 1,
                    children: 0,
                    infants: 0,
                    distribution: "1"
                }
        }
    }
    translateDateRange(dates: DateRangeParameter): DateRangeAsString {
        return {
            start: dates.start.format("YYYY-MM-DD"),
            end: dates.end.format("YYYY-MM-DD")
        }
    }
    translateSite(site: SiteParameter): string {
        return site
    }
}
```

(Implementación de **DespegarParameterTranslator)**

```
export class DespegarHotelsParameterTranslator extends DespegarParameterTranslator {
    translateCity(city: CityParameter): string {
        switch(city) {
            case CityParameter.BARCELONA:
                return "CIT_576"
            case CityParameter.BUENOS_AIRES:
                return "CIT_982"
            case CityParameter.CANCUN:
                return "CIT_1569"
            case CityParameter.MADRID:
                return "CIT_4361"
            case CityParameter.MIAMI:
                return "CIT_4544"
            case CityParameter.NEW_YORK:
                return "CIT_5227"
            case CityParameter.PARIS:
                return "CIT_5543"
            case CityParameter.RIO_DE_JANEIRO:
                return "CIT_6381"
            default:
                return "CIT_982"
        }
    }
}
```

*(Implementación de **DespegarHotelsParameterTranslator**)*

En este último se puede ver cómo se redefine únicamente el método **translateCity** para utilizar el formato numérico en vez del IATA del destino.

Tags

Los tags son marcas que se incluyen dentro del envío de la métrica como metadata con el fin de poder aislar ciertos escenarios o incluso poder versionar las métricas.

Se utilizó en las primeras versiones de la herramienta donde todavía las métricas estaban en un estadío experimental, de forma de saber que esos número provenían de una versión temprana y que podían no ser del todo confiables.

Las métricas que se trackean

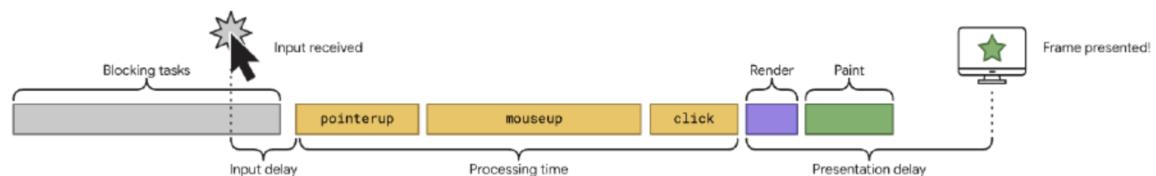
Como resultado de la ejecución de las pruebas obtenemos las métricas en las cuales nos basaremos para ver su evolución en el tiempo, establecer valores de referencia con el objetivo de determinar si es necesario mejorar y para poder comparar con los otros sitios. *(Los valores de referencia fueron obtenidos desde las fuentes citadas en la bibliografía)*

Estamos trackeando las siguientes métricas:

INP (**I**nteraction to **N**ext **P**aint)

Evalúa la capacidad de respuesta general de una página ante las interacciones del usuario mediante la observación de la latencia de todas las interacciones de clic, presión y teclado que ocurren durante la visita del usuario a una página.
El valor final de la INP es la interacción más larga observada, sin tener en cuenta los valores atípicos.

Para clarificar esta definición podemos usar este gráfico:



En la imagen vemos el ciclo de una interacción, la primera parte gris muestra tareas que están corriendo en el hilo principal, como puede ser requests HTTP, parseo de JSON, lógica de la aplicación, etc.
Cuando se genera un click, tenemos un tiempo marcado como *"Input delay"* que es el tiempo entre que el hilo principal termina lo que está ejecutando, para poder atender la interrupción del sistema por el clic y finalmente disparar todos los eventos correspondientes, luego el renderizado del resultado del click, el pintado y por último la presentación en pantalla. Esta métrica va a tomar como valor la suma desde el inicio del **"input delay"** hasta la finalización del **"presentation delay"**

Esta métrica nos indica qué tan rápido reacciona nuestra aplicación ante eventos del usuario, es un indicador de capacidad de respuesta.

Valores de Referencia

- **Bueno**: menor a 200 milisegundos
- **Mejorable**: Entre 200 y 500 milisegundos
- **Malo**: Mayor a 500 milisegundos

TTFB (**T**ime **T**o **F**irst **B**yte)

Mide el tiempo entre la solicitud de un recurso y el momento en que comienza a llegar el primer byte de una respuesta, esta compuesto por la suma de las siguientes fases de la solicitud:

- Hora del redireccionamiento
- Tiempo de inicio del service worker (si corresponde)
- Búsqueda de DNS
- Conexión y negociación de TLS
- Solicitud, hasta el momento en que llega el primer byte de la respuesta

Es una métrica que representa las capacidades del servidor para responder ante navegaciones

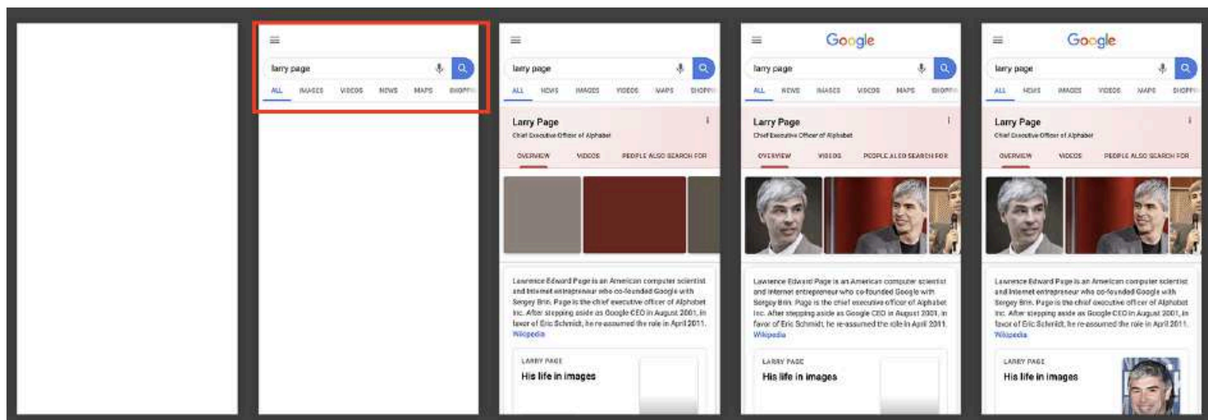
Valores de Referencia

- **Bueno**: menor a 800 milisegundos
- **Mejorable**: Entre 800 milisegundos y 1,8 segundos
- **Malo**: Mayor a 1,8 segundos

FCP (**F**irst **C**ontentful **P**aint)

Mide el tiempo que transcurre desde que el usuario navega a una página por primera vez hasta que se renderiza en la pantalla cualquier parte del contenido de la página. Para esta métrica, "contenido" hace referencia al texto, las imágenes (incluidas las de fondo), los elementos <svg> o los elementos <canvas> que no son blancos.

El FCP incluye el tiempo de descarga de la página anterior, el tiempo de configuración de la conexión, el tiempo de redireccionamiento y TTFB.



En la imagen se pueden ver 5 pasos de la carga del sitio, el recuadro rojo marca el FCP, es la carga del primer contenido.

Es una métrica centrada en medir la velocidad de carga percibida por el usuario, ya que marca el primer punto en el que el usuario puede ver algo en pantalla. Un FCP rápido ayuda a la percepción de progreso en el usuario.

Valores de referencia

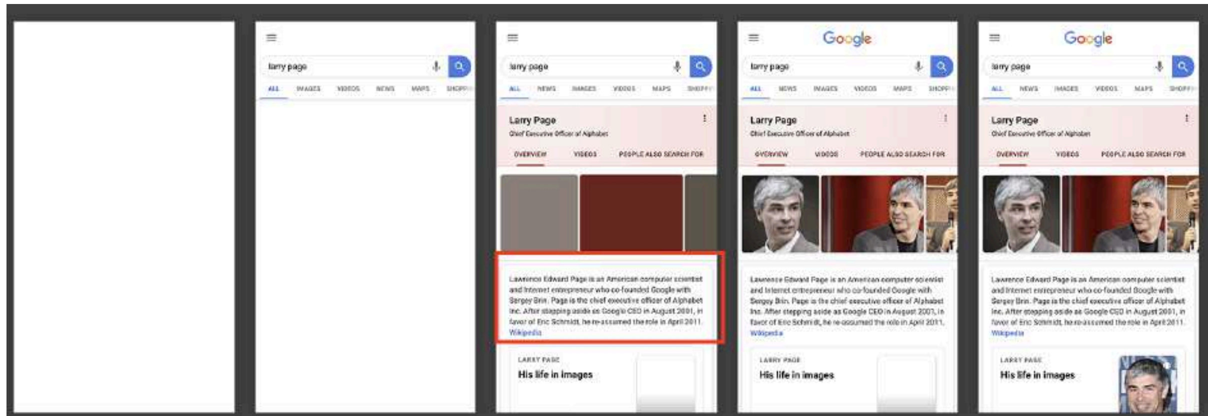
- **Bueno**: menor a 1,8 segundos
- **Mejorable**: Entre 1,8 y 3 segundos
- **Malo**: Mayor a 3 segundos

LCP (**L**argest **C**ontentful **P**aint)

Informa el tiempo de renderización del bloque de imagen o texto más grande visible en la pantalla, en relación con el momento en que el usuario navegó a la página por primera vez, esto quiere decir que la métrica incluye el tiempo de TTFB y FCP.

Cuando decimos “bloque de imagen o texto más grande” la métrica toma en cuenta los siguientes tags HTML

- Elementos **<img>**
- Elementos **<image>** dentro de Elementos **<svg>**
- Elementos **<video>** (Se utiliza el tiempo hasta que se dibuja la miniatura)
- Un elemento con una imagen de fondo cargada con la función **url()** de CSS
- Elementos de bloque con texto como **<p>, <h1>**



Tomando como referencia el mismo ejemplo del FCP, en este caso el LCP está indicado con el recuadro rojo, donde se ve dentro de la pantalla el primer bloque grande de texto, las imágenes en este caso no serían tomadas en cuenta.

Valores de referencia

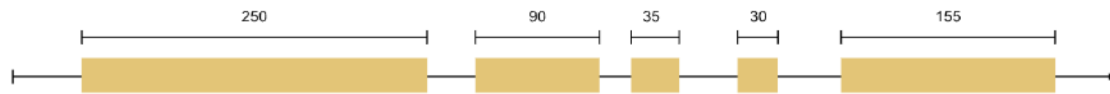
- **Bueno**: menor a 2,5 segundos
- **Mejorable**: Entre 2,5 y 4 segundos
- **Malo**: Mayor a 4 segundos

TBT (**T**otal **B**locking **T**ime)

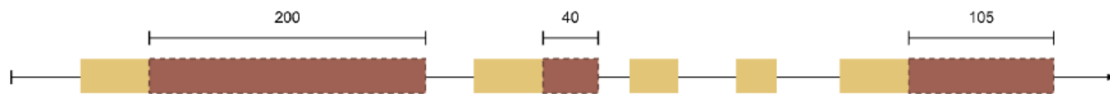
Mide la cantidad total de tiempo después del FCP durante el cual el hilo principal se bloqueó durante el tiempo suficiente para evitar la capacidad de respuesta de la entrada.

Se considera que está *"bloqueado"* cada vez que exista una *“Tarea larga”* que se ejecute en el hilo principal durante más de 50 milisegundos.
Decimos que el hilo principal está bloqueado porque el navegador no puede interrumpir una tarea en curso, Por lo tanto, en caso de que un usuario sí interactúe con la página en medio de una tarea larga, el navegador debe esperar a que esta finalice para poder responder.

Supongamos esta organización en la ejecución de tareas en el hilo principal



Cada barra amarilla representa una tarea y su duración en milisegundos, si a cada barra le restamos 50 milisegundos obtenemos el tiempo en el que el hilo está bloqueado



Luego, sumamos estos valores (200 + 40 + 105) y obtenemos el TBT de 345 milisegundos.

Esta métrica tiene como objetivo evidenciar si el usuario puede sufrir que sus interacciones no tengan una respuesta acorde en el tiempo en diversos momentos del uso de la aplicación

Valores de referencia

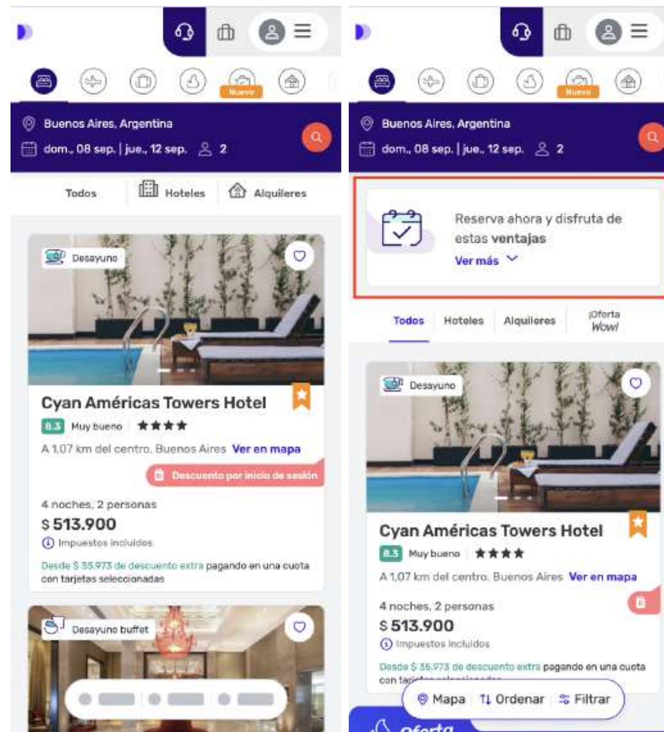
- **Bueno**: menor a 200 milisegundos, esto teniendo en cuenta el hardware móvil promedio.

CLS (**C**umulative **L**ayout **S**hift)

Mide el acumulado de los movimientos de los elementos en una página, cuanto más se *"mueva"* la vista más alto es el valor.
Un elemento se considera como que se movió cuando de un fotograma a otro está en una posición distinta, cada movimiento se le da una puntuación respecto a que tanto se movió.

Esta métrica es importante dado que un número alto genera que el usuario esté sufriendo de cambios en la pantalla que no espera, generando interacciones erróneas o llevando a que tenga que esperar más tiempo para poder interactuar correctamente. Es muy común con la carga asincrónica de elementos, por lo que se recomienda, en caso de agregar elementos que empujen a otros hacia abajo, representar este elemento en un estado de carga para que una vez cargado podamos reemplazar este elemento por el real y evitar los movimientos, además de darle información al usuario de que algo estará disponible en esa sección.

Por ejemplo, en estas imágenes podemos ver este fenómeno, en la primera se carga el sitio, mostrando los resultados de los hoteles, al fondo se puede ver el uso de un *"skeleton"* para indicar que la barra de abajo está cargando y que no aparezca repentinamente. Pero en la segunda imágen en el recuadro rojo, podemos ver cómo cargó un banner por encima de los resultados, generando un salto en la vista, este comportamiento es penalizado por la métrica.



Valores de referencia

- **Bueno**: menor a 0,1
- **Mejorable**: Entre 0,1 y 0,25 segundos
- **Malo**: Mayor a 0,25

Speed Index

Mide la rapidez con la que el contenido se visualiza en una página web durante la carga de la misma, Específicamente, el Speed Index calcula que tan rápido se renderiza el contenido dentro del *"viewport"* (lo que se ve sin hacer scroll) durante la carga de la página.

Un Speed Index bajo indica que la mayor parte del contenido se carga rápidamente, mientras que un Speed Index alto sugiere que el contenido tarda más en aparecer.

Valores de referencia

- **Bueno**: menor a 3,4 segundos
- **Mejorable**: Entre 3,4 y 5,8 segundos
- **Malo**: Mayor a 5,8 segundos

Score

Esta métrica sirve para asignar un número al conjunto de estas métricas, generando un puntaje, donde cada métrica tiene un peso particular en la ecuación.

Para el caso de Lighthouse 10 estas son las ponderaciones

- **FCP**: 10%
- **Speed Index**: 10%
- **LCP**: 25 %
- **TBT**: 30%
- **CLS**: 25%

Nos da un punto de referencia general de nuestro sitio, pero es necesario mirar en detalle cada una de ellas para identificar los puntos de mejora.

Dom Size

Simplemente es la cantidad de nodos que tiene nuestra página, cuanto más y con más profundidad tenga, peor será la performance, dado que cada renderizado debe barrer todos los elementos para modificarlos o comprobar si necesitan ser modificados.

Esta métrica nos evidencia la complejidad del maquetado de nuestro sitio, donde un número muy grande puede significar un mal maquetado o que el sitio tiene una carga cognitiva muy grande para el usuario, dado que tiene muchos elementos en pantalla.

Valores de referencia

- **Bueno**: menor a 800 nodos
- **Mejorable**: Entre 800 y 1.400 nodos
- **Malo**: Mayor a 1.400 nodos

Cómo es una ejecución

La ejecución de las pruebas sigue un ciclo que se repite todo el tiempo con el objetivo de recolectar muestras y minimizar errores en los datos por la variabilidad de los resultados. Como vimos antes en el apartado de Especificación tenemos un cron que dispara la ejecución de la prueba.

Al dispararse este cron, se toma como la especificación por defecto configurada, es enviada a un método que asegura mediante un lock que no exista otra ejecución en curso, si no existe una, empieza la prueba

```
    server.listen(PORT, () => {
        LOGGER.info(`Listening on Port: ${PORT}`)
        if(TIMER) {
            runEvaluation(runConfig)
            LOGGER.info(`Scheduled to run at every ${TIMER_STRING}`)
            INTERVAL = scheduleRun(runConfig)   ──▶ Se configura el timer
        }
    })
}

const runEvaluation = async (runConfig: RunConfiguration): Promise<void> => {
    if (RUNNING_LOCK) {
        LOGGER.debug('Already running, skipping this run.')   ──▶ Otra ejecucion en curso, ignora
        return Promise.resolve()
    } else {
        RUNNING_LOCK = true   ──▶ Setea el Lock y ejecuta
        return evaluate(runConfig).then(() => {
            LAST_RUN_STATUS = 'SUCCESS'
        }).catch((err) => {
            LOGGER.error(err)
            LAST_RUN_STATUS = 'FAILURE'
        }).finally(() => {
            RUNNING_LOCK = false   ──▶ Libera el Lock, sin importar el resultado
        })
    }
}

const scheduleRun = (runConfig: RunConfiguration): NodeJS.Timeout => {
    return setInterval(() => {
        runEvaluation(runConfig)
    }, TIMER)
}
```

Para cada **page y target** configurado se instancian las propiedades necesarias para la ejecución de Puppeteer y Lighthouse.

```
async function evaluate(config: RunConfiguration): Promise<void> {
    let skippedRunCount = 0
    const totalRunsToExecute = (config.pages.length * Object.values(config.presets).length)
    const startTime = Date.now()
    for (const page of config.pages) {   ──▶ Cada página
        LOGGER.info(`Working on ${page.name}: ${page.target.url}`)
        for (const preset of Object.values(config.presets)) {   ──▶ Con cada configuracion de dispositivos
            LOGGER.info(`+ Running preset ${preset.preset}`)
            const runStartedAt = new Date()
            const options: LighthouseRunOptions = { preset, page, runStartedAt, tags: config.tags, onlyCategories: config.onlyCategories}
            try {
                await Lighthouse.run(config.receivers, options)   ──▶ Ejecución de la prueba
            } catch(e: any) {
                LOGGER.warn(`Skipping run of ${page.name}: ${page.target.url} with preset: ${preset.preset}`)
                skippedRunCount = skippedRunCount + 1
            }

        }
    }
    const totalTime = (Date.now() - startTime)
    LOGGER.info(`*--------------------*****-------------------------------*`)
    LOGGER.info(`| Finished lighthouse runs, it took: ${Utils.toNearestMagnitude(totalTime)}`)
    LOGGER.info(`| Total runs executed: ${totalRunsToExecute - skippedRunCount}`)
    LOGGER.info(`| Total skipped runs: ${skippedRunCount}`)
    LOGGER.info(`*--------------------*****-------------------------------*`)
    if (skippedRunCount === totalRunsToExecute) {
        LOGGER.error("All runs were skipped")
    }

    for (const receiver of config.receivers) {
        receiver.afterEvaluation();
    }
}
```

Estadísticas de la ejecución

Se crea una instancia de ***BrowserManager*** donde su función es gestionar la creación de instancias de Chromium, configurar ciertos aspectos del mismo con el fin de deshabilitar ciertas funciones que no afectan al resultado pero sí a la velocidad de ejecución de las pruebas.

Una vez instanciado Chromium se configuran los interceptores para que en cada petición se agreguen los headers proporcionados para la prueba y luego se hace una primer navegación hacia la URL de *preheat*, con el fin de cachear recursos estáticos, para simular al momento de hacer la prueba realmente, que el usuario que navega es uno que al menos ya entró en el sitio una vez. Esto es importante, dado que no tener ningún recurso afecta negativamente a la prueba y pudimos corroborar que la gran mayoría de los usuarios que navegan ya lo hicieron con anterioridad.

```
private static async evaluate(options: LighthouseRunOptions): Promise<PerfResult | undefined> {
    const manager = new BrowserManager()  → Instancia el BrowserManager
    try {
        await manager.launch()  → Corre Chromium
        await manager.overridePermisions(options.page.target.home, [])
        const port = manager.getDebugPort();
        const page = await manager.getPage()  → Obtiene la Página actual de Chromium
        page.on("dialog", this.acceptBeforeUnload);
        const abHeader = this.buildABHeader(options.page.enhancements?.abs)
        const headers = {...options.page.enhancements?.headers, ...abHeader}  → Junta todos los headers configurados

        if (Object.keys(headers).length > 0) {
            await page.setRequestInterception(true);  → Habilita la intercepción de requests y agrega los headers
            page.on('request', interceptedRequest => {
                if (page.target()) {
                    this.addHeaders(interceptedRequest, options.page.target.domain, headers)
                }
            })
            LOGGER.debug(`On every request to ${options.page.target.domain} this headers will be added: ${JSON.stringify(headers)}`)
        }

        LOGGER.info(`Pre-heat URL ${options.page.target.preheat_url}`)
        await page.goto(options.page.target.preheat_url, {  → Navega hacia la URL de precalentamiento
            timeout: 30000,
            waitUntil: ['load', 'domcontentloaded', 'networkidle2']
        })
                                                  Agrega para ese dominio entradas en el localStorage
        if (options.page.enhancements?.localStorage) {
            await page.evaluate(this.addToLocalStorage, [options.page.enhancements.localStorage])
            LOGGER.debug(`Added to localstorage  ${JSON.stringify(options.page.enhancements.localStorage)}`)
        }
```

Haciendo un doble click sobre el **BrowserManager:**

```
export class BrowserManager {
    private static BROWSER_OPTIONS: PuppeteerLaunchOptions = {
        headless: config.has('lighthouse.headless') ? (config.get('lighthouse.headless') ? 'new' : false) : false,
        executablePath: config.has('chrome_exec_path') ? config.get('chrome_exec_path') : undefined,
        args: [
            '--no-sandbox',
            '--remote-debugging-port=0',
            '--incognito',
            '--disable-extensions',
            '--disable-add-to-shelf',
            '--disable-breakpad',
            '--disable-client-side-phishing-detection',
            '--disable-datasaver-prompt',
            '--disable-default-apps',
            '--disable-desktop-notifications',
            '--disable-dev-shm-usage',
            '--disable-features=TranslateUI,BlinkGenPropertyTrees',
            '--disable-infobars',
            '--enable-automation',
            '--force-device-scale-factor=1',
            '--user-agent=Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/75.0.3770.142 Safari/537.36'

        ]
    }

    private static instance: Browser | null = null
    private static debugPort: number | null = null

    async launch(): Promise<Browser> {
        if(!BrowserManager.instance){
            try {
                LOGGER.debug('Launching browser')
                BrowserManager.instance = await puppeteer.launch(BrowserManager.BROWSER_OPTIONS);
                BrowserManager.debugPort = this.setDebugPort(BrowserManager.instance);
            } catch(e) {
                BrowserManager.instance = null
                BrowserManager.debugPort = null
                LOGGER.error('Cannot start browser', e)
                return Promise.reject(e)
            }
        }
        return BrowserManager.instance
    }
```

Argumentos de Chromium para deshabilitar funcionalidades

Singleton para impedir más de un Chromium a la vez

Inicio del browser mediante puppeteer

Luego usando la librería de Lighthouse se corre la prueba sobre la página objetivo, se extrae el resultado y mediante el ***BrowserManager*** se destruye la instancia de Chromium.

```
    const flags: Flags = {
        port,
        throttling: options.preset.throttling,
        throttlingMethod: options.preset.throttlingMethod,
        formFactor: options.preset.formFactor,
        screenEmulation: options.preset.screenEmulation,
        disableStorageReset: true,
        skipAboutBlank: true,
        onlyCategories: options.onlyCategories
    }
    LOGGER.info('Running lighthouse')
    const flow: UserFlow = await startFlow(page, { flags })
    await flow.navigate(options.page.target.url, flags)
    await manager.closeBrowser()
    const flowResult = await flow.createFlowResult()
    const artifacts = (await flow.createArtifactsJson()).gatherSteps[0].artifacts
    const html = await flow.generateReport()
    const lhr = flowResult.steps[0].lhr
    LOGGER.info(`Finished lighthouse run for ${options.preset.preset}`)
    return {
        artifacts,
        html,
        lhr,
        json: flowResult
    };
} catch(e: any) {
    LOGGER.error(e);
    await manager.closeBrowser();
    return Promise.reject(new Error(e))
}
```

Configuraciones para Lighthouse

Creación del Flow de ejecución de Lighthouse

Ejecución de la prueba sobre el sitio

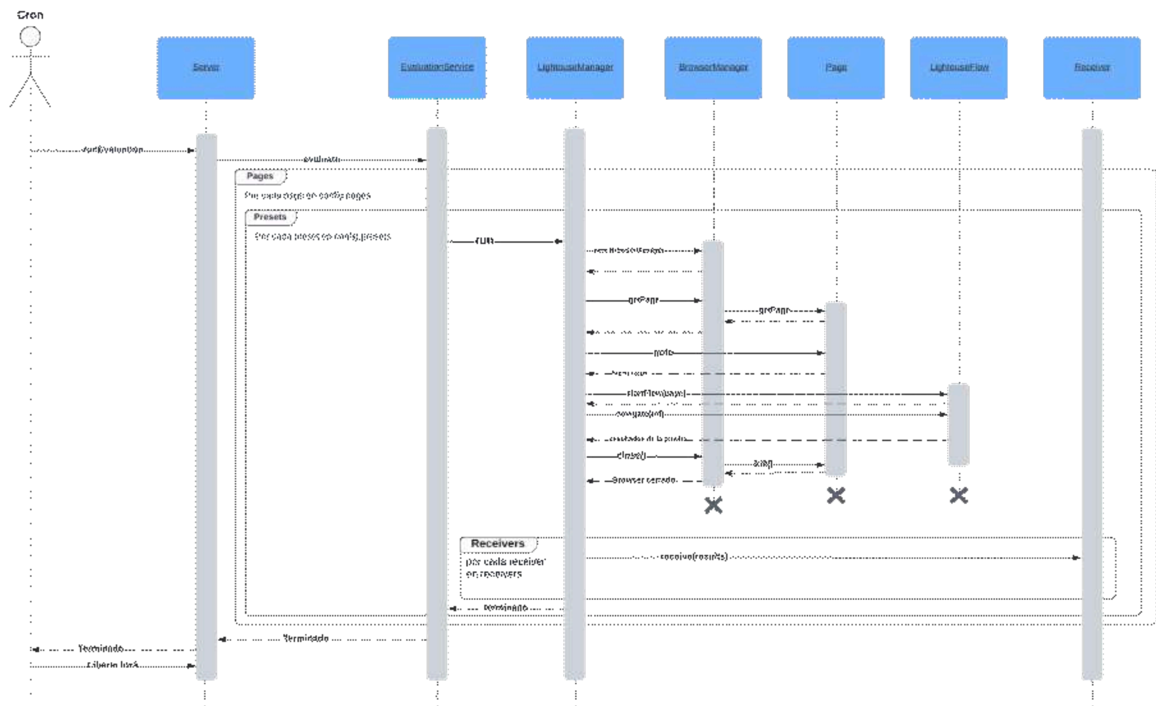
Obtención de las Métricas

Para finalizar, para cada **receiver** configurado, se le envía el resultado obtenido para que lo reporte donde corresponda.

```
static async run(receivers: Receiver[], options: LighthouseRunOptions): Promise<void> {
    const result = await Lighthouse.evaluate(options)
    if (result) {
        for (const receiver of receivers) {
            await receiver.receive(result, options)
        }
    }
}
```

Para cada Receiver
envía las métricas obtenidas

Por último veamos un diagrama de secuencia ilustrando el flujo completo.



Tableros

Una vez terminada la ejecución como mostramos anteriormente, los resultados se envían a una base de datos para su posterior procesamiento.
Este procesamiento está a cargo de una herramienta de BI llamada Metabase que nos va a permitir generar tableros en base a consultas sobre la tabla de esta base de datos.

Actualmente tenemos dos tableros, cada uno muestra 3 gráficos, éstos son:

- Evolución en el tiempo
- Acumulado en el tiempo

(Para todos los gráficos se tomó como periodo de tiempo, el rango desde el 1ro de Agosto al 24 de Agosto del 2024 y se utilizaron valores promedio.)

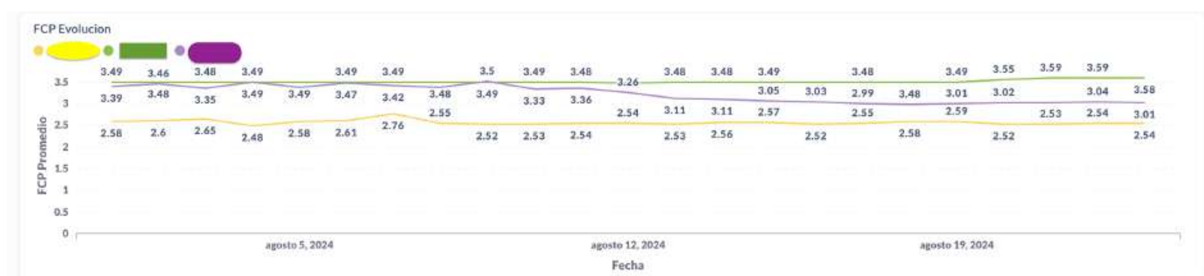
Una pregunta que surge es ¿por qué usar promedios y no percentiles ? Dado que nuestras pruebas ejecutan una simulación en la navegación, los valores resultantes tienen una dispersión más grande, si utilizáramos percentiles nos estaríamos quedando con los valores donde las pruebas arrojaron métricas que tuvieron peores resultados y no representan bien a toda la muestra.

(Por confidencialidad de los datos, las leyendas de a qué player pertenece cada valor fueron censurados en todos los gráficos)

Evolución en el tiempo

Estos tableros nos permiten hacer un seguimiento temporal de las métricas, poder monitorear desviaciones de la misma y buscar las causas en los casos que lo ameriten.

A veces debido a una funcionalidad nueva que se deploya, las métricas pueden verse afectadas, esto nos permite saber cuándo ocurrió y simplificar la búsqueda de la causa



(FCP en el tiempo para 3 Players)

(LCP en el tiempo para 3 Players)

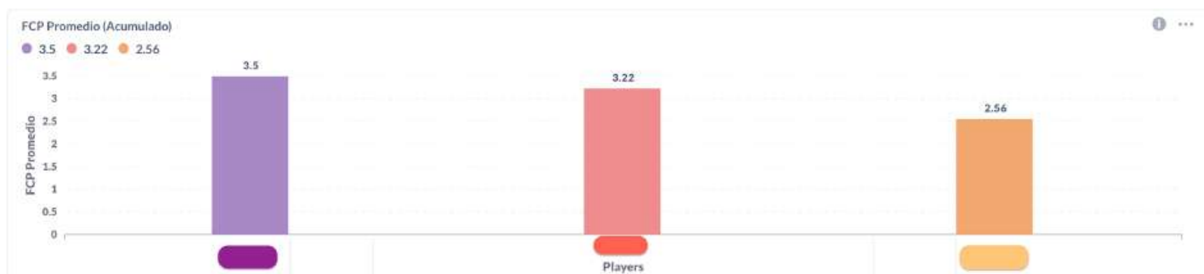
(Score en el tiempo para 3 Players)

A modo de ejemplo, la consulta que se ejecuta para obtener los datos, el ejemplo muestra para FCP pero es análogo para el resto de las métricas

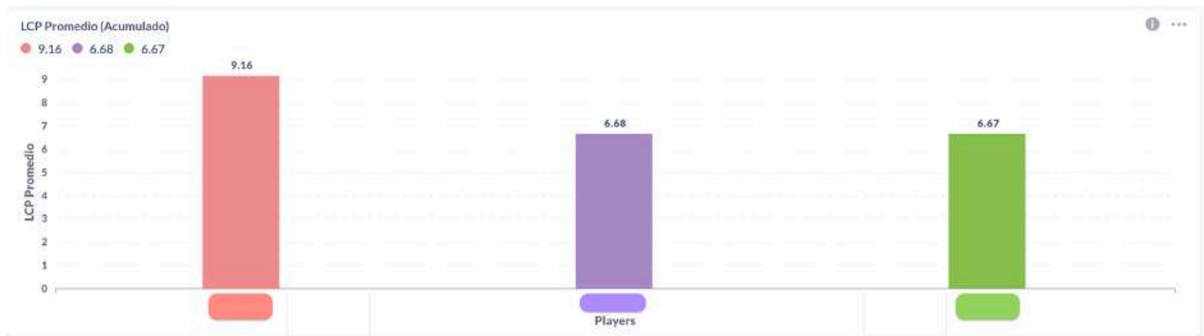
```
SELECT
count(*) AS TOTAL_SAMPLES,
m.event.player,
m.event.product,
m.event.step,
m.event.preset,
SUBSTRING(m.datetime, 1, 10) as eventDate,
(avg(m.event.fcp)/1000) AS AVG_FCP_IN_SECONDS        → Para cada grupo calcula el promedio y lo convierte a segundos
FROM "data".sot.front_perf_metrics m
WHERE date(SUBSTRING(m.datetime, 1, 10)) >= {{start_date}} AND date(SUBSTRING(m.datetime, 1, 10)) <= {{finish_date}}   ] Filtra los eventos por el rango de fechas, producto, player, paso y
AND m.event.score > 0 AND m.event.product = {{product}} AND m.event.preset = {{preset}} AND m.event.step = {{step}}     ] plataforma y descarta eventos erróneos
GROUP BY m.event.player,m.event.product,m.event.preset, m.event.step, SUBSTRING(m.datetime, 1, 10)
ORDER BY SUBSTRING(m.datetime, 1, 10) ASC
LIMIT ALL                                            → Agrupa los resultados por player, producto, plataforma, paso y fecha
```

Acumulado en el tiempo

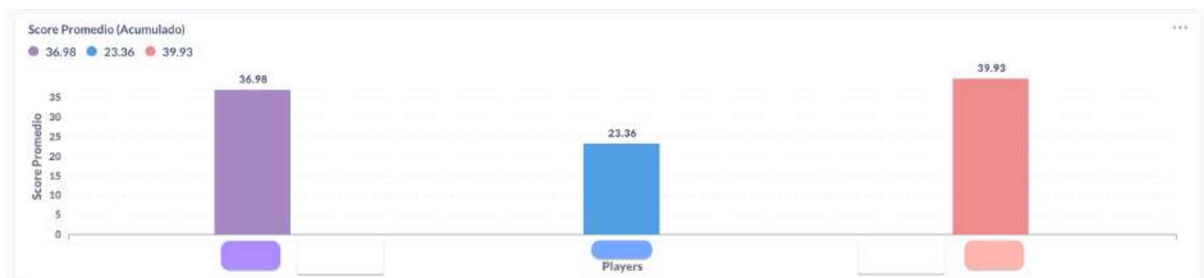
Estos tableros nos permiten ver un valor promedio de un rango de tiempo en un solo valor, estos gráficos son útiles para generar reportes mensuales y establecer un valor promedio para nuestra métrica.



(FCP acumulado para 3 Players)



(LCP acumulado para 3 Players)



(Score acumulado para 3 Players)

A modo de ejemplo, la consulta que se ejecuta para obtener los datos, el ejemplo muestra para FCP pero es análogo para el resto de las métricas

```
SELECT
count(*) AS TOTAL_SAMPLES,
m.event.player,
m.event.product,
m.event.step,
m.event.preset,
(avg(m.event.fcp)/1000) AS AVG_FCP_IN_SECONDS    --> Calcula para cada grupo el promedio y lo transforma a segundos
FROM "data".sot.front_perf_metrics m
WHERE date(SUBSTRING(m.datetime, 1, 10)) >= {{start_date}} AND date(SUBSTRING(m.datetime, 1, 10)) <= {{finish_date}}   ] Filtra los eventos por un rango de fechas, producto, plataforma y paso
AND m.event.score > 0 AND m.event.product = {{product}} AND m.event.preset = {{preset}} AND m.event.step = {{step}}     ] seleccionado
GROUP BY m.event.player,m.event.product,m.event.preset, m.event.step    --> Agrupa por player, producto, plataforma y paso
LIMIT ALL
```

Infraestructura y deploy

El último paso es el **empaquetado** y **deploy** de la aplicación, para esto se utilizó como herramienta **Docker**, para generar una imagen que luego es deployada en un contenedor dentro de **Kubernetes**, para automatizar este proceso de generación del empaquetado y su posterior deploy se utilizó **Github Actions**.

Dockerfile

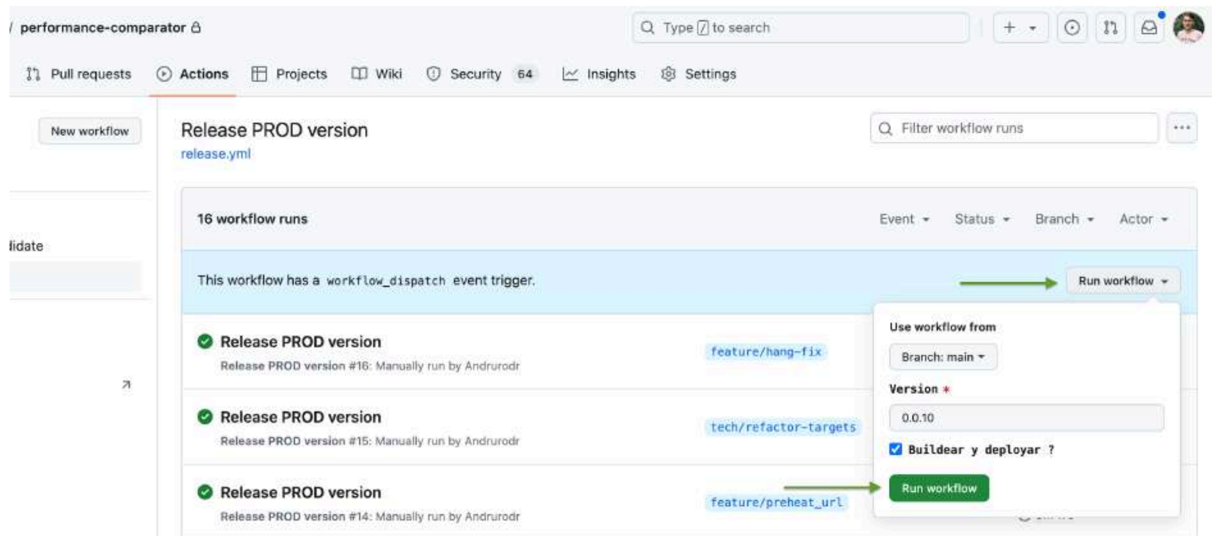
```
# Install dependencies
FROM node:18.18-alpine3.18 as dependencies  →  Imagen base, Node 18.18 sobre Alpine 3.18
RUN apk add dumb-init
WORKDIR /usr/src
COPY package*.json tsconfig.json nebula/entrypoint.sh /config/ /public/ /usr/src/  →  Copiar Código fuente y recursos dentro de la imagen
ENV PUPPETEER_SKIP_CHROMIUM_DOWNLOAD="true"
RUN npm ci  →  Descarga de dependencias

# Build artifact
FROM dependencies as build  →  Partiendo de la capa anterior
ARG NBL_SERVICE_TAG
WORKDIR /usr/src
COPY . .
## If is a valid SEMVER use the service tag, else use a prerelease version
RUN if echo "$NBL_SERVICE_TAG" | grep -qE '^[0-9]+\.[0-9]+\.[0-9]+$'; then npm --no-git-tag-version version $NBL_SERVICE_TAG; else npm --no-git-tag-ve
RUN npm run build  →  Generación del build de la app, inluye transpilacion de Typescript a Javascript
## Remove devDependencies after build
ENV PUPPETEER_SKIP_CHROMIUM_DOWNLOAD="true"
RUN npm ci --omit=dev  →  Descarga de dependencias, esta vez incluyendo solo las necesarias para producción
```

```
# Create final image
FROM node:18.18-alpine3.18 as final  →  Partiendo de una imagen limpia
RUN apk upgrade --no-cache --available \  →  Instalación de Chromium y fuentes
    && apk add --no-cache \
      chromium \
      ttf-freefont \
      font-noto-emoji \
    && apk add --no-cache \
      --repository=https://dl-cdn.alpinelinux.org/alpine/edge/community \
      font-wqy-zenhei

## Copy dumb-init
COPY --from=build /usr/bin/dumb-init /usr/bin/dumb-init
## Create nebula user
RUN addgroup --gid 2001 --system nebula-group && \
    adduser --uid 2000 --ingroup nebula-group --system nebula

## Increase the 8K bytes default header size
ENV NODE_OPTIONS=--max-http-header-size=16000

## Create necessary folders
RUN mkdir -p /home/nebula/app/node_modules
RUN mkdir -p /home/nebula/app/public

## Copy content to the app, config and dependencies folders
COPY --from=build /usr/src/node_modules/ /home/nebula/app/node_modules
COPY --from=build /usr/src/package.json /home/nebula/app
COPY --from=build /usr/src/config/ /home/nebula/app/config
COPY --from=build /usr/src/entrypoint.sh /home/nebula/app/entrypoint.sh
COPY --from=build /usr/src/build/ /home/nebula/app/
COPY --from=build /usr/src/public/ /home/nebula/app/public
    →  Copiamos desde la capa de build la app y sus dependencias

WORKDIR /home/nebula/app/
## Set proper permissions and flags
RUN chown -R nebula:nebula-group /home/nebula/app  →  Ownership y permisos sobre los archivos de la app
USER 2000
RUN chmod +x /home/nebula/app/entrypoint.sh
EXPOSE 9290:9290

ENTRYPOINT ["/usr/bin/dumb-init", "--"]
CMD ["/bin/sh", "./entrypoint.sh"]  →  Script que inicia la app
```

Recursos del contenedor

```
name: prod-5
region: us-east-1
service: performance-comparator
containers:
  - resources:
      cpu:          →  Cantidad de CPU
        request: 1
        limit: 2
      ram:
        request: 1     →  Cantidad de memoria principal
        limit: 2
        unit: Gi
      disk:
        request: 25    →  Tamaño de disco
        limit: 40
        unit: Gi
desired_size: 2     →  Cantidad de réplicas
environment: prod
health_check:
  initial_delay_seconds: 180
  timeout_seconds: 3
termination_grace_period_seconds: 30
traffic_rollout:
  percentage: 100
  waiting_seconds: 10
metric_config:
  runtime: javascript
  custom_metric: false
```

Pipeline de Integración Continua (CI) / Deployment Continuo (CD)




```
name: Release PROD version
on:
  workflow_dispatch:            → Se ejecuta manualmente
    inputs:
      version: ...
      deployMark: ...           → Parámetros para ejecutar
env: ...
jobs:
  prod:
    name: Release Prod
    runs-on: [self-hosted]
    continue-on-error: false
    outputs: ...
    steps:
      - name: Checkout ...               → Descarga el código fuente
      - name: Link Custom Actions ...
      - name: Get Branch ...
      - name: Check diff con main ...
      - name: Check tag ...
      - name: Validate Tag ...           → Valida que la versión sea única
      - name: Extract Jira Id ...
      - name: Get Pull request ...
      - name: Setup Node ...             → Configura Node para instalar dependencias y validar que se pueda buildear
      - name: Install dependencies ...
      - name: Build local ...
      - name: Release info ...
      - name: Create tag ...             → Se crea el tag en Github y se genera un Release
      - name: Create Release ...
      - name: Build Image ...            → Construye la imagen final con Docker y la sube al repositorio privado de imágenes
      - name: Set build status ...
      - name: Deploy on Sandbox ...
      - name: Deploy on PROD ...         → Deploya en un ambiente de pruebas y producción
```

Resultados obtenidos

Como mencionamos anteriormente, buscamos medir mejoras que se observen al modificar el comportamiento de la página de distintas maneras. Concretamente, estudiamos la performance reportada por *Performance Comparator* en los siguientes escenarios:

- Limitar la disponibilidad a tres (3) hoteles.
- Solicitar la disponibilidad usual, pero solo dibujar tres (3) hoteles.
- No dibujar la galería de imágenes de cada hotel, optando por una imagen default.
- No dibujar banners.

Para obtener el reporte de Lighthouse sobre nuestra aplicación con cada variante, utilizamos una serie de headers para cada prueba, que incluimos en *Performance Comparator*. Por ejemplo bastó con agregar el header **x-aloha-debug-request-limited-clusters = true** para la prueba planteada en el primer escenario

¿Cómo medimos?

Dado que las métricas obtenidas pueden variar entre cada ejecución, porque dependen no solo de la aplicación a ser medida sino del estado de la máquina que ejecuta *Performance Comparator*, cómo puede ser el uso de CPU por parte del sistema operativo, variaciones en tiempos de red, etc. Es necesario tomar varias muestras para aproximar lo mejor posible el valor

Sin embargo, utilizamos un método estadístico estándar para obtener el número de muestras mínimo que nos de un grado de confianza aceptable de que la estimación obtenida corresponde a las métricas reales. Para ello vamos a utilizar como referencia la métrica de **Score**. El proceso para esto es el siguiente:

1. Obtener un conjunto inicial de muestras y calcular el desvío estándar muestral, que notaremos **“σ”**. Intuitivamente, esto nos dará una idea preliminar de la varianza de los datos, y servirá para determinar un número de muestras óptimo.
2. Definir un intervalo de confianza y computar su **Z-Score**. Por ejemplo, si queremos tener una confianza del 95% de que la estimación del Score corresponde al real, obtenemos un Z-Score de 1.96. Este número es necesario para una fórmula que usaremos al final.
3. Determinar un margen de error **“E”** aceptable. Por ejemplo, si queremos que la estimación del Score no difiera en más de 1 punto del real, nuestro margen de error es 1.

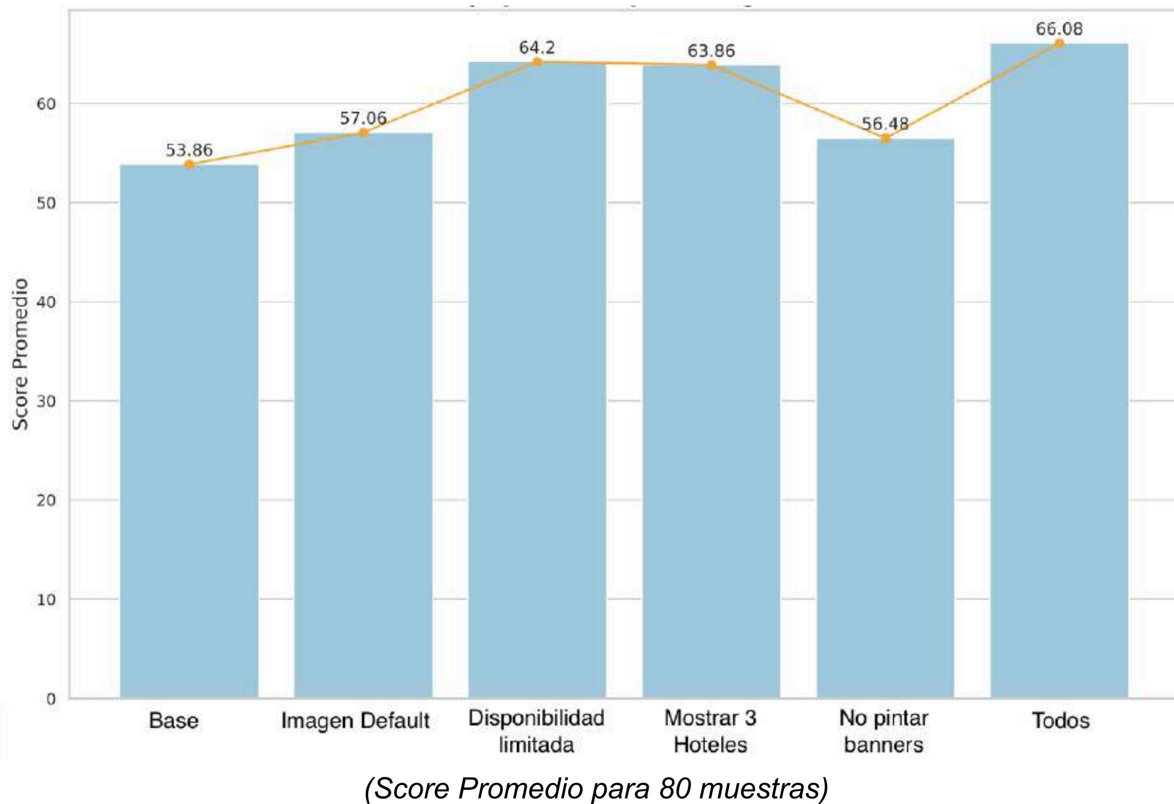
Finalmente, dado un desvío estándar muestral **“σ”**, un Z-Score **“Z”**, y un margen de error **“E”**, la cantidad de muestras óptima está dada por:

$$n = \left(\frac{Z \, . \, \sigma}{E}\right)^2$$

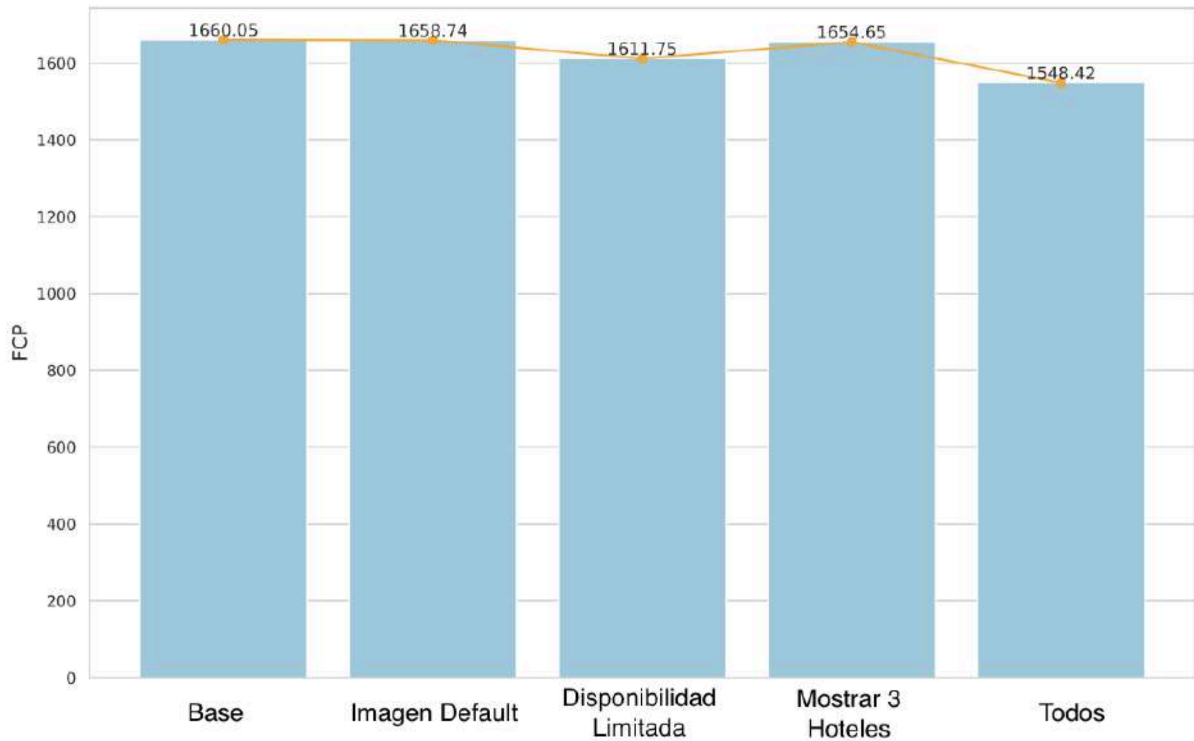
Con un intervalo de confianza del 95 %, un margen de error de 1, y el desvío estándar muestral obtenido luego de 40 corridas preliminares, obtuvimos un valor de $n = 76$, que redondeamos a 80.

Finalmente, teniendo un número de muestras que ofrecen confianza en que los resultados obtenidos son representativos de la realidad, utilizamos *Performance Comparator* para obtener las métricas para cada escenario planteado.
Adicionalmente, medimos las métricas sin ningún header para obtener una cifra base de referencia para poder hacer una comparación.

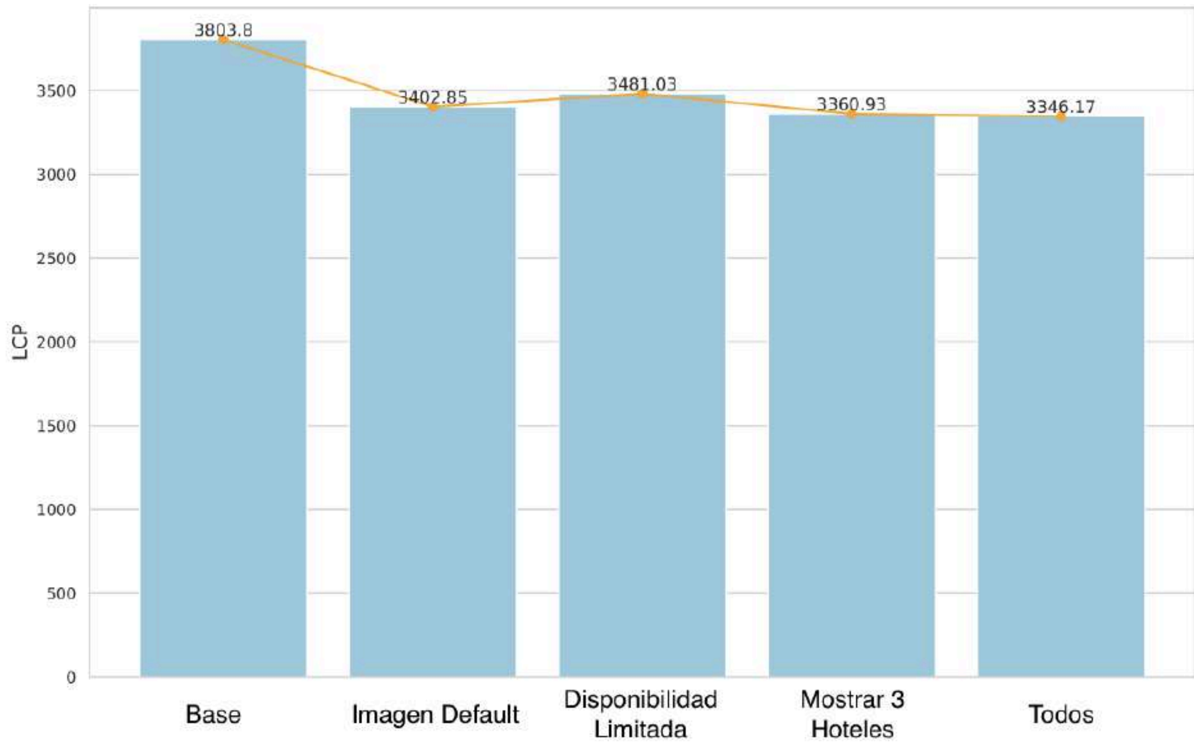
Aquí los gráficos con los resultados.



(Score Promedio para 80 muestras)



(FCP Promedio para 80 muestras)



(LCP Promedio para 80 muestras)

Luego de las pruebas con el Score, decidimos eliminar de la prueba el caso de “No mostrar banners” dado que no nos dio buen resultado y además no era un cambio que realmente pudiéramos hacer sobre el sitio de una forma sencilla.

Para el resto de los escenarios, se ve que todos los ajustes tienen un impacto observable y positivo. En particular, la mayor ventaja se presenta al limitar la cantidad de hoteles dibujados.

En este caso, se puede ver una similitud entre limitar la cantidad devuelta por la API y no limitar pero dibujar pocos. Dado que ambos escenarios tienen el efecto de mostrar una cantidad de hoteles limitada, y que pedir la cantidad usual no parece empeorar el rendimiento significativamente, podemos concluir que el escenario que nos aporta una oportunidad de mejora es el renderizado de una menor cantidad de resultados.

Líneas de trabajo futuros

Una observación en relación a la prueba fue el gran impacto que tuvo la utilización de una imagen por default en cuanto a la métrica de LCP. Dado que este desarrollo tiene un costo bajo comparado con el resto, fue lo que decidimos agregar a nuestro backlog como mejora a corto plazo.

Luego tenemos dos oportunidades más de mejora en este aspecto planeadas para el futuro

- **Actualización de Angular a su versión 16**: El frontend de Alojamientos utiliza Angular 15 como framework, la versión 16 incorpora cambios en la forma de renderizar componentes y promete una mejora de performance con respecto a su versión anterior, debido a dependencias compartidas con otros equipos, la actualización no es algo que se pueda hacer de forma trivial, estamos planificando su ejecución en lo antes posible
- **Llamado condicional a la "Full"**: Esta iniciativa tiene un doble objetivo, por un lado es reducir costos operativos y como efecto secundario creemos que puede mejorar la performance del sitio. A continuación voy a hacer un resumen de qué se trata brindando un poco de contexto.

Llamado condicional a la Full

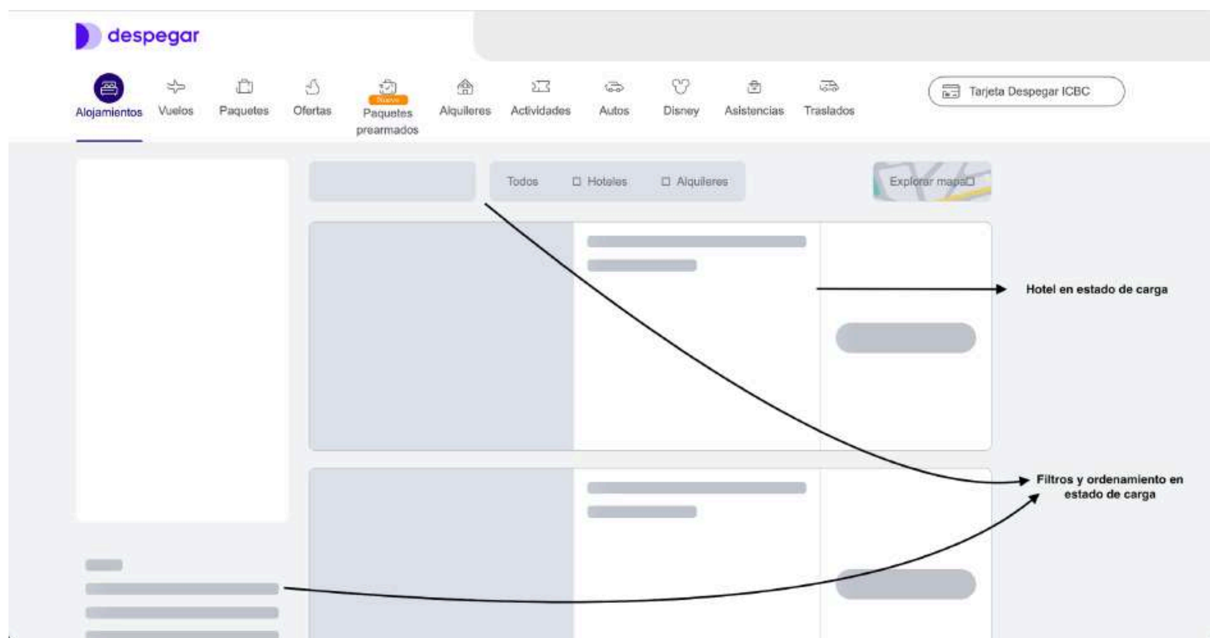
Hoy en día por cada búsqueda del usuario, la API de Alojamientos hace 2 llamados a la API de Disponibilidad de Alojamientos, identificadas con el nombre de:

- **Quick**: Corresponde a la disponibilidad de 42 Alojamientos, no genera información para filtrar, con el objetivo de mostrar al usuario resultados rápidamente.
- **Full**: Responde con toda la disponibilidad de Alojamientos para el destino, fechas y pasajeros solicitados, junto con información para filtrar, ordenar y paginar esos resultados

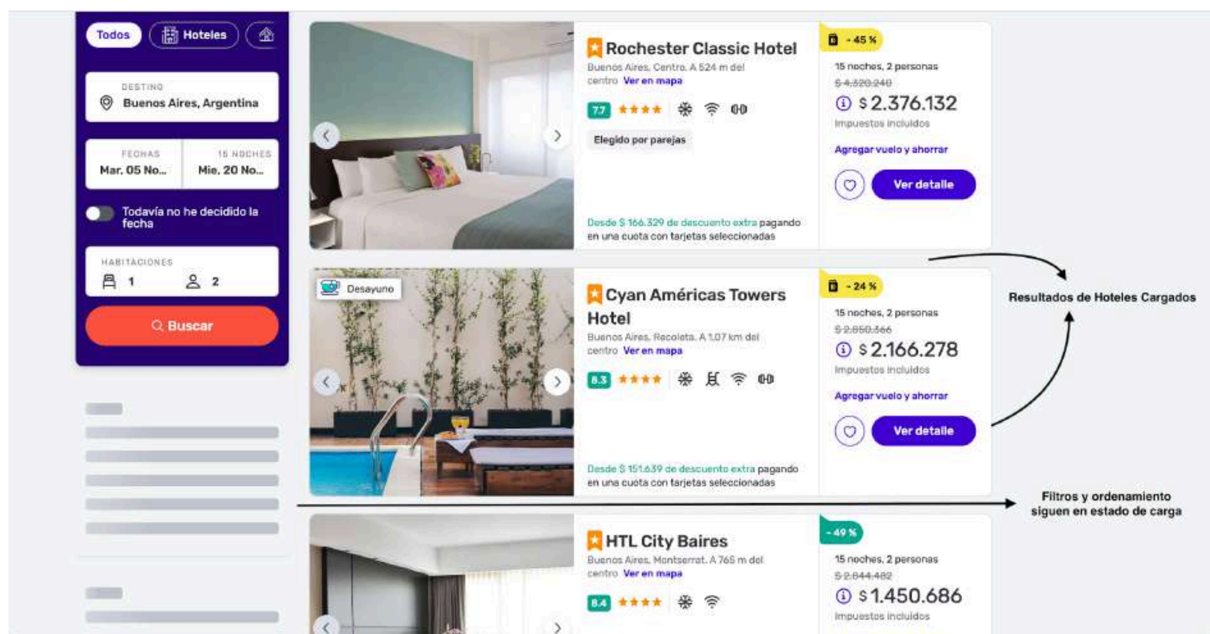
En el 3er cuarto del 2023 se ejecutó un **AB Test** en la vista mobile, el cual consistió en que al momento del usuario interactuar por primera vez con los elementos de ordenamiento, paginación y filtros, en vez de mostrar el resultado de la interacción instantáneamente se lo hizo esperar entre 5 y 8 segundos en un estado de carga, con el fin de simular lo que ocurriría si sólo hubiésemos solicitado la Quick.

Esta prueba arrojó como resultado que el **50%** de nuestros usuarios **nunca interactuó con estos componentes**, es decir no tuvo la necesidad de consultar la Full y con la Quick le fue suficiente.
Dentro del universo que si interactuó con estos componentes no se vieron afectadas las métricas de **conversión** o de **detail success**. Lo que nos dejó como conclusión que podíamos ahorrarnos este 50% de las llamadas a la **Full**.

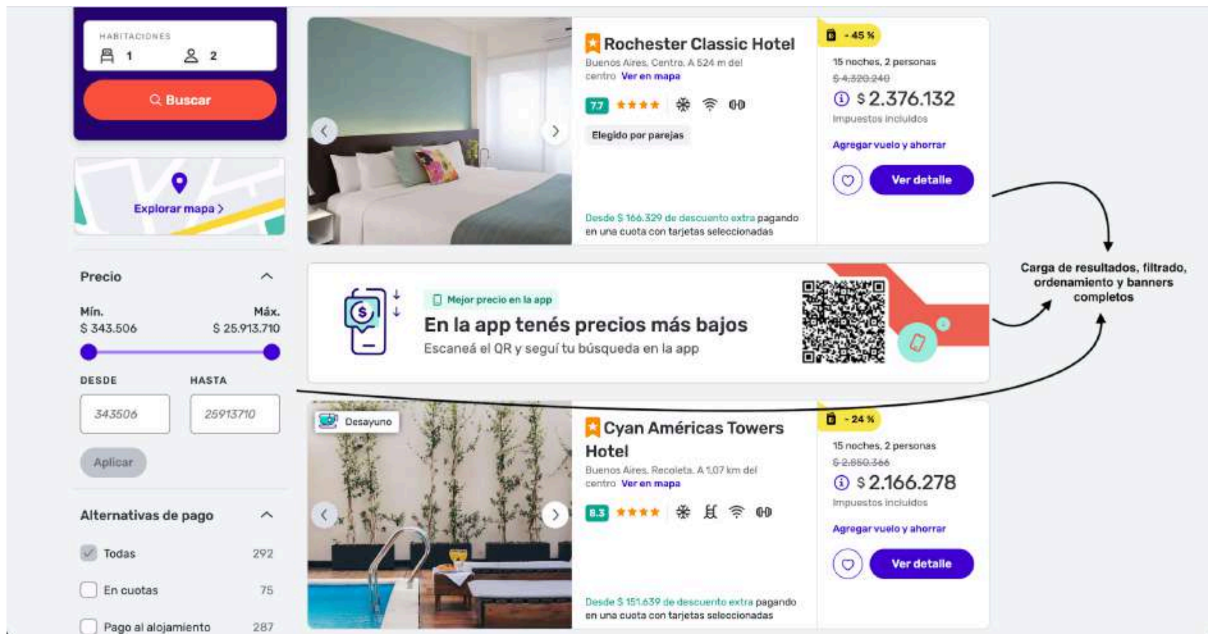
Este ahorro puede traducirse en mejoras respecto al costo en tiempos de carga inicial de la aplicación frontend de Alojamientos sobretodo si llevamos esta solución al sitio en su vista Desktop, pasando de esta secuencia de renderizado



(Fase inicial: Todo el sitio en modo de carga)



(Fase Quick: Se dibujan los resultados y la caja de búsqueda)



(Fase Full: El sitio terminó de cargar completamente)

Este desarrollo planificado, eliminaría la Fase Quick, pasando directamente a la Fase Full, pero utilizando la disponibilidad de la Quick.
Para esto se va a hacer el desarrollo para poder dibujar los filtros en esta fase, evitando el parpadeo y movimiento de elementos de la pantalla por los cambios y dejando al usuario disponible el sitio para su interacción más rápida.

Otra optimización importante de esta solución es la posibilidad de reducir un 50% los costos que tenemos al consultar disponibilidad hacia la API de proveedores externos, los cuales nos establecen una cuota de consultas mensuales, donde el excedente se debe abonar aparte.

Conclusiones

Ya habiendo hecho un recorrido de varios meses, empezando las primeras etapas de estudio de qué queríamos medir, qué propiedades tienen que tener estas métricas, para luego empezar una etapa de análisis de la solución, su posterior implementación, puesta en producción y validación. Podemos decir que nos encontramos con una manera de medir y monitorear a lo largo del tiempo como se comportan los flujos más importantes de nuestro sitio.

Permitiendo identificar fluctuaciones, encontrando la causa y corrigiendo para mantener esta métrica en los límites que proponemos como aceptables. También nos encontramos que más allá de que *Performance Comparator*, en su inicio tuvo como objetivo hacer una comparación con los sitios de la competencia, este rasgo no fue tan relevante como creímos. A la hora de comparar los sitios, no es tan trivial ni tan directa la comparación, pero nos aporta un grado de confianza al ver que tán lejos o cerca estamos de los números de la competencia, pero resultó no ser tan relevante como creímos.

Hoy *Performance Comparator* hace honor a su nombre en una comparación introspectiva, sirviendo como comparador de nosotros mismos en el pasado, para ayudarnos a mantener nuestro nivel y plantear objetivos a largo plazo que nos permitan mejorar.
A la vista de este trabajo está esto, donde los análisis que se hicieron fueron centrados en nuestros sistemas, encontrando puntos de mejora y estableciendo un norte claro hacia dónde asignar recursos.

Nos permitió cuestionar el *status quo* volviendo a analizar porqué hacemos las cosas como las hacemos, buscando formas de hacer lo mismo de manera más eficiente, ver qué soluciones que antes no eran posibles, por limitaciones del momento, ya no lo son y pensar en cambios de arquitectura que nos permitan llevar mayor calidad a nuestros clientes y bajar nuestros costos de mantenimiento de soluciones complejas que ya no tienen un sentido claro o incluso reduciendo costos en interacciones innecesarias con proveedores externos.

En la forma que está expuesto este trabajo da la sensación de una linealidad al momento del desarrollo de la herramienta, lo cual está alejado de la realidad, donde fue en un trabajo iterativo donde fuimos sumando funcionalidades, haciendo pruebas y validando resultados a medida que veíamos que la herramienta estaba siendo realmente útil.
Hubo momentos de cambios de prioridades donde se pausó su desarrollo, pero terminamos encontrando el tiempo para dedicarle, sabiendo que nos otorgaba un valor agregado.

Glosario alfabético

1. **A/B Test**: Método de pruebas en el que se comparan dos versiones de una aplicación o página web para determinar cuál rinde mejor.
2. **Angular**: Framework de desarrollo web basado en TypeScript para crear aplicaciones de una sola página (SPA).
3. **Backlog**: Lista de tareas pendientes que deben completarse en un proyecto, generalmente en un entorno ágil.
4. **Business Intelligence (BI)**: Procesos y tecnologías que transforman datos en información útil para la toma de decisiones estratégicas.
5. **Chromium**: Proyecto de navegador web de código abierto que es la base de muchos navegadores, incluyendo Google Chrome.
6. **Container**: Entidad virtualizada que agrupa una aplicación y sus dependencias, asegurando que se ejecute consistentemente en diferentes entornos.
7. **Conversión**: Medida de cuántos usuarios realizan una acción deseada, como completar una compra o registrarse en un servicio.
8. **Cron**: Herramienta que permite programar tareas automáticas a intervalos regulares.
9. **Deploy**: Proceso de colocar una aplicación o sistema en un entorno de producción donde puede ser utilizada por los usuarios.
10. **Deployment continuo**: Práctica en la que las actualizaciones del código se implementan automáticamente en producción después de pasar pruebas.
11. **Detail success**: Métrica que mide cuántos usuarios, en relación a la cantidad total que navegaron los resultados, hacen clic en un resultado y acceden a la página de detalle correspondiente

12. **DevTools**: Herramientas de desarrollo incluidas en navegadores como Chrome para depurar y optimizar aplicaciones web.
13. **Docker**: Plataforma que permite crear, ejecutar y administrar contenedores de software.
14. **Express.js**: Framework web minimalista para Node.js, utilizado para construir aplicaciones web y API de manera rápida y sencilla.
15. **Github Actions**: Herramienta de automatización de GitHub que permite crear flujos de trabajo para tareas como la integración continua y el despliegue.
16. **Google Lighthouse**: Herramienta automatizada de Google para medir el rendimiento, accesibilidad, SEO y más, de páginas web.
17. **IATA**: Asociación Internacional de Transporte Aéreo, conocida por establecer códigos de aeropuertos y aerolíneas a nivel global.
18. **Imagen**: En Docker, un paquete completo que incluye el código y las dependencias necesarias para ejecutar una aplicación.
19. **Integración continua**: Práctica de fusionar regularmente el código de todos los desarrolladores en un proyecto, generalmente acompañado de pruebas automáticas.
20. **Kubernetes**: Sistema de orquestación de contenedores que automatiza el despliegue, escalado y gestión de aplicaciones en contenedores.
21. **Local storage**: Almacenamiento web del lado del cliente que permite guardar datos localmente en el navegador.
22. **Lock**: Mecanismo utilizado para garantizar que solo un proceso tenga acceso a un recurso compartido a la vez.
23. **Metabase**: Herramienta de código abierto para análisis de datos y creación de dashboards interactivos.
24. **New Relic**: Plataforma de monitoreo y observabilidad para aplicaciones y servicios.
25. **NRQL**: Lenguaje de consulta de New Relic, utilizado para realizar análisis personalizados sobre los datos monitorizados.
26. **Node.js**: Entorno de ejecución para JavaScript del lado del servidor, construido sobre el motor V8 de Chrome.
27. **Pipeline**: Conjunto de pasos automatizados que llevan el código desde su desarrollo hasta su despliegue en producción.
28. **Preheat**: Proceso de precalentamiento en sistemas distribuidos, usado para optimizar el rendimiento antes de realizar una tarea intensiva.
29. **Puppeteer**: Biblioteca Node.js que proporciona una API para controlar navegadores web de manera automatizada.
30. **Renderizado**: Proceso de convertir el código HTML, CSS y JavaScript en una página web visible para el usuario.
31. **Service worker**: Script que el navegador ejecuta en segundo plano para manejar funcionalidades como caché o notificaciones push, incluso cuando la página no está abierta.
32. **Skeleton**: Diseño provisional que se muestra mientras una página o aplicación web carga sus datos, proporcionando una representación visual del contenido final para mejorar la experiencia del usuario.
33. **Tópico**: En Apache Kafka, un canal en el cual se publican y se leen mensajes. Los productores envían mensajes a los tópicos y los consumidores los leen de ellos.
34. **TypeScript**: Lenguaje de programación basado en JavaScript que añade tipado estático opcional y otras características avanzadas.
35. **Viewport**: Área visible de una página web en la pantalla de un dispositivo.

Referencias bibliográficas

Web

Métrica INP y valores de referencia: https://web.dev/articles/inp?hl=es-419
Métrica TTFB y valores de referencia: https://web.dev/articles/ttfb?hl=es-419
Métrica FCP y valores de referencia: https://web.dev/articles/fcp?hl=es-419
Métrica LCP y valores de referencia: https://web.dev/articles/lcp?hl=es-419
Métrica TBT y valores de referencia: https://web.dev/articles/tbt?hl=es-419
Métrica CLS y valores de referencia: https://web.dev/articles/cls?hl=es-419
Métrica Speed index y valores de referencia:
https://developer.chrome.com/docs/lighthouse/performance/speed-index?hl=es-419
Métrica Domsize y valores de referencia:
https://developer.chrome.com/docs/lighthouse/performance/dom-size?hl=es-419
Documentación de Google Lighthouse (Node Module):
https://github.com/GoogleChrome/lighthouse/blob/main/docs/configuration.md
Documentación Puppeteer: https://pptr.dev/guides/browser-management
Documentación Metabase: https://www.metabase.com/docs/latest/
Documentación Node.js 18: https://nodejs.org/docs/latest-v18.x/api/index.html
Documentación Express.js: https://expressjs.com/en/4x/api.html
Lucidchart para la creación de UMLs: https://www.lucidchart.com/pages/es/landing

Libros

Cohen, J. (1988). Statistical power analysis for the behavioral sciences (2nd ed.). Lawrence Erlbaum Associates.