

Universidad ORT de Montevideo  
Facultad de Ingeniería.

Obligatorio 2  
Arquitectura de Software en la Práctica

Facundo Arancet - 210696  
Marcel Cohen - 212426  
Eusebio Duran - 202741

Código fuente disponible en:  
<https://github.com/ArqSoftPractica/Coupons>

Aplicación desplegada disponible en:  
<https://coupons-ui.herokuapp.com>

Noviembre, 2019

# Índice

<b>Índice</b>	<b>1</b>
<b>Descripción de la arquitectura</b>	<b>2</b>
Particionamiento en microservicios	2
Justificación de particionamiento	3
Vista de componentes y conectores	5
Decisiones de diseño	5
Vistas de asignación	7
Vista de despliegue	7
Decisiones de diseño	7
Diagramas de secuencia	9
Autenticación y autorización	9
Creación de promoción	10
Evaluación de promoción	10
<b>Justificaciones de diseño</b>	<b>11</b>
RNF1. Performance	11
RNF2. Confiabilidad y disponibilidad	11
RNF3. Configuración y manejo de secretos	12
RNF4. Autenticación y autorización	12
RNF5. Seguridad	13
RNF6. Identificación de fallas	13
RNF8. Monitoriabilidad	13
RNF9. Independencia de aplicaciones	13
<b>SDK de evaluaciones</b>	<b>14</b>
Guía de uso	14
<b>Pruebas del sistema</b>	<b>16</b>
Pruebas unitarias y de integración	16
Pruebas de carga	17
Resultados de JMeter	18
Latencia de los pedidos	18
Metricas generales	19
<b>Descripción del proceso de deployment</b>	<b>20</b>
Coupons-promotions	20
Coupons-auth	21
Coupons-ui	22
Coupons-gateway	23
Coupons-reports	24
Coupons-mailing	25
<b>Patrones de Microservicios aplicados</b>	<b>26</b>
<b>The Twelve Factors</b>	<b>30</b>

# Descripción de la arquitectura

## Particionamiento en microservicios

En esta entrega, se pidió migrar la aplicación de la primera entrega a una arquitectura de microservicios. Se optó descomponer el sistema en 6 componentes:

- Una aplicación front-end. Que fue una adaptación de la aplicación Rails MVC que se entregó en el primer obligatorio.  
El código fuente se encuentra en: <https://github.com/ArqSoftPractica/Coupons>  
La aplicación desplegada se encuentra en: <https://coupons-ui.herokuapp.com>
- Un API Gateway, que es la implementación del patrón *External API*. La versión inicial fue construida en NodeJS.  
El código fuente se encuentra en:  
<https://github.com/ArqSoftPractica/coupons-api-gateway>  
Dicha versión fue descartada por el overhead que introducía, y se cambió por una implementación en Kong.  
La aplicación desplegada se encuentra en:  
<https://coupons-gateway.herokuapp.com>
- Un microservicio Auth Server, que contiene todo lo relativo a los usuarios, autenticación, y también las organizaciones. Fue construido en NodeJS + Typescript, utilizando una base de datos MongoDB.  
El código fuente se encuentra en  
<https://github.com/ArqSoftPractica/coupons-auth>  
La aplicación desplegada se encuentra en:  
<https://coupons-auth.herokuapp.com>  
La documentación de la API rest de este servicio si encuentra en:  
<https://documenter.getpostman.com/view/9634108/SW7gU5Lp?version=latest>
- Un microservicio que contiene las promociones y se encarga de las tareas de CRUD y la evaluación de estas. Fue construido en Rust, utilizando una base de datos Postgres.  
El código fuente se encuentra en:  
<https://github.com/ArqSoftPractica/coupons-promotion> (coupons-promotions ya fue tomado por otro equipo).  
La aplicación desplegada se encuentra en:  
<https://coupons-promotions.herokuapp.com>  
La documentación de la API rest de este servicio si encuentra en:  
<https://documenter.getpostman.com/view/5322177/SW7dURNX?version=latest>

- Un microservicio de reportes, que se encarga de calcular los reportes demográficos y de uso de las promociones evaluadas. Fue construido en Rails, utilizando una base de datos Postgres. Esta aplicación fue desplegada en dos aplicaciones Heroku, por las restricciones que la plataforma para los planes gratuitos. La aplicación tiene dos procesos principales: Uno web, al cual se acceden a los recursos y un conjunto de background workers de *Sneakers*, que reciben los mensajes de las *message-queues* y actualizan la base de datos. El problema que se encontró el equipo, fue que Heroku (en el plan gratuito) “pone a dormir” a toda aplicación que tenga al menos un contenedor web, que no haya recibido tráfico en 30 minutos, esto implica que ambos contenedores en la aplicación (web y worker) pasan al estado *idle*. Esto es un problema, ya que se necesita que el worker esté siempre despierto, para poder actualizar la base de datos y mantener la consistencia, por lo cual se desplegó como aplicación separada.

En resumen, serian dos aplicaciones desplegadas por separado que comparten la misma base de datos, pero consideramos que ambas son parte de la misma aplicación, no se tuvo intención de romper el patrón *Single Database per-Service*, que se ha aplicado en este sistema (Véase sección: Patrones de Microservicios Aplicados).

El código fuente se encuentra en:

<https://github.com/ArqSoftPractica/coupons-reports>

La aplicación web desplegada se encuentra en:

<https://coupons-reports.herokuapp.com>

La aplicación worker mencionada se encuentra desplegada en:

<https://coupons-reports-worker.herokuapp.com>

La documentación de la API rest de este servicio si encuentra en:

<https://documenter.getpostman.com/view/6603279/SW7gU5GK?version=latest#b6cc09e1-07e0-48dd-ae6-8121be400e74>

- Un microservicio que se encarga del envío y la programación de mails, de invitación y de expiración de promociones. Fue construido en Rails y utiliza Redis como base de datos en memoria.

El código fuente se encuentra en:

<https://github.com/ArqSoftPractica/coupons-mailing>

La aplicación desplegada se encuentra en:

<https://coupons-mailing.herokuapp.com/>

## Justificación de particionamiento

Inicialmente, la evaluación y actualización de reportes sobre promociones eran dos actividades que se realizaban en conjunto, esto restaba performance al sistema, ya que ambas son operaciones costosas. Tras un análisis, el equipo observó que la evaluación es

una operación que requiere una respuesta inmediata y el cálculo de reporte es una tarea que se puede realizar asincrónicamente, y además, el tener los reportes pre-calculados mejoraría considerablemente la performance a la hora de acceder a estos. Entonces se tomó la decisión de construir un microservicio que se encargue de guardar y actualizar los reportes, el cual recibiría mensajes asincrónicos cada vez que se evaluará una promoción. De ahí surgió *coupons-reports*.

También se decidió que las actividades de CRUD y evaluación de promociones eran altamente cohesivas, y no estaban tan relacionadas con el resto de las operaciones de la aplicación monolítica, adicionalmente, la evaluación de las promociones requería un alto grado de performance. Por ende, se decidió separar otro microservicio de promociones. El equipo también observó que las claves de aplicación estaban fuertemente relacionadas con la evaluación de promociones, por lo cual se decidió incluirlas en dicho microservicio. Se considera que idealmente, tendría que haber un microservicio por cada entidad del negocio, pero debido a la latencia provocada por tantos microservicios, el costo de performance no amerita tanta granularidad. El equipo intentó realizar un despliegue en una red privada para reducir la latencia, que por falta de recursos monetarios no se pudo implementar. Entonces, se incluyeron más de una entidad de negocio en un mismo microservicio. Como dicho microservicio tenía requerimientos de performance, entonces se aprovechó para implementarlo en una tecnología performante de por sí, Rust. De ahí surgió *coupons-promotions*.

Luego de dichas particiones, quedaban los usuarios y organizaciones junto con las vistas, en el monolito. Se decidió que incrementó considerablemente la mantenibilidad, tener la vista separada del resto de la lógica de negocio, de esta forma, también se podrían tener múltiples clientes distintos de la misma aplicación, potenciando el reuso. Adicionalmente, usuarios y organizaciones están fuertemente relacionados, son componentes de autenticación. Por ende, se separó otro microservicio con usuarios y organizaciones, en el cual se realizaría el registro de usuarios, organizaciones, y también el *log-in* en la aplicación. Dicha decisión dio lugar a *coupons-auth*.

El microservicio *coupons-mailing*, surgió de la observación de que existen más de una funcionalidad que requieren el envío de emails: Invitaciones de usuarios, y expiración de promociones. Para un buen agendamiento y envío de emails, es necesario apoyarse en servicios externos a la aplicación, como sidekiq, que a su vez se apoya en un servidor Redis. Se creyó una buena idea, separar las responsabilidades de envío de emails en un microservicio independiente, con los recursos mencionados.

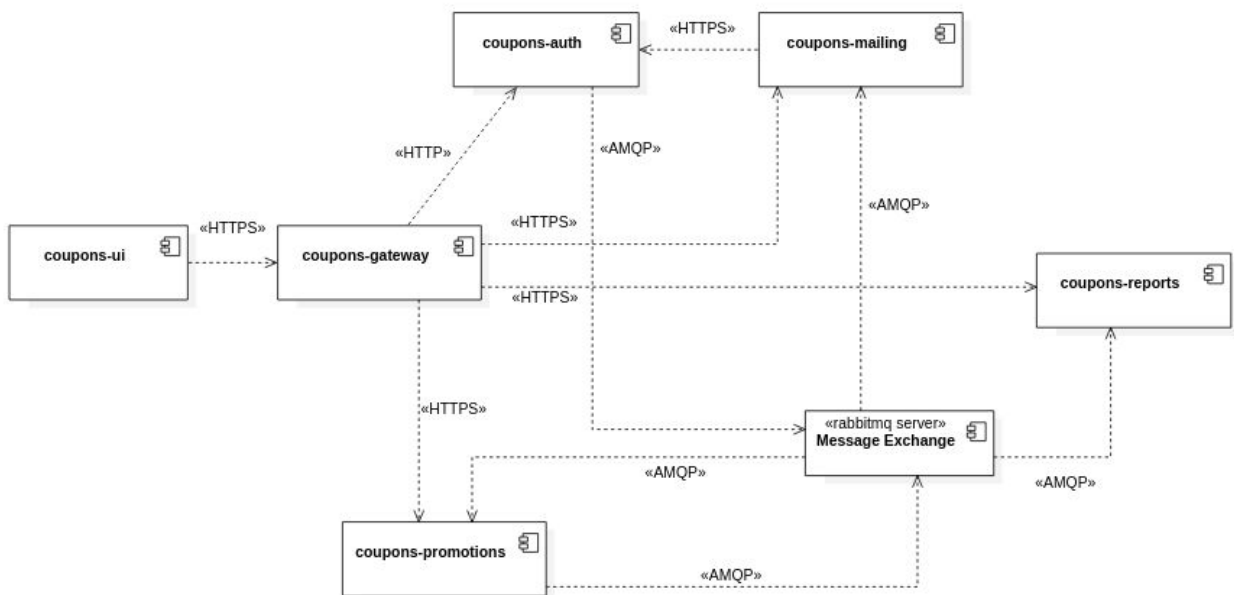
Lo que quedó del monolito original, solo contenía vistas, entonces se adaptó a una aplicación FrontEnd, *coupons-ui*.

Finalmente, teniendo ya el sistema particionado en microservicios, y considerando la idea de que podría a futuro haber más clientes (Ej. Mobile), se consideró útil tener una interfaz unificada que ocultara los microservicios, ya que las rutas podrían cambiar, o mismo su topología, si se sigue particionando. Entonces se aplicó el patrón de microservicios *External API*, creando un API-Gateway. Al principio se implementó en NodeJS, utilizando la librería *httpProxy*, pero al terminarla, se observó que era complejo y poco performante, por

lo cual, se abandonó el proyecto y se configuró uno utilizando Kong. El nombre de la aplicación creada es *coupons-gateway*.

## Vista de componentes y conectores

En el diagrama a continuación, se puede ver todos los microservicios y cómo interactúan entre ellos.



## Decisiones de diseño

Como se puede ver en el diagrama, se utilizaron dos formas de comunicación entre microservicios: una sincrónica, HTTP, y otra asincrónica AMQP (Advanced Message Queueing Protocol).

Se busco mantener la comunicación entre el cliente y los servicios sincrónica mientras que la comunicación entre los microservicios asincrónica. Ya que las llamadas sincrónicas entre microservicios rompen con la independencia entre microservicios y agregan más latencia innecesaria.

Por ejemplo, en el caso que el servicio de promociones se comunicara sincrónicamente con el servicio de usuarios, para validar que existe la organización de la promoción, si el servicio de usuarios dejase de responder, el servicio de promociones tampoco podría funcionar, lo que viola la independencia entre servicios. También a la latencia entre el cliente y el servicio de promociones se le agregaría la latencia entre las promociones y los usuarios lo que significa un impacto grande en la velocidad de respuesta.

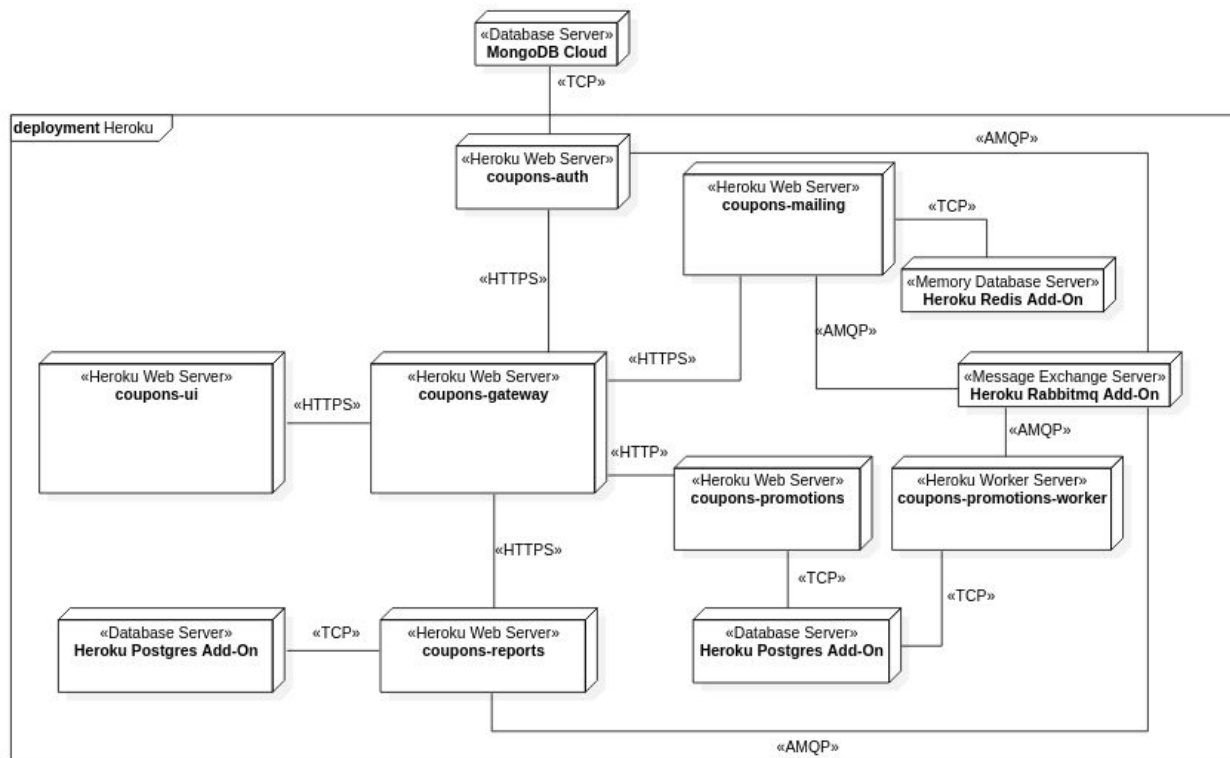
En cambio en nuestro sistema, el servicio de promociones se comunica asincrónicamente con el servicio de usuarios por lo que no se agrega latencia y en el caso que el servicio de los usuarios se caiga las promociones siguen funcionando sin problema alguno.

El uso mensajes y publicación de eventos, hacen que el sistema siga un modelo de consistencia eventual. Esto quiere decir que en un momento dado, los datos del sistema no son consistentes, pero convergen en algún momento. Este fue un sacrificio hecho para poder construir un sistema distribuido en varios componentes, se tuvo que “relajar” algunas reglas de negocio. Por ejemplo, al evaluar una promoción, se emite el evento de promoción evaluada, el servicio de reportes recibe el mensaje y actualiza su base de datos de reportes, esto puede tomar un tiempo, ya que el servicio puede tener muchos mensajes en cola. En estos casos se incluyeron fechas de actualización de los reportes, para brindar más confianza.

## Vistas de asignación

### Vista de despliegue

La vista a continuación muestra el diagrama general del despliegue.



### Decisiones de diseño

Como se puede ver en el diagrama, en esta ocasión se eligió Heroku como nube para hacer el despliegue de nuestras aplicaciones. Cada una de las aplicaciones está desplegada como aplicación independiente. Los microservicios se apoyan en componentes externos:

- Se utilizan servidores de bases de datos, los microservicios de reportes y promociones utilizan el que provee heroku como un add-on, en el diagrama se utilizó el mismo nodo para simplificar, pero ambos tienen bases de datos separadas. También, en el caso del servicio de autenticación, se utilizó una base de datos hosteada en la nube de MongoDB, externa a Heroku.
- El servicio de mailing utiliza el add-on de redis provisto por heroku, ya que es una aplicación worker hecha en ruby que utiliza la herramienta sidekiq para agendar envíos de mails. Dicha herramienta se apoya en redis.



- Se utilizó un servidor de mensajes de rabbitmq, también provisto por heroku. Este servidor se encarga de recibir los mensajes de eventos, y routearlos a los respectivos microservicios suscriptos a dichos eventos.

En el diagrama no aparece, por motivos de simplicidad, pero en cada nodo está corriendo un container de Docker con la respectiva aplicación. A continuación un ejemplo de la imagen de coupons-mailing desplegada en su nodo.

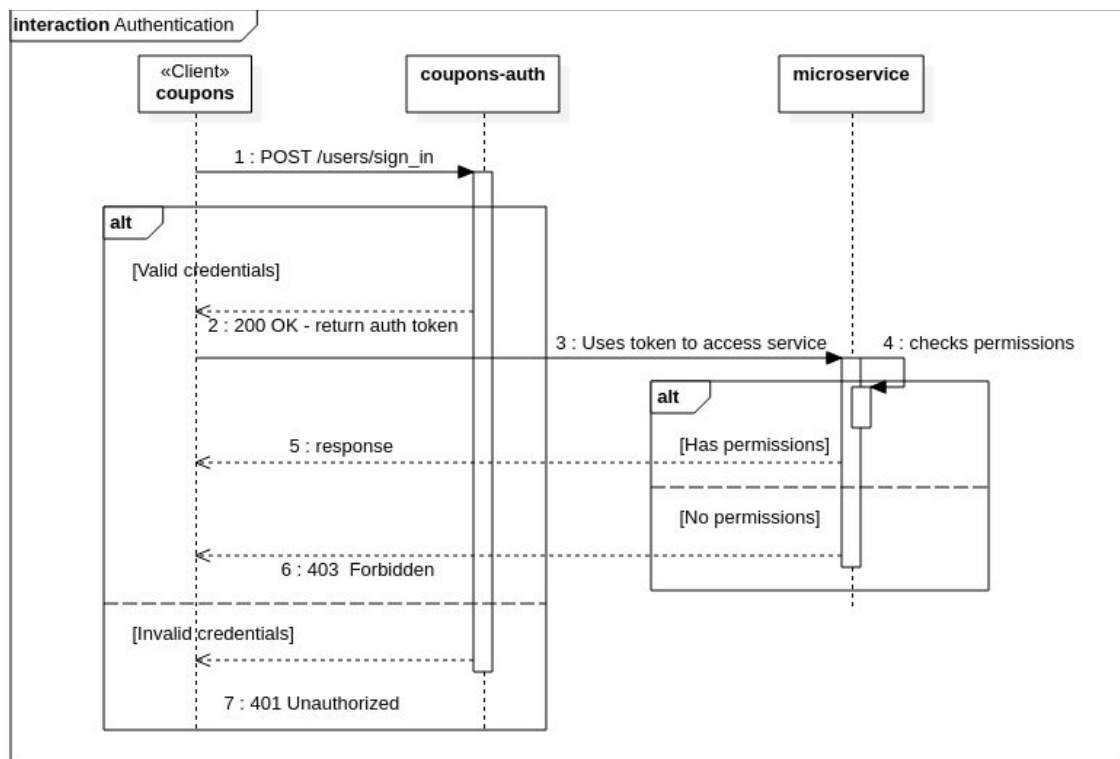


En Heroku se desplegaron dos tipos de aplicación: aplicaciones web y aplicaciones worker. Las web son las típicas aplicaciones que exponen datos a través de una interfaz, las aplicaciones worker son procesos que corren en background realizando tareas. Un ejemplo es el microservicio de mailing quien recibe mensajes de colas (Invitaciones de usuarios o promociones creadas/editadas) y envía o agenda el envío de emails. Otro ejemplo es el worker de la aplicación coupons-reports que actualiza la base de datos de reportes en base a eventos recibidos.

# Diagramas de secuencia

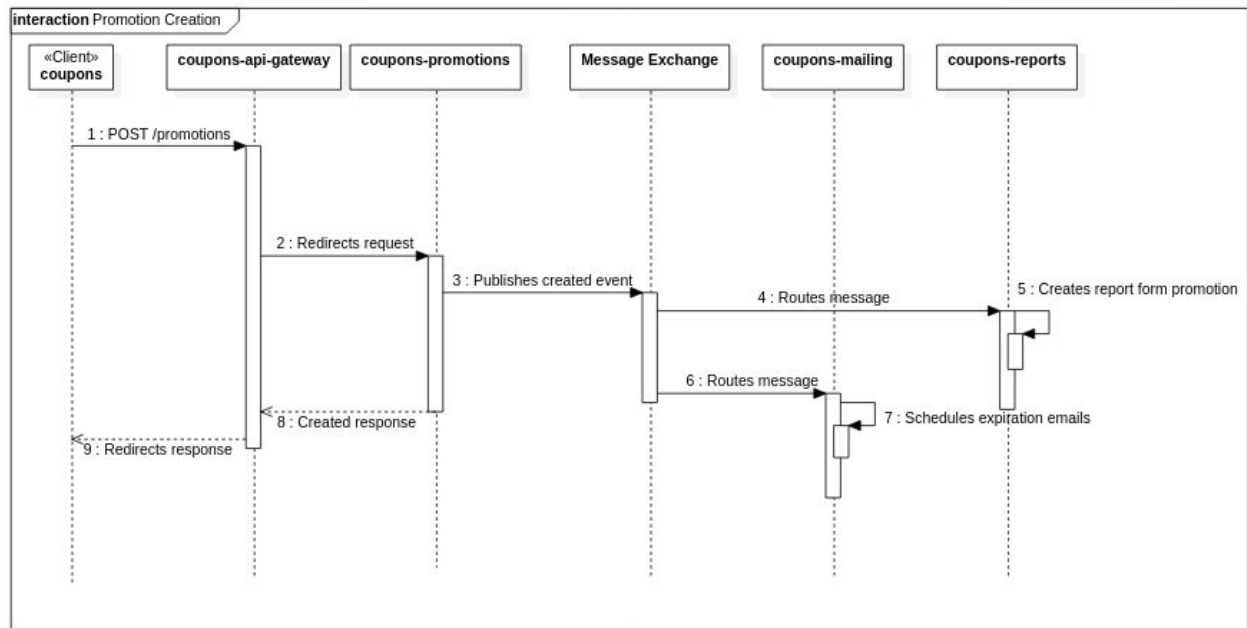
## Autenticación y autorización

Para poder interactuar con cualquier microservicio, el cliente debe autenticarse. La autenticación se hace mediante web tokens. En el diagrama a continuación se muestra el flujo de la aplicación para autenticarse y acceder a los microservicios. Para mantener la simplicidad, se omitió el api-gateway, quién es el intermediario entre el cliente (coupons) y los demás servicios, todas las llamadas pasan por el.



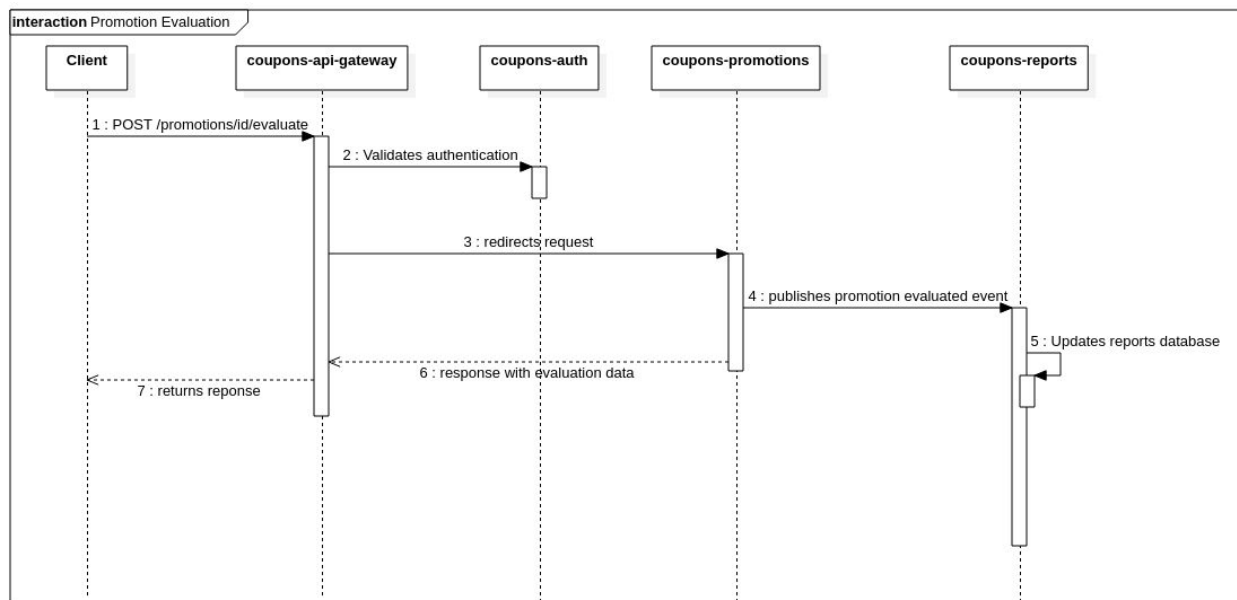
El equipo diseñó una arquitectura donde la autenticación es centralizada y la autorización es distribuida. Esto quiere decir, que existe un solo servicio que conoce que usuarios pertenecen al sistema, este servicio sería coupons-auth, que tiene el rol de Auth-Server. Los clientes se autentican (siempre a través del API Gateway) con el Auth-Server, quien les otorga una token, que contiene encriptada información relevante de los usuarios, como la organización a la que pertenecen y los permisos que tienen. Por ende, la autenticación es centralizada. Luego, cada microservicio descrypta la token (El Auth-Server y los microservicios comparten el mismo secreto), y valida que tenga los permisos para acceder al recurso o realizar la operación. Entonces, la autorización es distribuida. Para los diagramas a continuación, se asume que el cliente ya se autenticó con el auth-server, y posee una token.

## Creación de promoción



Se publican eventos en forma de mensajes para comunicar cambios de estados entre los microservicios. De esta forma, se logra desacoplar los microservicios, y se evita que utilicen llamadas sincrónicas para comunicar cambios.

## Evaluación de promoción



# Justificaciones de diseño

A continuación se explica, en detalle, las tácticas aplicadas para conseguir cada requerimiento no funcional exigido por la especificación. Los nombres de las tácticas aplicadas son tomados del libro *Software Architecture in Practice: Third Edition*, mencionado anteriormente y que fue utilizado como libro del curso.

## RNF1. Performance

Se aplicó la táctica, *Increase Resource Efficiency*, al implementar la parte del sistema que mayores requerimientos de performance tiene (Evaluación de promociones), en Rust, una tecnología de más bajo nivel, que aprovecha mejor los recursos y es considerablemente más óptima que la utilizada anteriormente. En la sección Pruebas de Carga, se pueden observar los resultados.

También, se aplicó la táctica *Increase Resources*, aumentando la cantidad de servidores que alojan los microservicios.

Por último, se aplicaron las tácticas *Reduce Overhead* y *Limit Event Response*, al utilizar colas de mensaje como forma de comunicación entre microservicios. Los microservicios, como Reports Service, consumen procesan los mensajes a su ritmo y al evitar hacer llamadas sincrónicas, se reduce el overhead.

## RNF2. Confiabilidad y disponibilidad

Al migrar a una arquitectura de microservicios, ya se gana disponibilidad, siempre y cuando estos estén desacoplados. Los microservicios del sistema se comunican mediante colas de mensajes en la mayoría de las ocasiones, esto significa que si un servidor está bajo, los demás pueden seguir funcionando. Adicionalmente, los mensajes enviados a este servicio persisten en su cola, alojada en el servidor de mensajes, por lo tanto, en el momento que el servicio se reintegre y se conecte a su cola, puede incorporarse fácilmente procesando los mensajes que quedaron en su cola.













Con el fin de poder monitorear la salud y disponibilidad del sistema, se implementó la táctica de *ping/echo*, mediante un Endpoint que responde con un 200 si todos los componentes del sistema funcionan. Esta táctica fue implementada utilizando la gema *'health\_check'*, la cual define un endpoint (*/health\_check*), por el cual se puede consultar por el estado de salud del sistema. Dicho endpoint responde con 200-Success cuando el sistema está funcionando correctamente.

También utilizamos la técnica de *retry*, ya que si el servidor se llega a caer por algún error este es reiniciado.

### RNF3. Configuración y manejo de secretos

Para este requerimiento se utilizaron dos mecanismos: variables de entorno y credenciales de rails.

Para manejar los datos de configuración, se utilizaron variables de entorno. Esto permite desplegar el código en cualquier servidor sin tener que cambiar código, es una aplicación de la táctica de mantenibilidad *Defer Binding*. Adicionalmente se utilizaron las variables de entorno para manejar credenciales privadas, para que no sean visibles en el código fuente. Con este sistema se esconden variables como la dirección de la base de datos y sus credenciales.

Config Vars		Hide Config Vars
CLOUDAMQP_APIKEY	3e8d95a7-5ecc-459c-aaf6-e1a927d57911	 
CLOUDAMQP_URL	amqp://zbixfgda:fBcm6ElyQvux4ynj5MMUi	 
CREATED_PROMOTIONS_BINDING_KEY	promotion.created	 
CREATED_PROMOTIONS_QUEUE	created_promotions	 
DATABASE_URL	postgres://njcsbuodzrimjr:f7c0f1e61c55	 
DB_HOST	postgres://njcsbuodzrimjr:f7c0f1e61c55	 

Para los entornos de development y de testing, se utilizaron credenciales con valores por defecto para facilitar el desarrollo, pero estas solo sirven para correr el sistema localmente y no pueden ser usadas para acceder el sistema en production. En el entorno de producción es necesario utilizar las variables de entorno.

### RNF4. Autenticación y autorización

El equipo utilizó las tácticas de *Authenticate users* y *Authorize users*, para que cada usuario solo pueda acceder a sus correspondientes funcionalidades y datos. Esto fue implementado manejando sesiones con la gema *devise* y utilizando *Json Web Tokens*. Aquí se utilizó la táctica de *Verify Message Integrity*, ya que se verifica la firma de los tokens para validar que estos son válidos y que fueron creados por la aplicación.

## RNF5. Seguridad

Para evitar que se filtre información confidencial se utilizó la táctica arquitectónica *Encrypt data*, en los datos y las comunicaciones. Esto se logró utilizando el protocolo HTTPS para la comunicación entre el FrontEnd, Api-Gateway, y los microservicios. También se utilizó SSL para encriptar la comunicación entre los microservicios y las bases de datos. A su vez se encriptaron todos los datos confidenciales dentro de la aplicación, como por ejemplo las contraseñas de los usuarios.

## RNF6. Identificación de fallas

Para facilitar la detección e identificación de fallas se optó por la táctica *Maintain audit trail*. El sistema crea logs de las operaciones y errores que surgen. Estos logs son enviados al STOUT, que luego Papertrail (un recurso para el manejo de logs) recibe, muestra en tiempo real por el Log Stream, y también los guarda en archivos de log que son accesibles. Estos perduran más de 24 horas, como especificado en el requerimiento no funcional.

Paperclip también nos permite filtrar los logs por importancia, endpoint, código de HTTP o por otros atributos que permite identificar errores fácilmente.

## RNF8. Monitoriabilidad

Para recolectar métricas de el estado del sistema y para la detección e identificación de fallas utilizamos dos servicios Papertrail y Librato.

Papertrail nos permite ver logs en tiempo real, ver el histórico y filtrar por diferentes campos los logs. Esto nos permite detectar e identificar errores y ver los tiempos de respuesta por cada endpoint.

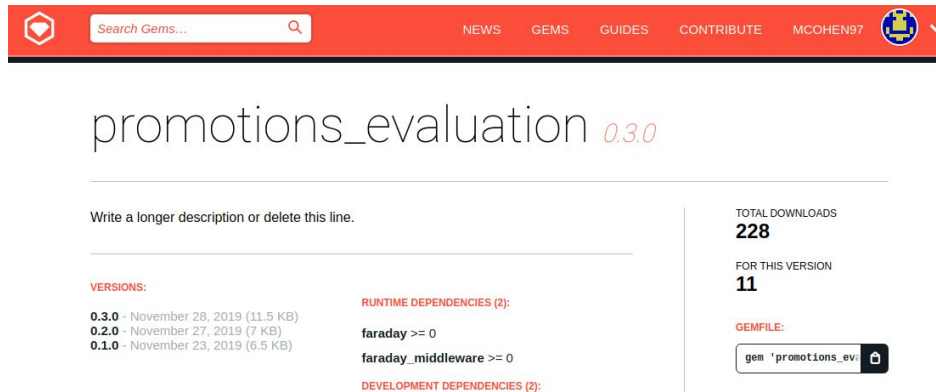
Librato nos permite calcular métricas sobre el funcionamiento del sistema como el throughput, el response time y la disponibilidad.

## RNF9. Independencia de aplicaciones

Nuestro sistema fue dividido en 6 microservicios independientes, cada uno teniendo un desarrollador responsable de este. Utilizamos diversos Stacks según los requerimientos de cada servicio. Como ya se ha mencionado anteriormente, se intentó mantenerlos lo más desacoplado posible.

# SDK de evaluaciones

Se decidió implementar el sdk de evaluación de promociones como una gema de ruby.



Link a la gema: [https://rubygems.org/gems/promotions\\_evaluation](https://rubygems.org/gems/promotions_evaluation).

El código fuente está disponible en:

<https://github.com/ArqSoftPractica/coupons-evaluation-sdk>

## Guía de uso

Para instalar la gema en su aplicación, hay que incluirla en el archivo Gemfile:

```
gem 'promotions_evaluation'
```

Luego ejecutar el comando:

```
bundle
```

O, también se puede instalar directamente:

```
gem install promotions_evaluation
```

La gema provee una interfaz sencilla para utilizar el sistema de evaluaciones. Se puede ver un ejemplo de uso en el archivo test.rb.

Para evaluar la promoción, se necesita invocar el método `perform_evaluation`, de la clase `PromotionsEvaluator`:

```
PromotionsEvaluator.perform_evaluation(<codigo de promocion>,  
<argumentos>)
```

Los argumentos deben pasarse utilizando el DTO provisto por el SDK

EvaluationParameters:

```
EvaluationParameters.new(country: <País>, city: <Ciudad>, birth_date:  
<Fecha de nacimiento>, attributes: <JSON con atributos de evaluación>,  
app_key: <clave de aplicación>, transaction_id: <id de transacción>)
```

El resultado de la función invocada devuelve un objeto `EvaluationResponse`, que posee los siguientes parámetros:

`successful` : que indica con `true` o `false`, si la petición fue exitosa.

`payload` : que contiene los datos devueltos por coupons-promotions al hacer la evaluación, en caso que se haya realizado éxito (sea aplicable o no).

`message`: contiene el mensaje de error en caso que no se haya podido evaluar (`transaction_id` ya utilizado, etc)



# Pruebas del sistema

## Pruebas unitarias y de integración

A nivel de microservicio, se crearon pruebas unitarias para comprobar su correcto funcionamiento, en los frameworks de pruebas provistos por las distintas tecnologías.

### TestCase en Rails.

```
PROBLEMS 109 OUTPUT DEBUG CONSOLE TERMINAL

(base) marcel@marcel-Inspiron-5759:~/Desktop/Reports Service/Reports_Service$ rails test
Running via Spring preloader in process 17904
Run options: --seed 63564

# Running:

.....

Finished in 2.358278s, 8.0567 runs/s, 27.1384 assertions/s.
19 runs, 64 assertions, 0 failures, 0 errors, 0 skips
(base) marcel@marcel-Inspiron-5759:~/Desktop/Reports Service/Reports_Service$
```

### Mocha, en NodeJS

```
> coupons-auth-service@1.0.0 test /home/cantuccis/Documents/Arq/Obligatorio2/AuthService/coupons-auth-service
> mocha -r ts-node/register test/**/*.test.ts

Role
  ✓ should create a new role with information provided
  ✓ should have permission asked
  ✓ should not have permission asked
User
  ✓ should create a user with information provided
  ✓ should have permission asked
  ✓ should not have permission asked
  ✓ should have permissions asked
  ✓ should not have permissions asked
  ✓ should be an admin
  ✓ should not be an admin
User Service
  ✓ should create a user service with repositories provided
  ✓ should save an admin user
  ✓ should create the organization when new admin is created and assign an org_id to it
  ✓ should hash the user's password
  ✓ should delete the user's password when retrieved
  ✓ should not add users with same username
  ✓ should add a invited user
  ✓ should set org_id to invited user when added
  ✓ should set role to invited user when added
  ✓ should not add a user without an invitation
  ✓ should not add a user with an invalid invitation

21 passing (34ms)
```

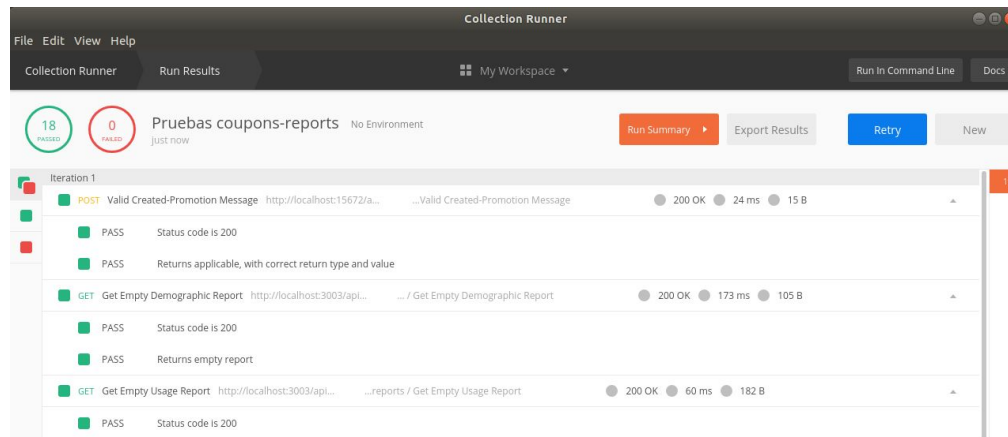
### Cargo Test, en Rust

```
eusebio ~ > Documents > evaluations > cargo test
Finished dev [unoptimized + debuginfo] target(s) in 0.09s
Running target/debug/deps/evaluations-731fa3951df423f0

running 11 tests
test models::demographics::tests::invalid_city_code ... ok
test models::demographics::tests::city_name_instead_of_code ... ok
test models::demographics::tests::country_name_instead_of_code ... ok
test models::demographics::tests::invalid_country_code ... ok
test models::demographics::tests::valid_data ... ok
test models::expression_parser::tests::invalid_coupon_code ... ok
test models::expression_parser::tests::valid_coupon_code ... ok
test models::expression_parser::tests::invalid_discount_code ... ok
test models::expression_parser::tests::valid_discount_code ... ok
test models::expression_parser::tests::valid_discount_code_or ... ok
test models::promotion_repo::tests::crud_test ... ok

test result: ok. 11 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
eusebio ~ > Documents > evaluations
```

También se crearon colecciones de pruebas de Postman, las cuales se encuentran en las carpetas de las aplicaciones, con el fin de probar los endpoints públicos.



## Pruebas de carga

### Capturas de pantalla de los resultados de las pruebas de carga

Adaptamos las pruebas de carga pasadas a el nuevo sistema. Estas pruebas realizan evaluaciones a promociones a promociones con valores en los atributos, transactions ids, y coupon codes randomizados dentro de los valores aceptables. La colección de pruebas de carga se encuentra en el repositorio Github de coupons-promotions.

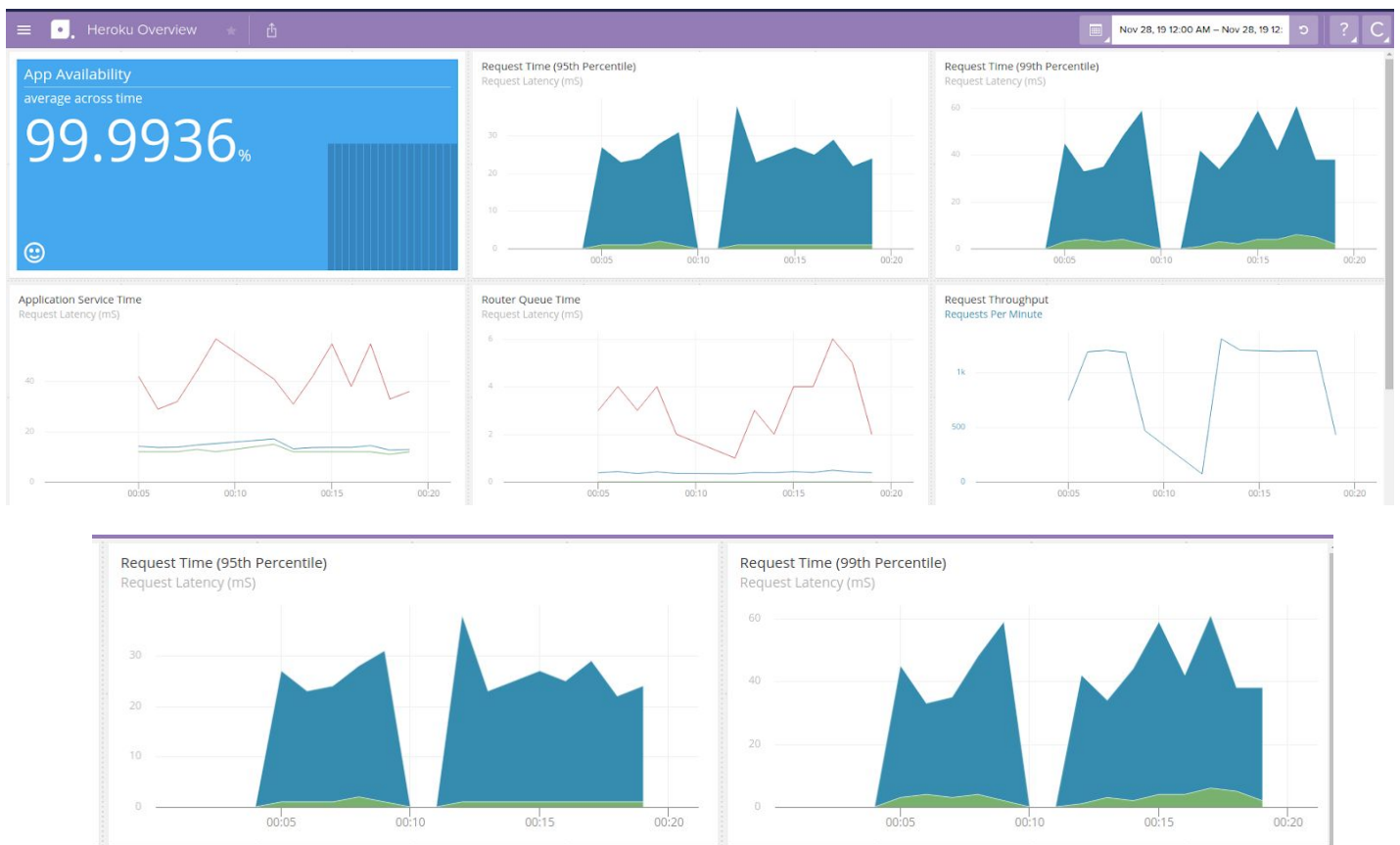
Los resultados en estas entrega sobrepasaron ampliamente los a los resultados pasados, a pesar de que ahora las evaluaciones son más complejas. En la entrega pasada tuvimos que utilizar 20 instancias de nuestro servicio en servidores de aptos para uso en producción, para llegar a un throughput por debajo de los 1200 request por segundo y con una latencia en promedio de 150 ms, con picos de 400 ms.

Mientras que ahora nuestro microservicio de promociones con una sola instancia hosteada en un dyno gratis de heroku, llegó a un throughput de 1700 request por segundo que luego de unos minutos se estabilizó en 1220 request por segundo con una latencia en promedio de 40 ms, con un pico máximo de 60 ms. Sin considerar la latencia de heroku, como se puede ver en los logs, la latencia es promedio es de 23 ms.

## Resultados de JMeter

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
Evaluate prom...	100	412	319	2030	245.46	21.00%	15.5/sec	3.97	7.02	261.7
Evaluate prom...	100	395	321	1107	172.84	0.00%	18.9/sec	4.66	7.38	251.8
	7631	374	36	4216	212.78	16.17%	20.0/sec	4.98	8.40	255.3
TOTAL	7831	375	36	4216	212.82	16.03%	20.4/sec	5.08	8.57	255.4

## Latencia de los pedidos



## Metricas generales

		New Events UI <input checked="" type="checkbox"/>		Give Feedback			
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	13.422588 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	9.348456 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 116	"-	P	10.073776 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	10.523915 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	15.981388 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	400 53	"-	P	6.819605 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	10.258500 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	15.589939 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	9.338190 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	13.849380 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	400 53	"-	P	6.387506 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 118	"-	P	10.891743 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	10.988188 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	15.484635 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	9.403336 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	17.895381 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	11.923056 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 130	"-	P	10.309151 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	12.213761 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	13.626080 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	9.794042 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	10.850378 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	9.051974 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	12.448615 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	18.532928 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 116	"-	P	11.177158 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	15.352723 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	12.135494 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 118	"-	P	20.098757 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	16.587405 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 118	"-	P	9.679375 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	17.816986 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 116	"-	P	10.120552 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	11.252752 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	12.767358 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	13.449119 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	400 53	"-	P	6.715800 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	7.690779 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	400 53	"-	P	7.336019 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	12.809404 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	20.937573 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 118	"-	P	13.252428 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	11.113973 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	9.412530 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	400 53	"-	P	9.979917 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	16.168303 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	400 53	"-	P	11.178550 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 72	"-	P	14.179263 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	17.440739 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 117	"-	P	11.167616 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	200 116	"-	P	12.228615 ms'	
"POST	/v1/promotions/chori promo/evaluate	HTTP/1.1"	200 111	"-	P	14.641365 ms'	
"POST	/v1/promotions/macri promos/evaluate	HTTP/1.1"	400 53	"-	P	9.277812 ms'	



# Descripción del proceso de deployment

## Coupons-promotions

1. Iniciar sesión en heroku desde el CLI con los comandos *heroku login* y *heroku container:login*
2. Estando situado en el root del proyecto ejecutar el comando *heroku container:push web*. Este comando construye la imagen y la sube a heroku.
3. Ejecutar *heroku container:release web*, que despliega la app en heroku.
4. Agregar a la app el addon de Postgres desde la web o el cli.
5. Agregar los addons de Librato y Papertrail.
6. Crear la base de datos con el comando *diesel migration database --database-url <url> setup* (en necesario tener instalado Diesel)
7. Introducir las siguientes variables de entorno en heroku.
  - **DATABASE\_URL:** URL del servidor de postgres
  - **RABBIT\_URL:** URL de la cola de rabbit.
  - **LOGGER\_FORMAT:** Formato del logeo, por ejemplo “%a %t “%r” %s %b “%{Referer}i” %P %D ms”
  - **RUST\_LOG:** Tipo de acciones que logear, por ejemplo “info, error” para logear mensajes de información y de error.
  - **SECRET:** Secreto de los tokens JWT.

## Coupons-auth

1. Crear una imagen de contenedor en docker para el servicio, ejecutando el siguiente comando en la carpeta raíz del proyecto:

```
$ docker build -t coupons-auth .
```

2. Para probar la aplicación antes de desplegar, correr la imagen con el comando:

```
$ docker run -p 8082:8082 -t coupons-auth
```

3. También se pueden correr los test previamente al despliegue con el comando:

```
$ npm run tests
```

4. Crear una aplicación en heroku. Para esta guía se considera coupons-auth.
5. Iniciar sesión en heroku desde el CLI con los comandos:

```
$ heroku login
```

```
$ heroku container:login
```

6. Estando situado en el root del proyecto, construir la imagen y subirla a heroku:

```
$ heroku container:push web --app coupons-auth
```

7. Desplegar la aplicación en heroku con:

```
$ heroku container:release web --app coupons-auth
```

8. Agregar los addons de Librato y Papertrail.
9. Crear una cuenta de MongoDB Cloud, crear un proyecto y obtener un connection string haciendo click en connect. Elegir la opción NodeJs. Poner el connection string en la variable de entorno **DB\_HOST**.
10. Introducir las siguientes variables de entorno en heroku.
  - **DEV\_MODE**: false para producción (esta variable expone endpoints que comprometen la seguridad del sistema, siempre mantener en false para producción).
  - **MQ\_HOST**: URL de la cola de mensajes rabbit.
  - **SECRET**: Secreto de los tokens JWT.

## Coupons-ui

1. Crear una imagen de contenedor en docker para el servicio, ejecutando el siguiente comando en la carpeta raíz del proyecto:

```
$ docker build -t coupons-ui .
```

2. Para probar la aplicación antes de desplegar, correr la imagen con el comando:

```
$ docker run -p 3000:3000 -e RAILS_ENV=production -e  
HOSTS=0.0.0.0 -e SECRET=whosjoe -t coupons-ui
```

3. Crear una aplicación en heroku. Para este servicio se considera coupons-ui.
4. Iniciar sesión en heroku desde el CLI con los comandos:

```
$ heroku login
```

```
$ heroku container:login
```

5. Estando situado en el root del proyecto, construir la imagen y subirla a heroku:

```
$ heroku container:push web --app coupons-ui
```

6. Desplegar la aplicación en heroku con:

```
$ heroku container:release web --app coupons-ui
```

7. Agregar los addons de Librato y Papertrail.
8. Crear una cuenta de MongoDB Cloud, crear un proyecto y obtener un connection string haciendo click en connect. Elegir la opción NodeJs. Poner el connection string en la variable de entorno **DB\_HOST**.
9. Introducir las siguientes variables de entorno en heroku.
  - **GATEWAY\_URL**: URL del gateway
  - **HOSTS**: URL de coupons-ui
  - **PORT**: 3000
  - **RAILS\_ENV**: production
  - **RAILS\_MASTER\_KEY**: Secret de rails
  - **SECRET**: Secreto de los tokens JWT

## Coupons-gateway

Coupons-gateway se implementó con Kong, a través del siguiente repositorio

<https://github.com/heroku/heroku-kong>

Allí pueden seguirse los pasos para levantar el gateway y configurar servicios con sus rutas.

Con el siguiente comando se puede acceder a la terminal del microservicio, desde donde se agregan los servicios y rutas al gateway:

```
$ heroku run bash --app coupons-gateway
```

A continuación se muestra un ejemplo de cómo crear un servicio en kong y agregar rutas a esté usando curl, habiendo realizado previamente los pasos de ADMIN que se encuentran en el repositorio anterior:

Agregar el servicio coupons-promotions y su url:

```
curl -H "apikey: coupons" -i -X POST \
  --url localhost:8001/services/coupons-promotions \
  --data 'url=https://coupons-promotions.herokuapp.com/'
```

Redirigir las request de coupons-gateway/v1/promotions a coupons-promotions/v1/promotions :

```
curl -H "apikey: coupons" -i -X POST \
  --url http://localhost:8001/services/coupons-promotions/routes \
  --data 'paths[]=v1/promotions/' \
  --data 'strip_path=false' \
  --data 'methods[]=GET' \
  --data 'methods[]=POST' \
  --data 'methods[]=PUT' \
  --data 'methods[]=DELETE' \
  --data 'protocols[]=https'
```



## Coupons-reports

Esta aplicación se despliega en heroku utilizando 2 containers: uno para el proceso web, otro para el proceso worker.

Para esto, se hace uso de un archivo heroku.yml, que es el que se encuentra en la raíz del directorio. Dicho archivo declara cuantos containers tiene la aplicación, y de que Dockerfiles se Heroku debe construir sus imágenes. En este caso tenemos un worker, que se construye de Dockerfile.worker, y un proceso web, que se construye de Dockerfile.web, como especificado en el mismo archivo YAML.

1. Ingresar a Heroku con las credenciales de la cuenta (Se debe tener una).

```
$ heroku login
```

2. Crear una aplicación.

```
$ heroku create
```

3. Hacer push de la aplicación, a la rama master de la aplicación creada.

```
$ git push Heroku <rama actual>:master
```

4. Esperar a que se termine el build de ambos containers.

5. Acceder a la aplicación desde el perfil de la cuenta Heroku, declarar las variables de entorno en Settings -> Config vars.

6. Agregar los Add-Ons necesarios, Base de datos de Postgres. También se necesita la ruta de un servidor RabbitMq, si no se tiene uno, también se puede adjuntar como Add-On a la aplicación.

```
$ heroku addons:create heroku-postgresql:hobby-dev  
$ heroku addons:create cloudamqp:lemur
```

7. Agregar en los config vars, la dirección de la base de datos asignada, y la de el servidor de mensajes rabbitmq.

8. Hacer el scaling, para asignarle un container a cada proceso.

```
$ heroku ps:scale web=1 worker=1
```

## Coupons-mailing

Esta aplicación se despliega en heroku utilizando 1 container de proceso worker, que recibe eventos de la cola, y envía los respectivos emails.

Para esto, se hace uso de un archivo heroku.yml, que es el que se encuentra en la raíz del directorio. Dicho archivo tiene declarado el tipo de proceso a desplegar (worker) y de qué archivo se hará el build, en este caso el archivo Dockerfile.

1. Ingresar a Heroku con las credenciales de la cuenta (Se debe tener una).

```
$ heroku login
```

2. Crear una aplicación.

```
$ heroku create
```

3. Hacer push de la aplicación, a la rama master de la aplicación creada.

```
$ git push Heroku <rama actual>:master
```

4. Esperar a que se termine el build del worker.

5. Agregar los Add-Ons necesarios, en este caso, se necesita una base de datos en memoria Redis, que es la que usa el proceso *Sidekiq* del worker. También se necesita la ruta de un servidor RabbitMq, si no se tiene uno, también se puede adjuntar como Add-On a la aplicación.

```
$ heroku addons:create heroku-redis:hobby-dev
```

```
$ heroku addons:create cloudfoundry:rabbitmq
```

6. Acceder a la aplicación desde el perfil de la cuenta Heroku, declarar las variables de entorno en Settings -> Config vars. Incluir la dirección de Redis y del servidor rabbitmq.

7. Hacer el scaling, para asignarle un dyno de worker a la aplicación.

```
$ heroku ps:scale worker=1
```

# Patrones de Microservicios aplicados

A continuación se explica, en detalle, los patrones de microservicios aplicados para cumplir con los requerimientos del sistema. Los nombres de los patrones son tomados del sitio Microservice Architecture (<https://microservices.io/>).

## Application patterns

### Decompose by business capability

Dividimos nuestro sistema en servicios basándonos en el dominio del problema, en lugar la tecnología. Esto significa que se dividieron los microservicios de forma que cada uno implementara un conjunto de funcionalidades fuertemente relacionadas. De esta manera logramos tener servicios cohesivos y desacoplados. Esto es muy importante ya que minimiza el impacto del cambio tenderán a impactar a solo un microservicio.

### Service per team

Cada servicio tiene su dueño que es el responsable de realizar cambios en este. Esto nos permite trabajar de forma desacoplada y nos da autonomía lo que resulta en un desarrollo más rápido ya que se pierde menos tiempo en coordinar. Esto tiene la desventaja dificultar probar todos los componentes del sistema juntos, y también, o al menos en nuestro proyecto, surgen dificultades de interoperabilidad entre los microservicios, por ejemplo: el formato de los mensajes que un microservicio espera recibir.

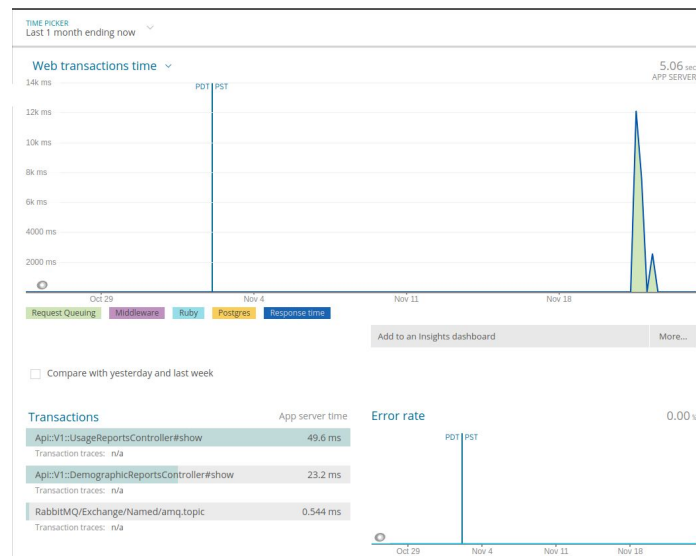
### Database per service

Cada servicio utiliza su propio repositorio de datos privado que utiliza para exponer una interfaz estable para comunicarse con los otros servicios. Esto mejora mucho el desacoplamiento ya que aunque se cambien los datos o la implementación de la persistencia, la interfaz se mantiene estable por lo que estos cambios no afectan al resto del sistema.

## Application metrics

Utilizando el servicio de librato, calculamos métricas como la disponibilidad, los tiempos de respuesta y la carga para monitorear el estado del sistema y para identificar y solucionar errores.

También, para algunos microservicios, se utilizó la herramienta newrelic, para monitorear el tráfico de las aplicaciones.



## Application Infrastructure patterns

### Health check API

En algunos microservicios se incorporó un endpoint de health check que permite monitorear el estado del servicio para determinar el estado del sistema.

### Exception tracking

Todas la excepciones que no son capturadas son logeadas con el fin de identificar defectos en el sistema. Esto es de mucha utilidad ya que los servicios están desplegados remotamente en la nube y no se tiene acceso directo a los hosts. Para facilitar esto utilizamos Papertrail para guardar y filtrar los logs. Papertrail tiene funcionalidades de notificación de logs por email.

[Papertrail] "Platform errors"

alert: 3 matches (at 2019-11-27 05:05) Inbox ☆



Papertrail Alerts 10:05  
to me ▾



**3 events** matched your **Platform errors** search in the past day.

```
Nov 26 12:17:44 coupons-evaluation
app/web.1: [2019-11-26T20:17:43Z ERROR
lapin::channel] Channel 1 closed by 60:40
=> Soft(ACCESSREFUSED) => ACCESS_REFUSED
- access to exchange 'user_events' in
vhost 'zbixfgda' refused for user
'zbixfgda'

Nov 26 12:17:44 coupons-evaluation
app/web.1: [2019-11-26T20:17:43Z ERROR
lapin::connection] Connection error

Nov 26 13:35:00 coupons-evaluation
heroku/router: sock=backend at=error
code=H18 desc="Server Request
-----
```

## Infrastructure patterns

### Service deployment platform

Utilizamos la plataforma de heroku ya que nos permite automatizar el despliegue que nos permite realizar despliegues más seguido y mantener el ambiente de producción más cercano a él de desarrollo. Además nos provee una serie de abstracciones y servicios muy útiles como bases de datos, caches y servidores de mensajes.

### Service instance per-Container

Al tratarse de microservicios distintos, creados en lenguajes y frameworks distintos, los cuales se querían desplegar de forma independiente, encapsulando los detalles de su tecnología y de forma que se puedan escalar fácilmente. Se encapsulo cada microservicio en un contenedor Docker. Lo cual nos permite lograr todo lo mencionado anteriormente. Heroku, ofrece funcionalidades para desplegar aplicaciones en contenedores fácilmente.

## Single service per host

Como ya se mencionó anteriormente, se buscó la independencia entre microservicios. Se aplicó este patrón para lograr que cada microservicio tuviera sus recursos, evitando conflictos, y para poder re-desplegar cada microservicio por separado, sin afectar la disponibilidad del resto.

## Api gateway

Consideramos que el particionado de los microservicios puede cambiar con el tiempo, también podrían alojarse en una red privada para que puedan comunicarse de una forma más rápida y segura (De todas formas, no se pudo llevar a cabo por que no pudo conseguirse un plan gratuito con una red privada). Considerando lo mencionado, se incluyó una aplicación que actuara como api gateway, funcione como reverse-proxy, entre el cliente y los microservicios.

# The Twelve Factors

## I. Codebase

One codebase tracked in revision control, many deploys

Cada servicio tiene uno repositorio en donde se encuentra la totalidad de su código. Se utilizan distintos environments para realizar despliegues distintos según el ambiente. Por ejemplo en utilizamos un ambiente development que utiliza recursos locales y tiene utilidades que facilitan el desarrollo y también tiene un ambiente production en donde se utilizan recursos externos y se optimiza para una mejor performance.

Este código también está manejado por sistema de control de versiones para mantener un versionado del código.

## II. Dependencies

Explicitly declare and isolate dependencies

Todas las dependencias de librerías están manejadas por manejador de paquetes de manera de tenerlas definidas explícitamente y aisladas del resto del sistema. En el caso de Rails se usó bundler que guarda la especificación de las gemas en un Gemfile. En NodeJS se utilizó NPM que guarda la especificación en el package.json y en Rust se utilizó Cargo que guarda las dependencias en el Cargo.toml.

Aun así los frameworks utilizados y/o las librerías a veces dependen de herramientas del sistema operativo por lo que se utilizó docker de manera de estandarizar el ambiente de ejecución y isolar las dependencias.

## III. Config

Store config in the environment

Para manejar datos que cambian según el ambiente, que contienen información confidencial o que se quieren cambiar con frecuencia utilizamos variables de entorno. De esta manera no hay que realizar cambios en el código para cambiar los datos de configuración sino que solo modificar las variables de entorno.

## IV. Backing services

### Treat backing services as attached resources

Nuestros servicios están desacoplados de los recursos adjuntos como las bases de datos, colas de mensajes u otros servicios externos. Esto quiere decir que podemos cambiar los recursos usados por un servicio en tiempo de ejecución sin tener que modificar el código. Esto fue implementado utilizando configuración por variables de entorno.

## V. Build, release, run

### Strictly separate build and run stages

Nuestro proceso de despliegue está separado en los siguientes 3 pasos:

- **Build:** En este paso se obtienen todas las dependencias necesarias y el código base se convierte en un ejecutable. En nuestro caso esto ocurre cuando creamos la imagen del contenedor docker. Este proceso varía según el servicio ya que utilizamos lenguajes compilados, transpirados e interpretados.
- **Release:** En este paso se combina el código ejecutable y la configuración para dejar todo listo para ejecutar la aplicación. En nuestro caso este paso se corresponde a el comando `container:release` de heroku
- **Run:** En este paso nuestro release es ejecutado en el ambiente de producción.

## VI. Processes

### Execute the app as one or more stateless processes

Nuestros procesos no persisten datos en memoria ni manejan sesiones locales de ningún tipo. La memoria local solo es utilizada como caché, para poder persistir datos se utilizan backing services como bases de datos. Además utilizamos métodos de autenticación stateless como JWT.

## VII. Port binding

### Export services via port binding

Nuestros servicios son completamente autocontenido, es decir que no dependen de que se le inyecten dependencias. Esto se logró incluyendo los servidores web como una dependencia más en el administrador de paquetes (en Rails se incluye puma como una gema, en NodeJS se incluye la librería de Koa y en Rust se incluye Actix-web como un crate de Cargo)

Cada servicio se adjunta a un puerto específico que en producción es envuelto por una capa de ruteo.

## VIII. Concurrency



## Scale out via the process model

Para poder escalar de una manera simple y confiable utilizamos distintos tipos de procesos que no comparten nada y son divisibles horizontalmente. Utilizamos procesos webs para manejar las peticiones inmediatas y procesos de workers para manejar operaciones de fondo como el envío de mails.

## IX. Disposability

### Maximize robustness with fast startup and graceful shutdown

Los procesos de nuestros servicios son descartables es decir que en cualquier momento pueden ser prendidos o apagados. Esto facilita la escalación, la rapidez de los despliegues y cambios de configuración. Esto también es muy importante en ambientes como heroku en donde las aplicaciones son suspendidas y reanudadas según la actividad.

## X. Dev/prod parity

### Keep development, staging, and production as similar as possible

Para minimizar el tiempo y los errores de despliegue utilizamos las siguientes técnicas:

- El código es desplegado rápidamente luego de ser codificado.
- Los mismos desarrolladores son los responsables de el despliegue.
- Se utilizaron las herramientas más cercanas a las de producción posibles. Por ejemplo se usaron los mismos tipos de bases de datos y mensajes de colas en development que en production.

## XI. Logs

### Treat logs as event streams

Nuestros procesos no son responsables de el manejo, ruteo y persistencia de los logs, sino que cada servicio simplemente escribe sus logs a stout. En el ambiente de desarrollo estos son vistos por la terminal, mientras que en producción estos son leídos por heroku y visibles por el stream de logs de heroku. Además son leídos por otros recursos externos como Papertrail que nos provee persistencia y análisis de los logs.

De esta manera como se especifica en *Backing Services* podemos cambiar el sistema de logueo en tiempo de ejecución sin modificar nuestro código.

## XII. Admin processes

Run admin/management tasks as one-off processes

Para realizar acciones singulares de administración utilizamos procesos separados de los procesos web que realizan las acciones regulares del sistema pero utilizando las mismas configuraciones y código base.

Por ejemplo en Rust utilizamos el ORM Diesel para realizar las migraciones y cambios a la base de datos y el mismo framework de rails con db:migrate.