

Universidad ORT de Montevideo
Facultad de Ingeniería.

Obligatorio 1
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://couponsweb.azurewebsites.net>

Octubre, 2019

Índice

Índice	1
Descripción de la arquitectura	3
Vistas de módulos	3
Vista de descomposición	3
Representación primaria	3
Decisiones de diseño	4
Vista de uso	4
Representación primaria	4
Decisiones de diseño	4
Vista de componentes y conectores	5
Representación primaria	5
Decisiones de diseño	6
Vistas de asignación	7
Vista de despliegue	7
Diagramas de secuencia	8
Evaluación de promoción	8
Justificaciones de diseño	10
RNF1. Performance	10
RNF2. Confiabilidad y disponibilidad	10
RNF3. Configuración y manejo de secretos	10
RNF4. Autenticación y autorización	11
RNF5. Seguridad	11
RNF8. Identificación de fallas	12
Pruebas del sistema	13
Pruebas unitarias y de integración	13
Pruebas de carga	14
Descripción del proceso de deployment	15
Deployment de la aplicación	15
Instalación de herramientas de consola	15
Variables de entorno	15
Configuraciones en Rails utilizando las variables de entorno	16
Production.rb	16
Routes.rb	17
Entrypoint.sh	17

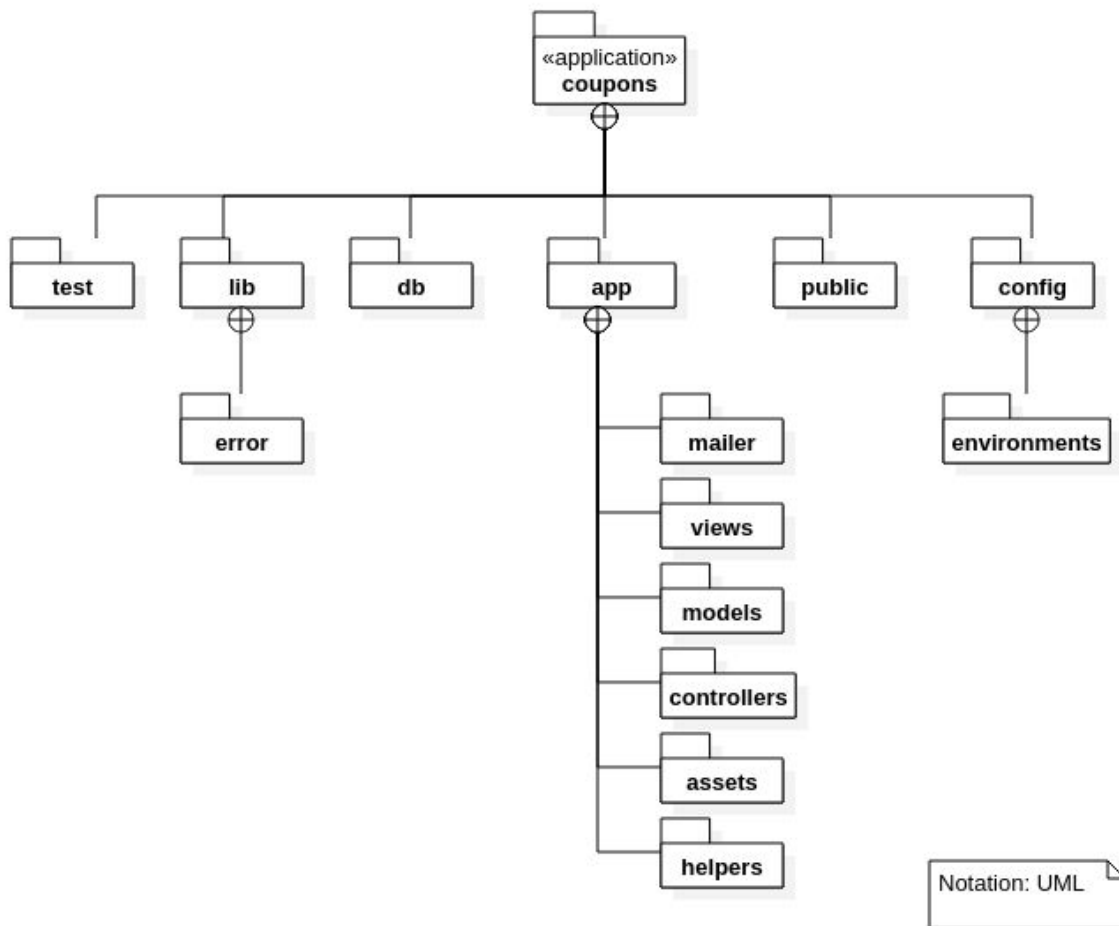
Database.yml	18
Storage.yml	18
Application.rb	19
Precompilación de assets	19
Configuración de Docker	20
Dockerfile	20
Docker-compose.yml	20
Build & Push de la Docker Image	21
Deploy del blob storage para imágenes subidas por usuarios	22
EDITOR="nano" bin/rails credentials:edit	22
Deployment de la Base de Datos Postgres	22
Variables de entorno en Azure	24
Realizar pruebas de carga	25

Descripción de la arquitectura

Vistas de módulos

Vista de descomposición

Representación primaria



La estructura de módulos se basó en la que viene por defecto en las aplicaciones Rails, la decisión se justifica en que el equipo decidió seguir el paradigma 'Convention over configuration' que proponen los creadores de este framework. En la representación solo se incluyen los módulos que se utilizaron, y que se consideran relevantes. Cabe destacar que el módulo test no es relevante en la funcionalidad en de la aplicación en sí, pero juega un

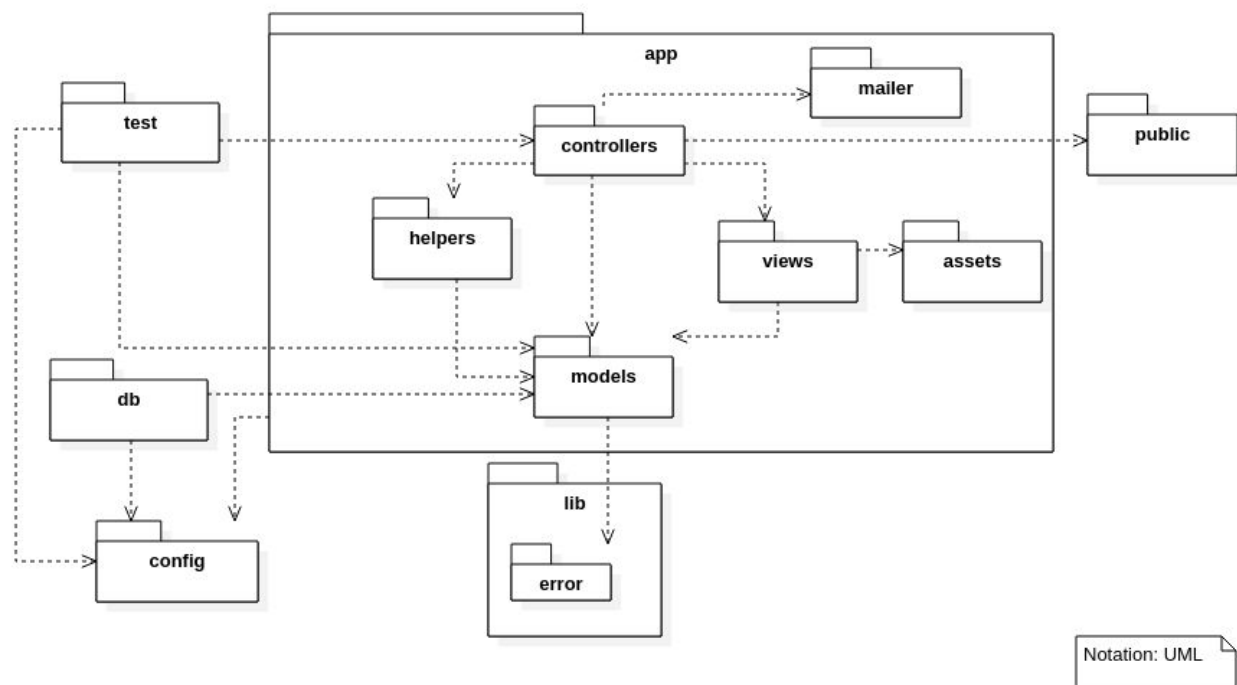
rol importante en dar confiabilidad a la lógica de la aplicación, además de que las pruebas son exigidas como requerimiento no funcional (RNF7).

Decisiones de diseño

En la vista ya se puede observar que se implementó una arquitectura monolítica, es decir, la aplicación está formada de un solo componente arquitectónico. Se optó por una arquitectura sencilla y fácil de implementar, ya que el equipo desconocía de casi todas las tecnologías empleadas en el proyecto. El equipo considera que esta arquitectura debe ser modificada en el futuro, de forma de poder mejorar atributos de calidad del sistema, como escalabilidad, mantenibilidad y disponibilidad, entre otros.

Vista de uso

Representación primaria



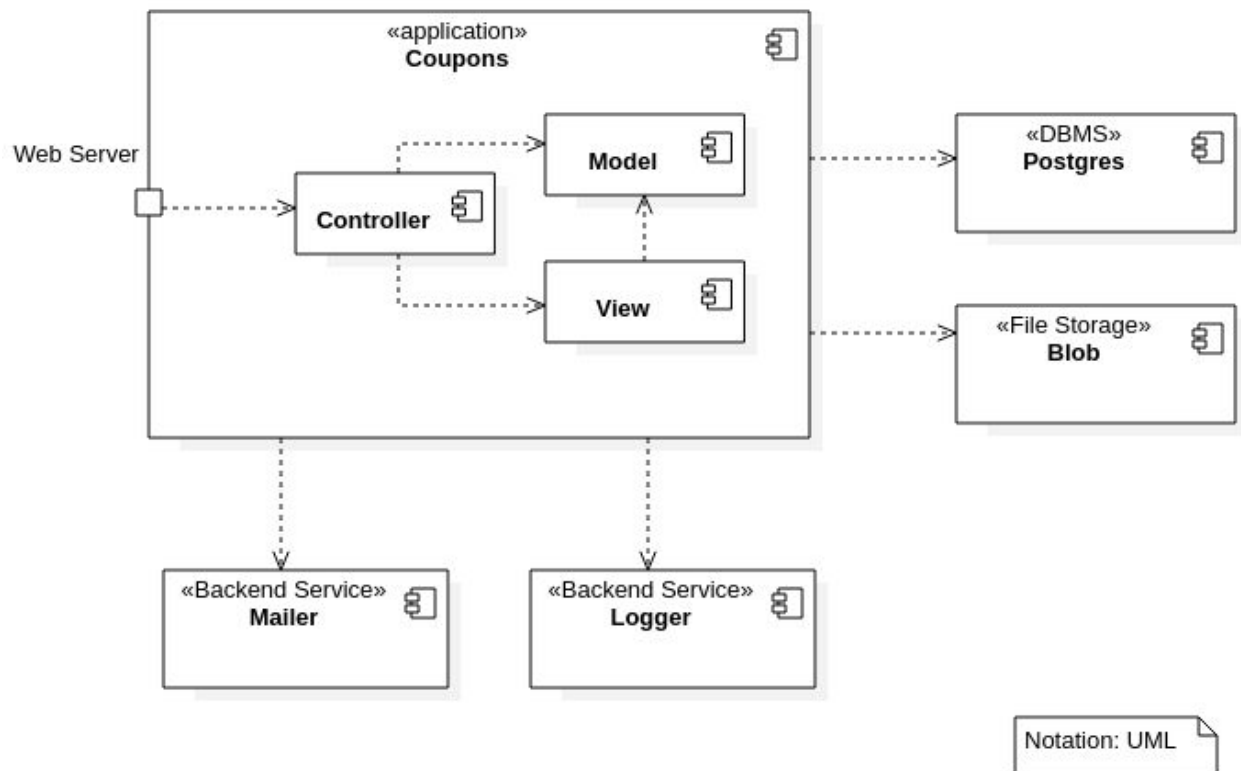
Decisiones de diseño

En cuanto a las dependencias entre los módulos, no hubieron muchas decisiones, porque, como se mencionó en la vista anterior, se siguieron convenciones, y tanto la estructura de los módulos como sus dependencias son bastante convencionales. Se creó un módulo error dentro de lib para guardar los errores propios de la lógica de negocio, también siguiendo

una práctica recomendada. El equipo optó por utilizar sus propios errores porque, basándose en Clean Code¹, se considera que hacen la lógica de la aplicación más sencilla, porque separa la lógica de los errores y esta no es opacada por el manejo de errores.

Vista de componentes y conectores

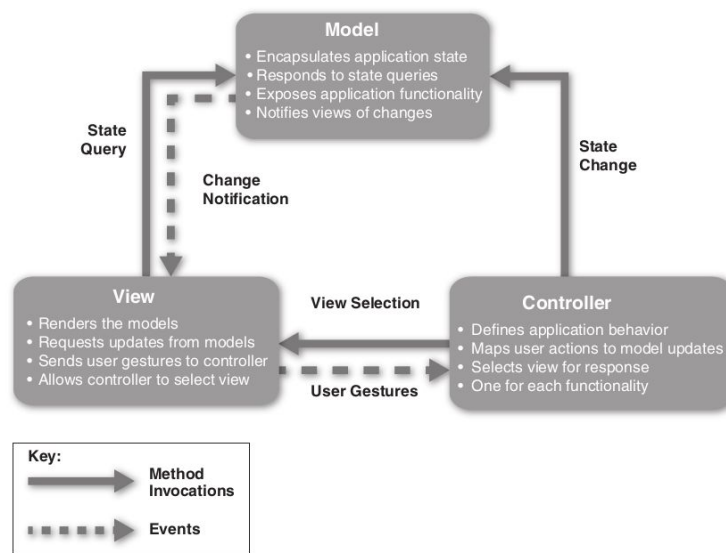
Representación primaria



¹ Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. ch. 7.

Decisiones de diseño

Para la construcción de la aplicación se utilizó el patrón arquitectónico Model-View-Controller². Una de las principales razones es que fue la arquitectura vista en clase para Rails, y el equipo prefirió apegarse a la convención ante una tecnología desconocida y un rango de tiempo relativamente corto para construir la solución. Este patrón busca separar los datos y la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario.

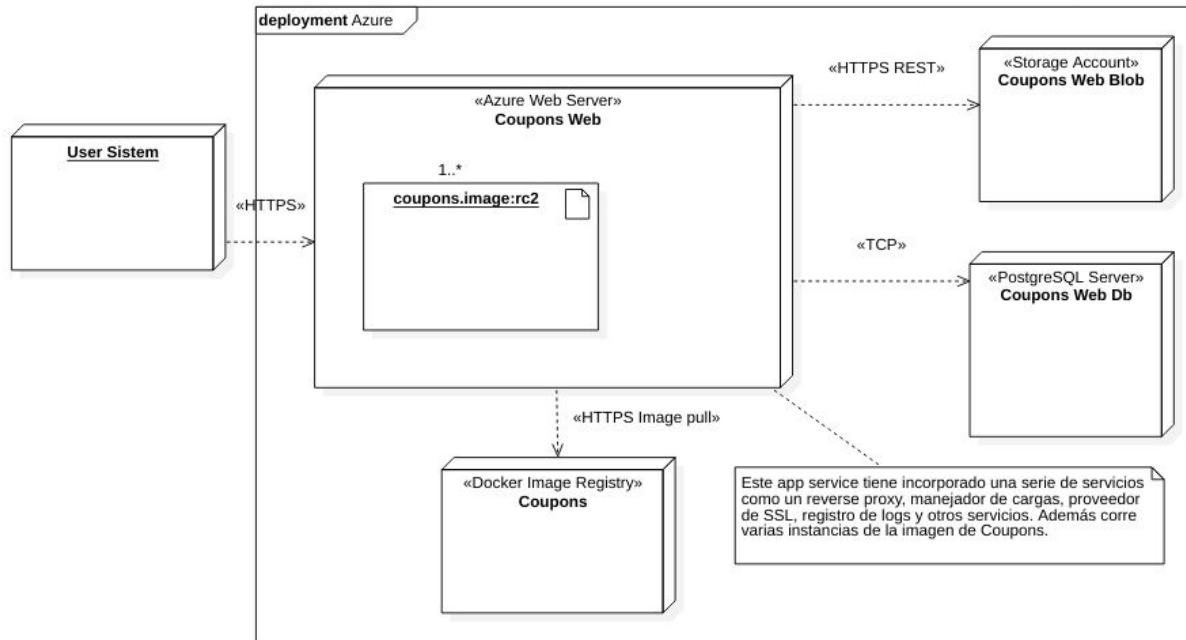


Otra razón, fue que el equipo encontró favorable este patrón, porque se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

² Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice: Third Edition. Addison-Wesley, 1997. p 212.

Vistas de asignación

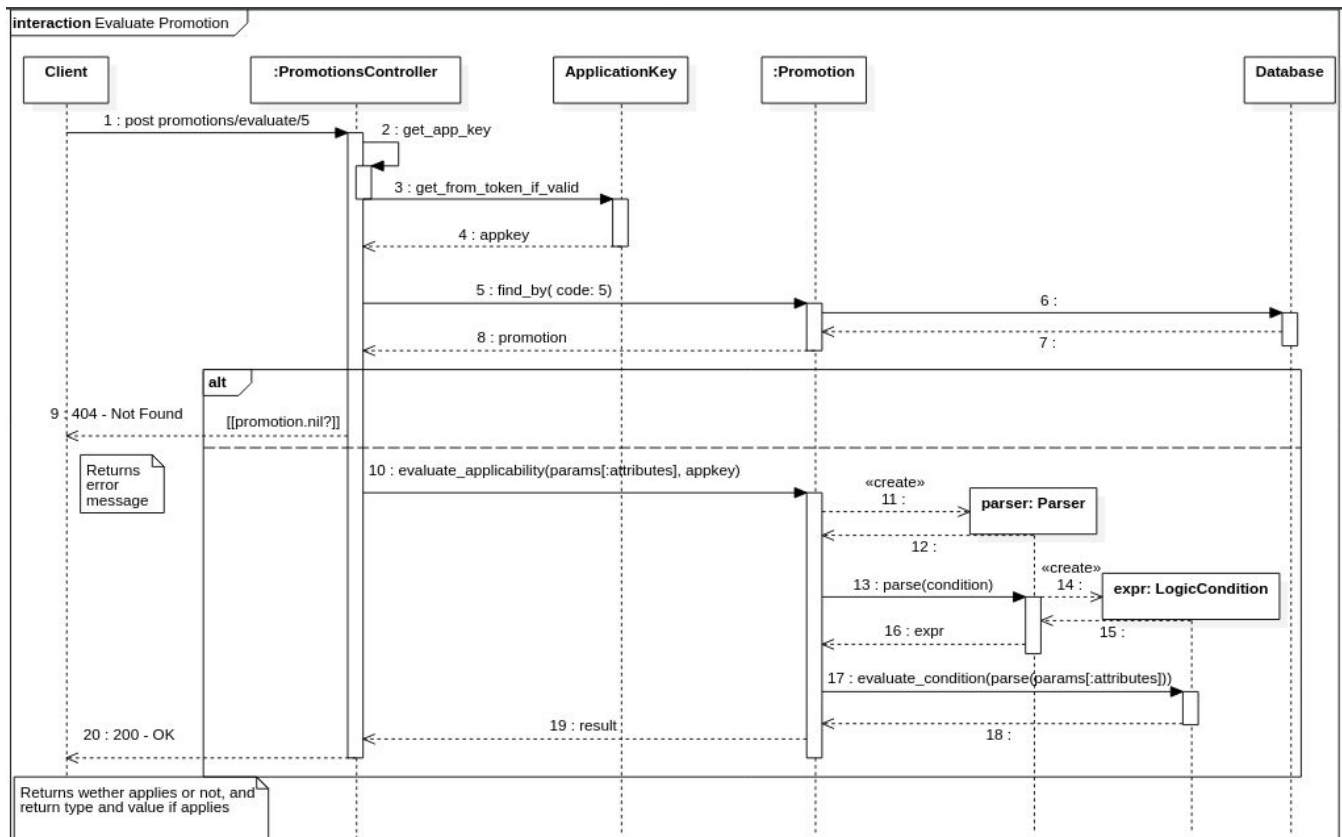
Vista de despliegue



Diagramas de secuencia

Evaluación de promoción

En el diagrama a continuación se muestra el flujo de la aplicación en el proceso de evaluación de promociones, el cual es considerado por el equipo como el más importante y complejo dentro de la aplicación.



Para implementar la especificación y evaluación de las condiciones de las promociones, se consideraron dos posibles opciones:

La primera fue crear estructuras en formas de árboles que representaran las distintas expresiones lógicas, las cuales serían evaluadas recorriendo el árbol recursivamente. Pero se descartó, ya que resultaba muy difícil de almacenar en la base de datos, y adicionalmente, como se optó por utilizar una base de datos relacional (Postgres), la recuperación de dicha estructura de la base de datos, puede implicar muchas operaciones JOIN, y se estimó que eso podría impactar fuertemente en la performance de la aplicación.

La segunda, que fue la que se halló más favorable, fue definir en la lógica de negocio, la gramática de las expresiones, y cómo debían de ser procesadas y validadas. De esta forma

se guardaría la expresión como un string simple en base de datos. A pesar de que el string de la condición debía ser procesado cada vez que se traía de la base de datos, el tiempo de procesamiento sería inferior a los accesos a disco de la primera opción.

Por último, para ahorrarse el parsing del string de condición cada vez que se evaluara la condición de una promoción, se podría aplicar caching a la expresión parseada. Esta última táctica no se llegó a implementar por que se alcanzó el requisito de performance antes de considerarlo.

Para describir la solución en más detalle, se construyó un parser para expresiones lógicas con notación infija, el cual aparece en el diagrama. Dicho parser procesa los strings de condiciones aplicando validaciones y construyendo objetos que representan la expresión lógica, que también se pueden observar en el diagrama como LogicCondition. Dicho objeto es el que puede utilizarse para evaluar la condición dados los parámetros de la promoción (Ej. total, quantity, etc).

El equipo optó por modelar la condición lógica como la parte variable dentro de la condición inicial, ignorando las partes mandatorias. Utilizando el ejemplo provisto en la letra:

```
○ IF valid_coupon_code AND total > 100 AND products_size >=2  
  THEN apply_discount
```

Nótese que la parte roja de la condición es la que varía entre las distintas promociones (de tipo cupón en este caso). Entonces, esa es la parte de la condición que se almacena como string en base de datos, y se procesa y evalúa con el parser.

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

Para aumentar la performance utilizamos múltiples instancias de nuestro servidor, junto con un balanceador de carga (Incorporado como parte del WebServer de Azure). Cada instancia a su vez corre varios threads independientes que atienden peticiones.

También utilizamos la técnica de *Multiple Copies of Data*, creando caches de los datos más accedidos de manera de evitar tener que calcularlo devuelta o tener que buscarlo devuelta en la base de datos. Se utilizó la interfaz Rails.cache, provista con el framework. La implementación seleccionada para el cache fue Memory Store, la cual no es recomendada en producción, pero fue la más sencilla de implementar y permitió mejorar la performance.

Por último se encontraron procesos en los cuales se ejecutaban varias queries para obtener entidades relacionadas, y se sustituyeron por una sola query.

RNF2. Confiabilidad y disponibilidad

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 datos que en configuración cambian, utilizamos variables de entorno de manera de poder desplegar el código en cualquier servidor sin tener que cambiar código, además las utilizamos para manejar credenciales privadas para que no sean visibles desde el código. Con este sistema guardamos variables como la dirección de la base de datos y sus credenciales.

Para el manejo de secretos utilizamos el sistema de credenciales de rails, agregamos nuestros secretos al `master.key` utilizando `credentials:edit`. Aquí guardamos el secreto de la encriptación de devise y la key para el storage blob. La única excepción fue el secreto de los tokens de JWS lo guardamos en una variable de entorno, porque en ese momento no sabíamos como funcionaba la credenciales de rails.

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

RNF4. Autenticación y autorización

El equipo utilizó las tácticas de *Authenticate users* y *Authorize users*, para que cada usuario pueda solo 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, el BackEnd y el blob storage. También se utilizó SSL para encriptar la comunicación entre el BackEnd y la bases de datos. A su vez se encriptaron todos los datos confidenciales dentro de la aplicación, como por ejemplo las contraseñas de los usuarios.

Otra medida de seguridad que se tomó fue aplicar la táctica de *Limit access*. Esta táctica fue implementada restringiendo el acceso a recursos que no deberían poder ser accedidos por cualquier usuario. Tanto la base de datos como el blob storage sólo pueden ser accedidos por servicios de Azure y por las IPs de la aplicación.

RNF8. 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 el WebServer de Azure recibe, muestra en tiempo real por el Log Stream, y también los guarda en archivos de log que son accesibles mediante FTP. Estos perduran mas de 24 horas, como especificado en el requerimiento no funcional.

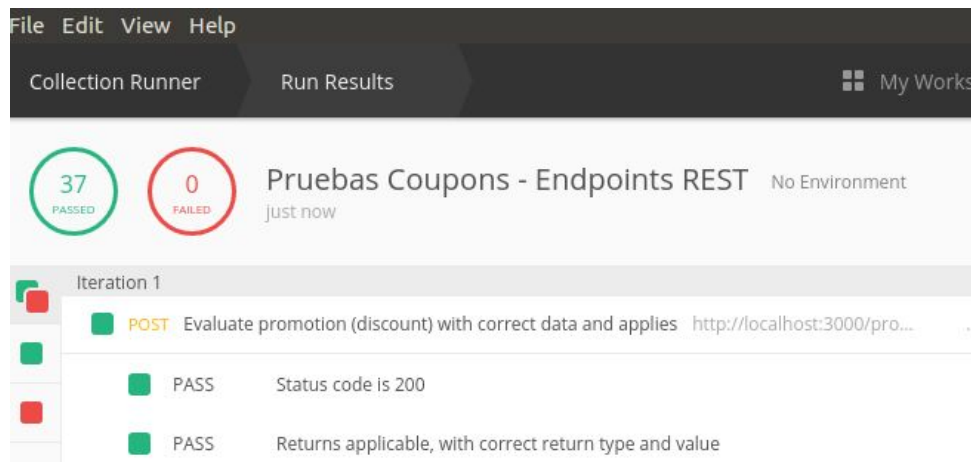
Pruebas del sistema

Pruebas unitarias y de integración

Para asegurar la correcta funcionalidad del sistema, el equipo utilizó dos tipos de pruebas automatizadas provistas por el framework: pruebas unitarias (`TestCase`) y pruebas de integración (`IntegrationTest`). Las pruebas unitarias se utilizaron para hacer pruebas a bajo nivel sobre las entidades del modelo, mientras que las pruebas de integración se utilizaron para hacer pruebas de los posibles casos de uso, integrando los controllers, modelos y vistas.

```
Finished in 8.864545s, 14.2139 runs/s, 34.9708 assertions/s.  
126 runs, 310 assertions, 0 failures, 0 errors, 0 skips  
(base) marcel@marcel-Inspiron-5759:~/Desktop/Coupons$
```

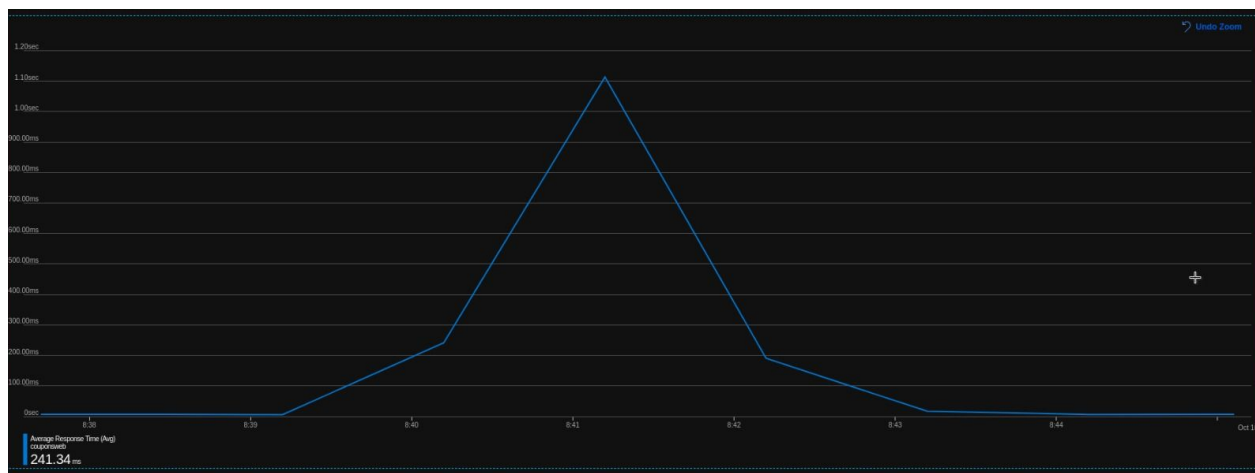
También se creó una colección de pruebas de Postman, la cual se encuentra en la carpeta de la aplicación, con el fin de probar los endpoints públicos.



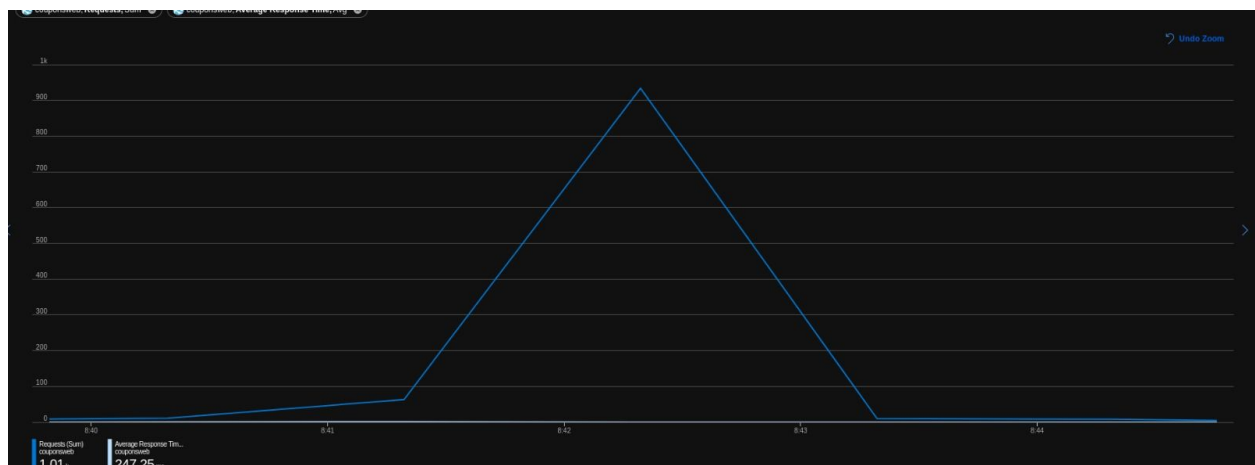
Pruebas de carga

Se creó un script de JMeter para realizar las pruebas de carga. Estas pruebas simulan una serie de acciones que son, aproximadamente, lo que los usuarios harían en la realidad, pero con un volumen mucho mayor de requests. Estas pruebas son realizadas sobre el sistema desplegado en Azure de manera de que esta sea lo más representativo al sistema real posible.

Para hacer las mediciones del tiempo de respuesta se utilizaron las métricas de Azure, ya que estas no son afectadas por la latencia que surge entre la distancia entre el cliente y el servidor.



Gráfica de el tiempo de respuesta en promedio.



Gráfica de la cantidad de requests procesadas por segundo.

Como se puede ver llegamos a un tiempo de respuesta de 241ms con un throughput de 1100 requests por minuto. Bajo un ambiente local la latencia se redujo a 34ms con un throughput de 1200 request por minuto.

Descripción del proceso de deployment

Para realizar el despliegue de este sistema se utilizaron contenedores *Docker*, que corren en Microsoft Azure. Se despliega la aplicación en un container, la base de datos en otro y un blob storage de Azure para las imágenes de perfil. La guía se realizará en base a un sistema operativo linux.

Deployment de la aplicación

Instalación de herramientas de consola

Para empezar, es necesario instalar Azure CLI y Docker en la máquina. Para ello, se pueden seguir las guías de Azure y Docker que explican cómo realizar la instalación en cada sistema operativo.

<https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest>

<https://docs.docker.com/install/>

<https://docs.docker.com/compose/install/>

Una vez instaladas las herramientas, se precisa ingresar con el usuario de Azure (como indica la guía de Azure). Para este sistema, se creó una cuenta utilizando la suscripción gratuita estudiantil provista por Microsoft 365.

Variables de entorno

Se necesita configurar el ambiente de producción y otros archivos de configuración para que el sistema pueda correr en Azure. Para esto, se hace uso de las variables de entorno para parametrizar esta configuración.

- **DB_HOST:** Indica la dirección de la base de datos Postgres.
- **DB_USERNAME:** Nombre de usuario de la base de datos.
- **DB_PASSWORD:** Contraseña del usuario de la base de datos.
- **HOSTS:** Dirección del sitio web que proporciona Azure.
- **JWT_SECRET:** Secreto para encriptación de tokens JWT.

- **MAILER_USER_NAME:** Dirección de correo electrónico que será utilizada para enviar los mails del sistema.
- **MAILER_PASSWORD:** Contraseña de la dirección de correo electrónico.
- **RAILS_ENV:** Ambiente de ejecución de Rails.
- **RAILS_LOG_TO_STDOUT:** Permite ver logs de la aplicación en azure.
- **RAILS_SERVE_STATIC_FILES:** Permite la utilización de assets ubicados en la carpeta public, utilizado para renderizar views de error (por ejemplo, la 404).
- **RAILS_MASTER_KEY:** Key utilizada para descriptar el archivo credentials.yml.enc.
- **STORAGE_ACCOUNT_NAME:** Nombre del blob storage account de Azure.
- **STORAGE_CONTAINER:** Nombre del container del storage account anterior.

Configuraciones en Rails utilizando las variables de entorno

A continuación, se muestra el uso de las variables de entorno anteriores y los elementos principales que deben aparecer en los archivos de configuración.

Production.rb

```

Rails.application.routes.default_url_options[:host] = ENV.fetch("HOSTS") {
'localhost' }

raise 'JWT secret not set in enviroment' unless ENV['JWT_SECRET'].present?

config.jwt_secret = ENV['JWT_SECRET']

config.require_master_key = true
config.assets.compile = false
config.assets.digest = true
config.serve_static_assets = false

Rails.application.config.assets.precompile += %w( *.js ^[^\]*.css *.css.erb )

config.public_file_server.enabled = ENV['RAILS_SERVE_STATIC_FILES'].present?
config.active_storage.service = :microsoft

config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  address: 'smtp.gmail.com',
  port: 587,
  domain: 'coupons.com',
  authentication: 'plain',
  enable_starttls_auto: true,

```

```

    user_name: ENV['MAILER_USER_NAME'],
    password: ENV['MAILER_PASSWORD']
  }

  if ENV['RAILS_LOG_TO_STDOUT'].present?
    logger = ActiveSupport::Logger.new(STDOUT)
    logger.formatter = config.log_formatter
    config.logger = ActiveSupport::TaggedLogging.new(logger)
  end

```

Routes.rb

```

Rails.application.routes.draw do
  devise_for :users, controllers: {
    registrations: 'users/registrations',
    confirmations: 'users/confirmations',
    sessions: 'users/sessions'
  }

  devise_scope :user do
    get '/users', to: 'devise/registrations#new'
  end

  get 'home/index', to: 'home#index', as: 'home'
  get 'home/invitation', to: 'home#invitation', as: 'invitation'
  post '/home/invite', to: 'home#invite', as: 'invite'
  get '/login/index', to: 'login#index', as: 'login'
  post '/login/create', to: 'login#create', as: 'register'
  post '/promotions/evaluate', to: 'promotions#evaluate', as: 'evaluate_promotion'
  get '/promotions/report/:id', to: 'promotions#report', as: 'generate_report'

  root 'home#index'

  resources :promotions
  resources :application_keys
end

```

Entrypoint.sh

```

#!/bin/bash
set -e

# Remove a potentially pre-existing server.pid for Rails.

```

```
rm -f /myapp/tmp/pids/server.pid

# Then exec the container's main process (what's set as CMD in the Dockerfile).
exec "$@"
```

Database.yml

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>

development:
  <<: *default
  username: <%= ENV['DB_USERNAME'] %>
  password: <%= ENV['DB_PASSWORD'] %>
  host: <%= ENV.fetch("DB_HOST") { "localhost" } %>
  database: Coupons_development

test:
  <<: *default
  username: coupons
  password: coupons
  host: localhost
  database: Coupons_test

production:
  <<: *default
  database: Coupons_production
  username: <%= ENV['DB_USERNAME'] %>
  password: <%= ENV['DB_PASSWORD'] %>
  host: <%= ENV['DB_HOST'] %>
```

Storage.yml

```
test:
  service: Disk
  root: <%= Rails.root.join("tmp/storage") %>

local:
  service: Disk
  root: <%= Rails.root.join("storage") %>
```

```

microsoft:
  service: AzureStorage
  storage_account_name: <%= ENV['STORAGE_ACCOUNT_NAME'] %>
  storage_access_key: <%= Rails.application.credentials.dig(:azure_storage,
:storage_access_key) %>
  container: <%= ENV['STORAGE_CONTAINER'] %>

```

Application.rb

```

require_relative 'boot'

require 'rails/all'
require 'rake'

Bundler.require(*Rails.groups)

module Coupons
  class Application < Rails::Application
    config.load_defaults 6.0
    config.before_configuration do
      env_file = File.join(Rails.root, 'config', 'local_env.yml')
      if File.exist?(env_file)
        YAML.safe_load(File.open(env_file)).each do |key, value|
          ENV[key.to_s] = value
        end
      end
    end
  end
end

```

Lo que se debe hacer en esta última configuración, es buscar un archivo llamado “local_env.yml”, que se encuentra en la carpeta config y define todas las variables de entorno que utilizamos. Esto nos permite cambiar fácilmente entre un ambiente de desarrollo y uno de producción ya que solo hace falta cambiar las variables para que se ajusten a un ambiente u otro.

Precompilación de assets

Todos los assets utilizados por el sistema deben estar ubicados en la carpeta app/assets. Sin embargo, cuando la aplicación se encuentre en producción, los assets nunca serán obtenidos de allí, sino que son precompilados y ubicados en la carpeta public/assets por

defecto. Desde ahí, las copias precompiladas son servidas como static assets por el web server.

Antes de realizar la imagen de docker, tenemos que precompilar los assets. Para eso, hay que ejecutar:

```
RAILS_ENV = production rails assets:clobber
```

```
RAILS_ENV=production rails assets:precompile
```

Configuración de Docker

Para configurar la imagen de docker, se agregan dos archivos de texto en la carpeta raíz del sistema:

Dockerfile

```
FROM ruby:2.6.4
RUN apt-get update -qq && apt-get install -y nodejs postgresql-client npm
RUN mkdir /myapp
WORKDIR /myapp
COPY Gemfile /myapp/Gemfile
COPY Gemfile.lock /myapp/Gemfile.lock
RUN bundle install
COPY . /myapp
RUN npm install yarn -g
RUN rails assets:precompile

# Add a script to be executed every time the container starts.
COPY entrypoint.sh /usr/bin/
RUN chmod +x /usr/bin/entrypoint.sh
ENTRYPOINT ["entrypoint.sh"]
EXPOSE 3000

# Start the main process.
CMD ["rails", "server", "-b", "0.0.0.0"]
```

Docker-compose.yml

```
version: '3'
services:
  web:
    build: .
    image: coupons.image:rc2
```

```
command: bash -c "rm -f tmp/pids/server.pid && bundle exec rails s -p 3000 -b '0.0.0.0'"
volumes:
  - ./myapp
ports:
  - "3000:3000"
network_mode: "host"
```

También, antes de realizar la imagen de Docker, se deben crear algunos recursos en Azure: un Resource Group, un Service Plan y un Container Registry. Estos pueden ser creados utilizando el portal de azure. **Todos los recursos son creados en East US**, debido a que da menor latencia en comparación a otros servidores para requests desde Uruguay.

El plan utilizado para el deployment de este obligatorio es P1V2, el cual puede adquirirse con la suscripción de estudiante. El equipo consideró que lo máximo que se podía invertir en recursos para cumplir con los requerimientos no funcionales era la cota superior del plan de estudiantes.

Build & Push de la Docker Image

Una vez que se tienen los recursos creados, se procede a ejecutar los siguientes comandos, ubicándose en la carpeta raíz del sistema:

Antes de realizar el build es necesario eliminar el archivo local_env.yml.

```
docker-compose build
```

```
docker tag coupons.image:rc2 coupons.azurecr.io/coupons:v1
```

```
docker push coupons.azurecr.io/coupons:v1
```

En este caso, nuestro Container Registry se llama coupons y la url provista por azure, donde subiremos la imagen del contenedor, es coupons.azurecr.io. El v1 es un tag que se le puede poner a la imagen para diferenciarla de otras.

Una vez subida la imagen, se debe crear un App Service con ella. La misma, puede ser creada desde el Container Registry, en el portal de azure: Coupons >> Repositories >> coupons, allí se pueden ver todos los tags subidos. Seleccionando la opción “Deploy to web app”, que aparece al desplegar el menú de opciones del tag subido, se crea el App Service que contendrá una instancia del Docker. Consideraremos el nombre couponsweb para el App Service.

Deploy del blob storage para imágenes subidas por usuarios

Se necesita crear un Storage Account de Azure para guardar las imágenes que suben los usuarios. Se pueden poner en un storage de Azure, al cual accederán todas las instancias de la aplicación.

Para crear el blob storage, se debe ir a Storage Accounts y hacer click en add, le llamaremos couponswebblob. Dentro de couponswebblob se debe ir a container y agregar uno nuevo, eligiendo la opción "Container".

En Access keys, se obtendrá la storage_access_key propia, la cual se pondrá en credentials.yml.enc. Con un editor de texto en consola, se puede poner la storage_access_key de esta forma:

```
EDITOR="nano" bin/rails credentials:edit

GNU nano 2.9.3 /tmp/13152.credentials.yml

# aws:
#   access_key_id: 123
#   secret_access_key: 345

# Used as the base secret for all MessageVerifiers in Rails, including the one p$
secret_key_base: 2540392838032f1b338462d0205c1e95ad038a492d2337224c99b2b81ff747f$

azure_storage:
  storage_access_key: BvG5hMyDn2YnNIz6sVzwYNn0/UKuQbM6EGKXErHPGzbbJEAUDrlns4QyuF$

EDITOR="nano" bin/rails credentials:edit
```

Deployment de la Base de Datos Postgres

Para desplegar la base de datos postgres, se utilizará un servidor postgres provisto por azure. El recurso se llama Azure Database for PostgreSQL y puede ser creado desde el portal de azure. Se utilizará la opción single server y consideraremos el admin username coupons.

Para mejorar la seguridad del sistema, se debe activar la opción Enforce SSL connection en Connection security y agregar cada una de las Additional Outbound IP addresses que aparecen en couponsweb >> Properties. Con esto se está permitiendo que solo las IPs de la

aplicación puedan conectarse a la base de datos. También se debe agregar la IP de la PC utilizada, para poder configurar la base de datos.

En este punto, ya se tienen todos los recursos creados para asignar los valores a las variables de entorno definidas.

Se deben definir los valores de las variables de entorno en el archivo `local_env.yml` (se crea nuevamente) en la PC utilizada. El mismo debería verse similar a este:

```
DB_USERNAME: 'coupons@couponsweb-db'
DB_PASSWORD: '*****'
DB_HOST: 'couponsweb-db.postgres.database.azure.com'
STORAGE_ACCOUNT_NAME: 'couponswebblob'
STORAGE_CONTAINER: 'couponsstoragecontainer'

TEST_DB_USERNAME: 'coupons'
TEST_DB_PASSWORD: 'coupons'

MAILER_USER_NAME: 'coupons.arancet.cohen.duran@gmail.com'
MAILER_PASSWORD: '*****'

RAILS_SERVE_STATIC_FILES: 'true'

JWT_SECRET: '*****'

RAILS_MASTER_KEY: '*****'

HOSTS: 'localhost'
```

`DB_HOST` se obtiene en el Overview del recurso Database for PostgreSQL creado. `DB_USERNAME` es el nombre de usuario de admin que se creó junto con el recurso, seguido de “@” y el nombre de la base de datos.





















Luego se debe conectar a la base de datos desde la PC utilizada y correr las migraciones con el siguiente comando:

```
RAILS_ENV=production rails db:setup
```

(En caso que se quiera borrar la base de datos, se debe correr el siguiente comando)

```
RAILS_ENV=production DISABLE_DATABASE_ENVIRONMENT_CHECK=1 rails db:drop
```


Variables de entorno en Azure

COUPONS_DATABASE_PASSWORD	 Hidden value. Click show values button above to view
DB_HOST	 couponsweb-db.postgres.database.azure.com
DB_PASSWORD	 Hidden value. Click show values button above to view
DB_USERNAME	 coupons@couponsweb-db
DOCKER_ENABLE_CI	 true
DOCKER_REGISTRY_SERVER_PASSWORD	 Hidden value. Click show values button above to view
DOCKER_REGISTRY_SERVER_URL	 https://coupons.azurecr.io
DOCKER_REGISTRY_SERVER_USERNAME	 coupons
HOSTS	 couponsweb.azurewebsites.net
JWT_SECRET	 Hidden value. Click show values button above to view
MAILER_PASSWORD	 Hidden value. Click show values button above to view
MAILER_USER_NAME	 coupons.arancet.cohen.duran@gmail.com
RAILS_ENV	 production
RAILS_LOG_TO_STDOUT	 true
RAILS_MASTER_KEY	 Hidden value. Click show values button above to view
RAILS_SERVE_STATIC_FILES	 true
STORAGE_ACCOUNT_NAME	 couponswebblob
STORAGE_CONTAINER	 prueba
WEBSITE_ENABLE_SYNC_UPDATE_SITE	 true
WEBSITE_HTTPLOGGING_RETENTION_DAYS	 100

Realizar pruebas de carga

Para ejecutar las pruebas de carga se debe abrir el archivo *Pruebas de carga.jmx* con jMeter. Luego se deben configurar las siguientes variables, definidas en *User Defined Variables*:

- DOMAIN: La dirección donde se encuentra el servidor.
- PORT: Puerto del servidor
- APPKEY: El token llamado “All pedidos ya” de la organización Pedidos Ya. (Logearse con el usuario pv@gmail.com contraseña 123456)
- COUPON: Nombre del cupón de prueba, esta variable no hace falta cambiarla.
- COUPON_KEY: Token de una app_key que contenga al cupón de la variable COUPON.

Luego de esto las pruebas están listas para ser corridas.