BCIO

Branded Content Inventory Optimization

TECHNICAL IMPLEMENTATION

Functional documentation.

Introduction to the big three components of the tool:

This tool provides a solution for the creation and management of sponsored content for a final website. The main objective is the injection of sponsored offers on a web into a given space, also including a solution for content management and results analysis.

The injection of offers is based on an analysis of the interests of the user who visits the final website. Depending on the interests registered in the user data, the application will select the offer that best suits those interests and will inject it on the front web page replacing/rotating other offers of the same active campaign.

For these purposes, a cloud system has been designed that provides the content creator a web environment as a backoffice to see campaigns, see the clicks and impressions as the main goals of the associated offers and manage it to adapt to the branded content needs.

This design is divided into three main components:

- Backoffice.
- Recommendation API.
- Module Web (script).

Backoffice:

Backoffice web application is designed to provide to the BCIO content creator a web environment where create and manage sponsored content campaigns with associated offers.

Once a campaign is created and a valid offer is associated, the tool will send the offers to the final page as long as the campaign is still active on dates.

Backoffice also provides a logic to create and manage campaigns and offers.

Campaign functional logic:

A campaign is created with mandatory fields and optional dates.

Fields list:

- **Page and Position**:
  Mandatory field. Final page where offers will be injected and the associated position where the offer has to match.
  Page and position values are included via insert into the database. These fields are selected by the main user and set on the installation.

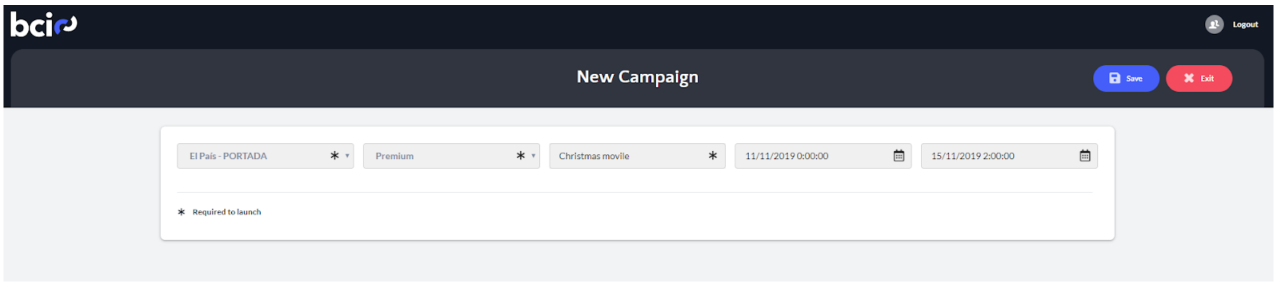
  These fields are designed to apply a control over the final page and its layout of the content of the offers. Page field indicates the identifier of the web page and position its associated template. Example:

  - Page 1 has two associated positions, Premium and Standard.
    This is, each position has one or more associated sizes for the offer image, kicker colors and an html template that will be used to find the html article/class on the final page and replace the original offer with the new one.

    The html template is set on the installation tool as the page and positions are but the template is indicated in a file instead of database. These templates must be prepared for variable substitution. This is, indicate a {{variable}} field related to each field of the offer. This code will be replaced by the value of the offer field and injected as plain html on the final page and replacing the initial offer.

- **Campaign title:**
  Mandatory field filled by the campaign creator. Indicates the title/description of the campaign.

- **Dates. Start and End:**
  Optional fields. Determines start and end of the campaign. Campaigns can be created without these two fields and be set in next steps but then they will be mandatory to launch.



El País - PORTADA *
Premium *
Christmas movile *
11/11/2019 0:00:00
15/11/2019 2:00:00

* Required to launch

Campaign states:

Depending on dates, associated offers status or user decisions campaigns have a state/status attribute which determines its activity. States can be *DRAFT*, *LIVE*, *PAUSED*, *SCHEDULED* and *CLOSED* and campaign behavior varies depending on their status.

- **DRAFT**

  Innital and default status of a campaign of new creation. In this states the campaign is not able to be launched because of missing or incomplete default associated offer. Will stay as DRAFT as long as the user wants or until the default offer is filled and launch the campaign.

- **LIVE**

  Main state of a campaign. In this state a campaign has at least a valid default offer and an active/current date. A campaign passes to LIVE status when it gets saved and the user clicks on the 'Launch' button. Then data is saved, validated and status is marked as LIVE because of current dates.
  At this point the system is running and sending the offers to the page and position of the campaign.

- **SCHEDULED**

  Same behavior as LIVE state but at this point campaign is launched but its start date has not been reached. Campaign will pass to LIVE automatically when its start date arrives.

- **PAUSED**

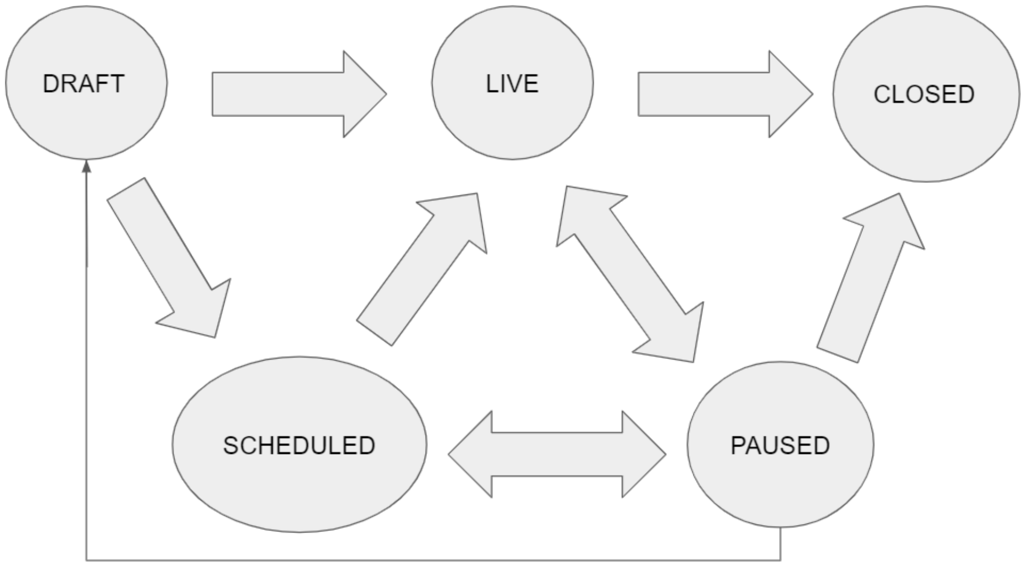
  Only LIVE or SCHEDULED campaigns can be paused. This campaign status prevents offers from being sent in case of LIVE campaign or pass to LIVE from SCHEDULED ones.
  A paused campaign can be resumed but this state implies certain controls:
  - A campaign can only be resumed by the user.
  - When the user resumes a paused campaign, data is validated and the campaign passes to LIVE or SCHEDULED. Default offers must be valid to resume a campaign but if there is some other offer with invalid data the campaign will resume and the invalid offer will not be injected.
  - A paused campaign can not be resumed if its dates overlaps with some LIVE or SCHEDULED campaigns.
  - If a paused campaign exceeds its end date automatically passes to CLOSED.

  To pause a campaign click on Pause toggle on the campaign list page or on the Pause button on modify campaign page (replaces Launch button).

- **CLOSED**

  Final state. When a campaign reaches its end date automatically its marked as closed and users cannot launch or modify it anymore. Its campaign and offer data can be consulted but not modified.

Campaign state diagram:



Campaign state diagram explanation:

From	To	Operative
DRAFT	LIVE	Campaign created, saved and launched.
DRAFT	SCHEDULED	Campaign created, saved and launched if its beginning is in some date after this very moment.
LIVE	CLOSED	Campaign wich has reached its end date.
LIVE	PAUSED	Campaign gets deactivated and will send no offers till resumed. Even if its end date is overpassed paused status remains active.
PAUSED	LIVE	After clicking the resume button data gets validated and campaign is relaunched.
PAUSED	SCHEDULED	If a paused campaign gets resumed. Data is validated and status gets updated as scheduled due to dates.
PAUSED	DRAFT	Special transition in case of a scheduled campaign tries to pass to live due to dates but its default offer is paused.
SCHEDULED	LIVE	Reached start date on a valid campaign. This change is automatic made by a background process.
SCHEDULED	PAUSED	Clicking on the pause button in a scheduled campaign changes its status and the campaign gets freezed.
PAUSED	CLOSED	Reaches end date on a paused campaign.

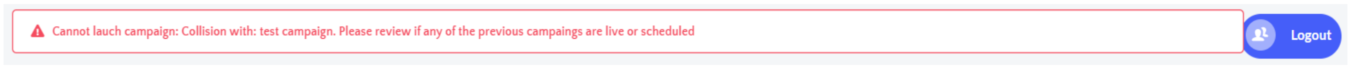
Campaign filter by status:

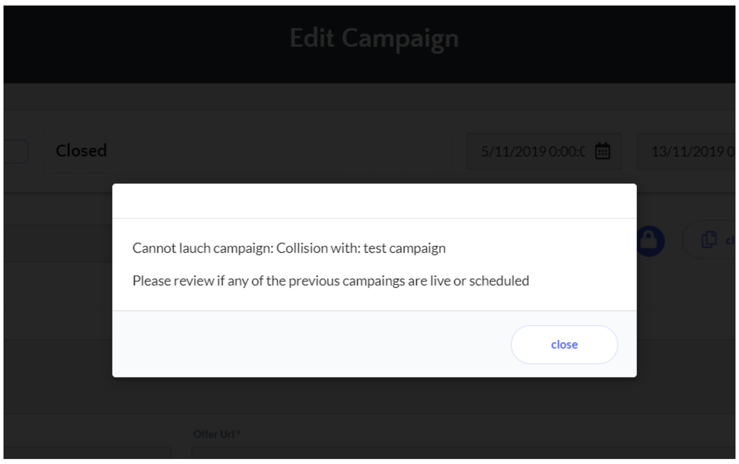
On the home page there is an option to filter the list of campaigns by it's status. Paused campaigns will be shown with the LIVE check even if the resumed status is LIVE or SCHEDULED.

Collisions on campaigns:

Two campaigns are in collision when it's dates overlap each other and it's final page and position are the same. This means a live campaign can not be injecting offers to the same page and same position at the same time as another campaign.

To handle this behaviour backoffice will not allow to launch/resume a campaign when a collision exists with another LIVE/SCHEDULED campaign.
When this happens an error/warning message will appear on the top of the page or a popup indicating the overlap.

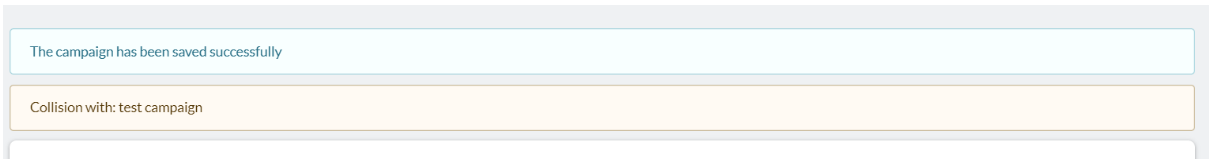

Cannot lauch campaign: Collision with: test campaign. Please review if any of the previous campaings are live or scheduled



Cannot lauch campaign: Collision with: test campaign

Please review if any of the previous campaings are live or scheduled

Warning messages will always appear indicating the conflicting campaign when saving data if two campaigns share page, position and dates interval even if the collisioned campaign is not in LIVE/SCHEDULED status.


The campaign has been saved successfully

Collision with: test campaign

Notifications:

The notification system will notify the user of certain situations related to a campaign that requires their attention. This system is designed following these basic features:

- Notifications will be independent per user. Each user will have all notifications and will mark them as read or deleted as they consider but that will not affect other users, who will continue to have those notifications on their panel.
- Clicking the notifications button at the homepage will display a side panel with the list. There will be a Read button to mark as read and an X in to clear the notification.
- A limited number of notifications will be displayed. Suppose there are 100 unread notifications, as a maximum of X will be displayed (eg 30) that will appear scrolling. No paging or loading of previous notifications.

There are three types of notifications implemented:

- Notification in the app when a campaign pass to live and some offer is incomplete:
  - The system detects that a campaign will pass to live (from SCHEDULED or PAUSED) and it has invalid or paused offers (not the default one).
- Notification in the app when the start date is near and the campaign is in DRAFT state:
  - The system detects that a draft campaign has saved dates and the start date will be soon.
- Notification when the default offer is retirated of the cover page (from the widget):
  - The system cannot find the default offer of the page to inject the offers of the campaign.
  - The campaign will be set as PAUSED.

Offer functional logic:

An offer can be created with mandatory and optional fields. An offer is the single branded content unit which belongs to a single campaign and represents the content that will be injected into the page and position of its father campaign indicates.
On modify campaign page campaign basic data can be updated. On the same page each offer data is loaded as a section of the page and can be modified too. By default the first offer is set as default offer. This can be changed by switching on the lock icon of the offer.

Fields list:

- **Offer Name:**
  Auto generated by default when a user creates a campaign or when an offer is cloned.
- **Offer Description:** Optional and 20-250 characters long.
- **Brand Name:** Name of the branded content company.
- **Offer Url:**
  Mandatory. Url of the default offer of the destination page. BCIO tool will use this url to find the offer into the html code of the final page. Then will replace the original offer with the new one injecting its data and the html template.
- **Headline:** Mandatory. Main title input.
- **Subtitle:** Subtitle input.
- **Kicker Url:** Associated url of the kicker.
- **Kicker class name:** Name of the .class associated with the kicker style. On the final page this class will be styled with css.
- **Kicker text:** This text will be shown on the kicker image.
- **Offer Goal:**
  Mandatory. Number of clicks marked as offer main goal. If some offer goal is reached, the offer will not be a priority and the system will show other sibling offer. If every offer has reached the goal but campaign remains as live, the system will send a random one of its offers.
- **Image:**
  Mandatory. Main image of the offer content. Must be loaded with every size that campaign position marks on the load modal.
- **Tags:** Documentation or segmentation tags. It's a way of classifying the offer. Tags must be loaded in the installation inserting it into the database.
- **Author:** Author of the linked content.
- **URL Author Page:** Link to portfolio/personal space of the author.
- **Photographer:** Author of the offer image.
- **Image Caption:** Text associated to the image.
- **Image Copyright:** Copyright of the image if any.

Offer states:

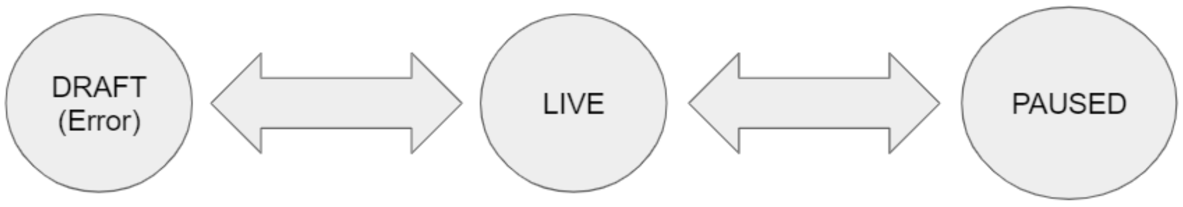
Offers have its own status attribute. There are three states and each one determines if the offer is valid or ready to be sent.

- **DRAFT**
  Initial state of every offer. Once an offer is created it is in draft state when it's required data is empty. When the user fills the required data with valid information (p.e url field needs to be valid via GET call) the offer passes to LIVE even of the state of it's campaign. An offer in DRAFT state will never be sent to the final page.

- **LIVE**
  An offer is in live state when every mandatory field is filled, valid and saved independently of the campaign status.

- **PAUSED**
  Only a LIVE offer can be paused. When an offer is in PAUSED status it will not be sent to the final page on LIVE campaign.
  Campaigns default offer can not be paused if campaign is in LIVE status. If the campaign is in SCHEDULED status it's default offer can be paused but when campaign reaches the start date will pass to DRAFT if default offer is still as paused.

Offer state diagram:



From	To	Operative
DRAFT	LIVE	Required offer data filled, validated and saved.
LIVE	DRAFT	Saved data has some invalid (empty mandatory) inputs. Shown as ERROR on a live campaign list. If the user fills a field of a live offer with invalid data its offer won't be saved but if the mandatory field is empty theres is no validation and the state passes to DRAFT
LIVE	PAUSED	Offer paused and not running into the campaign.
PAUSED	LIVE	Offer reactivated after successful offer data validation.

Recommendation API

The recommendation api is the module of the solution that serve the content to the widget. In the output side, this service will provide the required information of the campaigns and the information of the user to choose the best offer for each user. In the input side the widget will send events and errors to this API.

Endpoints

- **Get active campaigns by page.** This service is used when the webpage is loaded, the widget will request the active campaigns with offers and the templates. This information is read from the redis database.
- **Get user custom data.** When the widget is found a known user will make a request of the custom order for the offers for that user. This information is also in the redis database and is loaded by the data team.
- **New impression (post event).** Each impression of any offer by the widget will send an event to the recommendation API. This event will go to BigQuery. And will use to update the impression count of each offer by the back office.
- **New click (post event).** Each click in any offer by the widget will send an event to the recommendation API. This event will go to BigQuery. And will use to update the click count of each offer by the back office. The clicks amount reach the goal defined for the offer the widget will stop to impress that offer.
- **New error.** If the widget can not find the default offer's url in the webpage will send an error to the recommendation api. The back office will check the errors of each campaign. If one campaign has 10 or more errors by day will notificate the user and pause the campaign.

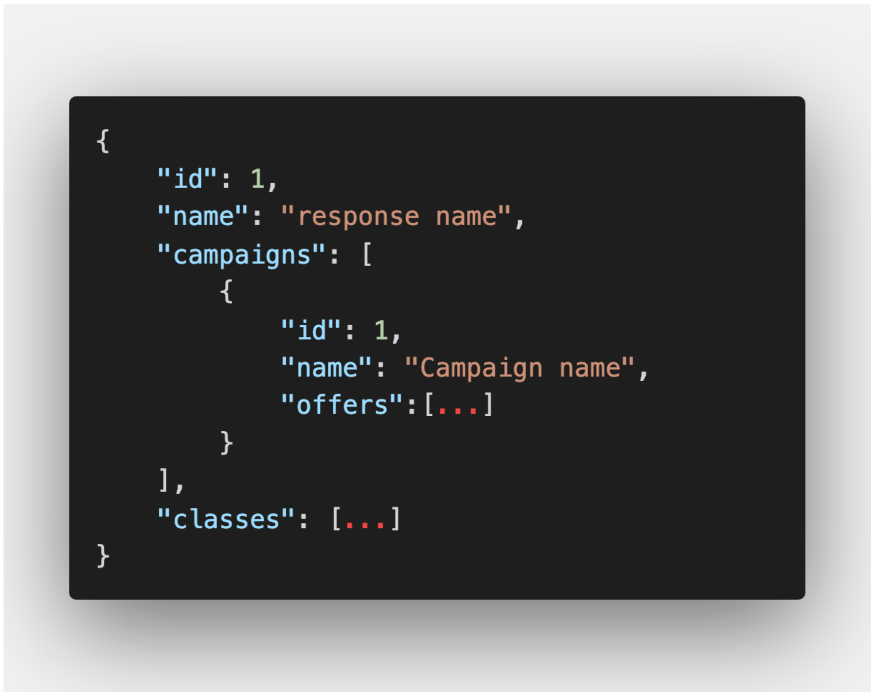
Widget Script (Module Web)

The Bcio widget is one of the two software parts which allows to rate posted/published ads and replace them accordingly.

How does it work?

Once the widget is initialized, it will make a request to the api URL passed in its params, getting all the ads available for the pageID set. This response comes with the next format:

- An array of campaigns available for the pageID passed.
    - An array of offers for that campaign.
- Array of css classes available for the pageID passed.

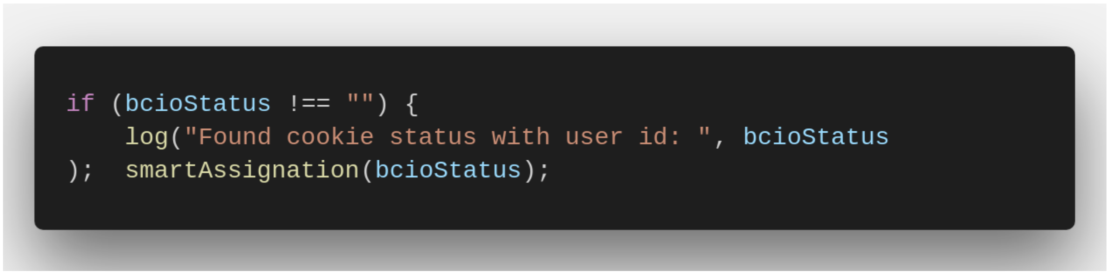


```
{
    "id": 1,
    "name": "response name",
    "campaigns": [
        {
            "id": 1,
            "name": "Campaign name",
            "offers":[...]
        }
    ],
    "classes": [...]
}
```

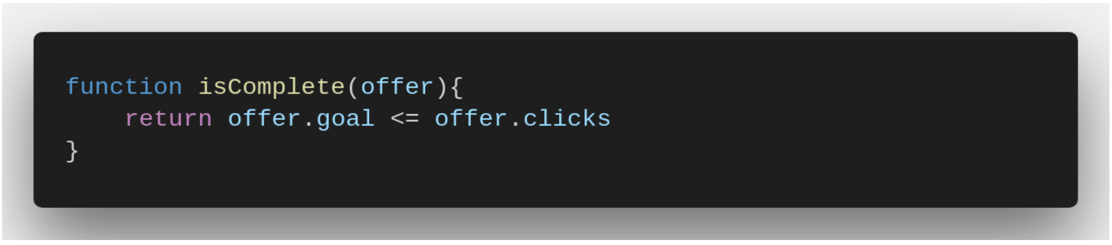
Right after it gets the response mentioned above it will check if the user has visited the site previously checking a cookie that was set before in order to make another request asking which ads are more relatable for this user.

In this cookie evaluation there are several conditionals to contemplate:

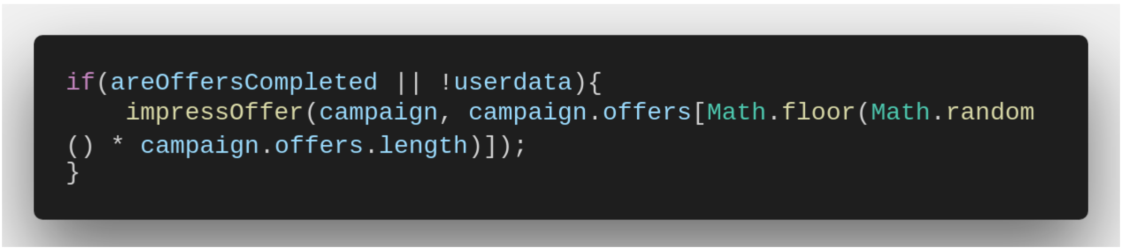
1. If a cookie is found the user goes to a smartAssignation process:
    a. It evaluates the first offer obtained, since they are return in order the first one would be always the more related to the user, if the first offer its completed (has already achieved its goal) or there is not user data stored the widget will render a random offer:



```
if (bcioStatus !== "") {
    log("Found cookie status with user id: ", bcioStatus
);  smartAssignation(bcioStatus);
```



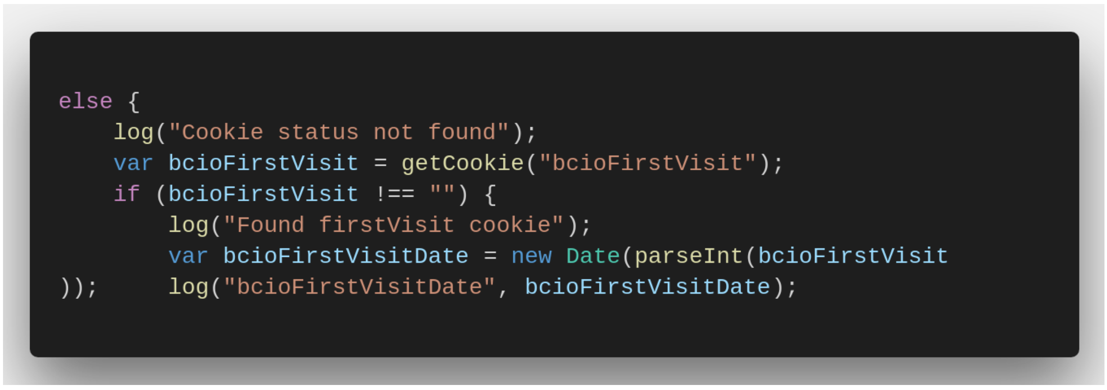
```
function isComplete(offer){
    return offer.goal <= offer.clicks
}
```



```
if(areOffersCompleted || !userdata){
    impressOffer(campaign, campaign.offers[Math.floor(Math.random
() * campaign.offers.length)]);
}
```

    b. If the user data is received successfully and the offers has not achieved its goal the widget will render the first more related to the user.

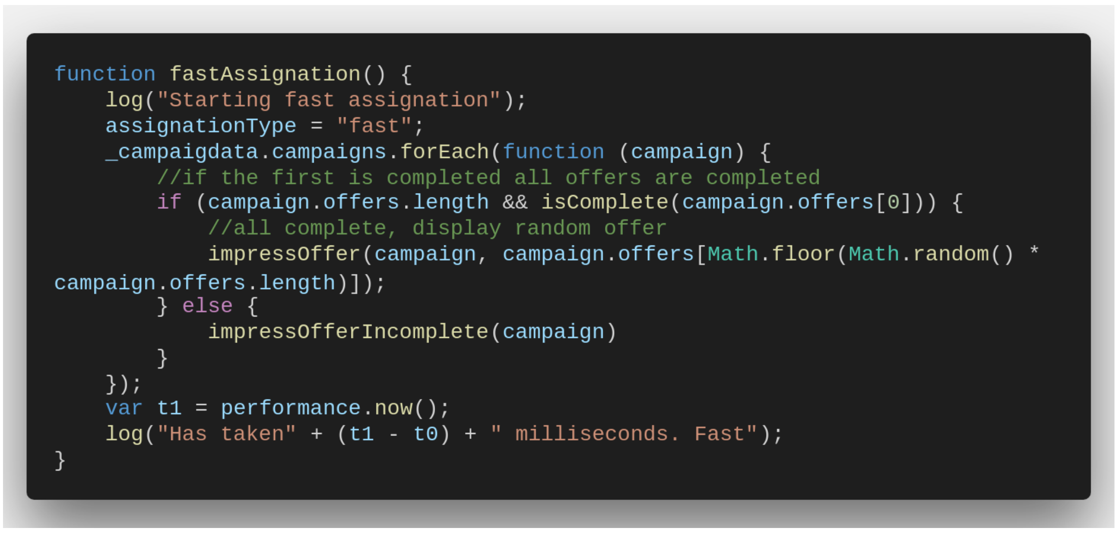
2. On the contrary, if a cookie has not been set previously, it evaluates if the user has the first visit cookie and when it was set.



```
else {
    log("Cookie status not found");
    var bcioFirstVisit = getCookie("bcioFirstVisit");
    if (bcioFirstVisit !== "") {
        log("Found firstVisit cookie");
        var bcioFirstVisitDate = new Date(parseInt(bcioFirstVisit
));     log("bcioFirstVisitDate", bcioFirstVisitDate);
```

a. If the cookie's date is the day before the widget will save the cookie and send the user to the smartAssignation process explained before.
b. If the cookie has been set the same day the user will go to the fastAssignation process. Where if all offers have achieved their goal the widget will render a random offer.

On the contrary, if there are incomplete offers it will use the compensation Index configured to show a random offer or the one that needs more clicks to reach its goal.

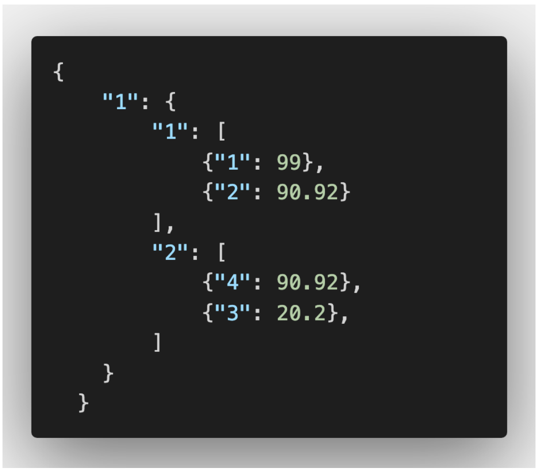


```
function fastAssignation() {
    log("Starting fast assignation");
    assignationType = "fast";
    _campaigdata.campaigns.forEach(function (campaign) {
        //if the first is completed all offers are completed
        if (campaign.offers.length && isComplete(campaign.offers[0])) {
            //all complete, display random offer
            impressOffer(campaign, campaign.offers[Math.floor(Math.random() *
campaign.offers.length)]);
        } else {
            impressOfferIncomplete(campaign)
        }
    });
    var t1 = performance.now();
    log("Has taken" + (t1 - t0) + " milliseconds. Fast");
}
```

There are several steps to render the selected offers, after we evaluate which ones should be shown:

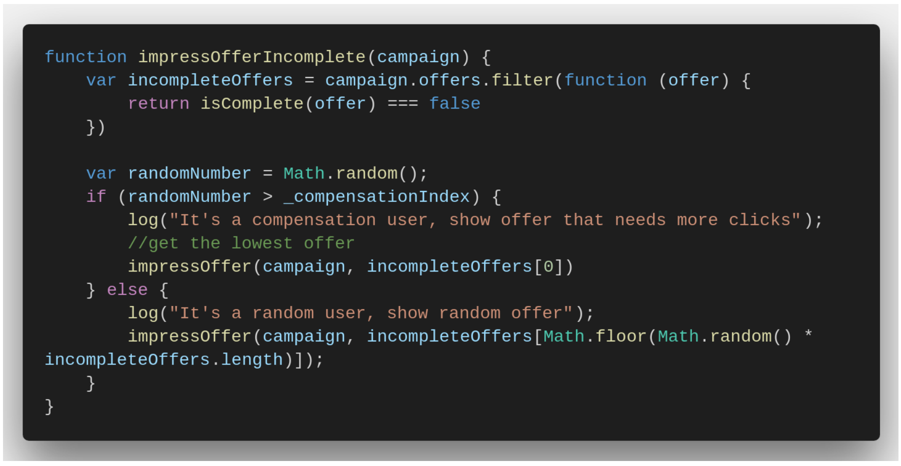
The response with the offers comes with the next format:

- The pageID.
    - Campaign Ids
        - An array of offers ordered by how relatable are they for the user:



```
{
    "1": {
        "1": [
            {"1": 99},
            {"2": 90.92}
        ],
        "2": [
            {"4": 90.92},
            {"3": 20.2},
        ]
    }
  }
```

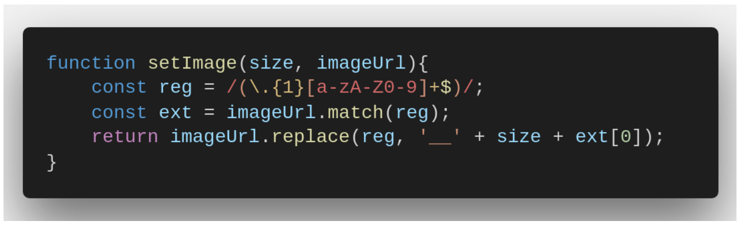
The next step is filtering the offers which have been gotten in the first request and match which one with the most relatable in the second request. Once it gets the offer for a campaign it will look into the page for the default offer through its url, once it finds it, it will search its node parent and replace it with the offer selected, getting first all the offer values to be rendered, like image and template.



```
function impressOfferIncomplete(campaign) {
    var incompleteOffers = campaign.offers.filter(function (offer) {
        return isComplete(offer) === false
    })

    var randomNumber = Math.random();
    if (randomNumber > _compensationIndex) {
        log("It's a compensation user, show offer that needs more clicks");
        //get the lowest offer
        impressOffer(campaign, incompleteOffers[0])
    } else {
        log("It's a random user, show random offer");
        impressOffer(campaign, incompleteOffers[Math.floor(Math.random() *
incompleteOffers.length)]);
    }
}
```

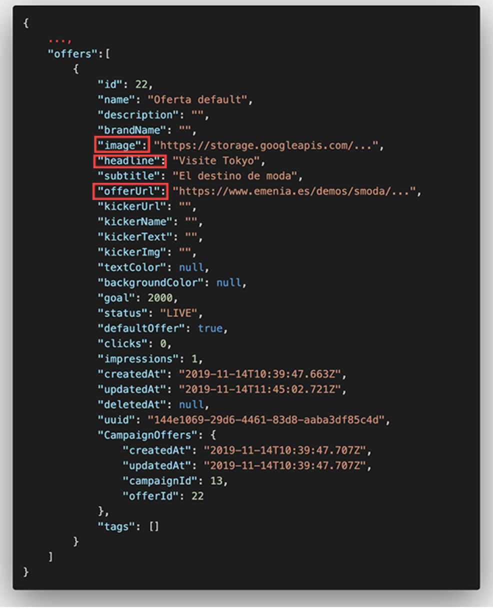
- Image: in order to set the image it will need to match the imgUrl with the corresponding size:



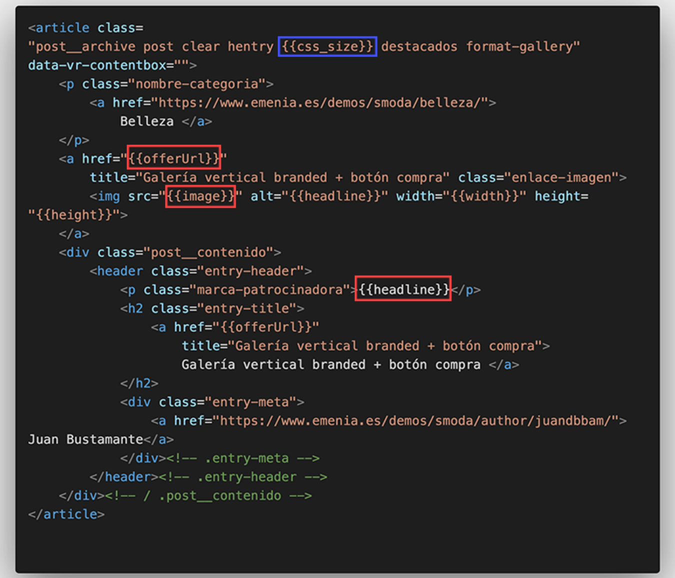
```
function setImage(size, imageUrl){
    const reg = /(\.{1}[a-zA-Z0-9]+$)/;
    const ext = imageUrl.match(reg);
    return imageUrl.replace(reg, '__' + size + ext[0]);
}
```

- Template: in order to choose the template it will need to find the class in the parent node and replace it with the one that comes in the response object.

Finally, to render the offer it will match all the templates params and replace them in the corresponding spot:



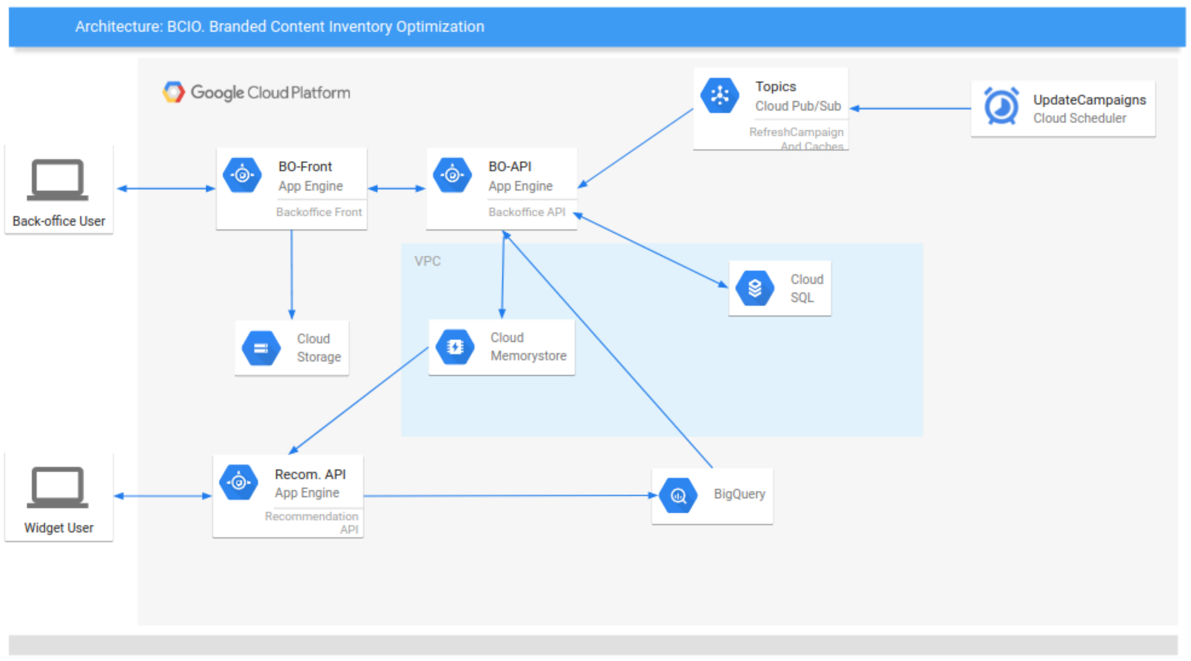
```
{
    ...,
    "offers":[
        {
            "id": 22,
            "name": "Oferta default",
            "description": "",
            "brandName": "",
            "image": "https://storage.googleapis.com/...",
            "headline": "Visite Tokyo",
            "subtitle": "El destino de moda",
            "offerUrl": "https://www.emenia.es/demos/smoda/...",
            "kickerUrl": "",
            "kickerName": "",
            "kickerText": "",
            "kickerImg": "",
            "textColor": null,
            "backgroundColor": null,
            "goal": 2000,
            "status": "LIVE",
            "defaultOffer": true,
            "clicks": 0,
            "impressions": 1,
            "createdAt": "2019-11-14T10:39:47.663Z",
            "updatedAt": "2019-11-14T11:45:02.721Z",
            "deletedAt": null,
            "uuid": "144e1069-29d6-4461-83d8-aaba3df85c4d",
            "CampaignOffers": {
                "createdAt": "2019-11-14T10:39:47.707Z",
                "updatedAt": "2019-11-14T10:39:47.707Z",
                "campaignId": 13,
                "offerId": 22
            },
            "tags": []
        }
    ]
}
```



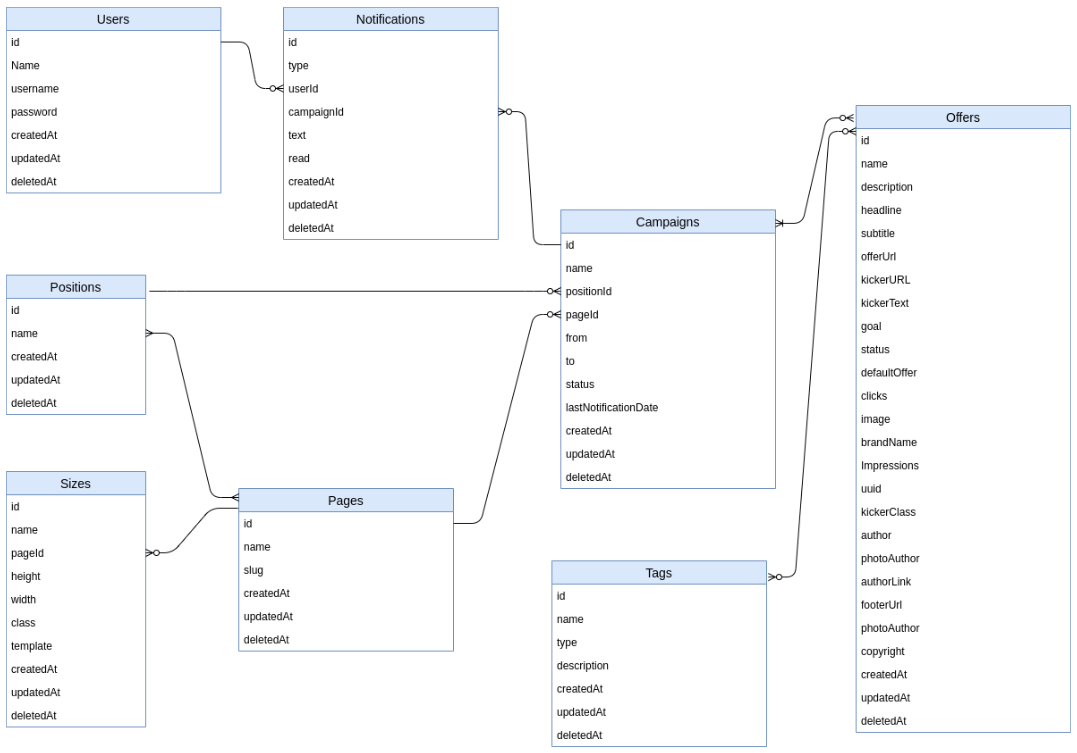
```
<article class=
"post__archive post clear hentry {{css_size}} destacados format-gallery"
data-vr-contentbox="">
    <p class="nombre-categoria">
        <a href="https://www.emenia.es/demos/smoda/belleza/">
            Belleza </a>
    </p>
    <a href="{{offerUrl}}"
        title="Galería vertical branded + botón compra" class="enlace-imagen">
        <img src="{{image}}" alt="{{headline}}" width="{{width}}" height=
"{{height}}">
    </a>
    <div class="post__contenido">
        <header class="entry-header">
            <p class="marca-patrocinadora">{{headline}}</p>
            <h2 class="entry-title">
                <a href="{{offerUrl}}"
                    title="Galería vertical branded + botón compra">
                    Galería vertical branded + botón compra </a>
            </h2>
            <div class="entry-meta">
                <a href="https://www.emenia.es/demos/smoda/author/juandbbam/">
Juan Bustamante</a>
            </div><!-- .entry-meta -->
        </header><!-- .entry-header -->
    </div><!-- / .post__contenido -->
</article>
```

Architecture documentation.

Global map of main components.



Entity Relation Diagram.



Technology Stack (Main components)

- **Back office - Front:**
  - **React:** A JavaScript library for building user interfaces
  - **Jest**: Al JavaScript Testing Framework with a focus on simplicity
  - **Puppeteer**. Headless Chrome Node.js API
  - **Redux**. Predictable state container for JavaScript apps

- **Back office - API / recommendation API:**
  - **Nodejs:** A JavaScript runtime built on Chrome's V8 JavaScript engine
  - **Jest**: A JavaScript Testing Framework with a focus on simplicity
  - **Express**: Fast, unopinionated, minimalist web framework for Node.js
  - **Passport:** A authentication middleware for Node.js
  - **Sequelize:** Sequelize is a promise-based Node.js ORM

GCP Components description

- **App Engine** is the fully managed serverless application platform of Google Cloud Platform and we use as the main process component to execute our code for back office and the recommendation API. The location of this service can not be changed after it is selected so is important double check it to select a zone with VPC connector service. There are 3 services running on App Engine Standard:
  - **Recommendation API**: This microservice provides the needed information for the widget.
  - **Back office Front**: Serving the static code of the frontend
  - **Back office API**: The backend of the backoffice. This component will attend massive requests

- **Cloud SQL** (Postgres). Cloud SQL is the managed service of GCP for SQL database like MySql or Postgres. No ops service with automated backups. This database will be used by the back office so don't need a big instance.

- **VPC** (Default). Virtual Private Cloud for GCP services. The redis and postgres bd will run inside the vpc to block the access from outside. This VPC is created with the project so no manual action is required.
- **Memorystore** (Redis). Cloud Memorystore for Redis provides a fully managed in-memory data store service built on scalable, secure, and highly available infrastructure managed. We use it as a cache to keep the data of the campaigns and users ready to serve to the request from the widget. The Data team will upload the users result of the model in order to show the best content for each user.

- **VPC Connector**

    - **App Engine → VPC**. This connector allows App Engine process to connect to the elements in the VPC like the Redis database.

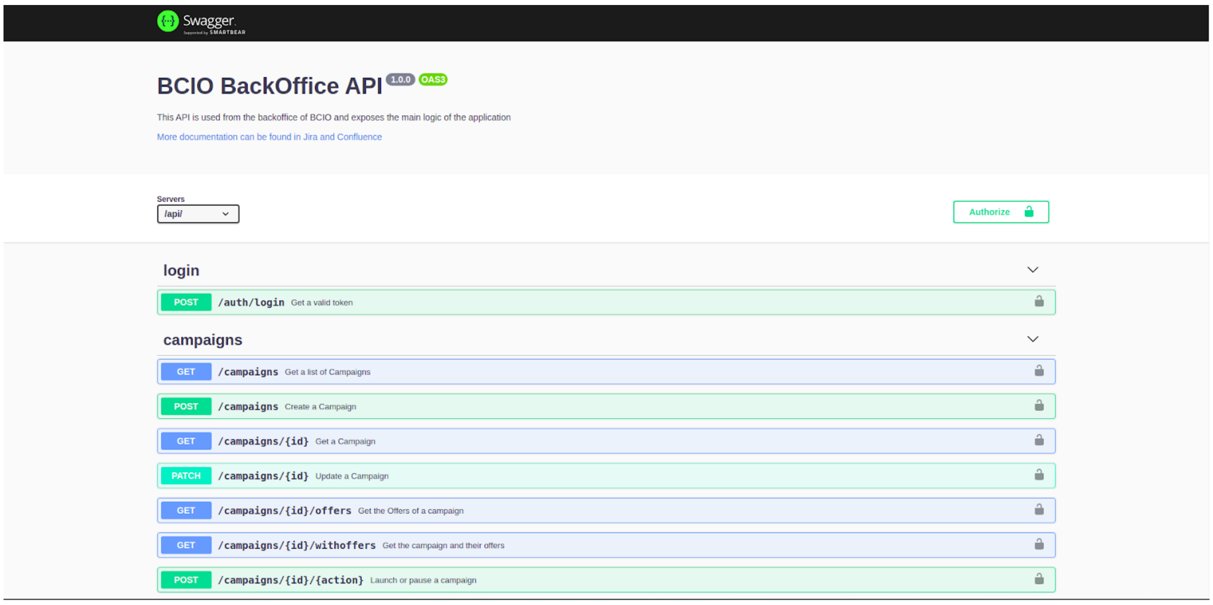
- **PubSub** is shorthand for publish/subscribe messaging, an asynchronous communication method in which messages are exchanged between applications without knowing the identity of the sender or recipient. Serverless product you pay for use
    - **RefreshCache**. In this topic the Scheduler service will publish a new message every 14 minutes. The backoffice service is subscribed and will update the cache in the redis with a TTL of 1200 seconds (20 minutes)
    - **RefreshCampaigns**. In this topic the Scheduler service will publish a new message every 15 minutes. The backoffice service is subscribed and will update the campaigns with the data of the events.
    - **DraftReview**. In this topic the Scheduler service will publish a new message every day at midnight. The backoffice service is subscribed and will check the next day campaign in draft to send notification if necessary.

- **BigQuery** is a serverless, highly-scalable, and cost-effective cloud data warehouse with an in-memory BI Engine and machine learning built in.
    - Event Dataset
        - Event table.
    - Error Dataset
        - Error table.

- **Cloud scheduler** is a fully managed enterprise-grade cron job scheduler. It allows you to schedule virtually any job, including batch, big data jobs, cloud infrastructure operations, and more.
    - Publish **refreshCache**. Each 14 minutes publish in the refreshCache topic
    - Publish **refreshCamapaigns**. Each 15 minutes publish in the refreshCamapaigns topic.
    - Publish **draftReview**. Each midnight publish in drafReview topic.

Swagger.

There is a swagger for no productive environments of the backoffice API module. You can review it easily by going to /api/api-docs/ in the backoffice-api microservice.



BCIO BackOffice API

This API is used from the backoffice of BCIO and exposes the main logic of the application

More documentation can be found in Jira and Confluence

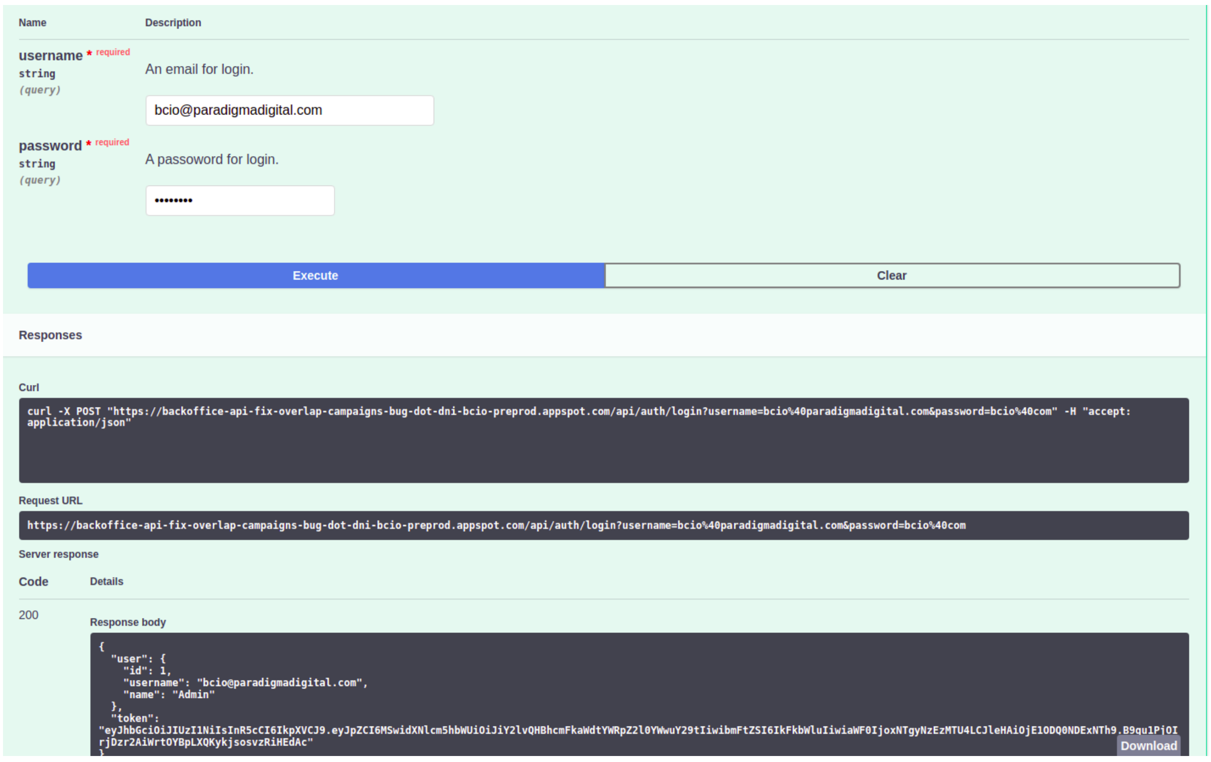
login

POST	`/auth/login`	Get a valid token

campaigns

GET	`/campaigns`	Get a list of Campaigns
POST	`/campaigns`	Create a Campaign
GET	`/campaigns/{id}`	Get a Campaign
PATCH	`/campaigns/{id}`	Update a Campaign
GET	`/campaigns/{id}/offers`	Get the Offers of a campaign
GET	`/campaigns/{id}/withoffers`	Get the campaign and their offers
POST	`/campaigns/{id}/{action}`	Launch or pause a campaign

The auth/login endpoints allow you to get a request token that you can use to authenticate you for the other request.



Name	Description
username * required string *(query)*	An email for login. bcio@paradigmadigital.com
password * required string *(query)*	A passoword for login. ••••••••

Responses

Curl

```
curl -X POST "https://backoffice-api-fix-overlap-campaigns-bug-dot-dni-bcio-preprod.appspot.com/api/auth/login?username=bcio%40paradigmadigital.com&password=bcio%40com" -H "accept:
application/json"
```

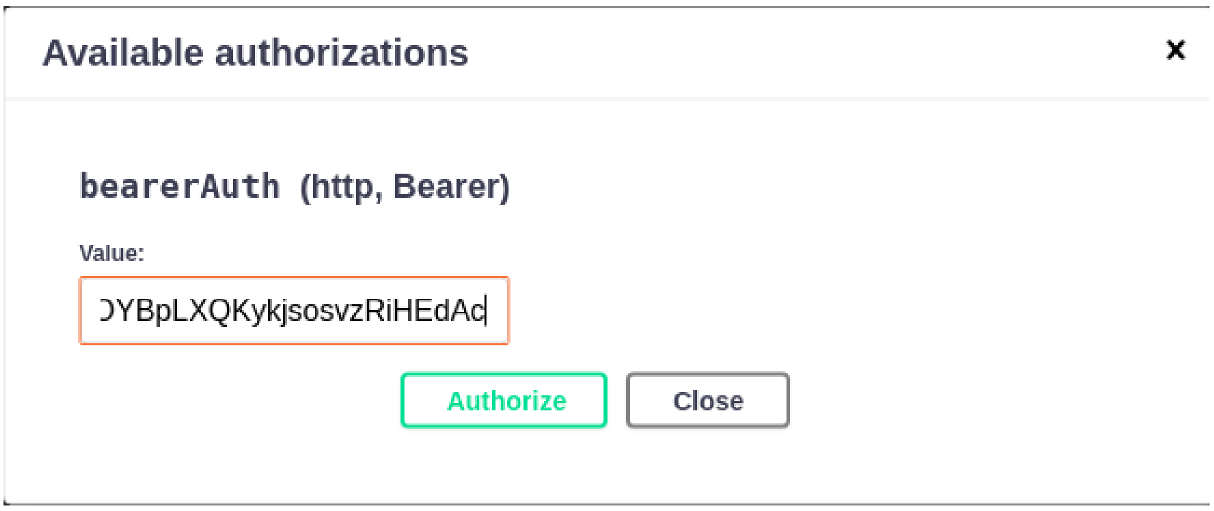
Request URL

```
https://backoffice-api-fix-overlap-campaigns-bug-dot-dni-bcio-preprod.appspot.com/api/auth/login?username=bcio%40paradigmadigital.com&password=bcio%40com
```

Server response

Code	Details
200	Response body

```
{
  "user": {
    "id": 1,
    "username": "bcio@paradigmadigital.com",
    "name": "Admin"
  },
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJiY2lvQHBhcmFkaWdtYWRpZ2l0YWwuY29tIiwibmFtZSI6IkFkbWluIiwiaWF0IjoxNTgyNzEzMTU4LCJleHAiOjE1ODQ0NDExNTh9.B9qu1PjOI
rjDzr2AiWrtOYBpLXQKykjsosvzRiHEdAc"
}
```



Available authorizations

bearerAuth (http, Bearer)

Value:

```
ЭYBpLXQKykjsosvzRiHEdAc
```

Installation

In Google Cloud Platform you can run infrastructure as code. Using terraform the main components of the architecture can be created easily.

Readme.md file in Widget github repository contains the required commands and files to do this:

BCIO project has been designed to deploy in a GCP project. This guide will teach you how to install all the components to have the BCIO solution working.
There are 3 parts in this guide.

1. Platform installation: Using terraform we will set all the infrastructure.
   a. Create project
   b. Activate GCP APIs
   c. Terraform
   d. Plan and Apply
2. Github Three main components:
   a. backoffice-api
   b. backoffice-front
   c. recommendation-api
3. Widget (script) installation.

Platform Installation:

Create GCP project

To execute the BCIO solution you must have a GCP project. If you have not an account you can start here:
https://cloud.google.com/

Activate GCP APIs:

Enter in the project and open the cloud shell (icon on upper right). One you are in the shell follow this steps:

1. Clone this repo:

```
$ git clone https://github.com/BCIO-Project/widget.git
```

2. Ensure we work in the correct project:

```
$ gcloud config set project my-project-id
```

3. Enable the APIs (Be patient, this cloud takes a while...)

```
$ gcloud services enable vpcaccess.googleapis.com redis.googleapis.com
sqladmin.googleapis.com appengine.googleapis.com
```

Terraform

Using docker to help us:

```
$ docker run -v $HOME/bin:/software sethvargo/hashicorp-installer
terraform 0.11.10

$ sudo chown -R $(whoami):$(whoami) $HOME/bin/

$ export PATH=$HOME/bin:$PATH

$ terraform -v
```

Go to the terraform folder inside the clone repo

```
$ cd bcio-widget/terraform
```

Edit in main.th projectId variable. Check other vars to ensure they are Ok.
Initialize terraform and install dependencies

```
$ terraform init
```

Plan and Apply

Run plan to plan the installation

```
$ terraform plan -out "bcio.out"
```

Apply the changes

```
$ terraform apply "bcio.out"
```

When the installation ends you will see on the shell some ENV VARS you will need soon, please write them down. An example of this vars:

```
instance_ips = [
   PLEASE USE THESE ENV_VARS: ,
   DB_HOST_PREPRO: ,
   34.XX.XXX.XXX,
   REDIS_HOST_PREPRO:,
   10.XXX.XX.147,
   BUCKET_NAME_PREPRO:,
   dni-bcio-XXX-images
]
```

Upload a default test code to default service

Go to next dir in the repo

```
$ cd gae-default-service
```

Deploy the example code

```
$ gcloud app deploy
```

Add correct role to img bucket

There is a bug in terraform that only adds the legacy reader role to our bucker, to work properly you should manually look for the bucket in cloud storage and add to all users the role of storage object viewer.

Add user and network to SQL instance

Now you need to go to set the user and password for the SQL user in the GCP.

Update the parametric information.

The solution has the following data that can be modified easily changing the seeds files in the backend-api. These files control the initial data that will be used to populate each new branch.

In the seeders directory we have the following files:

- **initial_tags**: Load the tag information for the system
- **generate-pages**: The pages elements
- **login-user**: The users of the system
- **generate-positions**: The different positions for each page

- **size-generations**: For each page, all the different sizes of the pictures, the class to look at the page and the template to load the new offer.

If it is necessary to upload or new data you can update the database or load new data using sequelize command line.

Widget (script) installation.

Bcio widget is one of the two parts software which allows to rate posted/published ads and replace them accordingly.

Property	Value
pageID	Represents the actual view being rendered, it's needed to know how many ads spots are there.
parentSelector	it's the css class in the parentNode of the ad to be replaced.
api	URL of the backoffice api to make all the requests.

Installation

You will need to import two scripts in the <head> tag which are;

```
<script type="text/javascript" src="/static/bcio-client.js"></script>

<script type="text/javascript" src="/static/bcio-config.js" defer></script>
```

The bcio-config.js file (name is optional) have the init values of the widget. It looks like this:

```
(function(window){
  var options = {
    pageID: 1,
    parentSelector: '.articulo',
    api: 'https://recommendation-api-milestone-3-dot-dni-bcio-int.appspot.com'
  }
  BCIO(options);
})(window)
```

How does it work?

Once the widget is initialized, it'll make a request to the API URL passed in its params, getting all the ads available for the pageID set. Right after it gets the ads it'll check if the user has visited the site previously checking a cookie that was set before in order to show the ads related to the user's interest, otherwise it will show random ads based on the lowest ad's score clicked.

Managing the Database with ORM.

The back office API uses Sequelize as ORM. So we must use it for access and evolve the database. The CLI is called by npx and helps using Sequelize.

Model generation example (Create the initial migration and the model file):

```
npx sequelize-cli model:generate --name Page --attributes name:string
```

Migration generation example:

```
npx sequelize migration:create --name add_slug_to_page
```

Execute all the pending migrations:

```
npx sequelize-cli db:migrate
```

Undo the last migration:

```
npx sequelize-cli db:migrate:undo
```

Create a seed

```
npx sequelize-cli seed:generate --name demo-user
```

Apply all the seeds

```
npx sequelize-cli db:seed:all
```

Apply one seed

```
npx sequelize-cli db:seed --seed ./path_to_file
```

Undo one seed

```
npx sequelize-cli db:seed:undo --seed ./path_to_file
```

Run locally

This project is full cloud and uses several managed cloud services like redis or bigquery, you can run locally to code fast. For use the cloud service first you should set the env var with a service account with the needed roles, please follow this doc: https://cloud.google.com/docs/authentication/getting-started
Let’s see how to do this in the different services:

Back office front end:

In the root folder there is a file called .env that contains the api url REACT_APP_API_BASE=http://localhost:8080/api. You can start the dev environment with the:

```
npm run dev
```

Back office API:

In the file config/dev/.env you can find the vars for local development. Once modified you can start the node process with:

```
npm start
```

Recommendation API:

In the file config/dev/.env you can find the vars for local development. Once modified you can start the node process with:

```
npm start
```

Smart Assignation

Introduction

A personalized content recommendation model was implemented based on the history of frequent user interactions with the content displayed during the last 30 days. For each frequent user, the model is able to predict their interest in each of the new branded content.

Recommendation Model

The purpose of the recommendation model is to suggest relevant new content based on the affinity of the users to previously published content.

The implemented model is composed of two modules:

- Data preprocessing: responsible for getting and transforming the information in order to make it consumable by the machine learning method.

- Smart algorithm: contains the whole machine learning logic and carries out the recommendation process itself.

Below, both blocks are going to be described.

Data Preprocessing

This module performs the task of preparing the information to be used by the smart algorithm and is composed of two steps:

1. *GetInfoAPI*: obtain the information of the active campaigns using one of the services of the Recommendation API module.

2. *Modelo30DaysCampaignInfoCTR*: information from the published campaigns in the last 30 days is crossed with the navigation information of users through the site in the same period time.

GetInfoAPI

This process downloads the information of the active campaigns from the Recommendation API module and stores it in parquet files every day.

To connect the API endpoint to download the data, a token to pass the authentication process is needed. This token needs to be renewed, so every day the first action is to connect the API to get a new token.
The final result of this process is a parquet file with the information of all the active campaigns.

Modelo30DaysCampaignInfoCTR

In this process, the information of the campaigns published in the last 30 days and previously stored in parquet format is crossed with the data about the interactions of every user with those campaigns. The data regarding the interaction of users on the site is obtained from the Event Dataset and the Event Table available in BigQuery.

With this matching it is possible to calculate the click-through rate (CTR) between the clicks and the impressions that every user has done in one offer of a certain campaign. This information will be useful for the smart algorithm.

The CTR is calculated for each campaign-user pair and each of the tags present in the offers, in other words, for each campaign-offer-user-tag combination, there is a record.

On the other hand, for the offers belonging to the current day, it is also done the same separation by tag, although there is no navigation information. In this case, only the separation of the tags in different records is performed.

The results of this process are the following files also stored in parquet format:

- *CampaignInfo30days*: the last 30 days information of the campaigns with the tags separated in different records.
- *newInfo*: the same as the previous file but for the active campaigns in the current day.
- *CTR_tags*: contains the CTR by tag in each offer.
- *userId_tags*: store the CTR for each pair user-tag, for all the users and all the tags.

Smart Algorithm

The algorithm designed is based on the collaborative filtering of user clusters and the prediction of the CTR of new branded contents using the CTR of similar branded contents published in the last 30 days.

Next, the elements involved in the system will be defined and each step of the algorithm will be explained.

Notations

u represents a user.
n is the number of tags available in the system.
CTR_{tag_t} represents the click-through rate of the tag t.
c_i represents a group of users.
k is the number of user groups.
U_{c_i} represents the set of users in group c_i.
b_j represents branded content published in the last 30 days.
m is the number of branded contents published in the last 30 days.

b_{new} is a brand new branded content.
$sim(b_{new}, b_j)$ is a measure of similarity between branded contents.
$w(u,\ c_i)$ is a measure of membership of user u in group c_i.

Algorithm steps

1. Calculation of user preferences attributes.
   Each user is profiled according to the content tags they have visited in the last 30 days. Then, a user u is represented vectorially as follows:
   $$u = (CTR_{tag_1},\ CTR_{tag_2},\ CTR_{tag_3},\ \ldots,\ CTR_{tag_n})\ . \tag{1}$$

2. Segmentation of users based on their preferences.
   User clusters with similar interests are created using the k-means unsupervised clustering method.

3. Calculation of similarity between new and old branded contents using a cosine similarity measure.

4. The interest of each cluster for each old branded content is calculated in the form of CTR using the following equation:
   $$CTR(U_{c_i}, b_j) = \frac{\sum_{p \varepsilon U_{c_i}} clicks_{p,b_j}}{\sum_{p \varepsilon U_{c_i}} impressions_{p,b_j}}\ . \tag{2}$$

5. Calculation of the interest of each cluster for each new branded content in the form of CTR prediction using the following equation:
   $$CTR'(c_i, b_{new}) = \sum_{j=1,m} sim(b_{new}, b_j) * CTR(U_{c_i}, b_j)\ . \tag{3}$$

6. For each user, an interest score is calculated for each new branded content using the previous information as follows:
   $$score(u, b_{new}) = \sum_{i=1,k} w(u, c_i) * CTR'(c_i, b_{new})\ . \tag{4}$$
   This information is loaded in a JSON file in a Redis database. Next, the branded content with the highest score will be the recommended one to show the user.

Summary Description

A summary table is presented below with a brief description of all the steps of the recommendation model and their parameters.

Process Name	Description	Parameters
GetInfoAPI	This process obtains the campaign-offer information from the Recommendation API and is stored	**BCIOBasePath:** the base path, which serves as root for all the other file paths.

	daily in a parquet file.	**API:** the relative path where the campaign-offer information is stored daily. **url_api_token:** the URL of the API endpoint to obtain the token in order to be able to connect to the API. **url_api_campaigns:** URL of the API endpoint where the campaign-offer information is obtained from. **header_auth:** the header needed to make possible the authentication.
Modelo30DaysCampaignInfoCTR	Information from the published campaigns in the last 30 days is crossed with the navigation information of users through the site in the same period time.	**BCIOBasePath:** the base path, which serves as root for all the other file paths. **API:** the relative path where the campaign-offer information is stored daily. **CampaignInfo30days:** the relative path to store the file with the information of each campaign-offer-tag combination for the published campaigns in the last 30 days. **newInfo:** the relative path to store the file with the information of each campaign-offer-tag combination for the active campaigns in the current day. **CTR_tags:** the relative path to store the file with the CTR for each offer-tag combination. **userId_tags:** the relative path to store the file with the CTR values for each user-tagl combination. **project_id:** name of the project in BigQuery. **dataset:** dataset where the table with the user navigation data is stored. **table:** name of the table with the user navigation data. **gs_bucket:** Google Storage bucket where the library used to read from BQ writes its temporary information. **location:** the location of the project.
FormattingData	Each user is represented by a vector of preferences.	**BCIOBasePath:** the base path, which serves as root for all the other file paths. **userVStagsPath**: the relative path to store the file with the CTR values for each user-tagl combination. **formatedDataPath**: the relative path to store the file with the vector representation of users.
TrainClustering	User clusters with similar interests are created using the k-means unsupervised clustering method.	**BCIOBasePath:** the base path, which serves as root for all the other file paths. **numCluster**: number of clusters to create. **userVStagsPath**: the relative path to store the file with the vector representation of users. **modelPath**: the relative path to store the trained clustering model.
PredictClustering	Each user is assigned their cluster.	**BCIOBasePath:** the base path, which serves as root for all the other file paths. **userVStagsPath**: the relative path to store the file with the vector representation of users. **modelPath**: the relative path to store the trained clustering model.

		preditedDataPath: the relative path to store the file with the vector representation of users and their assigned clusters.
ContentSimilarity	Calculation of similarity between new and old branded contents using a cosine similarity measure.	**BCIOBasePath:** the base path, which serves as root for all the other file paths. **oldContentsPath**: the relative path to store the file with the information of each campaign-offer-tag combination for the published branded content in the last 30 days. **newContentsPath**: the relative path to store the file with the information of each campaign-offer-tag combination for the active branded content in the current day **similarityPath**: the relative path to store the file with the similarity between new and old branded contents.
CalculateCTR	The interest of each cluster for each old branded content is calculated in the form of CTR.	**BCIOBasePath:** the base path, which serves as root for all the other file paths. **ctrPath**: the relative path to store the file with the CTR for each previously published branded content. **clusteredDataPath**: the relative path to store the file with the vector representation of users and their assigned clusters. **ctrByClusterPath**: the relative path to store the file with the CTR of each old branded content in each cluster.
CalculateScore	Calculation of the interest of each cluster for each new branded content in the form of CTR prediction. For each user, an interest score is calculated for each new branded content. This information is loaded in a JSON file in the Redis database.	**BCIOBasePath:** the base path, which serves as root for all the other file paths. **modelPath**: the relative path to store the trained clustering model. **clusteredDataPath**: the relative path to store the file with the vector representation of users and their assigned clusters. **ctrByClusterPath**: the relative path to store the file with the CTR of each old branded content in each cluster. **similarityPath**: the relative path to store the file with the similarity between new and old branded contents. **redisIP**: redis IP address. **redisPort**: redis port. **ttl**: the validity time of the keys in the JSON file.

All the parameters mentioned above must be defined in a configuration file called *BCIO.properties*. The *BCIO.properties* file is required to run the Recommendation Model.

Infrastructure

This section details the steps followed for the installation and configuration of the necessary elements for the operation of BCIO scripts in Github, as well as the deployment of machines and data in Google Cloud Platform (GCP).

GCP Project: DNI BCIO PRO

Service Account Creation

(Location: GCP | IAM & Admin / IAM)
First, a new Service Account (SA) must be created and assigned the required roles for operation.

Created SA:

bcio-runner-pro@dni-bcio-pro.iam.gserviceaccount.com

Roles:

- *Dataproc_run_role*
- *github bigquery role*
- *Infraestructura_role*
- BigQuery User
- Dataproc Worker
- Service Account Token Creator
- Service Account User
- Viewer

The first three are custom roles that have been assigned the following permissions:

- *infraestructura_role*

compute.regions.list
dataproc.clusters.create
dataproc.clusters.delete
dataproc.clusters.get
dataproc.clusters.list
dataproc.operations.get
Dataproc.operations.list

- *dataproc_run_role*

dataproc.agents.create
dataproc.agents.get
dataproc.agents.update
dataproc.clusters.use
dataproc.jobs.cancel
dataproc.jobs.create
dataproc.jobs.delete
dataproc.jobs.get
dataproc.jobs.list
dataproc.jobs.update
dataproc.operations.delete
dataproc.operations.get
dataproc.operations.list
dataproc.tasks.lease
dataproc.tasks.listInvalidatedLeases
dataproc.tasks.reportStatus
dataproc.workflowTemplates.create
dataproc.workflowTemplates.delete
dataproc.workflowTemplates.get
dataproc.workflowTemplates.instantiate
dataproc.workflowTemplates.instantiateInline
dataproc.workflowTemplates.list
dataproc.workflowTemplates.update
storage.buckets.get
storage.objects.create
storage.objects.delete
storage.objects.get

storage.objects.list

- *github_bigquery_role*

bigquery.config.get
bigquery.datasets.get
bigquery.datasets.update
bigquery.jobs.create
bigquery.jobs.get
bigquery.jobs.list
bigquery.jobs.listAll
bigquery.jobs.update
bigquery.savedqueries.create
bigquery.savedqueries.delete
bigquery.savedqueries.get
bigquery.savedqueries.list
bigquery.savedqueries.update
bigquery.tables.create
bigquery.tables.delete
bigquery.tables.export
bigquery.tables.get
bigquery.tables.getData
bigquery.tables.list
bigquery.tables.update
bigquery.tables.updateData
bigquery.transfers.get
bigquery.transfers.update
resourcemanager.projects.get
resourcemanager.projects.getIamPol
icy

Runner Machine Creation

(Location: GCP | Compute Engine / VM instances)

A Virtual Machine (VM) is created in GCP to pick up and execute jobs. Therefore, this machine will be the Runner for the project.

The resources of the VM are low since this machine is only in charge of executing the script that creates the *dataproc* cluster. The details of the VM are shown below:

VM name:	instance-github-runner
OS image:	Ubuntu 18.04 LTS
Machine type:	n1-standard-1 (1 vCPU, 3.75 GB memory)
Service Account:	SA
Zone:	europe-west1

Github Project Files

- bcio.sh

The values defined in the BCIO.properties file are used as parameters for this script. Besides, if the date value is empty, it calculates the current date and runs process specified in the source parameter.

Inputs:

- source: name of the process to execute
- jarpath : path of the JAR file
- cluster: name of the created cluster in dataproc.sh script
- region: zone where the cluster is located
- fechaParam: execution date. By default it is empty and is defined in the schedule.

- BCIO.properties

Properties file for each process to run.

Appendix

Enable “Cloud Dataproc API”

If the following error appears, it is necessary to enable the *cloud dataproc API* from GCP/PRODUCTS/APIs & Services/Dashboard/Cloud Dataproc API/Overview:

> *“ERROR: (gcloud.beta.dataproc.clusters.create) PERMISSION_DENIED: Cloud Dataproc API has not been used in project xxxxxxxxxxxxxx before or it is disabled. Enable it by visiting https://console.developers.google.com/apis/api/dataproc.googleapis.com/overview?project=xxxxxxxxxxxx then retry. If you enabled this API recently, wait a few minutes for the action to propagate to our systems and retry.”*