

# Academic Platform for ONline Experiments (APONE)

USER GUIDE

TU Delft, February 2018

# INDEX

<b>INDEX</b>	<b>2</b>
<b>1.INFORMATION RETRIEVAL AND A/B EXPERIMENTS</b>	<b>3</b>
<b>2. APONE OVERVIEW</b>	<b>7</b>
<b>3. TECHNICAL DETAILS</b>	<b>8</b>
APONE 1.0.0	8
Web GUI	9
ClientE 1.0.0	9
<b>4. HOW TO CREATE AN EXPERIMENT</b>	<b>11</b>
Step 1. Create and Start the Experiment in APONE	12
Step 2. Install and Deploy the Client	14
Step 3. Test your Client	15
Step 4. System in Production	16
<b>5. HOW TO REGISTER EVENTS</b>	<b>18</b>
<b>6. WEB INTERFACE USER REFERENCE</b>	<b>21</b>
6.1 Experiment::Manage	21
Start/Stop an experiment	21
6.2 Experiment::Create New	22
6.3 Experiment::Create New (Advanced)	23
6.4 Events::Manage	24
6.5 Monitoring::Experiments	25
6.6 Monitoring::Users	27
<b>8. ADVANCED EXPERIMENTS</b>	<b>28</b>
8.1 MORE CONTROL IN OUR EXPERIMENT	28
8.2 PLANOUT	29
PlanOut in APONE	31
<b>9. RELEVANT REFERENCES</b>	<b>33</b>
Surveys	33
Tips	33
Experiment Life-Cicle	33
Interleaving	33
Large-scale	34
PlanOut	34

# 1. INFORMATION RETRIEVAL AND A/B EXPERIMENTS

In this section, we will introduce very briefly how Information Retrieval systems are evaluated, and we will compare the process followed in traditional evaluation with the one followed in A/B experiments.

Information Retrieval has been traditionally evaluated through *offline evaluation*, following the so called *Cranfield Paradigm*, because of the Cranfield projects initiated in 1957 by Cleverdon, which provided the foundations for the evaluation of IR systems. This paradigm consists on evaluating the results of the systems against a) a document collection, b) a predefined set of information needs or topics, c) the relevance judgements of the documents in the document collection in relation to the queries, and d) measures to compare the results (ranking and relevance of documents) obtained by the systems for the set of queries.

This approach provides a way to evaluate with reliable judgments, collected from experts or using crowdsourcing techniques, but it has two major disadvantages: the cost to obtain such judgments, and the lack of a user model that may interfere in the results obtained (eg. interaction, personalized results, impact of interface changes).

In order to include the user in the evaluation, two approaches have been used: *interactive evaluation* in controlled environments, and *online evaluation*. In the first case, a group of users interact with the systems in a controlled environment. In the second approach, the users are *exposed* to the systems with no idea that an experiment is running.

All of them are controlled experiments, and they aim to test one or more hypotheses. In offline experiments the aim is to test if the results obtained by the systems are good enough to decide if one system is better than another. In the case of interactive and online experiments, the aim is to test if the changes in user behaviour have been caused by changes to the IR system (be them in the back-end or the front-end). In both cases though, we try to statistically prove that one system is better than other, and the dependent variable that measure the effect of the changes is known as *Overall Evaluation Criteria (OEC)*.

*A/B testing* is one specific type of online experiment where users are confronted with two approaches, usually a new approach (*treatment*), and an existing approach (*control*). Each user is typically exposed to one of the *variants* (treatment or control), and his behaviour is recorded in order to decide which one has been more successful. The *experimenter* has to define and implement the metrics to measure user behaviour (eg. clicks, dwell-time, time to click, number of queries), decide how to aggregate them in an OEC to test the hypothesis, and estimate the required sample size, conditioned by the OCE. Therefore, the experimental design should be planned together with the methodology for data analysis.

It is also relevant to note that the entity which is assigned to a treatment or a control is called the *experimental unit*. It determines the granularity at which the experimental conditions are compared (eg. the clicks, the number of queries). Although the experimental unit is typically the user, depending on the experiment we could have smaller units (eg. queries, user-day) or larger (eg. network of users). In general, we will assume that the experimental unit is the user in the examples contained in this document unless otherwise stated.

It is also important to note that the assignment of variants to the experimental unit has to be random in order to guarantee the validity of the experiment, and the same experimental unit is always assigned to the same variant (usually it is desirable that the same user always interacts with the same variant during the experiment, specially if changes in the interface are involved).

Let's see an example. Suppose we are working on a new ranking algorithm B for our search engine, and we want to test if it is actually better than the state-of-the-art ranking algorithm A. We can follow different evaluation methods:

### Traditional offline evaluation

1. Get data: get a test collection, that is, a collection of documents, queries, and judgments of relevance over these documents for those queries. The latter information is typically a tuple of three values: `query_id`, `document_id`, and `relevance_document_query`. The [Cranfield Collection](#) is an example of such a test collection. More examples can be found in [Text REtrieval Conference \(TREC\)](#).
2. Index documents and run queries: index the collection of documents in our system, launch the queries and save the ranking of documents obtained by systems A and B (these are called *runs*). Typically the results are saved as tuples of four values: `system_id`, `query_id`, `document_id`, and `relevance_document_query`.
3. Select an Overall Evaluation Criteria: select a specific OEC to compare results, for example, mean reciprocal rank (mean of the inverse of the rank of the first relevant document obtained for each query).
4. Analyse results: calculate the results of both systems for the OEC selected in relation to the relevance judgments we have. For example, we could see that system A obtains a mean of  $1/3$  (on average, the first relevant document is in the third position), meanwhile system B obtains a mean of  $1/2$  (on average, the first relevant document is in the second position)<sup>1</sup>. As a conclusion, and in relation to the test collection and metric used, system B improves system A.

### A/B testing

---

<sup>1</sup> These results can be calculated automatically for several standard metrics using the `trec_eval` tool located at [http://trec.nist.gov/trec\\_eval/index.html](http://trec.nist.gov/trec_eval/index.html)

1. Get Data: get a collection of documents.
2. Index documents: Index the collection of documents in the search engine.
3. Develop a client: implement a client that let the users make queries. Identify the experimental unit and implement a random algorithm to assign algorithm A or B to each unit. In this case, the experimental unit would typically be the user, so the client must identify the user and assign algorithm A or B in her searches. Provide means to automatically register the behaviour of the user that can be used to calculate the Overall Evaluation Criteria selected.
4. Select an Overall Evaluation Criteria: what are the interactions of the user that could indicate that one ranking algorithm is better than the other? Several metrics could be used, for example the click-rank (average positions of documents clicked on the search results page), where the most common variant is the mean reciprocal rank (average of the position of the first document *clicked*<sup>2</sup>).
5. Analyse results: after running the experiment and saving the results, we calculate the OEC for the group of users exposed to A, and for the group of users exposed to B, and compare the mean reciprocal rank obtained as we explained in offline evaluation.

APONE reduces the time needed to setup an A/B experiment by reducing the work to develop a client (step 3 above). For example, if the experimental unit is the user, the experimenter just has to determine what are the web pages displayed for control and treatment. In the previous example, we could develop a web page that will be displayed when the user is assigned to the control (ranking A), and a different one that will be displayed when he is assigned to the treatment (ranking B). The platform will redirect each user (according to his unique identifier) to the appropriate URL. The platform guarantees that the assignment of users to variants is random. It also offers the means to monitor and control the running experiments, and endpoints to save and retrieve the information we will have to analyze in step 5.

Finally, we should make sure that the results obtained are statistically significant. We could compare both algorithms with only three users in A/B testing, or only three queries in offline evaluation, but would those results be reliable? In the case of online experiments, the monitoring of the OEC during the experiment could help the experimenter decide when to stop.

It is also recommended to take into consideration possible external elements that may condition the behaviour of the user. For example, we could pay attention to the date and/or to the

---

<sup>2</sup> Note that we are assuming that if the user clicks on a document, then that document is relevant. Usually, several metrics are combined in an OEC, for example, the click-rate plus the dwell-time, which measures how much time a user spends in a document. Those two metrics have to be then aggregated in some way, and this is one of the research problems in online evaluation in general.

user-agent information saved in the event. Sometimes the behaviour (or even the users themselves) are different during weekdays than in the weekends, the use of a mobile device or a different browser may also change completely their behaviour, and the first time a user access the experiment may have a behaviour completely different (novelty effect). For example, a bad search engine may receive more queries than a good one at first because the users do not find what they are looking for!

## 2. APONE OVERVIEW

The *Academic Platform for ONline Experiments* (APONE, or just *platform*) aims at the evaluation of new approaches in interfaces and algorithms, by means of the interaction of users in online real environments. These new approaches (*treatments*) could affect the user interfaces as well as the back-end algorithms or a combination of both, and the user interactions could be compared with those obtained from existing approaches (*control*).

The objective of the platform is to speed up the setup of controlled A/B experiments on the Web, as well as to keep a shared resource of methods and data that can help in research. APONE builds upon [PlanOut](#) to define experiments, and offers a web GUI to easily create, manage and monitor them.

A second component is the client, accessed by the users (directly or via redirection through the experimental platform) and developed by the experimenter, which will interact with the RESTful web services of the platform directly to i) get the variant (treatment or control) and parameters (optionally) corresponding to the user, and ii) register the events that occur during the user interaction. The analysis of this information will allow the experimenter decide which treatment is best.

In order to show the capabilities of the platform as well as to ease the development of such a client, an example is also provided (Client Example --ClientE--).

APONE is not restricted to any specific domain of research. The domain of the experiments defined and run is actually given by the client linked to it, which determines the experience of the user. Nonetheless, the provided ClientE and most examples in this user guide are limited to the Information Retrieval domain.

The purpose of this guide is to explain how to define, run and control an experiment. We assume that the experimenter has online access to APONE. It is also possible to download and install the platform following the installation instructions in github.

### 3. TECHNICAL DETAILS

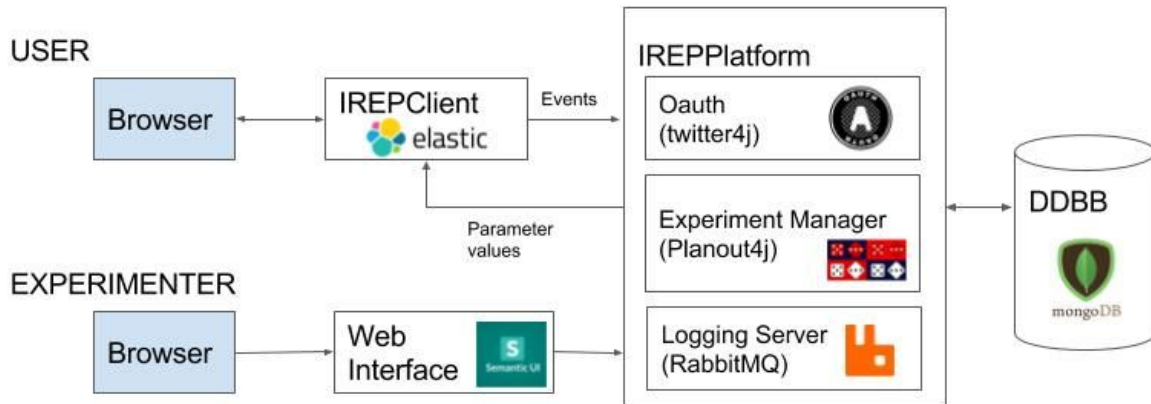


Figure 1. IREPlatform. Main Components and Technologies

#### APONE 1.0.0

RESTful Web Services developed in Java ([Jersey](#)) to be deployed in [Tomcat](#) (at least 8.5). Requires connection to [RabbitMQ](#) and [MongoDB](#) services. It uses [Twitter4j](#) as a library to delegate Twitter to grant access to the users via Oauth, and [PlanOut4j](#) as a library to randomize the assignment of experimental units to variants.

RabbitMQ is a message broker in charge of handling and saving in MongoDB the events sent by the client. This way, the client does not have to wait for the platform to take care of the events. At the same time, the message broker keeps control of the number and type of events registered, which is useful for monitoring the experiments.

MongoDB is a NoSQL database that stores the definition of the experiments, their configuration and status, and the related events sent by the client and handled by RabbitMQ. The events are saved as JSON objects, with predefined properties (see section 6.4). Depending on a property that identifies the data type of the information to be saved, it will be saved as a string, binary or JSON object. This is where the flexibility of MongoDB comes into play. Nonetheless, it is possible to change the database used by implementing the interface `tudelft.dds.irep.data.database.Database` in the platform.

The [PlanOut library](#) (or its Java port *PlanOut4j*) offers an open source framework to define complex experiments, guaranteeing the randomization of the variant assignment that controls



the user experience. It uses a domain-specific language (see section 8.2) to define parameters whose values will change per variant. Its use is optional when we create an experiment in the platform. The library and the language are well documented, with user guides and scientific papers (see section 9), and different tools that support it (eg. online editor <http://planout-editor.herokuapp.com/> ). Therefore, the use of this library makes it easier to reproduce the experiments.

## Web GUI

Web GUI included in APONE project. It uses [JSP](#), and requires [Semantic-UI](#) and [jQuery](#). It supports the definition, visualization and monitoring of the experiments, as well as the visualization of the events and users of the platform.

Semantic-UI is a framework to ease the development of web interfaces. For this project, only CSS and JavaScript libraries on the client-side have been used.

## ClientE 1.0.0

The experimenter can follow two main approaches to implement a client. The simplest way would be to indicate a different URL for each variant when defining the experiment, so the user is redirected from the platform to the appropriate URL. In this scenario, the experimenter has to provide the client hosted at that URL (see Figure 2).

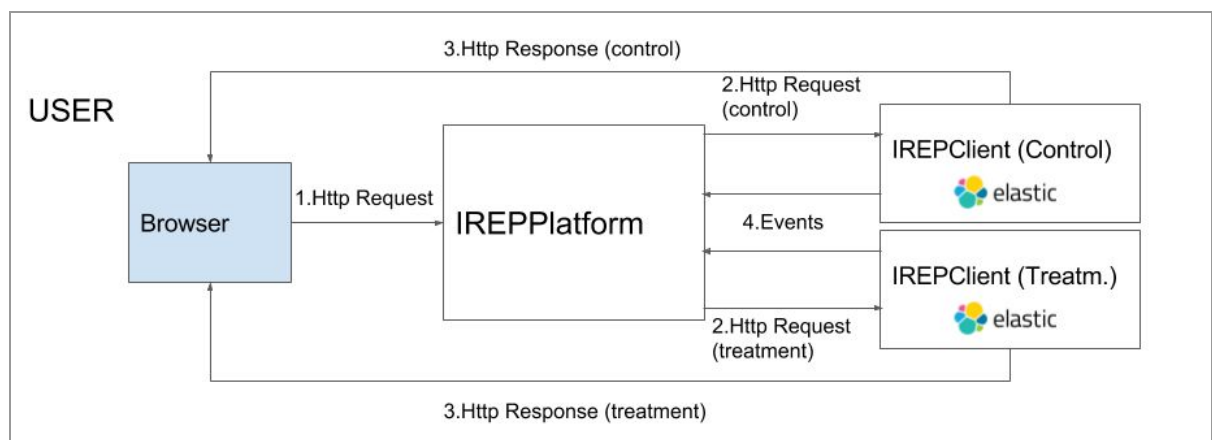


Figure 2: the users access the experimental platform from the browser, and it redirects them to the different variants (same user is always redirected to the same variant).

Another approach would be to develop the client, which asks to the platform if a user should receive control or treatment. In this case, the user would access directly the website developed by the experimenter, which is responsible of loading or redirecting to the appropriate interface (see figure 3).

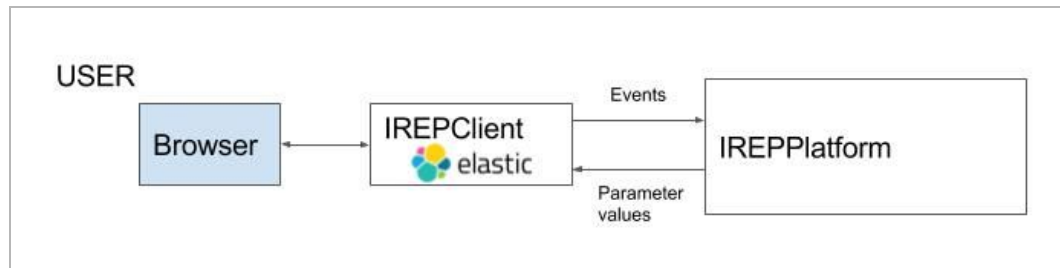


Figure 3: the users access a specific web service developed by the experimenter, and this service is responsible of asking the platform what variant should be displayed to the user, and load/redirect it.

In both cases, control and treatments may have associated pairs parameter-values by using PlanOut scripts (see section [PlanOut](#)). You can read more about this second approach as well as about the use of PlanOut in section 8 (Advanced experiments).

For demonstration purposes, in section 4 we will show how to implement two different experiments using the first approach with ClientE (figure 2). We will use [ElasticSearch](#) as search engine, and we will use the Cranfield documents to set-up those experiments: one which tests changes in the interface (different color in the links of the search results), and another one which tests changes in the ranking algorithm (default ElasticSearch ranking algorithm versus BM25). This client, in HTML and Javascript, will make use of the platform endpoints to register the events, so the experimenter can analyze the results later and decide the best variant according to the behaviour of the users exposed to them.

ElasticSearch is a distributed and open source Search Engine based on [Apache Lucene](#). It has been implemented in Java and provides a RESTful interface.

## 4. HOW TO CREATE AN EXPERIMENT

In this section, we will see how to create and run an experiment using ClientE as example, running locally in your computer. We will assume that APONE is running at <http://ireplatform.ewi.tudelft.nl:8080/IREPlatform>

ClientE serves as an example for two different experiments at the same time. One of them implies changes in the front-end (changes in the color of the links of the search results). We will call it *ColorExp*. The other one involves changes in the back-end (changes in the ranking algorithm used). We will call it *RankingExp*.

In *ColorExp* we want to test if it would be better to change the color of the links of the search engine results from blue to green. In this case, we are going to assume that the *Click-through Rate (CTR)* over the first search results page is a good indicator of the success of the new approach (green color) with respect to the previous one (blue color)<sup>3</sup>. We will consider as experimental unit a user, that is, each user will always receive the same variant (control:blue color, or treatment:green color). In order to evaluate the results of our experiment, we will work with one value per user: the mean CTR for all the queries made by that user during the experiment.

In *RankingExp* we want to test if our new ranking algorithm (let's suppose we are the proud inventors of the Okapi BM25 ranking algorithm, and we make it work with Elasticsearch) is better than the default algorithm implemented in Elasticsearch. In this case, we will assume that the *Reciprocal Rank (RR)* is a good indicator of the success of our BM25 with respect to the default ranking algorithm, and, as previously explained in *ColorExp*, our experimental units are the users. At the end, we will compute an average per user, obtaining the *Mean Reciprocal Rank (MRR)*: the mean of the RR over all the queries made by that user during the experiment.

Note that, depending on the decisions you make at this point, the design will be affected: the Overall Evaluation Criteria you are going to use will determine the information (events) that need to be registered, as well as the number of different users we will need in order to get statistically significant results.

### Step 1. Create and Start the Experiment in APONE

In order to use the services of the platform, for example to register events, the client needs to communicate with an experiment defined and running in APONE. We will see here how to create our own experiments *ColorExp* and *RankingExp*.

---

<sup>3</sup> This is a simplification. Usually different metrics are combined to get a more reliable *Overall Evaluation Criteria*. In Hofmann et al. (see [references](#)) there is a extensive list of metrics that could be used in this kind of IR experiments.

First, you have to access to the platform (<http://ireplatform.ewi.tudelft.nl:8080/IREPlatform>) and log-in with a twitter account. Once you are authenticated by Twitter, you will be redirected to the platform (this may take a while).

To create *ColorExp* we go to the interface *Experiment::Create New*, fill the information, and press the button *Add*. You will see that the *Experimenter* field is already set with the user name in Twitter you used to access the platform. In figure 4 you can see an example of the information to fill in (go to section 4 to see an explanation of each of the fields of the form).

The screenshot shows the 'Create New Experiment' form in the IREPlatform. The form is organized into three main sections: **Experiment**, **Variants**, and **Configuration**.

- Experiment Section:**
  - Experiment:** A text input field containing 'Color'.
  - Experimenter:** A text input field containing 'socialdatadelft'.
  - Description:** A text area containing 'Experiment to test the change in the color of the links resulting from a search. Search engine used: ElasticSearch, ranking algorithm: default, collec'.
- Variants Section:**
  - Two rows of variant information:
    - Variant:** 'Blue' (Control) and 'Green' (Treatment).
    - Description:** 'Search result links in blue.' and 'Search result links in green.'
    - Client URL:** Both variants share the same URL: 'http://ireplatform.ewi.tudelft.nl:8080/IREPClient-simple-/col'.
    - Is Control:** A checkbox that is checked for the 'Blue' variant and unchecked for the 'Green' variant.
- Configuration Section:**
  - Configuration:** A text input field containing 'test'.
  - Date to end:** A date picker field.
  - Max.Units:** A text input field.
  - Below these fields, there are two rows for 'Variant' (Blue and Green) with 'Units (%)' set to 50.

Figure 4. Example of definition of ColorExp.

The most important part here are the client URLs. We will add the following URLs:

Blue (control):

<http://localhost:8080/ClientE/WebContent/color.html?rankingAlg=default&linkColor=blue>

Green (treatment):

<http://localhost:8080/ClientE/WebContent/color.html?rankingAlg=default&linkColor=green>

Note that in both cases we are using the same URL, but with different values for the query parameters *linkColor*<sup>4</sup>. This information will be used by the client to decide what to do (in this case, set the color of the search results to blue or green).

To create *RankingExp* we follow a similar approach, but this time the client URLs are:

Default (control):

<http://localhost:8080/ClientE/WebContent/ranking.html?rankingAlg=default&linkColor=blue>

<sup>4</sup> The platform does not have any restrictions regarding the URLs that can be used. You could use, for example, different URLs for each variant in other experiments, with or without query parameters.

BM25 (treatment):

<http://localhost:8080/ClientE/WebContent/ranking.html?rankingAlg=bm25&linkColor=blue>

Note that this time the query parameter whose value changes is *rankingAlg*, to indicate the client to use the default ranking algorithm in ElasticSearch, or to use BM25 instead. In figure 5, you can see an example of the information to fill in.

The screenshot shows the 'RankingExp' definition form. It includes fields for 'Experiment' (Ranking), 'Experimenter' (socialdatadelft), and 'Description' (Experiment to test the change in the ranking algorithm. Search engine used: ElasticSearch, ranking algorithm: default and bm25, collection: cran (l)). The 'Variants' section lists two variants: 'Default' (Default ranking algorithm in Elast) and 'BM25' (BM25 implementation in ElasticS), both with their respective client URLs and 'Is Control' checkboxes. The 'Configuration' section includes 'Configuration' (test), 'Date to end' (Date/Time), and 'Max.Units' (Max. units). Below these are two rows for 'Variant Default' and 'Variant BM25', each with a 'Units (%)' field set to 50.

Figure 5: Example of definition of *RankingExp*.

Once we have created both experiments, we have to start them. In order to do that, we select them from the right window in *Experiments::Manage*, and press the button Start, located at the bottom. The status should then change from *OFF* to *ON*.

## Step 2. Install and Deploy the Client

At this point you will need to get the client ready in your local machine. In order to do that you will need to download and deploy it in a server. One of the lightest servers is SimpleHTTPServer, included in the python library, but you can use any other server.

Download the client from <https://github.com/marrerom/ClientE> and update the two HTML files, *color.html* and *ranking.html*, located in the folder *WebContents*, with the corresponding identifier of the experiment running in the platform. You can get that identifier from *Experiments::Manage*, it is the number that appears right to each experiment in the list (you can also see the identifier at the top of the window displayed when clicking on the experiment). Replace the parameter in the method *init* (triggered when loading the page) of the HTML files with the corresponding identifier of our experiments (the identifier of our *ColorExp* in the file *color.html*, and the identifier of our *RankingExp* in the file *ranking.html*).

Deploy the client. In case you use SimpleHTTPServer, go to the folder where the client is located and launch the following command from console:

```
python -m SimpleHTTPServer 8080
```

It deploys the contents of the folder where it is launched. Therefore, now you should see in <http://localhost:8080> the project. You can test it is working by accessing the web pages *color.html* or *ranking.html*. In both cases, you should see a search website (see figure 6). However the query functionality still does not work because we do not have yet the parameters needed (user, rankingAlg, and linkColor).



Figure 6: main search web page. The user id is displayed at the top right corner just for demonstration purposes (the user is not supposed to see this!).

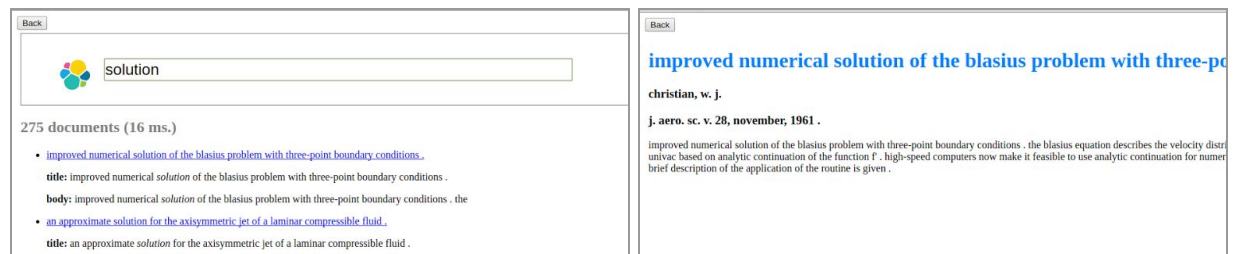


Figure 7: search results interface (left), and contents of a specific document retrieved (right). The user can navigate the search results pages, click on the documents retrieved, and go back to the previous page with the button at the top left.

## Step 3. Test your Client

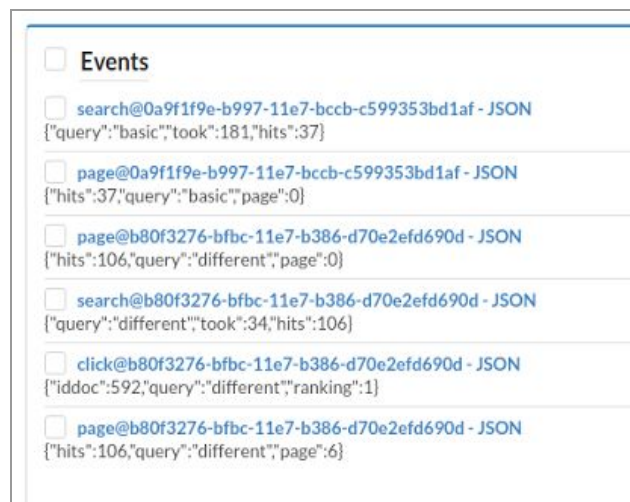
We can test if our client is working properly by simulating we are users of the experiment. That way, we can test if the platform is properly redirecting the users to the different variants, if the client receives the parameters and display the corresponding variant, and if the interactions are saved as events in the platform. In order to do so, go to *Experiments::Manage*, select your experiment (*RankingExp* or *ColorExp*), and press the button *Test* at the bottom. Every time you press that button, you will be automatically assigned a random identifier and redirected to one of

the variants of the experiment, that is, one of the client URLs defined in the variants, which will appear as a pop-up window.

In the URL received we will have not only the query parameters we defined (if any), but also two additional parameters: identifier of the user assigned randomly by the platform (`_idunit`), and name of the variant assigned to that user (`_variant`). All that information is properly encoded<sup>5</sup>.

Now you can interact with the client as a user, issuing queries and checking results (see figures 6 and 7). You can test that the search engine works (the ElasticSearch service is located in `ireplatform.ewi.tudelft.nl:9200`, and it is open to make queries over the `cran` collection).

During this interaction, there should be events registered in the platform. In the case of `ClientE`, the events registered are *search*, *click* and *page*, corresponding to the user actions of searching, clicking on a document retrieved, or going to the next (or any other) page of results (see figure 8).



<input type="checkbox"/>	Events
<input type="checkbox"/>	search@0a9f1f9e-b997-11e7-bccb-c599353bd1af - JSON {"query":"basic","took":181,"hits":37}
<input type="checkbox"/>	page@0a9f1f9e-b997-11e7-bccb-c599353bd1af - JSON {"hits":37,"query":"basic","page":0}
<input type="checkbox"/>	page@b80f3276-bfbc-11e7-b386-d70e2efd690d - JSON {"hits":106,"query":"different","page":0}
<input type="checkbox"/>	search@b80f3276-bfbc-11e7-b386-d70e2efd690d - JSON {"query":"different","took":34,"hits":106}
<input type="checkbox"/>	click@b80f3276-bfbc-11e7-b386-d70e2efd690d - JSON {"iddoc":592,"query":"different","ranking":1}
<input type="checkbox"/>	page@b80f3276-bfbc-11e7-b386-d70e2efd690d - JSON {"hits":106,"query":"different","page":6}

Figure 8: Events registered by `ClientE`. The content of these events is saved in JSON format.

You should see those events in the right window in *Events::Manage*, together with the event *exposure*, which is registered every time the client page is loaded. Click on the events to check the information they contain, or filter, select and download those of interest for you in CSV or JSON format by using the buttons at the bottom.

You can also check the status of the experiment in *Monitoring::Experiments*. There you should see the experiments running, the number of exposures and users who have completed the experiments they have been exposed to (we will talk about this in the next section). Clicking on

---

<sup>5</sup> The value of each query parameter in the URL will be then URLEncoded. The whole query string after the symbol '?' will be then Base64 encoded and then URLEncoded. For example:  
`http://myclient?URLEncode(Base64encode(_variant=URLEncode("Blue")&_idunit=URLEncode("1234")))`

those numbers, you will see the information about the number of users per variant in each case<sup>6</sup>.

## Step 4. System in Production

When you develop your own client, after making sure everything works as expected, you will need to host it in a public domain, so any user can access it. Create a new experiment in the platform with the proper URLs, and remember to update the new experiment identifier in your client.

Once the client is up and running in the new host, you can turn on the experiment in the platform. The users of the platform will be able to participate in the running experiments created by other users by pressing the button *Participate in random experiment* from the menu *Monitoring::Users*. There you can also see how many experiments you have created, in how many experiments you have participated (and completed the experiment) and the remaining number of experiments you can participate in.

Maybe it is a good idea in general to test first the experiment with real users for a short period of time. To this end, you can create the experiment (maybe with more users assigned to control if it is safer), and check the events are properly registered. Afterwards, selecting the experiment from *Experiment::Manage*, you can create a new experiment with the button *New conf.* which will create an experiment with exactly the same metadata (except for the identifier) and variants, but you can assign a different distribution of users among variants. Now you can stop or remove the previous one, and start the new one. Again, remember to update the experiment identifier in your client.

---

<sup>6</sup> Note that the events may not be immediately saved in the database, as they are sent to a message broker platform (RabbitMQ). A possible delay here will also affect the monitoring system, as it depends on the events *exposure* and *completed* already registered by the message broker.



## 5. HOW TO REGISTER EVENTS

The client is responsible of sending to the platform the information related to the user interaction, so the experimenter can analyze what variant is better after the experiment has finished. In order to do that, it has to call to the following endpoint:

POST /IREPlatform/service/event/register<sup>7</sup>

which consumes a JSON with the information of the event we want to save (see Table 1).

name	type	comment
idconfig	string	Id of the experiment
idunit	string	Id of the experimental unit (eg. user)
ename	string	Name of the event (e.g. 'click'). Reserved names are 'exposures' and 'completed'.
evaluate	string	Information we want to save
etype	Enum: ["BINARY", "STRING", "JSON"]	Data type of the information contained in evaluate. It will determine the type of data used to save the contents in the database. If the contents are binary, they should be previously encoded in Base64 <sup>8</sup> .
paramvalues	object	If you use PlanOut (see section 8.2), parameter values received (the platform can not calculate them as some may have been overwritten). If not, empty object: {}
timestamp	string	Timestamp of the event (ISO 8601 format). This is optional, if it does not exist, then the platform will assign the current timestamp when it starts processing the petition.

Table 1. JSON consumed by the endpoint to register an event.

---

<sup>7</sup> Two different content-types are supported for this endpoint, APPLICATION/JSON and TEXT/PLAIN. The latter should be used in order to easy cross-domain communications, where the calls are made directly from the client-side to a server located in a different domain (Cross-Origin Resource Sharing).

<sup>8</sup> For long binary information, the platform offers an additional endpoint where the information is consumed as Multipart-form-data, and all the parameters are string except for *evaluate*, which is an InputStream. In this case, the contents are sent without encoding.

When we register an event, the variant name assigned to the user will also be saved automatically, together with the information about the user agent, the owner of the experiment, and a unique identifier of the event.

In the specific case of `ClientE`, it includes a general Javascript function called `registerEvent`, which makes an AJAX call to the servlet `Register`. That servlet then calls the `platform` and send the information of the event to be registered. In this case we will register three types of events:

- Search: we save the query made by the user, the time it took and the total documents retrieved in JSON. Function `registerSearch`.
- Page View: we save the page viewed by the user (only the first 10 pages, each one with 10 results, are retrieved), the query made, and the total number of documents retrieved in JSON. Function `registerPageView`.
- Document View: we save the query made, the document the user clicked on, and the rank of that document with respect to the total number of documents retrieved in JSON. Function `registerDocView`.

There are two special type of events you can send and check, called *exposure* and *completed*. The first indicates that the user has been exposed to the experiment, while the second indicates that the user has already finished that experiment. It is up to the client to send or not these events, and the moment they should be sent<sup>9</sup>. In `ClientE`, only the event *exposure* is sent when loading the page (function `init`). The function to register and check the event *complete* are implemented but not used (functions `registerCompleted`, `checkIfCompleted`). The platform will use these events to monitor the experiment, and in the case of *complete* events, to stop sending that user to the same experiment (from *Monitoring::Users* -> *Participate in random experiment*). The number of completed experiments, together with the number of exposures can be monitored in *Monitoring::Experiments*.

To check if a user has already completed the experiment from the client (i.e. exists at least one event *completed* for that user/experiment), we can make a call to the following endpoint:

```
GET /IREPlatform/service/user/checkCompletedExp/{id experiment}/{id user}
```

the response will be a string 'true' or 'false' that can be used by the client to decide what to do (for example, avoid registering new events from a user who has already completed the experiment).

---

<sup>9</sup> You can send more than one *completed* event for the same user, same experiment, although they are counted as one in terms of monitoring.

## 6. WEB INTERFACE USER REFERENCE

### 6.1 Experiment::Manage

The manage experiments interface shows the existing experiments in the database, be them running or not. For each experiment the following information is displayed:

```
[experiment name]:[configuration name]@[experimenter] - Status: [ON/OFF] -  
[Identifier]  
[Description of the experiment]
```

If we click on it, a window with the full information about the experiment will pop-up. The unique identification of the experiment will be at the top. This identifier will be used when calling the platform's endpoints to ask for services like getting parameters or registering events while that experiment is running.

We can control the experiments that are displayed by applying filters. In order to do so, we fill partially or completely the form at left and then click on the 'Filter' button. The experiments that match ALL the stated conditions in the filter will be displayed in the right panel. In the case of the *Experiment.Description*, *Variants.ClientURL*, *Variants.Description*, *Variants.Definition* and *Configuration.Client*, the search is made by using the input as a regular expression (Perl Compatible Regular Expression). This is noted with the word *regex* in the alt-text that appears when we place the mouse on them. For the other fields, the match is exact, even for the dates (date started, ended, and date to end), which match the exact day (independently of the time).

Several of the experiments displayed can be selected to be tested, removed, started, or stopped (see subsection *Start/Stop an experiment*) with the corresponding buttons below. We can also show the events saved for a specific experiment with the button *Show Events*, or create a new experiment by replicating the metadata and variants of the selected experiment with the button *New Configuration*. In the first case, the interface *Events::Manage* will be displayed. In the second case, the interface *Experiments::Create New* will be displayed, with the metadata and variants of the experiment already filled and disabled (to enable the form, click button *Clear*).

#### Start/Stop an experiment

An experiment can be started or stopped by selecting it in the interface *Experiments::Manage* and clicking the button *Start* or *Stop*. If the experiment is already running or has been previously run, we can also start or stop it in the *Monitoring::Experiments* interface by sliding the button in the *Status*.

Note that the same experiment can be started/stopped as many times as we want. Only while the experiment is running, it is possible for the platform to redirect to the predefined client URLs,

get parameter values, or register events. Every time we start or stop an experiment, the date and time is saved. This information is displayed in the fields *Started* and *Ended* when we click on an experiment to display its information in the interface *Experiment::Manage*. Those dates should be taken into account when analyzing the data.

## 6.2 Experiment::Create New

A new experiment has three areas of information to be filled: metadata about the experiment, definition of the variants that will be displayed to the user, and configuration to determine how it will be run (percentage of users per variant, and conditions to finish).

### Metadata:

- **Experimenter:** name or any other identification of the person in charge of the experiment. This information will appear already pre-filled with the name of the Twitter user authenticated to access to the platform.
- **Name:** name or any other identification of the experiment. Useful to look it up or identify it quickly when searching or monitoring our experiments. This is not a unique identifier of an experiment, it is just to make it easier the search and visualization of a specific experiment.
- **Description:** a description of the experiment. It should be a brief summary of objectives and methodology, that is, what we want to prove with the experiment, and how we are going to do it.

### Variants:

- **Name:** name to identify the specific variant of the experiment (eg. in *ColorExp*, *Blue* and *Green*).
- **Description:** brief description of the approach the variant represents.
- **Client URL:** URL of the specific variant. It is used to redirect the user to the assigned variant. You can use query parameters, as is the case of *ClientE*.
- **Is Control:** checked if the variant is the control. We can only have one control per experiment, but it is possible to define multiple treatments. The idea behind is that we can compare the results obtained from the treatments with something that we *already* know. For example, if we test directly two new approaches for our ranking algorithm, we may see which one is better, but they both might actually be bad approaches. In offline evaluation, we assume that the relevance judgements are the truth. In online evaluation we need to know the impact of one of the variants in order to give some validity to the results obtained from the others.

## Configuration:

- Name: used to identify the specific configuration. Note that we can create a new experiment with the same metadata and variants, but different configuration. For example, we can run the same experiment at first with just 10% of the users accessing the treatments instead of the control for testing purposes. Then, we can create a new experiment with the same metadata and variants but with a different distribution, this time 50% for treatment and 50% for control. See in section *Experiment::Manage* how to create easily a new experiment from an existing one, replicating metadata and variants.
- Date to end: condition to finish the experiment. Once reached, the experiment will stop automatically, that is, the client we are using will no longer be redirected from the platform, able to get the parameters, or register events for that experiment.
- Maximum completed units<sup>10</sup>: maximum number of different experimental units (eg. users) to complete the experiment (note that in order to consider an experiment *completed*, the client must send at least one event of this type (see section 5). It is a condition to finish the experiment. Once reached, the experiment will stop automatically.
- Percentage of units per variant: percentage of units accessing each one of the variants defined in the experiment. At this moment, the percentage **has to be an integer** between 0 and 100. If the sum of the percentages are less than 100, the rest will be automatically assigned to the control variant.

## 6.3 Experiment::Create New (Advanced)

We can define an experiment in advanced mode, which additionally will let us include PlanOut scripts in the variants, and exclude the definition of client URLs (see section 8). Each variant may have the following additional information:

- Unit: variable we are going to use to identify the experimental unit in the PlanOut script. It is necessary when we use random operators in the script (see section 8.2). For example, in the following definition we use the variable 'userid' to identify each user:

```
rankingAlgorithm = uniformChoice(choices=rankings, unit=userid);
```

Note that this is just the variable used in the script, not its actual value (which we don't know and will be different for each user).

- Definition: definition of the parameter-values in PlanOut language. There is a link to an external application ([PlanOut Experimental Editor](#)) which can be used to validate and

---

<sup>10</sup> Note that there may be a delay to stop the experiment when the stopping conditions match (*date to end* or *max. completed units*) as these conditions are checked every five seconds.

test the script before creating the experiment, or to check why there was an error when we try to create it.

In this mode, we may also include information about the client in the Configuration section:

- Client: information about the specific client used. Could be the URL and version to identify the code in a software repository, could also include the URL where it is running, etc. From a research perspective, it is important to identify uniquely the specific version of the client used in an experiment. For example, if we wanted to run an experiment with ClientE, we should note that here (link to repository in github and version).

## 6.4 Events::Manage

When an experiment is running, we can register events from our client through calls to the platform. Each event contains information about the experiment, and about the specific information we want to save about the interaction of the user.

### Experiment:

- Experiment ID: unique identifier of the experiment
- Experimenter: owner of the experiment
- Variant: name of the variant the user is being exposed to.
- Parameters and values: name and values of the PlanOut parameters for that specific user (if any).

### Interaction:

- Unit id: identification of the experimental unit (typically a user).
- Timestamp: timestamp of the event.
- Name: user-defined name assigned to the event we want to register (eg. click, page-view, etc.). The names *exposure* and *completed* can be used, but only to indicate that there is a new exposure, or to register that the user has completed the experiment.
- Value: the specific information we want to save about the interaction with the user (eg. number and positions of clicks, issued queries, search results obtained, etc.).
- Data type: data type of the specific information (value) we want to save: binary, JSON or string format. Note that the *exposure* and *completed* events do not have any value, but still the data type, when they are automatically registered, is string.

- User-agent: user agent information used by the user in the interaction with the client.

The interface shows the existing events in the database. For each event the following information is displayed:

```
[experiment]:[event name]@[experimenter] - [binary|string|JSON]
[Value -first 100 characters, and only if not binary-]
```

If we click on it, a window with the full information about the event will pop-up. The unique identification of the event will be displayed at the top.

We can control the events that are displayed by applying filters. In order to do so, we have to fill partially or completely the form at left and then click on the *Filter* button. The events that match ALL the stated conditions in the filter will be displayed in the right panel. In the case of user-agent, the search is made by using the input as a regular expression (Perl Compatible Regular Expression). In the case of the field value, it depends. If the type of event selected is string or any, then the input is used as a regular expression (but it won't work for events with type JSON). However, if the type selected is JSON, then the input will be considered as a JSON document in the search (**note that the input in this case has to be a valid JSON document!**). For the other fields, the match is exact, even for the timestamp, which match the exact day (independently of the time).

Finally, the events displayed can be selected to be removed or downloaded as CSV or JSON with the buttons at the bottom of the right panel.

## 6.5 Monitoring::Experiments

There are two special types of events called *exposure* and *completed*. They are optionally sent by the client, however both events are vital to monitor the experiment, so we can see the number of *different* users exposed to the experiment so far, and how many of them completed the experiment.

We can see the experiments running in the top panel, while the experiments that already finished are displayed in the bottom panel. Each row shows information about an experiment. The *unit* column shows the number of different users exposed to the experiment, and also the number of users who have completed the experiment. If we click on these numbers, a new window will pop-up displaying a more detailed information about the exposures or the completion of the experiments: you will see the total number per variant, and also the total number per parameter-values assigned in case we used PlanOut scripts<sup>11</sup> (see Figures 10 and 11).

---

<sup>11</sup> We could have different parameter values in the same variant if, for example, in the script definition of the variant we use a random assignment operator like *uniformChoice* (check section 8.2).

Treatment Monitoring Experiment id 59fae9142ada010f733a5b80	
<b>BM25</b>	
Values	Exposures
{}	1
TOTAL	1
<b>Default</b>	
Values	Exposures
{}	3
TOTAL	3

Figure 10. Detailed information about exposures in our ranking experiment with no PlanOut script associated (no pairs of parameter-values). In Monitoring::Experiments

Treatment Monitoring Experiment id 59b00e102ada012c1fa0b3ee	
<b>Treatment2</b>	
Values	Exposures
{"a":11,"b":true}	6
{"a":10.0,"b":true}	3
TOTAL	9
<b>Treatment1</b>	
Values	Exposures
{"a":11,"b":true}	15
{"a":5.0,"b":true}	7
TOTAL	22

Figure 11. Detailed information about exposures in an experiment with a PlanOut script associated. Note that, depending on the definition of the script, we may have different values for the same parameter in the same variant. In Monitoring::Experiments

From the dashboard it is also possible to start/stop the experiments by sliding the button in *Status*.

## 6.6 Monitoring::Users

In this interface we can see information about the current user (all the users of the platform if we are administrators). For each user, we will see the experiments owned by the user, the experiments the user has completed<sup>12</sup>, and the current number of remaining running experiments the user can still participate in.

<sup>12</sup> Experiments where the user identifier in the event *completed* is the name of the user in the platform.



At the bottom of the window we can see the button *Participate in random experiment*. If we press that button, we will be redirected to one of the experiments to participate as a user.

## 8. ADVANCED EXPERIMENTS

### 8.1 MORE CONTROL IN OUR EXPERIMENT

The use of the platform to redirect the users automatically to the experiments currently running from *Monitoring::Users* is useful, but imposes some limits:

- The users has to authenticate themselves first in the platform in order to participate in the experiments, and, at the moment at least, they can not decide what experiment they want to participate in. On the other hand, the experimenter does not have any control in the users who can or can not participate in the experiment.
- The experimental unit has to be the user: the assignment of the variants is done per user identifier, that is, the user name authenticated in the platform.
- When using the platform to redirect users, the information about the variant assigned to that specific user (as well as the user identification) is sent in the URL. Although it is encoded, it is not encrypted: what happens if the user modify this information? What happens if the user access again this URL directly from the browser with new parameters?

In order to avoid all these situations, we provide a specific endpoint (see below) to get the information related to the assigned variant, given an experimental unit identifier. This way, in the definition of the experiment we can leave empty the fields *client URL* (although we should fill this information in the field *Controller* at least for information purposes). Then, from our client, we assign a user identifier and then we call the endpoint to get the variant (and parameters) assigned. This way, the users can access directly this URL to participate in the experiment.

The endpoints to get directly the information related to the variant assigned to a experimental unit are:

```
GET /IREPlatform/service/experiment/getparams/{id experiment}
```

```
GET /IREPlatform/service/experiment/getparams/{id experiment}/{id unit}
```

As a result of these calls, we will receive a JSON object with the parameter-value pairs from the PlanOut script, if any (params), as well as three additional pieces of information: the identification of the user (`_idunit`), the variant assigned (`_variant`), and the URL assigned to that variant in the definition of the experiment, if any (`_url`).

We have an additional endpoint which consumes a JSON:

POST /IREPlatform/service/experiment/getparams/

With this endpoint it is possible to set values of existing parameters in the defined PlanOut scripts (see section 8.2), so they may condition the value of the rest of the parameters when running the script (see Table 2).

name	type	comment
idconfig	string	Id of the experiment
idunit	string	Id of the experimental unit (eg. user)
overrides	object	Parameter values we want to set before running the corresponding PlanOut script.

Table 2. JSON consumed by the endpoint to get parameters.

For example, we can use a parameter named *browser* in the script, as explained in section 8.2. Depending on the browser the user is actually using, desktop or mobile, we can overwrite the value of this parameter *before* PlanOut calculates and sends back the values of the other parameters for that user.

Finally, note that there are two similar endpoints, depending on the previous knowledge of the identifier of the unit or not. If we don't know it, then the platform will assign one randomly (UUID version 1) **every time we call that endpoint**: the client has to make sure that the same physical unit (be it the user, a query, etc.) corresponds always to the same identifier.

## 8.2 PLANOUT

In this section we will explain briefly what PlanOut is and how we can use it to define experiments. It is used by the platform to define experiments, however, the experimenter does not need to use it directly, as it is possible to define experiments just by indicating the URL's of the variants instead of using PlanOut scripts.

[PlanOut domain-specific-language](#) lets us define univariate or multivariate experiments, by defining parameters<sup>13</sup> and their possible values. The experiment is defined with a script that determines the parameters, values and operators used to assign values to parameters. For example, for our *ColorExp* we can define a script where our parameter is 'linkColor' and the assignment is made over two predefined values: 'blue' and 'green' with a probability of 0.5. This selection will depend on the **experimental unit**<sup>14</sup>, for example the user id (represented in the

---

<sup>13</sup> Predefined parameters used by the platform start with “\_”, so it is better to avoid names starting by that symbol.

<sup>14</sup> In PlanOut scripts it is possible to include more than one unit at the same time, for example, the user id and the query id. However, APONE is limited to one unit.

example below with the variable *userid*), so for the same user id, the PlanOut library will always assign the same color:

```
possibleColors = ['blue', 'green'];  
linkColor = uniformChoice(choices=possibleColors, unit=userid15);
```

When we ask from our client to the platform for the value of these parameters, we send the unique identifier of the experimental unit, and then we will get the parameter values according to that identifier. The client developed by the experimenter transforms the values of the parameters received in changes in the interaction with the user. In this example, it is expected that for some users, when they issue queries, the links of the results will be blue, while other users will see them green.

This way, we are separating experimental design from application logic, with a twofold objective:

- a) It is easier for the experimenter to set-up an experiment because the PlanOut library takes care of the randomization in the assignment of units to parameter values. The experimenter just has to ask from her code what is the *linkColor* assigned to a specific user id, and implement the corresponding changes in the interface (or back-end) that the user is going to interact with. That way we could have a general client which, according to the values of the parameters defined in the experiment, makes the corresponding changes instead of having multiple codes, one for each variant we want to test.
- b) PlanOut scripts may be used as a concise summary of an experiment's design and manipulations, which makes it easier to communicate about, and replicate experiments.

An important additional benefit of using PlanOut is the possibility of running full factorial experiments easily. For example, instead of using just the link color in the previous example, we may want to change also the ranking algorithm as in *RankingExp*:

```
possibleRankings = ['default', 'BM25'];  
possibleColors = ['blue', 'green'];  
rankingAlgorithm = uniformChoice(choices=possibleRankings,  
unit=userid);  
linkColor = uniformChoice(choices=possibleColors, unit=userid);
```

As a result, we would get all the different combinations of parameters *rankingAlgorithm* and *linkColor*, that is, we will have users exposed to  $2^2$  different variants (two factors or independent variables, two levels each).

---

<sup>15</sup> The name of this variable has to be specified in the field *Unit* when creating a new experiment in advanced mode.

It is also possible to use conditional statements, as well as [other random assignment operators](#) that let us define complex experiments. The use of conditional statements is useful to assign parameters that depend on the assignment of previous parameters, but it is also possible to overwrite others that depend on the user, so we could end up writing something like this:

```
colorsMobile = ['blue', 'green'];
if (browser == 'mobile') {
    linkColor = uniformChoice(choices=colorsMobile,unit=userid);
} else {
    linkColor = 'blue'; #no treatment
}
```

Here, the variable *browser* can be overwritten with the actual browser used by a user every time we ask for the value of the parameters for him. However, bear in mind that in this case the assignment of the variables is not random (*browser* is a *quasi-independent* variable), and the conclusions of the experiment may be compromised. In any case, if we still want to filter by mobile browsers, desktop users shouldn't be part of the experiment, even if they receive only the control (they are not really part of the control population!). The client should previously filter those users and avoid them to be part of the experiment.

## PlanOut in APONE

To define an experiment making use of PlanOut you have to go to *Experiment::Create new (Adv.)*, and fill in the box called *Definition* with the PlanOut script associated to each variant. You can make use of the PlanOut editor located in the link *Planout DSL* to make sure the script is correct before creating the experiment.

The values of the parameters defined will be sent in the client URL (if defined) as query parameters when a user participates in an experiment pressing the button *Participate in random experiment*, or they will be included in the JSON returned when we call the endpoint `getparams` (see section 8.1). In the latter case, we could overwrite one or more params, so the values of the remaining parameters may change.

The downside of using PlanOut is that you may not be able to define what is the control and what the treatments, specially if you want to run a full factorial experiment. In that case, you should use only one script, which will be part of one variant (which will be *control* for the platform). All the users will go to that variant, but depending on the identifier of the user, they will get different parameters. Therefore, in order to monitor the experiment and analyze the results, you must pay attention to the values of the parameters, instead of just the name of the variant, which will be the same in all cases.

In general, the use of PlanOut will be conditioned by three main factors:

- Reuse of the client for different experiments
- Number of variables to test

- Full factorial experiment

If you are going to reuse the client for multiple experiments, you will probably have several parameters that condition the behaviour or interface showed to the user. Keeping them in a script that can be reused is a cleaner solution: you can comment there the parameters, add new parameters or modify existing ones, keeping in those scripts the logic of the each experiment.

If you have a big number of variables to test, then maybe it is not a good idea to add all of them to each one of the client URL defined in each variant (probably a lot of them too). If you have a big number of variables to test, and you want a full factorial experiment, then it is clear you need PlanOut. Otherwise you will have to define all the possible variants manually. See the flowchart in figure 9.

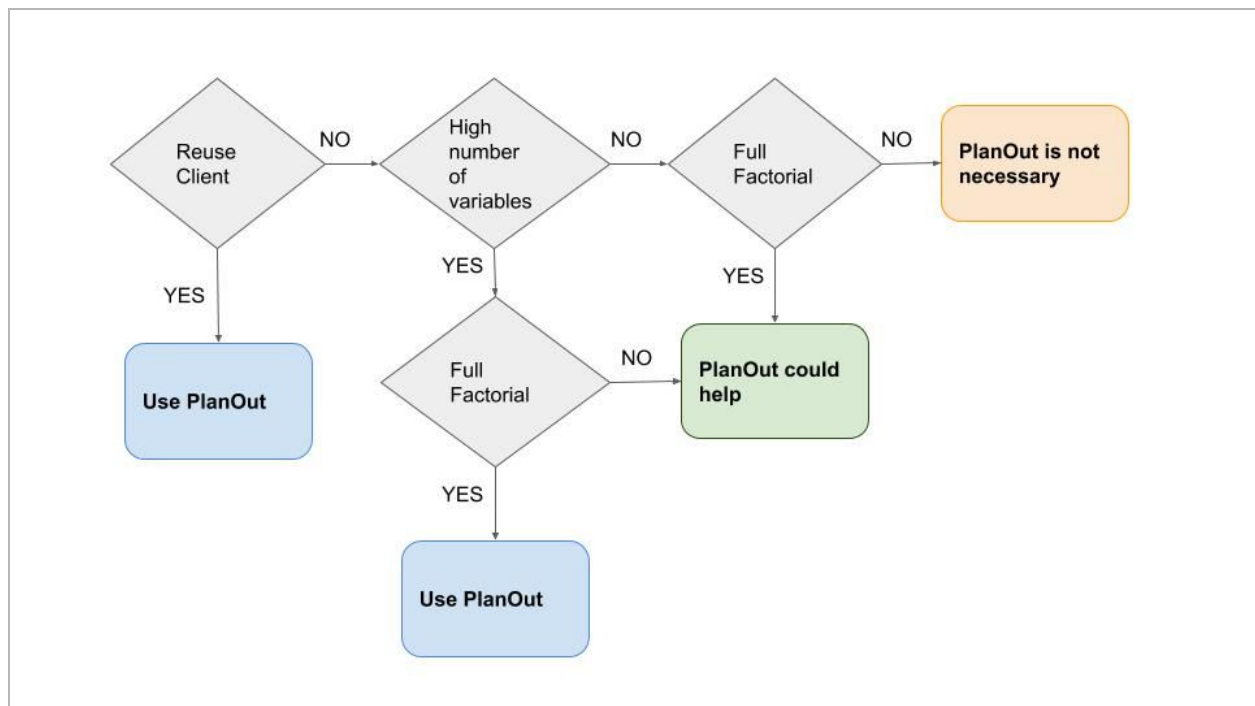


Figure 9: Flowchart to decide if it is worth it to use PlanOut in APONE as part of the definition of an experiment.

## 9. RELEVANT REFERENCES

### Surveys

Hofmann, Katja, Lihong Li, and Filip Radlinski. "Online evaluation for information retrieval." *Foundations and Trends in Information Retrieval*, 10(1), pp. 1-117, 2016

Kohavi, Ron, et al. "Controlled experiments on the web: survey and practical guide." *Data mining and knowledge discovery* 18(1), pp. 140-181, 2009

## Tips

Kohavi, Ron, et al. "Trustworthy online controlled experiments: Five puzzling outcomes explained." *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 786-794, 2012

Kohavi, Ron, et al. "Seven rules of thumb for web site experimenters." *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1857-1866, 2014

## Experiment Life-Cicle

Kevic, Katja, et al. "Characterizing experimentation in continuous deployment: a case study on bing." *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pp. 123-132, 2017.

## Interleaving

Joachims, Thorsten. "Optimizing search engines using clickthrough data." *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 133-142, 2002

Chapelle, Olivier, et al. "Large-scale validation and analysis of interleaved search evaluation." *ACM Transactions on Information Systems*, 30(1), pp. 6:1–6:41, 2012

## Large-scale

Kohavi, Ron, et al. "Online controlled experiments at large scale." *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1168-1176, 2013

## PlanOut

Bakshy, Eytan, Dean Eckles, and Michael S. Bernstein. "Designing and deploying online field experiments." *Proceedings of the 23rd international conference on World wide web*, 283-292, 2014