TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

ITV40LT

Olga Orlova 120976IAPB

# WEB ANALYTICS BASED APPROACH FOR FEATURE INTRODUCTION IN WEB APPLICATIONS

Bachelor's thesis

Supervisor: Jaagup Irve
Master of Science
Software Engineer

Tallinn 2016

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

ITV40LT

Olga Orlova 120976IAPB

# ANALÜÜTIKALE TUGINEV FUNKTSIONAALSUSE RAKENDUSSE JUURUTAMINE

Bakalaureusetöö

Juhendaja: Jaagup Irve
Magister
Tarkvarainsener

Tallinn 2016

# Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Olga Orlova

22.05.2016

# Abstract

When some major changes, e.g. new features, an updated version or redesign, are introduced to a software product in general, and a web application in particular, it is crucial for product owners to monitor the usage of the new parts of application, and to be able to compare it against the way it used to be before. Data collection, research and analysis help to reveal whether or not the issues that new features were expected to solve are actually gone, how satisfied the customers are with the new features, which new issues were possibly caused.

This thesis is based on a project I implemented at my workplace. It describes the process of designing and developing a web analytics tracking for measuring the web application usage during a new feature introduction, and gives a brief insight into the consequent data visualization and analysis.

This thesis is written in English and is 48 pages long, including 5 chapters, 11 figures and 6 tables.

# Abstract

## Analüütikale Tuginev Funktsionaalsuse Rakendusse Juurutamine

Kui tarkvaratootesse, iseäranis veebirakendusse juurutatakse olulisi muudatusi, näiteks uut funktsionaalsust, kujundust või versiooniuuendusi, on tarnijal kriitiliselt tähtis jälgida, kuidas lõpptarbija rakenduse muutunud osi kasutab ning võrrelda seda varasema seisuga. Andmete kogumise, jälgimise ning analüüsi abil selgub kas muudatused avaldavad soovitud mõju, kui rahul klient uue funktsionaalsusega on ning ka seda milliseid uusi probleeme muudatus põhjustas. Bakalaureusetöö vaatleb tööl läbiviidud projekti. See kirjeldab kuidas kavandati ja arendati veebirakenduse seiremoodul, mis jälgis uue funktsionaalsuse juurutamist ning annab põgusa ülevaate järgnenud andmete visualiseeringu ja analüüsi etappi.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 48 leheküljel, 5 peatükki, 11 joonist, 6 tabelit.

# List of abbreviations and terms

| | |
|---|---|
| ISO | The International Standards Organization |
| CRM | Customer relationship management |
| UI | User interface |
| UX | User experience |
| SPA | Single-page application |
| ms | Millisecond |
| NRQL | New Relic Query Language |

# Table of Contents

# List of Figures

## List of Tables

# 1 Introduction

The project described in this thesis is a typical task for an IT business in the life stage of rapid growth. Old parts of the application are being gradually but relatively quickly replaced by their updated versions that are implemented using new approaches and contain new or improved features.

Our task was to test and measure the new feature's usability employing a web analytics based approach. To complete this task we employed a web analytics tool, namely New Relic Insights, and the A/B testing methodology.

This thesis follows the development process emphasizing the importance of data consistency retention, as well as the collected data visualization and analysis. The latter stages allow us to estimate how successful the feature introduction was.

# 2 Theoretical Background

Since in this project we use web analytics tools to try to determine the level of customer satisfaction with a newly introduced web application feature and discover potential issues, it is necessary to introduce some theoretical background first, such as terminology and the existing approach to usability metrics.

## 2.1 Definition of Usability

The essence of the examined new feature is improved usability. Let us define this term.

Probably one of the most precise definition of usability is given by The International Standards Organization (ISO 9241-11). It identifies three aspects of usability, defining it as "the extent to which a product can be used by specified users to achieve specified goals with *effectiveness*, *efficiency*, and *satisfaction* in a specified context of use" [6] . This definition suits our context, since the three aspects: effectiveness, efficiency and user satisfaction, are what we will try to measure with the help of web analytics.

## 2.2 Definition of Usability Metrics

Metrics is a collection of data related to the object of investigation, that can be a whole product or a particular feature or part of it. It is important that the researcher sets the questions that can help to evaluate usability before referring to the obtained data. "Some studies, in the words of one critic, 'use statistics as a drunk uses a street lamp, for support rather than illumination'" [5] . Based on these questions the usability metrics should be designed.

All usability metrics must be *quantifiable* – they have to be turned into a number or counted in some way. All usability metrics also require that the thing being measured represent some aspect of the user experience, presented in a numeric format [6] .

## 2.3 The Importance of Usability Metrics

Some usability issues can be identified at the pre-launching stage, but they are so minor or specific that it is not clear whether or not they will influence a statistically significant number of users. Without usability metrics, the magnitude of the problem is just a guess [6] .

Usability metrics show whether the user experience improves between different product versions. Usability metrics are the only way to really know if the desired improvements have been realized. By measuring and comparing the current with new, “improved” product and evaluating the potential improvement, you create a win-win situation. There are three possible outcomes:

- The new version tests better than the current product.
- The new version tests worse than the current version: Steps can be taken to address the problem or put remediation plans into place.
- No difference between the current product and the new product is apparent: The impact on the user experience does not affect the success or failure of the new product. However, improvements in other aspects of the product could make up for the lack of improvement in the user experience [6] .

Another way usability metrics can be employed when measuring customer satisfaction is to test intuitive assumptions. In fact there is inevitably some guesswork involved in creating a new product or feature, however sometimes the right design solutions are counterintuitive [6] .

## 2.4 Definition of A/B Testing

We used A/B testing to evaluate the usability of the examined new feature. It means that we enabled the feature for approximately 50% of customers, and were measuring the user experience of both groups, comparing the new version of the product with the previous one. The concept of A/B testing is simple: show different variation of your website to different people and measure which variation is the most effective [4] .

# 3 Overview of Pipedrive’s Web Application

Pipedrive Inc develops a web-based sales CRM (Customer relationship management) software. It is a tool for salespeople, primarily coming from small and medium enterprises, that enables them to better organize their work process. Pipedrive’s web application is based on one of the sales methodology’s principles, which is called Sales Pipeline.

The company was founded in 2010, and originally the project was implemented as a traditional web application using PHP and Javascript. Later the conversion into a single-page application has begun, and currently it is completed by approximately 85%. For simplicity’s sake, further on we will refer to the original application and the single-page version using the company’s insider terms – *old* and *new* respectively.

Figure 1 and Table 1 show Pipedrive’s web application structure. Since its front-end part is currently built mostly with OOP principles using Backbone.js, it is easy to imagine the architecture of the application being familiar with its main objects and viewing modes.

As seen from Figure 1, the main objects of the application are: Deals, Persons and Organizations (grouped together under “Contacts” in the application’s toolbar), Products, Activities and Pipelines – the latter being either default or custom set of stages that a particular Deal can pass.

Table 1 summarizes various viewing modes available in Pipedrive’s web application. The former two, Pipeline and Timeline, are specific for Deal-type objects, whereas any object collection can be viewed as a List. The difference between the *old* and the *new* list views lies in its architecture, UI/UX design and, to a little extent, the variety of features provided.

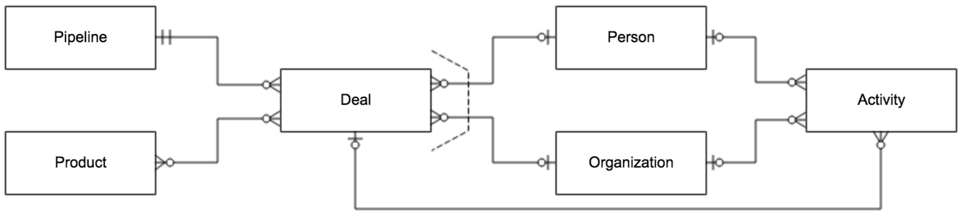


Figure 1. Main objects and their relations in Pipedrive's web application.

Table 1. Viewing modes in Pipedrive's web application.

| | **Pipeline** | **Timeline** | **List** | |
|---|---|---|---|---|
| | | | ***Old*** | ***New*** |
| Deal | + | + | | + |
| Person | ▨ | ▨ | | + |
| Organization | ▨ | ▨ | | + |
| Activity | ▨ | ▨ | + | + |
| Product | ▨ | ▨ | + | |

## 3.1 The Reasons Pipedrive Is Switching to Single-page Application

Whereas traditionally, web applications left the heavy lifting of data to servers that pushed HTML to the browser in complete page loads, now this relationship has been inverted – client applications pull raw data from the server and render it into the browser when and where it is needed. Developers commonly use libraries like Backbone.js to create *single-page applications* (SPAs). SPAs are web applications that load into the browser and then react to data changes on the client side without requiring complete page refreshes from the server [3] .

The main reason to switch from server-side application pattern to the client-side one is improving the user experience. The first thing users will immediately benefit from is page load speed increase. The traditional approach to web app development is very server-centric, requiring a complete page reload to move from one page to the next [3] . Client-side architecture provides a faster and more fluent experience for users: not all the page elements need to be reloaded simultaneously.

SPAs can also take advantage of browser features like the History API to update the address shown in the location bar when moving from one view to another. These URLs also make it possible for users to bookmark and share a particular application state, without the need to navigate to completely new pages [3] .

Finally, client-side architecture potentially enables implementation of offline mode, when the application, once loaded in user's browser, runs independently and sends the data to the server side as soon as the network connection is restored.

## 3.2 The Process of Converting to Single-page Application (Using Pipedrive's Activity List View As an Example)

Creating and launching the *new* activity list view was a major project started on October 13, 2015 and completed in February 2016. The beta version first went live and became available for new sign-ups on January 7. By the beginning of February, when only approximately 3% of customers (i.e. newly signed up users) were using the *new* activity list view, we had developed performance and usage analytics trackers and started collecting data on user experience and behavior on both the *old* and *new* list view pages. On February 12 the *new* activity list view was enabled for another 40% of customers, and eventually, by the end of February, the transition was complete.

## 3.3 Web Analytics Tools Used in Pipedrive

Various web analytics tools are used in Pipedrive to monitor application performance, track user activity, process and visualize the collected data.

### 3.3.1 Action Mapper

Action Mapper is an in-house developed tracking tool that logs all the actions made on the back end side of the application. For instance, if there are several ways to mark an activity as “done” in the application, and a user completes this action using one of those ways, Action Mapper will log that the event “activity marked as done” has happened, but it does not provide the context, i.e. which of the UI paths the action was completed with.

### 3.3.2 New Relic Insights

New Relic Insights, on the contrary, keeps track of the events happening on the front end side of the application. It is possible to create custom events bound to various actions coming directly from user. As New Relic Insights was the tool used in implementation of the project which is the focus of this thesis, a more detailed description will be provided further on.

### 3.3.3 Segment

Segment is an analytics data warehouse in cloud, that is used in Pipedrive for marketing needs. It has a range of integrations available to be enabled or disabled depending on the company’s needs. For instance, Pipedrive is using MailChimp, Google Tag Manager, Kissmetrics, Amazon S3 and some others. Those services receive the data tracked by Segment and provide its analysis, visualizations, etc. The data tracked by the Segment script is more suited for marketing needs and contains user’s location, settings, time spent in the application, etc, as well as all the actions made by user on the page. This helps to create statistics on percentage of anonymous page visitors who chose to sign up for Pipedrive, new paid customers, busiest times, most active customers, geographical distribution of customers, etc.

# 4 Web Analytics Tools Usage in Feature Introduction

## 4.1 Planning Stage

As a tool to measure the *old* and *new* activity list view usage we chose to use New Relic Insights. Its choice over the other tracking tools used in Pipedrive is evident, as we needed to track custom-defined events happening on the front end side of the application.

The data was to be collected during an A/B testing, since one of our main goals was to compare the usage of the *new* activity list view against that of the *old*. Another reason why A/B testing is well-suited to our needs, is the planned gradual introduction of the new feature to the users.

The next step now was to decide on the data we would like to collect based on our needs and assumptions. Table 2 summarizes the performance aspects and user actions that were decided to be tracked and measured.

Table 2. Data types to track in A/B testing.

| **Event** | **Measured property** | **Assumption (compared to the *old* page)** |
|---|---|---|
| Page load | Time (ms) | Is as fast or faster |
| Time range filters usage | Filter name (e.g. "overdue", "today", "next week", "custom date range") | "Custom date range" filter is used more frequently |
| Bulk edit usage | Count | Increases or remains the same |
| Marking activity as "undone" | Count | Is done more frequently |

As can be seen from Table 2, we decided on measuring page loading time and were expecting it to be faster in the *new* activity list view. Page load speed increase is one of the major benefits of single-page architecture for the end user, and it is therefore important to control that the new version of our application indeed provides it.

Both of the activity list view versions have a range of quick filters that allow the user to filter all the activities by type (e.g. “meeting”, “call”, “lunch”) and time range. Since the time range filters, unlike the type ones, are the same for every customer and reflect the needs of users, we decided to focus only on them. Time range filters are the following:

- “Planned” - all activities with the status “undone”, both due and overdue
- “Overdue” - all overdue activities with the status “undone”
- “Today”, “tomorrow”, “this week” and “next week” - all activities planned for the respective time period, both due and overdue
- “Custom date range” - all activities planned for the date/date range selected by user, both due and overdue

As can be seen from Figure 2, the *new* activity list view page has an improved date picker for the custom time range filter: unlike the *old* one it allows the user to select a date range, not merely one day, while it is still possible to filter activities of one particular day – by leaving either of the date fields empty, or filling them out with the same date. It adds value for users, and that is the reason why we were assuming this time range filter usage to significantly increase on the *new* activity list view page.

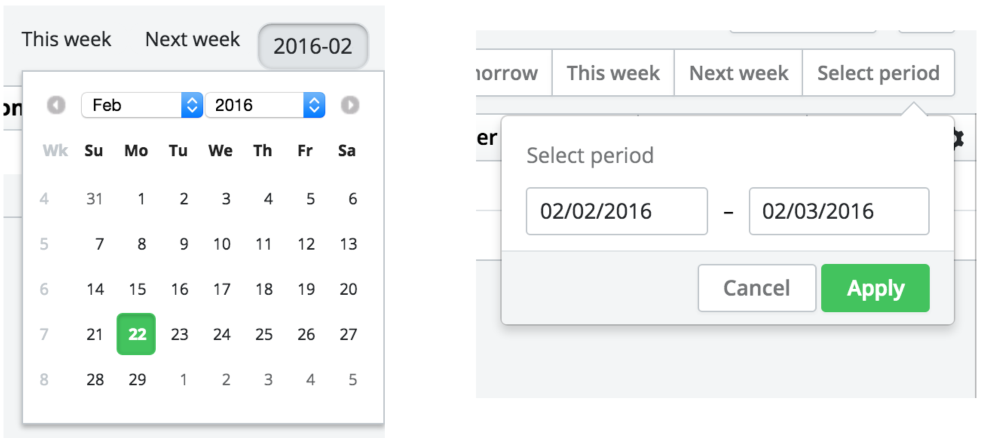


Figure 2. Selecting custom date/date range in the *old* and *new* activity list views.

The bulk edit feature was also improved, previously known issues with its usage were taken into account. The design elements prone to confuse the users in the *old* activity list view are shown and explained in Figure 3. As the bulk edit feature was fundamentally re-designed in the *new* page, we made an assumption that it might be used more frequently.

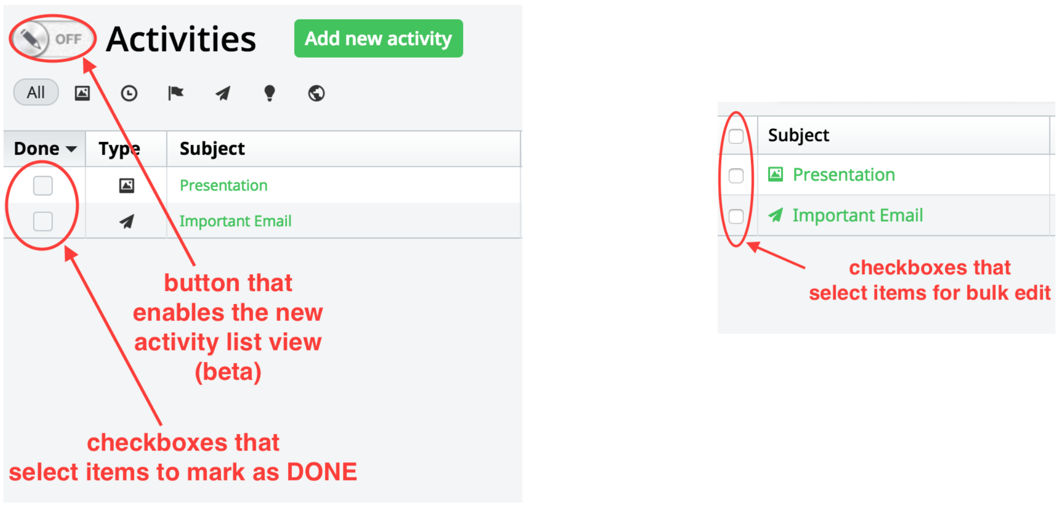

OFF
Activities
Add new activity
All
Done
Type
Subject
Presentation
Important Email
button that enables the new activity list view (beta)
checkboxes that select items to mark as DONE
Subject
Presentation
Important Email
checkboxes that select items for bulk edit

Figure 3. Confusing UI elements in the *old* activity list view and selecting items for bulk editing in the *new* activity list view.

As it has been already mentioned, one design flaw with the *old* activity list view was a confusing way to mark activities as “done”. From customers feedback we were aware of the usability flaw, that was resulting in an increase of changing accidentally marked as “done” activities back to the “undone” status. As the *new* activity list view has this issue fixed, it could be assumed that the total number of those events would decrease. However, the *new* page has a considerably more convenient and simple way of marking activities as “undone”, which leads to an assumption that the count might stay the same, or even decrease in the *new* page.

Such are the events we decided to track and measure with the help of New Relic Insights, and the assumptions we planned to prove or disprove based on the collected data.

## 4.2 Implementation Stage

### 4.2.1 Overview of the New Relic Insights Data Tracking Structure

In order to start using New Relic Insights for data tracking, first of all, its library should be added to the web application code, and afterwards a special class, that initializes the tracker as a global object and defines its methods, can be created. The New Relic class used in Pipedrive is contained in *newrelic.js* file. It defines a simple way to create custom New Relic Insights events (page actions) and adds a number of default attributes that will be sent with each custom event, such as *user_id*, *company_id*, *default_currency*, *timezone_server_offset*, etc.

### 4.2.2 Implementing Custom New Relic Actions for the Task

The New Relic class component for both the *old* and *new* parts of the application was already created and set up, thus the following steps were left to be taken:

- Grouping the data to track into New Relic events
- Examining the application code and determining the best place for each kind of data retrieval
- Implementing the tracking
- Manual testing in development environment, controlling that the data is tracked correctly

Table 3 shows the New Relic events (actions) that were to be created for tracking the data.

Table 3. New Relic actions for activity list view usage tracking.

| Event | NR action name | NR action's attributes |
|---|---|---|
| Page load | activityListMetrics:pageViews | activityListType: “new”/”old”<br>loadingTime<br>timeFilterName |
| Time range filters usage | | |
| Bulk edit usage | activityListMetrics:bulkEdit | activityListType: “new”/”old” |
| Mariking activity as “un-done” | activityListMetrics:markUndone | activityListType: “new”/”old”<br>newListOldHabit: *true*/*false* |

### 4.2.3 Measuring Page Load Time and Time Filters Usage

As the *old* and the *new* parts of the application are independent, the development was happening in two code branches. The server-side style *old* application reloads the page every time a filter is applied, changing the page's URL. That is why, in order to keep data consistent and to facilitate the implementation, we chose to consider a time filter switch as a page view – the data (a list of activities) matching a filter is loaded, so it is a suitable event for analysing the application's performance.

Table 4 shows the events that were defined as a page load, as well as the Backbone views where the beginning and the end of loading should be tracked in each of those cases.

Table 4. Defining events to track for "activityListMetrics:pageViews" NR action in the *new* activity list view.

| Event | User action to start event | Loading start point place in code | Criteria for event finish | Loading finish place in code |
|---|---|---|---|---|
| Direct page view (e.g. initial page load, coming from an external link, page refresh while viewing activities list) | Calling an URL | router.js:activities_beta() | The list of matching activities is fully loaded | Activities-analytics.js:trackPageViews() |
| "Stack-based navigation" | Going back to activities list after having switched to another page in the *new* application | router.js:switchCurrentView() | | |
| Time range filter switch | Clicking on a time range filter (in case of the "custom date range" filter – clicking on the "Apply" button) | quick-filters.js:handleFilterChange() | | |

Let us examine the former possibility as an example. This is the case of a direct page view.

Router is a Backbone element that provides methods for routing client-side pages, and connecting them to actions and events[1]. The Router class is extending *Backbone.Router* and contains all route configurations for Pipedrive's web application.

The path to the *new* activity list view page is */activities_beta*. When the router receives this key, it goes into a corresponding function – *activities_beta()*. This is the point that can be considered as page loading start, and that is why a new option, *loadingStart*, was added to store the timestamp. Inside the function the method *switchCurrentView* gets called, where the requested view either starts loading from scratch, or gets returned as the version saved in stack if it is available.

In case of a direct page view, the page has not been visited yet, and therefore is not available from stack. A new instance of the ActivitiesList view, a Backbone representation of a logical element of the interface, is created. Its *collection* attribute contains an array of requested activities, and we bind its *sync* event, which means that the collection is received from server, to a call to our *trackPageViews* function, located in *activities-analytics.js*. It takes only one parameter – *loadingStart,* logs the moment when the collection was received, calculates the difference between *loadingStart* and *loadingFinish*, determines other necessary attributes (*activityListType* and *timeFilterName*), and finally sends it to New Relic Insights cloud as a custom action named "activityListMetrics:pageViews". The code to illustrate the described process is below:

```
// from router.js
activities_beta: function(filterType, filter, action) {
  var opts = {
    action: null,
    loadingStart: $.now()
  };

  <...>

  require(['views/lists/activities'],
_.bind(function(ActivitiesListView) {
    this.switchCurrentView(ActivitiesListView, ‘activitiesList’,
opts);
  }, this));

  <...>
}

// from lists/main.js
initialize: function(options) {
  <...>

  if (this.filterType === ‘activity’) {
    this.collection.once(‘sync’, function() {
      activitiesAnalytics.trackPageViews(this.options.loadingStart);
    }, this);
  }

  <...>
}

// from activities-analytics.js
trackPageViews: function(loadingStart) {
  <...>

  app.nr.addPageAction(‘activityListMetrics:pageViews’, {
    ‘activityListType’: ‘new’,
    ‘loadingTime’: loadingFinish - loadingStart,
    ‘timeFilterName’: timeFilterName || null
  });
}
```

Everything mentioned above applies to the *new* application. The *old* application is structured differently, which created some obstacles.

The *old* application's architecture relies on the *main.js* file which loads the requested PHP templates and adds Javascript elements to the page. *Main.js* introduces the global variable *app* that has a range of attributes and methods to initialize and display various features.

As it is characteristic of a server-side application, the whole page is reloaded every time a new request is sent, e.g. each time a filter is applied to activities list. That is why the first step was to identify all possible URI patterns on this page, and decide for each of them on whether or not it should be considered to be a page view in terms of the metrics task. Table 5 shows the outcome of this investigation.

Table 5. URI patterns in the old activity list view, events leading to them and whether or not they should be counted as a page view in metrics.

| URI | Event | Counted as a page view? |
|---|---|---|
| /activities | A click on "Activities" in the toolbar | Yes |
| /activity/my_activities/by_type/* | A type filter applied/a link of this kind followed | No/*yes* |
| /activity/my_activities/by_time/* | A time filter applied/a link of this kind followed | Yes/yes |
| /activity/my_activities/by_user/* | A user filter applied/a link of this kind followed | No/*yes* |
| /activity/my_activities/by_time/date | "Custom date range" filter clicked | No |
| /activity/my_activities/by_time/date/* | "Custom date range" filter applied/a link of this kind followed | Yes/yes |

As in some cases it proved impossible to determine whether the user applied a filter or followed a link of the format (e.g. by refreshing the page after having applied a filter, or by copying the link and opening it in another browser tab), we decided not to track either of those URI patterns ("by_type" and "by_user"). Thus the data loss would only be negligible, while in the opposite case of always tracking both patterns, the data would be inconsistent compared to the *new* list view metrics. The task deliberately omitted type and user filter usage, since these are mostly custom, and different users can have different number of both activity types and users.

Another issue we encountered was page load time tracking in the *old* activity list view. In the *new* application page load time is calculated from the moment the user started a particular action. In the *old* page however, as it is a server-side application, the current client side stops existing and gets rebuilt anew as soon as the user requests another page.

With the goal to keep both *new* and *old* page metrics consistent, we decided to track page load time with the help of cookies storing the loading start timestamp. The process is represented in Figure 4. The main drawback of this approach is the fact that setting a cookie requires to be triggered by some user's action on the page. Thus, it does not enable us to track direct page views, which leads to a slight data loss.

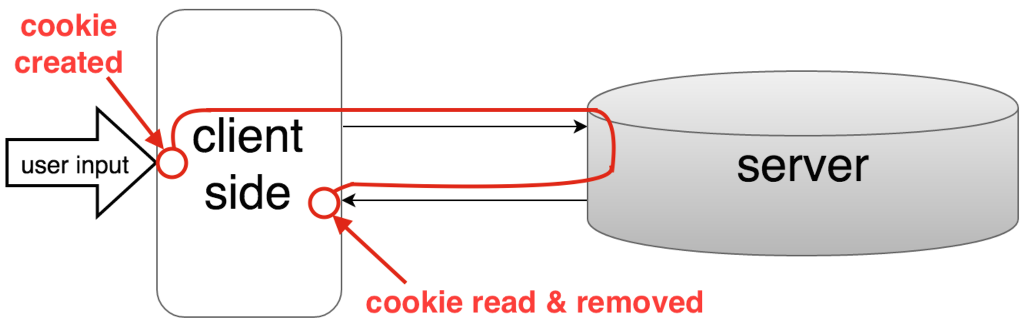


Figure 4. Using cookie to measure page load time in the *old* activity list view.

In the *old* activity list view the events causing what is considered to be a page view in terms of our task, are the same as in the *new* page, with the exception of the "stack-based navigation", which is a feature of the *new* application. This means that those events are a click on a time filter and a direct page view (except for the cases related to time and user filter usage).

As explained before, a time filter change differs from a direct page view in enabling to measure page loading time by setting a cookie with the current timestamp as soon as the user input occurred. Let us start with examining the implementation for this case.

The *loadingStart* cookie is when a time filter is clicked. The “custom date range” filter is a special case, since this filter implementation in the *old* list view is slightly faulty and resulted in loading the page twice. To avoid counting each “custom date range” filter usage as two page views, we added the cookie setting line to the *onSelect* function of the date picker, which means that the cookie is set only when the user has selected a date.

After a time filter is applied, the page is reloaded, and the global variable *app* is reinitialized in the *main.js* file. We added a variable *isPageView* that determines whether or not a particular page load should be tracked in the metrics. The first compulsory condition for a page load to qualify as an activity list page view is the presence of the *activities-list* class on the page, which means that the user has landed on the activity list view page, not on some other page of the *old* application. If this condition is met, a number of additional checks are performed to decide whether the page view represents one of the cases that were decided on for tracking.

Table 6 extends Table 5 and maps conditions used in the variable *isPageView* with the corresponding URIs that are to be tracked as activity list page views. The global variable *app* has an attribute *uri* with several methods that enable access to the full URI path, as well as to various parts of that path. Therefore, direct page views with the URI of “/activities” are detected by the check *app.uri.full_uri() === '/activities'*. When a time filter is applied, a cookie is placed on user’s machine, as mentioned before, so to detect those cases, it is enough to check if this cookie exists: *$.cookie('loadingStart')* returns either *true* or *false*. The cases when user refreshes the page after a time filter was applied or makes a direct page view from an external link of this type can be determined by the check *app.uri.segment(2) === 'by_time'* for all time filters except for the “custom date” filter, since all of them have the URI pattern “/activity/my_activities/by_time/*”, where the third segment is “by_time”. The “custom date range” filter is a more difficult case, since, as it was mentioned before, in the *old* list view it first loads the current day activities and then opens the date picker element which allows the user to select a particular date. The first load, as it was not actually requested by user, should not be tracked. It is possible to distinguish between those two loads by their URI paths: the first, automatic load uses URI path "/activity/my_activities/by_time/date", while the actual "custom date" filter is available by "/activity/my_activities/by_time/date/*", where the segment after "/date" is filled out with a particular date selected by user. To check whether or not the "custom date range" filter usage should be counted as a page view, we first check if the third segment is "by_time" and the fourth segment is "date": *app.uri.segment(2) === 'by_time' && app.uri.segment(3) === 'date'*. As meeting both of these conditions may still be the case of the first automatic load, another check is performed to detect if a fifth segment exists in the URI.

Table 6. Page view events representation in *main.js*.

| URI | Event | Condition |
|---|---|---|
| /activities | Click on "Activities" in the toolbar | `app.uri.full_uri() === '/activities'` |
| /activity/my_activities/by_time/* | A time filter applied | `$.cookie('loadingStart')` |
| | A link of this kind followed | `app.uri.segment(2) === 'by_time'` |
| /activity/my_activities/by_time/date/* | A "custom date range" filter applied | `$.cookie('loadingStart')` |
| | A link of this kind followed | `app.uri.segment(2) === 'by_time' && app.uri.segment(3) === 'date' && app.uri.segment(4)` |

After the variable *isPageView* is evaluated and if its value equals *true*, the data to send to New Relic Insights is prepared. This includes calculating page loading time (if the page loading start had been stored with a cookie), time filter name, determined by currently active link's ID, and user's company ID, which can be extracted from the global variable *user.*

The code to illustrate the described process is below:

```
/**
* from main.js
*
* 1. setting the cookie
*    1.1. for time filters (except for the “custom date range” filter)
*/
$(‘.sorting-filter-link’).on(‘click’, function() {
  $(this).addClass(‘active’);
  if ($(this).hasClass(‘time-filter’)) {
    $.cookie(‘loadingStart’, $.now(), {path: ‘/’ });
  }
});

/**
*    1.2. for the “custom date range” filter
*/
app.date_time_pickers = {
  init: function() {
    $(‘input.datepicker:not(.datepickerEnabled),
input.datepicker:not(.datepickerEnabled)’)
      <...>
      .datepicker({
        <...>
        onSelect: function(dateText, inst) {
          <...>
          $.cookie(‘loadingStart’, $.now(), {path: ‘/’ });
        },
        <...>
      });
    <...>
  },
  <...>
};

/**
* 2. sending data to New Relic Insights
*/
var isPageView = $(‘body’).hasClass(‘activities-list’) && (
                 $.cookie(‘loadingStart’) ||
                 app.uri.full_uri() === ‘/activities’ || (
                 app.uri.segment(2) === ‘by_time’ && (
                 app.uri.segment(3) === ‘date’ ? app.uri.segment(4) :
true)));
```

```
if (isPageView && _.isObject(window.newrelic)) {

  <...>

  newrelic.addPageAction('activityListMetrics:pageViews', {
    'activityListType': 'old',
    'company_id': user.company_id,
    'loadingTime': loadingFinish - loadingStart || null,
    'timeFilterName': statusFilterName || null
  });

  $.removeCookie('loadingStart', {path: '/' });
}
```

### 4.2.4 Measuring Bulk Edit Usage

In the *new* application bulk edit is implemented as a Backbone view used by all list view pages (deals, persons, organizations, activities). The bulk edit action should be sent to New Relic Insights when user had bulk edited some activities and saved the changes.

In the BulkEdit view the *updateModelsData* function is called on successful *save* event. The model of this view is a BulkEdit model that contains a collection of items. One of its attributes is *type* with the possible values of “deal”, “org”, “person” and “activity”. On model update we added a check for the model type. In case it equals “activity”, the bulk edit action will be tracked. The only attribute to the New Relic Insights action that is needed here is activity list view type: “old” or “new”.

Below are the changes added to the code:

```
// from bulk-edit/main.js
updateModelsData:
  function(data) {
    <...>

    if (this.model.type === ‘activity’) {
      app.nr.addPageAction(‘activityListMetrics:bulkEdit’, {
        ‘activityListType’: ‘new’
      }
    );
    <...>
  }
}
```

In the *old* application the bulk edit feature is designed differently. To begin bulk editing user is required to press the Bulk edit button. Then any number of list items can be selected, and each field type (column) is edited and saved separately. Consequently several *save* events can happen during one bulk edit action. When the user has finished editing, the Bulk edit button needs to be disabled. That is why we decided on the Bulk edit button disabling event to indicate the end of action in the *old* application.

The Bulk edit button is disabled with the *disableBulkEditMode* function in *main.js*. Since this function is used for other *old* list views, such as the product list view, we added a check for URI that determines that the action should only be sent to New Relic Insights if it happened in the activity list view. The other condition required is *app.bulkedit.changesMade*, which is a global variable attribute that indicates whether or not any changes were actually made during the bulk edit event. If both of the conditions are met, the bulk edit action is filled out with additional attributes and sent to New Relic Insights.

The code to illustrate the described process is below:

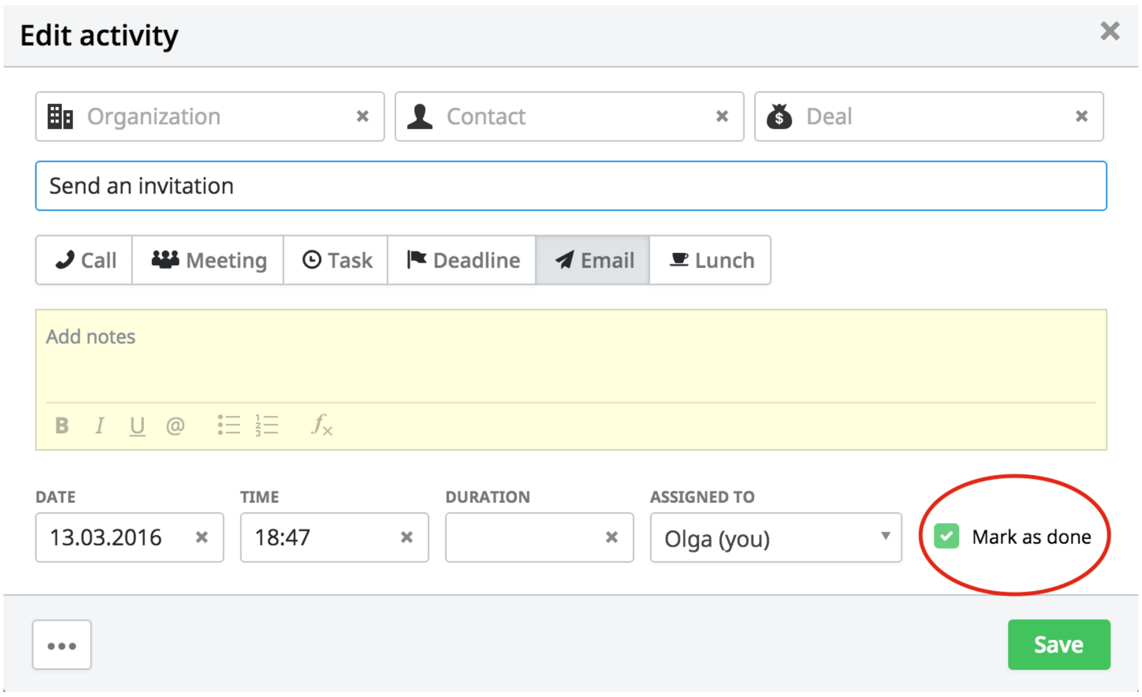
```
// from main.js
disableBulkEditMode: function(editScope, disableButton, leaveClass) {
  <...>

  if (_.contains(['activity', 'activities'], app.uri.segment(0)) &&
app.bulkedit.changesMade && _.isObject(window.newrelic)) {
    newrelic.addPageAction('activityListMetrics:bulkEdit', {
      'activityListType': 'old',
      'company_id': user.company_id
    }
  );
  app.bulkedit.changesMade = false;
  }
}
```

### 4.2.5 Measuring Marking Activities As "Undone" Usage

One of the UX improvements introduced with the *new* activity list view is the changed way for marking activities as "done"/"undone". In the *old* list view the UI for this action was confusing, users often mistakenly used the Mark as done checkboxes when they meant to bulk edit their activities. It led to the necessity to reverse those accidentally closed activities back to the "undone" status, which was impossible by the means of the *old* list view. Customers had to use a subpage of the Statistics section to do that.

Besides the case when marking activity as "undone" is used to correct marking it as "done" by mistake, there is little user stories for this action. While marking activity as "done" means that the activity was completed, e.g. a call was made or a task was accomplished, marking activity as "undone" may only mean that the activity completion conditions have changed, which is evidently not a very popular scenario in real life.

The *new* activity list view provides users with the possibility to mark activities as "undone". It can be done in 3 ways: by opening a particular activity in a modal (accessible by clicking on its subject), in the bulk edit panel, and in a single edit field if the Done column is visible. Figures 5-7 represent these options.


Edit activity

Organization | Contact | Deal

Send an invitation

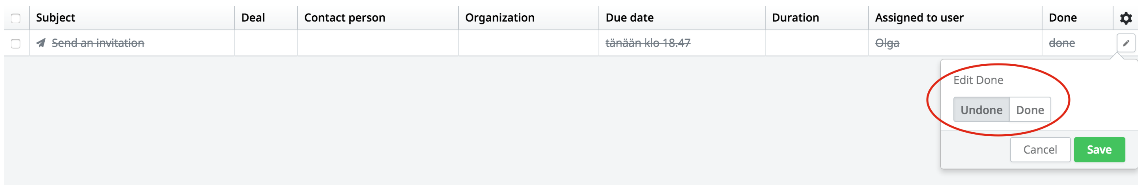
Call | Meeting | Task | Deadline | Email | Lunch

Add notes

| DATE | TIME | DURATION | ASSIGNED TO | |
|---|---|---|---|---|
| 13.03.2016 | 18:47 | | Olga (you) | ✓ Mark as done |

Save

Figure 5. Marking activity as "undone" through activity modal in the *new* activity list view.



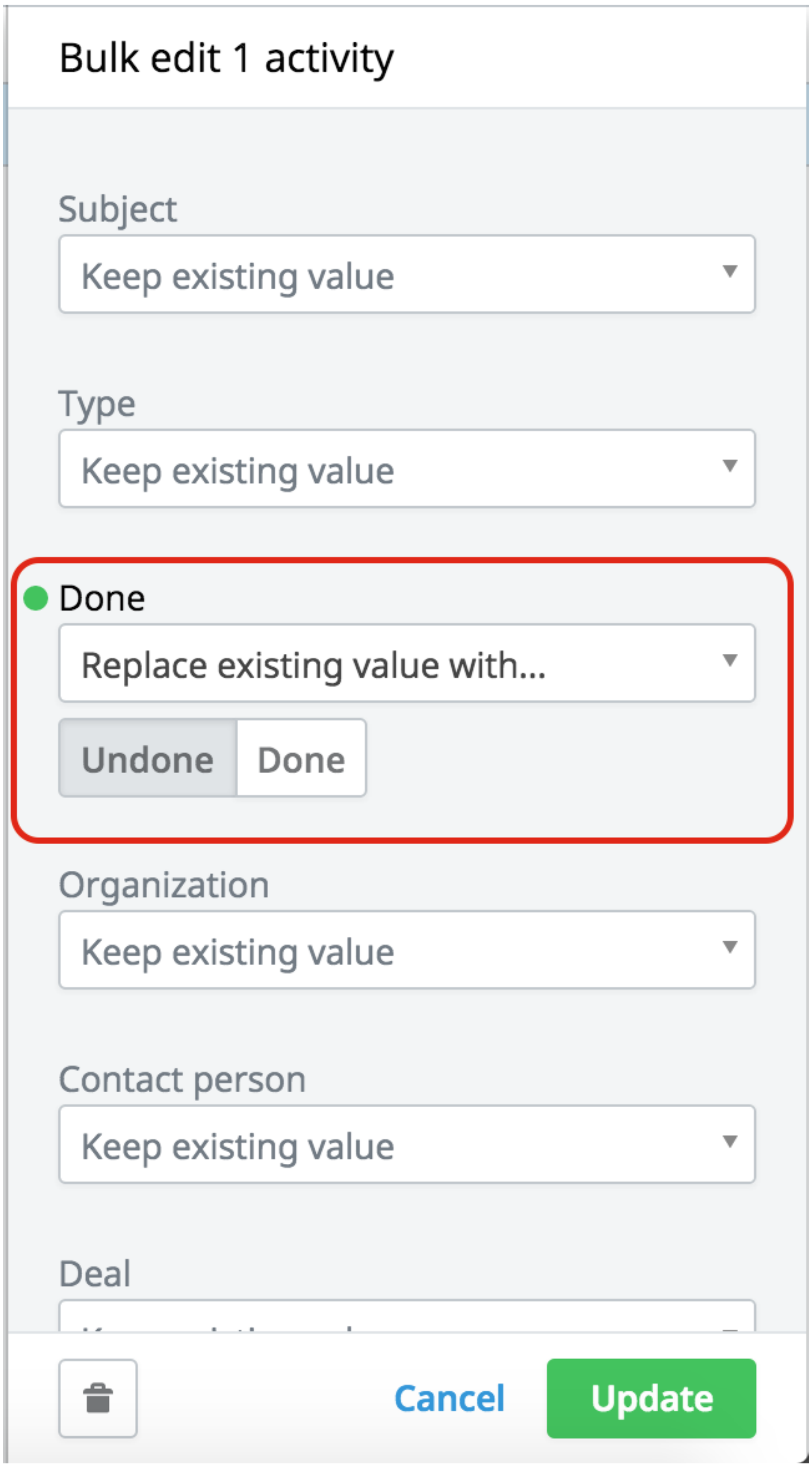
| Subject | Deal | Contact person | Organization | Due date | Duration | Assigned to user | Done |
|---|---|---|---|---|---|---|---|
| ~~Send an invitation~~ | | | | ~~tänään klo 18.47~~ | | ~~Olga~~ | ~~done~~ |

Edit Done

[✓ Undone] Done

Cancel Save

Figure 6. Marking activity as "undone" through single edit in the *new* activity list view.


Bulk edit 1 activity

Subject: Keep existing value

Type: Keep existing value

● Done: Replace existing value with...

[✓ Undone] Done

Organization: Keep existing value

Contact person: Keep existing value

Deal

Figure 7. Marking activity as "undone" through bulk edit in the *new* activity list view.

Table 7 maps each of the ways to mark activity as "undone" with user actions that should trigger the New Relic Insights action, and its place in the code.

| Event | User action | Place in code |
|---|---|---|
| Marking activity as "undone" through the modal | Open activity modal by clicking on its subject, click on a checked checkbox "Mark as done" in activity modal and save the changes | Activity-form.js: saveActivity() → activity-analytics.js: trackMarkedUndone() |
| Marking activity as "undone" through bulk edit | Select one or more activities for bulk editing, select "Replace existing value with..." in the bulk edit panel and change the value to "Undone", save the changes | Bulk-edit/main.js: updateModelsData() → activity-analytics.js: trackMarkedUndone() |
| Marking activity as "undone" through single edit | Open single select for the "Done" field, change the value to "Undone", save the changes | Fields/field.js : onSaved() → activity-analytics.js : trackMarkedUndone() |

To mark activity as undone through activity modal user should select a done activity (in the list it is distinguished by strikethrough font and grey text color) and click on its subject which makes the modal with all activity details appear. This modal is a Backbone view defined in the file *activity-form.js*. It uses a model of the class Activity which contains all the activity details as its attributes. A click on the Save button triggers the *saveActivity* function of this view. It results in the model receiving the changes and the corresponding function in *activities-analytics.js* being triggered to send the event to New Relic Insights. The call to it happens disregarding whether the *save* was successful or not due to some reasons, e.g. loss of internet connection. The reason for that is that our goal is to measure the usage of the application, not its performance or stability, in other words, how it is used, not how well it works.

The *data* variable that is passed to the *trackMarkUndone* function, contains an object, whose attributes are the attributes to be changed in the activity model. For instance, if subject and activity type were changed, the *data* object may look as follows: *{'subject': 'FE Architecture review', 'type': 'presentation'}*. The name of the attribute that represents activity completion status is "done", so in the case when user changes it from "done" to "undone" the *data* object will contain the following key-value pair: *'done': false*. The *trackMarkUndone* function checks if these particular attribute name and value are present, and if so, sends the page action *activityListMetrics:markUndone* to New Relic Insights. This action contains two attributes, one of which, *activityListType*, is similar to the previously described page actions and stores the type of the list view from where the action was performed, in this case it is "new". The second attribute is called *newListOldHabit*, and it is used for distinguishing the cases when a user whose account had already been switched to the *new* list view, did not realize that marking activities as "undone" can be done without leaving the list view, and used the Statistics page for this purpose. In case of marking activity as "undone" through the modal, *newListOldHabit* naturally equals *false*, likewise as for both of the remaining ways to do it with the means of the *new* list.

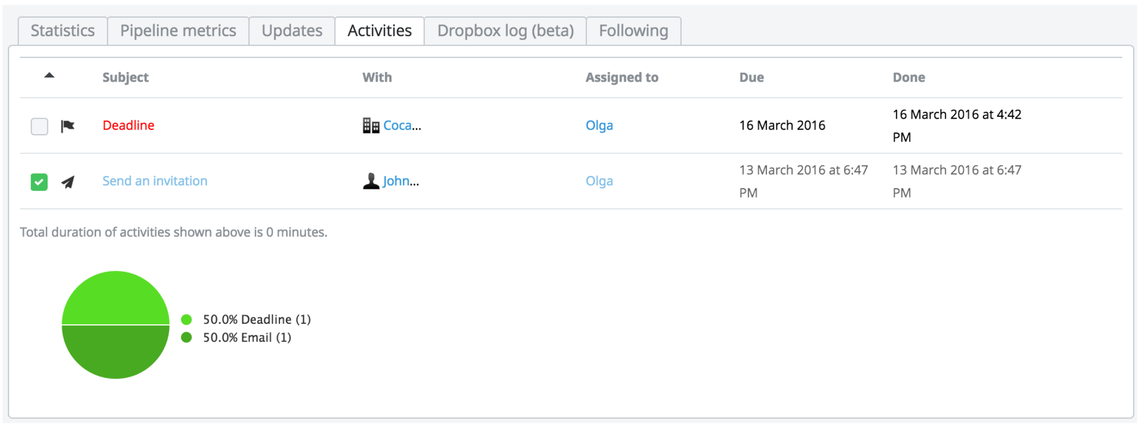
```
// from activity.js
saveActivity: function(ev) {

  <...>

  activitiesAnalytics.trackMarkUndone(data);
}

// from activities-analytics.js
trackMarkUndone: function(data) {
  if (data.done === false) {
    app.nr.addPageAction('activityListMetrics:markUndone', {
      'activityListType': 'new',
      'newListOldHabit': false
    });
  }
}
```

The logic behind the implementation of tracking bulk editing activities to the "undone" status is similar to the one described above. User selects one or more activities from the list for bulk edit, changes the value of selected activities' completion status to "undone" and clicks the Update button on the Bulk edit panel. It will fire the *save* method, which creates the *data* object that contains the changed values. At this point the New Relic Insights tracking function can be called and the *data* object passed to it.

Finally, let us examine the case when user marks activity as “undone” through single edit form. The Backbone view representing the edited field is defined in the file *fields/field.js*. The model used by this view is the activity being edited. By clicking on the Save button user triggers the *save* function, which runs a check for input validity and calls *saveModel()* if the input is valid. In the end of the saving process we sent the field value to the *trackMarkUndone* method of *activities-analytics.js* where the New Relic Insights action occurrence will be created if the value object contains attribute *done* set to *false*.

```
// from fields/field.js
save: function() {
  if (this.isSaveValid() {
    this.saveModel();
  }
},
<...>

saveModel: function() {
  <...>

  this.model.save(null, {
      query: query,
      success: _.bind(this.onSaved, this),
      <...>
    }, this)
  });
},

<...>

onSaved: function() {
  <...>
  activitiesAnalytics.trackMarkUndone(this.value);
}
```

Measuring marking activity as “undone” in the *old* application is very different. Since there is not way to perform it in the *old* activity list view itself, users have to go to the Personal Statistics page and choose the Activities tab, where they can view all the activities assigned to them and edit their status by selecting the checkbox in the first column. Figures 8 and 9 illustrate this.



| | Subject | With | Assigned to | Due | Done |
|---|---|---|---|---|---|
| ☐ | Deadline | Coca... | Olga | 16 March 2016 | 16 March 2016 at 4:42 PM |
| ✓ | Send an invitation | John... | Olga | 13 March 2016 at 6:47 PM | 13 March 2016 at 6:47 PM |

Total duration of activities shown above is 0 minutes.

Figure 8. Viewing all activities assigned to the user (both "done" and "undone") in the *old* activity list view.

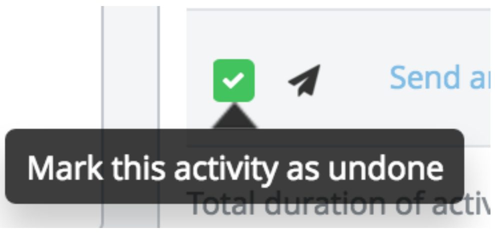


Figure 9. Marking activity as "undone" in the *old* activity list view.

As it was assumed before that marking activities as "undone" mostly occurs due to user's mistake, it was decided that it is sufficient to track the number of times this particular tab of the Personal Statistics section was visited.

Our solution for this tracking case is to send the data to New Relic Insights when the Activities tab is clicked on the Personal Statistics page. But before this happens, the *hasNewList* variable is created. Its purpose was described above, and in case of user marking activities as "undone" in the *old* way, the value of this attribute can be either *true* or *false*, depending on whether or not the user already has the *new* list view feature switched on.

```
// from main.js

app.partial_refresh = {
  init: function() {
    $(document).off('click', '.partial_refresh').on('click',
'.partial_refresh', function(event) {
      app.partial_refresh.load_partial($(this), event);
    });

    <...>
  },

  load_partial: function($obj, event) {
    var data = $obj.metadata();

    <...>

    if ($obj.text() === 'Activities' && _.isObject(window.newrelic)) {
      var hasNewList =
company_features.activity_list_beta.enabled_flag;

      newrelic.addPageAction('activityListMetrics:markUndone', {
        'activityListType': hasNewList ? 'new' : 'old',
        'newListOldHabit': hasNewList,
        'company_id': user.company_id
      });
    }
  },

  <...>
}
```

## 4.3 Data visualization and analysis

The data tracked by New Relic Insights page actions can be viewed, queried and visualized at New Relic Insights Website (http://insights.newrelic.com).

### 4.3.1 Querying Collected Data using NRQL

The queries are made with the special New Relic Query Language (NRQL), which is "an SQL-flavored query language for making calls against the Insights Events database" [2] . Basic NRQL queries examples can be found in the Appendix 1.

### 4.3.2 Conclusions on the Data Collected for the Project Based on New Relic Insights Visualizations

For visualizing, monitoring and analyzing the data, New Relic Insights provides users with the ability to create dashboards that group together several charts devoted to a particular topic. Having implemented the tracking for the project, we created a dashboard called "Old vs New activity list" in order to follow the process of the feature switch and see the results of A/B testing. Figure 10 shows a screenshot of the dashboard, taken in February 2016 when the *new* activity list view feature was introduced for approximately 50% of users.

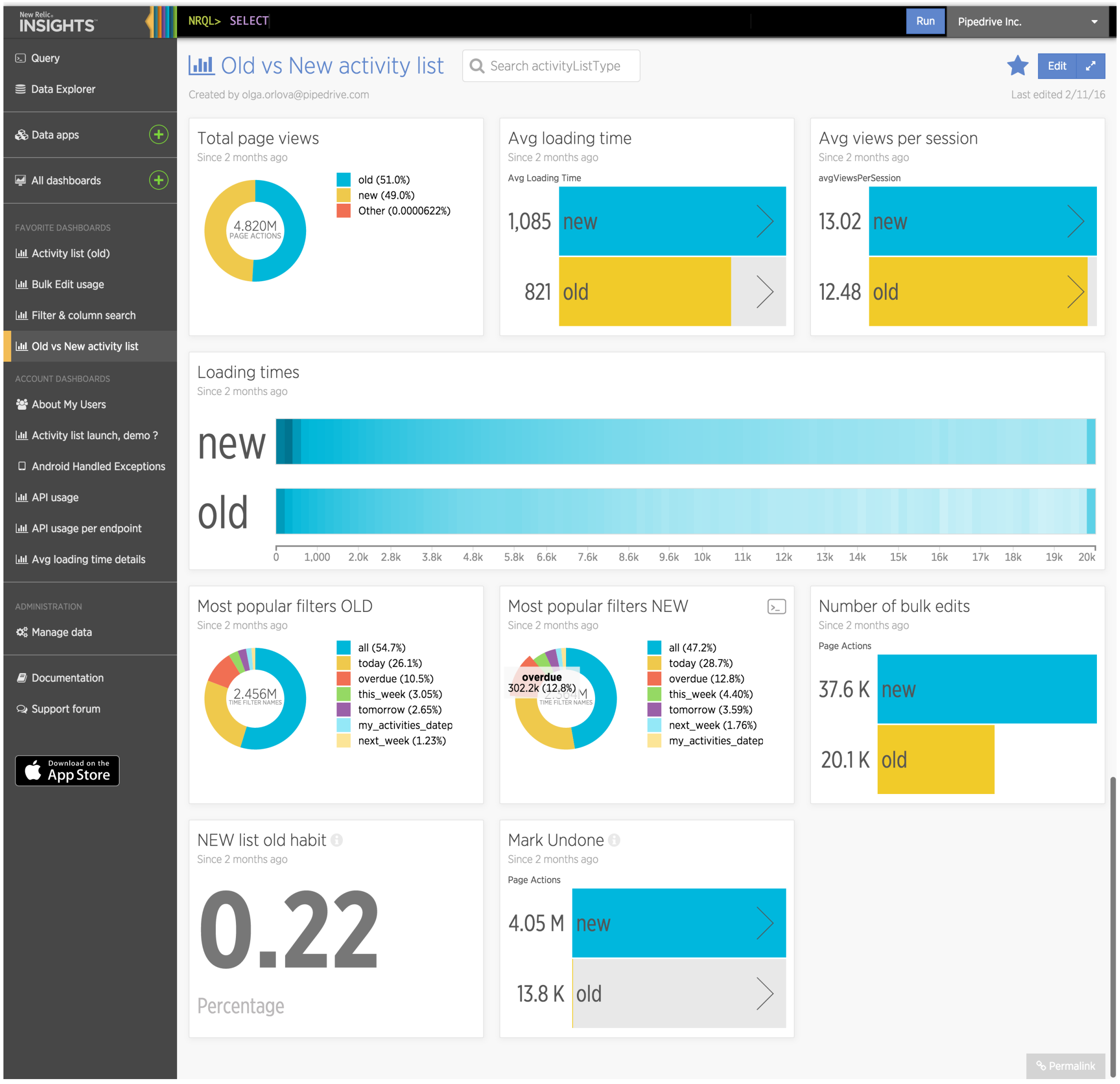


Figure 10. New Relic Insights dashboard devoted to the project.

First of all, we wanted to see how often the two list types were viewed in total. This proportion reflects the percentage of customers that use either of the list views, and as can be seen from Figure 10, at that moment it was approximately 50% for each type. Therefore it was a favourable moment to compare the usage of each of the list views and test our assumptions.

The query for this chart is:

```
SELECT count(*) FROM PageAction SINCE 2 months AGO WHERE actionName =
'activityListMetrics:pageViews' FACET activityListType;
```

We had assumed that the average loading time should go down, since the *new* activity list view introduces significant improvements in speed. With the help of New Relic Insights web analytics we can see that in February 2016, immediately after the initial introduction of the *new* activity list view feature, the average page loading time on the *new* list view pages was, contrary to expectations, higher than that on the *old* list view pages. This information prompted an investigation of possible reasons, and eventually the implementation of the *new* activity list view was debugged and improved. As a result, the *new* page loading time has decreased to 952 ms on average in April, as opposed to 1085 ms in February (Figure 11).

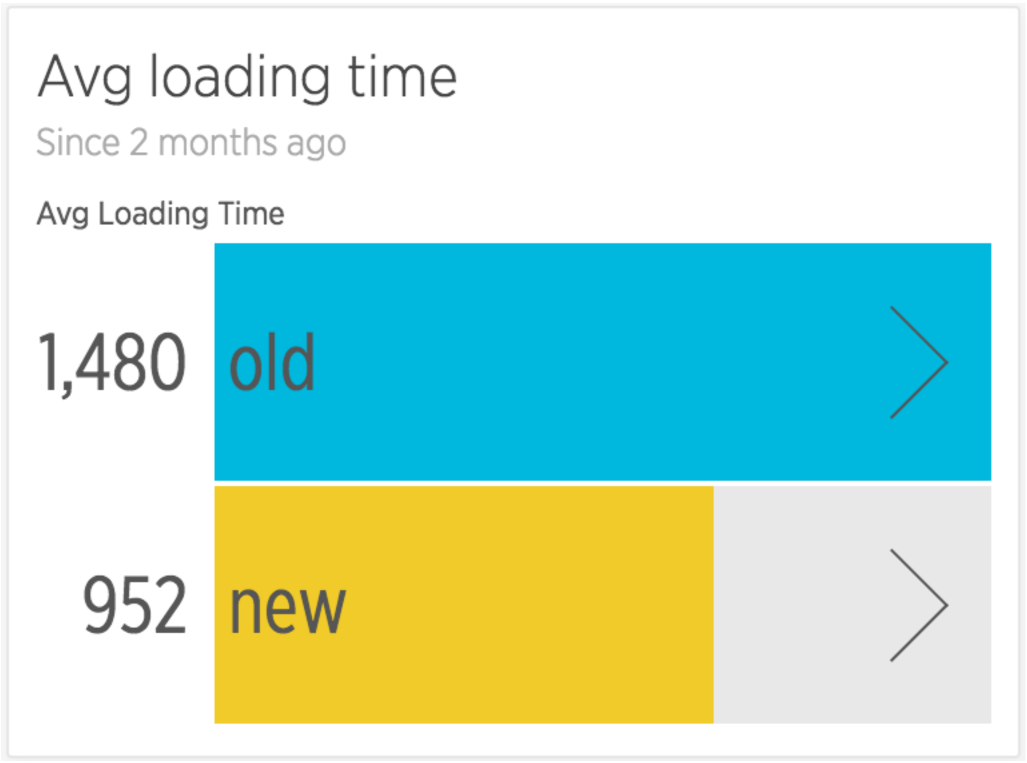


Figure 11. New Relic Insights chart depicting average page loading times in April 2016.

The query for this chart is:

```
SELECT average(loadingTime) FROM PageAction SINCE 2 months AGO WHERE
actionName = ‘activityListMetrics:pageViews’ AND loadingTime < 150000
AND loadingTime IS NOT NULL FACET activityListType;
```

As it was described in chapter 3, it was decided to count initial page loads as well as time range filter applications as a “page view” in this web analytics implementation. The chart of average views per session, thus, most probably shows a slight increase in time range filter usage, which was expected since time range filters became more convenient in the *new* activity list view version. The query was possible to execute because a unique hash code for every single session is stored as a default New Relic Insights action’s attribute.

The query:

```
SELECT count(*)/uniqueCount(session) AS ‘avgViewsPerSession’ FROM
PageAction WHERE actionName=’activityListMetrics:pageViews’ SINCE 2
months AGO FACET activityListType;
```

Apart from the average page loading time, we were also interested to see a histogram depicting a distribution of page loading times in the *old* and *new* activity list views. Despite the fact that in February the average loading time on the *new* page was higher than on the *old* page, the histogram proved that there definitely was an improvement, since the loading times in the *new* list are more focused in the left edge of the graph (0 to 800 ms).

The query:

```
SELECT histogram(loadingTime, 20000, 100) FROM PageAction SINCE 2
months AGO WHERE actionName=’activityListMetrics:pageViews’ AND
loadingTime < 150000 FACET activityListType;
```

The two time range filter ranking charts report which time range filters were selected by users of the *old* and the *new* activity list views. “All” is the default time range filter, and the charts indicate that it’s usage decreased in the *new* activity list view, therefore the conclusion that can be drawn here, is that the usage of the time filters has overall increased in the *new* list view.

The queries:

```
SELECT count(timeFilterName) FROM PageAction SINCE 2 months AGO WHERE
actionName = ‘activityListMetrics:pageViews’ AND
activityListType=’old’ FACET timeFilterName;
SELECT count(timeFilterName) FROM PageAction SINCE 2 months AGO WHERE
actionName = ‘activityListMetrics:pageViews’ AND
activityListType=’new’ FACET timeFilterName;
```

As for the bulk edit usage, we were unable to make a definite prediction on what the statistics should show. Since the page action was logged on submit, web analytics would not contain, for instance, the cases of turning the bulk edit mode on by mistake in the *old* list view. Therefore we could only assume that the bulk edit usage might remain at the same figure, or increase due to improved UX design.

The corresponding New Relic Insights dashboard chart shows a significant increase of 87% for the bulk edit usage in the *new* activity list view.

The query:

```
SELECT count(*) FROM PageAction SINCE 2 months AGO WHERE actionName =
‘activityListMetrics:bulkEdit’ FACET activityListType;
```

The way to mark activities as undone has been drastically changed in the *new* list view, so we decided to check if users have discovered the new way. The collected data showed that a negligibly small percentage of customers already switched to the *new* activity list view kept using the old way for marking activities as undone.

The query:

```
SELECT percentage(count(*), WHERE newListOldHabit IS true) FROM
PageAction WHERE actionName = ‘activityListMetrics:markUndone’ AND
activityListType=’new’ SINCE 2 months AGO;
```

Overall the *markUndone* event shows a major increase on the *new* page, even taking into account the fact that in the *new* activity list view each occurrence of the action was tracked separately, unlike the *old* activity list view. We can conclude that this solution might not have been justified, since it has evidently made the data inconsistent, as the difference between the number of events for each of the list view versions is too big to be plausible.

The query:

```
SELECT count(*) FROM PageAction WHERE actionName =
'activityListMetrics:markUndone' SINCE 2 months AGO FACET
activityListType;
```

To sum up, the collected data analysis allowed us to prove some of our assumptions: the introduction of the *new* list view increased the usage of this page and its improved features, such as time filters. New Relic Insights metrics indicated the overall more stable performance of the *new* page, however it also revealed a slowdown in the average page loading time, which made us become aware of the issue and start an investigation. Other assumptions, such as the predicted increase in the "custom date range" filter usage, were disproved, which is also a positive outcome, since it implies that this feature is not as valuable for users as we expected.

Another advantage of having a New Relic Insights dashboard with the metrics related to a new feature, is the opportunity to monitor the usage over time. Even after the A/B testing is completed, and the *new* activity list view is switched on for almost 100% of our customers, we are still able to see the usage trends.

However we also determined that using web analytics tools requires a very detailed planning and testing of the implementation. Even though we carefully analyzed the task requirements for the event of marking an activity as "undone", the metrics proved to be unreliable due to data inconsistency. Sometimes it might not be reasonable to invest a lot of time in the metrics development process, therefore web analytics tools might not be the universally best solution for measuring usability.

In Pipedrive we combined the conclusions retrieved with the help of New Relic Insights with the customers feedback in order to see the whole picture. The users reports are a valuable source of information that is impossible to obtain using web analytics tools. For instance, our customers complained about the removal of some minor features in the *new* page. As a result, out of approximately 15000 accounts switched to the *new* activity list view, 50 had to be switched back to the *old* version. The development process continued for several weeks more, with the focus on the issues revealed by both web metrics and customers feedback analysis.

# 5 Summary

In this thesis we gave a detailed description of web analytics based approach for feature introduction in web application. Although the examined case is a specific project of a particular company, it illustrates the principle of the methodology and the main stages of work: planning, implementation, data visualization and analysis.

A special emphasis was laid on the development process since implementing the metrics was my biggest personal contribution to this project. I took an initiative to create the New Relic Insights dashboard with the data visualizing graphs and charts as well, and although it was not my role to proceed with the data analysis, I was closely following the product managers' work on it.

The description of each stage reveals its potential obstacles. During the planning stage a particular attention should be paid to retain data consistency. The comprehensive description of the implementation process exposes the problems of injecting web analytics code into applications based on server-side and single-page architectures. Finally, the conclusive part briefly introduces NRQL queries, New Relic Insights dashboards and, most importantly, the basics of such kind of data analysis.

The amount and variety of work done to design and implement the metrics was beneficial for me, as it was my first task related to web analytics in general, and New Relic Insights in particular. It has been used widely in Pipedrive's application since this project, and based on my previous experience I was able to improve and accelerate the development. For instance, in later projects I chose to keep web analytics related code in separate files in order to prevent the business logic from being polluted, and call them by firing global events.

It can be said that we achieved our goal to test the web application usability when introducing a new feature. New Relic Insights has proved to be a suitable and efficient web analytics tools for the task. It cannot and was not meant to be a universal solution: as it was mentioned in the previous chapter, not all the potentially crucial usability aspects can be tracked by web analytics, therefore it is important to combine it with other sources of information, such as customers feedback. Furthermore, as New Relic Insights is designed for running and collecting data in the front-end side of an application, certain limitations may apply. We could notice that in this particular project it was not always possible to obtain all the required data from the *old*, server-side architecture based page. It was partially (except for the event of marking activities as “undone”) possible to overcome this obstacle in our project, however, it can be assumed that this tool is better suited for client-side architecture based applications.

## References

[1] Backbone.js Reference (http://backbonejs.org) (2.05.2016).

[2] NRQL Reference (https://docs.newrelic.com/docs/insights/new-relic-insights/using-new-relic-query-language/nrql-reference) (2.05.2016).

[3] Osmani, A., Developing Backbone.js Applications, O'Reilly Media, 2013, e-book.

[4] Siroker, D., Koomen, P., A/B Testing: The Most Powerful Way to Turn Clicks into Customers, John Willey & Sons, Hoboken, New Jersey, 2013.

[5] Tufte, E., Data Analysis for Politics and Policy, Prentice Hall College Dev, 1974.

[6] Tullis, T., Albert, B., Measuring the User Experience: Collecting, Analyzing and Presenting Usability Metrics, Morgan Kaufmann Publishers, 2008.

# Appendix 1 – NRQL Basic Queries Examples

The most basic NRQL query is simply returning all the events received from a specified page action. For instance:

```
SELECT * FROM PageAction WHERE actionName =
'activityListMetrics:pageViews';
```

This query returns a table containing all the attributes defined in this page action (*activityListType, loadingTime, timeFilterName*), as well as the attributes added to each New Relic Insights event by default (*timestamp, browserHeight, city, companyID*, etc).

To make the results into a pie chart or a graph, first of all it is necessary to specify what the query is reporting in the *SELECT* clause. There is a range of predefined functions that can be used here, such as *average(), uniqueCount(), max(), percentage()*, etc. The set of data on which to perform the function should be passed as an argument, moreover there may be other obligatory or optional parameters to include.

The *FACET* clause breaks the data by a defined attribute, thus allowing it to be visualized. For instance:

```
SELECT count(*) FROM PageAction WHERE actionName =
'activityListMetrics:pageViews' FACET activityListType;
```

This query creates a chart that shows the comparison between all recorded events by activity list type. As we have two activity list types, *old* and *new*, the result will be a chart with two columns: “old” and “new”, each of which will have a length corresponding to the total number of events where the *activityListType* attribute equals to *new* or *old* respectively.