CampaignChain Documentation

Release

October 16, 2016

Part I

Abstract

This is the CampaignChain documentation. It provides information to end users, administrators and developers.

Part II

What is CampaignChain?

With CampaignChain, marketers can plan, execute, monitor and optimize campaigns in a holistic overview. Programmers can integrate communication channels, tools and devices with our open-source platform.

Today, marketers face an increasing amount of online channels, tools and devices (IoT, beacons). As the opportunities for highly targeted marketing grow, so does fragmentation of workflows. CampaignChain is the only marketing technology that allows for a comprehensive real-time access to all marketing activities.

Our goal is to help everyone make innovative marketing ideas come true.

CHAPTER 1

What is CampaignChain?

CampaignChain is open-source campaign management software to plan, execute and monitor digital marketing campaigns across multiple online communication channels, such as Twitter[2], Facebook[3], Google Analytics[4] or third-party CMS, e-commerce and CRM tools.

For marketers, CampaignChain enables marketing managers to have a complete overview of digital campaigns and provides one entry point to multiple communication channels for those who implement campaigns.

1.1 Key Features

CampaignChain covers three main areas of outbound and inbound campaign management:

Planning

- Define campaign goals and milestones.
- Create and schedule campaign activities and operations on multiple online channels.
- View and modify campaign activities and operations using an interactive timeline.

Execution

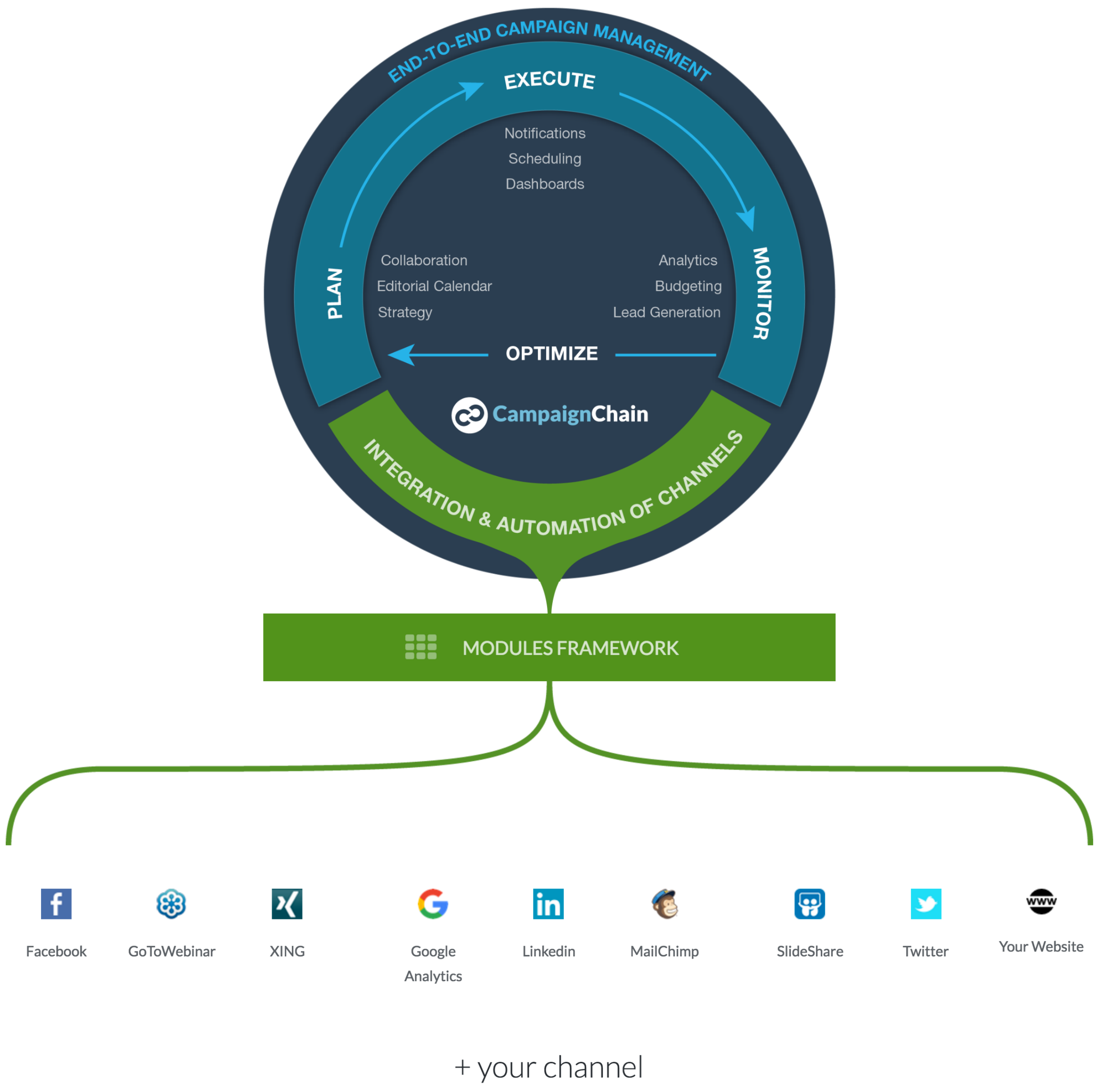
- Automatically execute scheduled activities and operations.
- Collect data for monitoring during campaign duration.
- Automatically notify responsible persons if errors occur during campaign execution.

Monitoring

- Analytics reports: Channel-specific reporting and analytics (number of Facebook views and comments, number of Twitter retweets, and so on) for accurate campaign ROI measurement.
- Budget reports: Defining budgets and spend per channel.
- Sales reports: Integrate with CRM and other tools to view and analyze leads generated by each campaign.

This is a high-level overview of the features and architecture:

[2] http://twitter.com
[3] http://facebook.com
[4] http://www.google.com/analytics



1.2 Basic Concepts

CampaignChain's software architecture has been designed along digital marketing terms and concepts in a specialized way, so this section gets you up to speed on CampaignChain's terminology and explains the main entities to you.

CampaignChain knows two types of entities, a Medium and an Action, which are:

Medium	Action
• Channel • Location	• Campaign • Milestone • Activity • Operation

1.2.1 Campaigns

Campaigns are at the core of CampaignChain, and are the "DNA of modern digital marketing"[1]. In CampaignChain, every campaign uses one or more communication **channels**. Campaigns also have **milestones** and **activities**.

Campaigns usually come in two variants: **manually scheduled campaigns**, which have a defined start and end date, and **triggered campaigns** (also called nurtured campaigns), which occur in response to user events. A campaign focused on a new product launch is an example of the former, whereas a drip email campaign that begins when a user fills up a registration form is an example of the latter.

1.2.2 Channels & Locations

Campaigns use online **channels**, which are the pathways by which campaign content reaches its audience. Common examples of channels include websites, blogs and social networks like Facebook and LinkedIn. For monitoring purposes, CampaignChain also allows connections to channels to retrieve traffic statistics (e.g. Google or YouTube Analytics) and lead generation data maintained in a CRM.

Every channel includes one or more **locations**, which allow granular publishing of campaign content. For example, a Twitter channel has only one location: the Twitter stream. However, a website channel might have various locations: a landing page, a banner on the home page, a "Contact Us" page with a form, and so on. Similarly, a LinkedIn channel might consist of two locations: a company profile page and a news stream. Locations are being created when connecting to a new Channel.

Furthermore, Locations can be created by an Operation. For example, an Operation that posts a Tweet on a Twitter stream is essentially creating a new Location (i.e. that Tweet) within a Location (i.e. a Twitter user's stream). Learn more about Operations below.

1.2.3 Milestones

Milestones are key events or reference points during a campaign. For example, the campaign go-live date could be a milestone, and a press tour could be a second milestone. When you set up campaign milestones, related actions can be defined. For example, you could compare analytics data between two milestones. Or you could notify a member of your marketing team to start working on the next set of tasks once a milestone has been reached.

1.2.4 Activities and Operations

Every location allows one or more **activities** which can be undertaken. For example, creating a new post is an example of an activity for a blog channel.

Every activity must always have at least one **operation**. For example, posting on Twitter is one activity which equals the operation.

In other cases, a single activity may encompass multiple operations. For example, defining and creating a Google AdWords campaign that runs for 3 months is a possible activity for the Google AdWords channel. However, this activity could consist of two operations: the first operation might be a Google Ad that runs for the first 2 months of the campaign, and the second operation would be a second, different Google Ad that runs for the remaining 4 weeks.

1.3 User Interface

CampaignChain's Web-based user interface is responsive and works on Desktop computers as well as mobile devices such as Tablets and Smartphones.

[1] This terminology was used by Lars Trieloff in his Feb 2014 presentation, which also inspires CampaignChain's architecture.

1.4 Footnotes

CHAPTER 2

What is CampaignChain?

CampaignChain is open-source campaign management software to plan, execute and monitor digital marketing campaigns across multiple online communication channels, such as Twitter[6], Facebook[7], Google Analytics[8] or third-party CMS, e-commerce and CRM tools.

For developers, CampaignChain is a platform to integrate key marketing campaign management functions with data from multiple channels. It is implemented in PHP[9] on top of the Symfony framework[10].

2.1 Key Features

CampaignChain covers three main areas of outbound and inbound campaign management:

Planning

- Define campaign goals and milestones.
- Create and schedule campaign activities and operations on multiple online channels.
- View and modify campaign activities and operations using an interactive timeline.

Execution

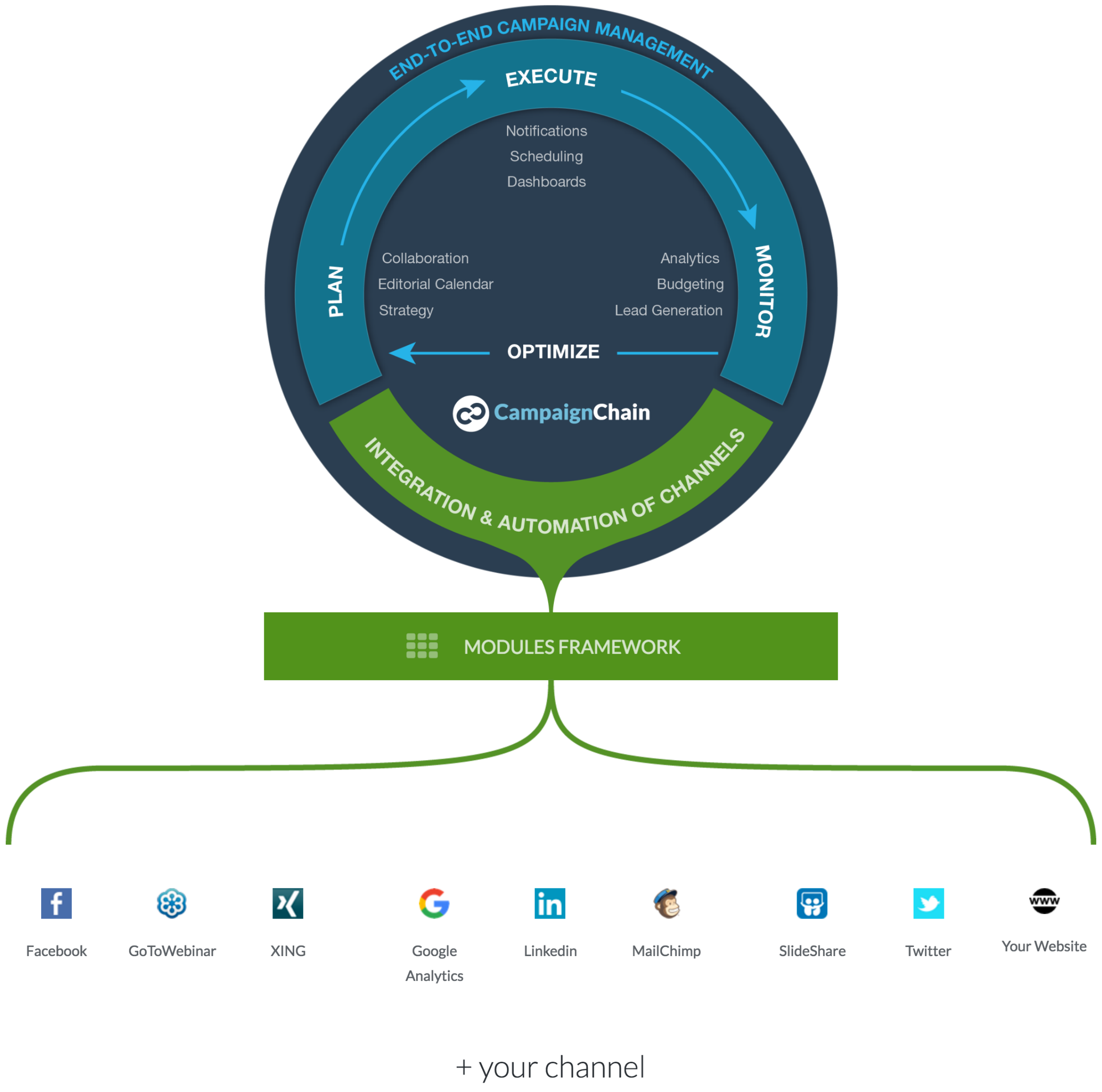
- Automatically execute scheduled activities and operations.
- Collect data for monitoring during campaign duration.
- Automatically notify responsible persons if errors occur during campaign execution.

Monitoring

- Analytics reports: Channel-specific reporting and analytics (number of Facebook views and comments, number of Twitter retweets, and so on) for accurate campaign ROI measurement.
- Budget reports: Defining budgets and spend per channel.
- Sales reports: Integrate with CRM and other tools to view and analyze leads generated by each campaign.

This is a high-level overview of the features and architecture:

[6] http://twitter.com
[7] http://facebook.com
[8] http://www.google.com/analytics
[9] http://php.net
[10] http://symfony.com



2.2 Basic Concepts

CampaignChain's software architecture has been designed along digital marketing terms and concepts in a specialized way, so this section gets you up to speed on CampaignChain's terminology and explains the main entities to you.

CampaignChain knows two types of entities, a Medium and an Action, which are:

Medium	Action
• Channel • Location	• Campaign • Milestone • Activity • Operation

2.2.1 Campaigns

Campaigns are at the core of CampaignChain, and are the “DNA of modern digital marketing”[1]. In CampaignChain, every campaign uses one or more communication **channels**. Campaigns also have **milestones** and **activities**.

Campaigns usually come in two variants: **manually scheduled campaigns**, which have a defined start and end date, and **triggered campaigns** (also called nurtured campaigns), which occur in response to user events. A campaign focused on a new product launch is an example of the former, whereas a drip email campaign that begins when a user fills up a registration form is an example of the latter.

2.2.2 Channels & Locations

Campaigns use online **channels**, which are the pathways by which campaign content reaches its audience. Common examples of channels include websites, blogs and social networks like Facebook and LinkedIn. For monitoring purposes, CampaignChain also allows connections to channels to retrieve traffic statistics (e.g. Google or YouTube Analytics) and lead generation data maintained in a CRM.

Every channel includes one or more **locations**, which allow granular publishing of campaign content. For example, a Twitter channel has only one location: the Twitter stream. However, a website channel might have various locations: a landing page, a banner on the home page, a “Contact Us” page with a form, and so on. Similarly, a LinkedIn channel might consist of two locations: a company profile page and a news stream. Locations are being created when connecting to a new Channel.

Furthermore, Locations can be created by an Operation. For example, an Operation that posts a Tweet on a Twitter stream is essentially creating a new Location (i.e. that Tweet) within a Location (i.e. a Twitter user’s stream). Learn more about Operations below.

2.2.3 Milestones

Milestones are key events or reference points during a campaign. For example, the campaign go-live date could be a milestone, and a press tour could be a second milestone. When you set up campaign milestones, related actions can be defined. For example, you could compare analytics data between two milestones. Or you could notify a member of your marketing team to start working on the next set of tasks once a milestone has been reached.

2.2.4 Activities and Operations

Every location allows one or more **activities** which can be undertaken. For example, creating a new post is an example of an activity for a blog channel.

Every activity must always have at least one **operation**. For example, posting on Twitter is one activity which equals the operation.

In other cases, a single activity may encompass multiple operations. For example, defining and creating a Google AdWords campaign that runs for 3 months is a possible activity for the Google AdWords channel. However, this activity could consist of two operations: the first operation might be a Google Ad that runs for the first 2 months of the campaign, and the second operation would be a second, different Google Ad that runs for the remaining 4 weeks.

2.2.5 Operations and Locations

Locations are created when connecting to a new Channel or by an Operation. Upon creation by a Channel, the URL of the Location is usually known and can be stored in the system when creating the new Location. For example,

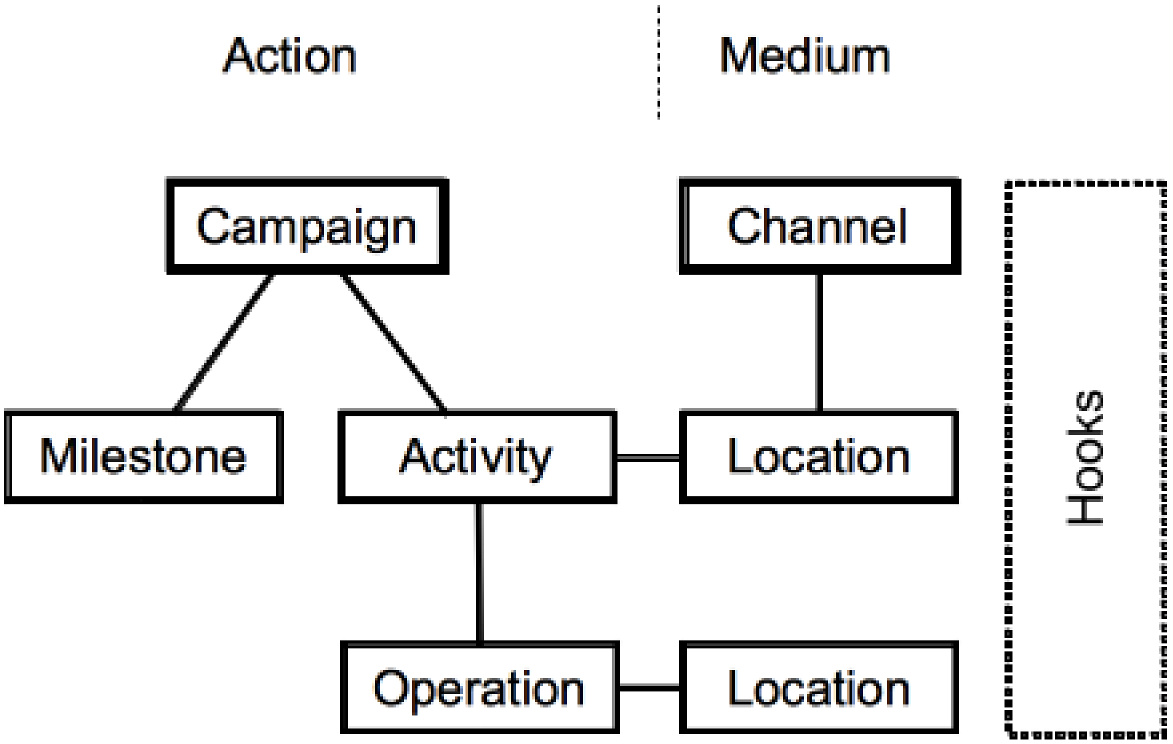
[1] This terminology was used by Lars Trieloff in his Feb 2014 presentation, which also inspires CampaignChain’s architecture.

when connecting a new Twitter user stream to CampaignChain, the user's URL on Twitter will be accessible (e.g. www.twitter.com/ordnas).

This is different when it comes to Operations. An Operation could well create a Location stub without the URL and only provide the URL after the Operation has been executed. For example, the URL of a scheduled tweet will only be generated by Twitter once the tweet has been posted. Hence, CampaignChain allows Operations to create Locations without a URL, but requires them to provide a URL when the Operation gets executed.

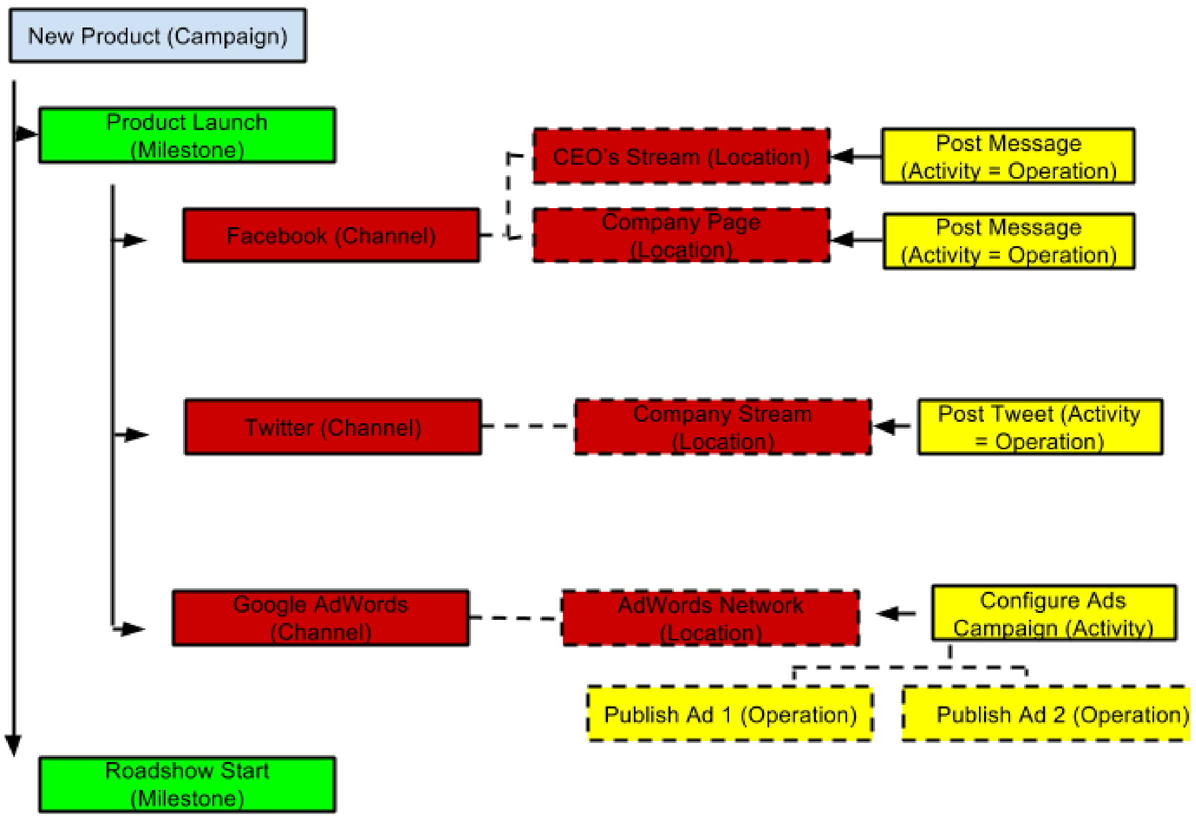
2.2.6 Visual Summary

The following diagram explains the relationship between the various entities.



It should be clear from this diagram an Activity is never related directly to a Channel. The relationship is always Channel -> Location -> Activity -> Operation.

A more concrete example of this relationship is illustrated below.


New Product (Campaign)
Product Launch (Milestone)
Facebook (Channel)
CEO's Stream (Location)
Company Page (Location)
Post Message (Activity = Operation)
Post Message (Activity = Operation)
Twitter (Channel)
Company Stream (Location)
Post Tweet (Activity = Operation)
Google AdWords (Channel)
AdWords Network (Location)
Configure Ads Campaign (Activity)
Publish Ad 1 (Operation)
Publish Ad 2 (Operation)
Roadshow Start (Milestone)

2.3 Modules and Hooks

CampaignChain has been designed so that it does not require you to replace existing digital marketing applications. Instead, it serves as a platform for integrating such applications and acts as a cockpit for managing digital marketing campaigns.

2.3.1 Modules

Due to CampaignChain's **modules** architecture, any online channel along with its locations can be integrated. Furthermore, custom campaigns, milestones, activities and operations can be developed. Given that CampaignChain is built on top of the Symfony framework, modules can use functionality provided by other modules.

2.3.2 Hooks

Hooks are reusable components that provide common functionality and can be used across modules to configure campaigns, milestones, channels, locations, activities and operations. CampaignChain already provides a number of hooks and developers can easily add new ones.

For example, CampaignChain comes with an assignee hook, which makes it possible to assign specific channels or activities to members of a marketing team. Similarly, CampaignChain's due date hook can be used to specify a due date for a Twitter post activity; the same hook can be reused to define a due date for a campaign milestone.

2.4 Call to Action

CampaignChain allows tracking Calls to Action across various Channels and Locations to understand which Operations had the highest impact. Imagine the following conversion funnel:

1. A Twitter post links to a landing page on a website.
2. The landing page includes a registration form to download something.
3. All the personal data collected in the form will be saved as leads in a CRM.

With CampaignChain, you will be able to understand how many leads have been generated by that specific Twitter post.

Learn more about the details of CampaignChain's Call to Action (CTA) Tracking (page 36).

2.5 User Interface

CampaignChain's Web-based user interface has been implemented with Bootstrap 3[11]. Thus, it is responsive and works on desktop computers as well as mobile devices such as tablets and smartphones.

2.6 Footnotes

[11] http://getbootstrap.com

Part III

CampaignChain Community Edition

The CampaignChain Community Edition is available under an Open Source license on GitHub. It is the first open-source platform for marketing integration.

CHAPTER 3

CampaignChain Community Edition

The CampaignChain Community Edition is available under an Open Source license on GitHub. It is the first open-source platform for marketing integration.

3.1 Developers

Get started with CampaignChain quickly in this step-by-step guide.

3.1.1 Quick Tour for Developers

Learn how to start developing with CampaignChain. This page gives you some pointers to the most important concepts behind CampaignChain, how to install it and a tutorial that explains how to create your first modules to connect to an online channel.

Installation

Install CampaignChain with some sample data in 10 minutes! Learn all about the system requirements, step-by-step installation and configuration as well as how to load the sample data in the `Community Edition installation` tutorial.

Architecture

Before you start developing with CampaignChain, please make yourself familiar with the following concepts of its software architecture:

1. Features, entities, calls to action (page 13)
2. General introduction to modules (page 25)

Development

Customizing and enhancing CampaignChain can all be done through modules. There is an in-depth tutorial available that shows you how to Connect a new Online Channel (page 47).

Learn the general concepts of developing with CampaignChain.

3.1.2 The Developer Book

Development Mode

It is highly recommended that you configure CampaignChain to run in development mode while you work on the code.

To enable development mode, set the `campaignchain.env``parameter to ``true` in *app/config/parameters.yml*.

You should only do this prior to a fresh installation of CampaignChain and not switch back to production mode for that installation.

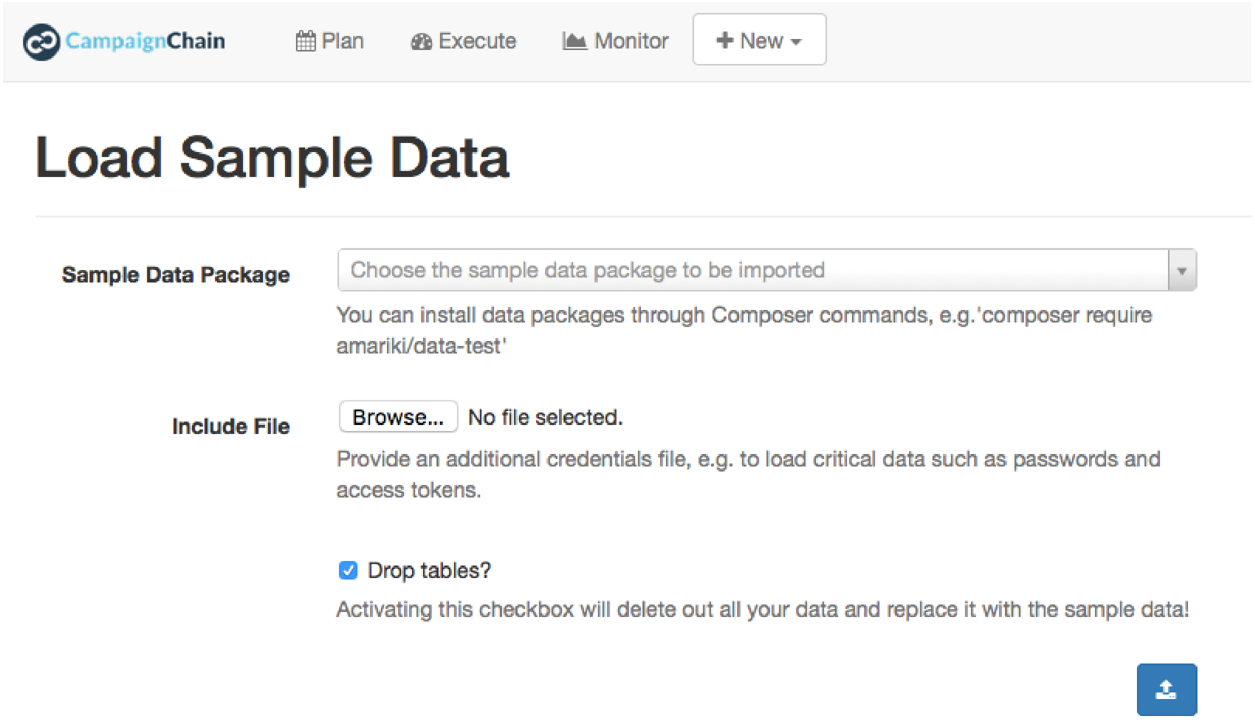
When in development mode, the following happens:

require-dev in composer.json

While in development mode, CampaignChain will upon installation also register CampaignChain modules which have been defined in the `require-dev` section of your project's *composer.json* file.

Development Tools

The navigation bar will display an icon to access various developer tools from within the CampaignChain user interface, such as loading sample data (page 40).



Load Sample Data

Sample Data Package: Choose the sample data package to be imported

You can install data packages through Composer commands, e.g.'composer require amariki/data-test'

Include File: Browse... No file selected.

Provide an additional credentials file, e.g. to load critical data such as passwords and access tokens.

[✓] Drop tables?

Activating this checkbox will delete out all your data and replace it with the sample data!

Modules Repositories

You can specify modules repositories that should be used in development mode instead of the ones you defined for the production instance.

The development modules repositories can be defined in the *campaignchain.yml* of a distribution module.

```
modules:
    repositories:
        - http://www.example.com/modules/
    repositories-dev:
        - http://www.example.com/modules/dev/
```

In case no development modules repositories have been defined under `repositories-dev`, CampaignChain will fall back and use the ones specified under `repositories`.

Module Basics

CampaignChain uses a modular architecture, allowing developers to integrate any online channel along with its locations, activities and operations.

This document provides an overview of common concepts that developers should know when developing CampaignChain modules, e.g. for custom channels, locations, activities and operations.

Framework

Symfony CampaignChain has been built on top of the PHP-based Symfony framework[13]. Therefore, custom modules should also be developed with Symfony.

Doctrine Within Symfony, CampaignChain uses Doctrine[14] as its Object-Relation Mapper (ORM). This allows usage of various databases in the back-end.

Bootstrap 3 CampaignChain's GUI is based on Bootstrap 3[15] and module developers should follow its best practices of responsive design.

Types of Modules

CampaignChain's core can be extended through various types of modules, each covering a certain feature set. The following pre-defined types exist:

- *Activity*, e.g. post on Facebook or Twitter.
- *Campaign*, to develop custom campaign functionality (e.g. nurtured campaigns).
- *Channel*, to connect to channels such as Facebook or Twitter.
- *Location*, to manage e.g. various Facebook pages.
- *Milestone*, e.g. to develop a new kind of milestone besides the default one with a due date.
- *Operation*, similar to Activity module type.
- *Report*, to create custom analytics, budget or sales reports for ROI monitoring.
- *Security*, e.g. functionality for channels to log in to third-party systems.

[13] http://symfony.com/
[14] http://doctrine-project.org
[15] http://getbootstrap.com

- *Distribution*, an aggregation of bundles and system-wide configuration, e.g. the CampaignChain Community Edition[16].

The concepts in this document apply to all types of modules.

- For information specific to building channel and location modules, refer to the Channels and Locations page.
- For information specific to building activity and operation modules, refer to the Activities and Operations page.

Packaging

CampaignChain modules are developed as Symfony bundles[17]. One Symfony bundle must contain at least one CampaignChain module and can contain various CampaignChain modules of the same type.

To allow CampaignChain to install a bundle along with its module(s), the bundle must contain the following two configuration files in its root:

- *composer.json*: CampaignChain modules (residing inside a Symfony bundle), are installed/distributed as Composer[18] packages. This file holds information relevant to Composer.
- *campaignchain.yml*: This file holds all the CampaignChain-specific module configuration parameters.

Since CampaignChain is built on top of the Symfony framework, modules can use functionality provided by other modules mainly through Symfony services[19].

Versioning

The version number of module packages for CampaignChain must follow the syntax laid out in the Semantic Versioning[20] specification.

Bundle Generation

When building an CampaignChain module, the first step is to create a new Symfony bundle[21].

Configuration Files

Every bundle with CampaignChain modules must have the following two configuration files, which are essential for CampaignChain to correctly identify and integrate the included module(s).

These files must be located in the root of the bundle directory.

composer.json The bundle's *composer.json* file follows standard Composer conventions. The *type* parameter must belong to the set of pre-defined module types as outlined previously. Here is a list of parameters typically seen in this file:

- require: A list of package dependencies
- description: A human-readable description for the bundle
- keywords: Additional descriptive keywords for the bundle

[16] https://github.com/CampaignChain/distribution-ce
[17] http://symfony.com/doc/current/cookbook/bundles/index.html
[18] https://getcomposer.org
[19] http://symfony.com/doc/current/book/service_container.html
[20] http://semver.org/
[21] http://symfony.com/doc/current/bundles/SensioGeneratorBundle/commands/generate_bundle.html

- homepage: A link to the bundles's website
- license: The license under which the bundle and its modules are made available
- authors: A list of package authors

Example:

```
{
   "name": "campaignchain/channel-twitter",
   "description": "Connect with Twitter.",
   "keywords": ["twitter","oauth"],
   "type": "campaignchain-channel",
   "homepage": "http://www.groganz.com",
   "license": "Proprietary",
   "authors": [
       {
           "name": "Sandro Groganz",
           "email": "sandro@campaignchain.com"
       }
   ],
   "require": {
       "campaignchain/core": "dev-master",
       "campaignchain/security-authentication-client-oauth": "dev-master"
   }
}
```

In addition to the schema of the composer.json file[22] developers of CampaignChain modules should also follow the best practices outlined below.

Parameter *name* The name of the bundle. Typically this is the application name or vendor name, followed by a separating slash (/), then the module type followed by a dash and the bundle's purpose.

The schematic representation of the syntax is: <application or vendor name>/<bundle type>-<purpose of bundle>

Example: campaignchain/channel-twitter

Parameter *type* The type of the bundle, which must be one of

- campaignchain-channel
- campaignchain-location
- campaignchain-activity
- campaignchain-operation
- campaignchain-report
- campaignchain-campaign
- campaignchain-security
- campaignchain-milestone

Custom types are not supported and CampaignChain will display an error if it encounters a type value outside the above allowed set.

[22] https://getcomposer.org/doc/04-schema.md

Other Parameters Required by CampaignChain

- description: A human-readable description for the bundle
- keywords: Additional descriptive keywords for the bundle
- homepage: A link to the bundles's website
- license: The license under which the bundle and its modules are made available
- authors: A list of package authors

campaignchain.yml The bundle's *campaignchain.yml* file specifies all CampaignChain modules contained in the bundle. Per module, it defines parameters such as the internal name of the module, used to reference it from other modules, as well as any associated Symfony routes[23] and Symfony services[24] or CampaignChain hooks. The information in the file varies depending on the module type and requirements.

The typical structure of the *campaignchain.yml* file is as follows:

```
modules:
    |module-identifier|:
        display_name: |display name|
        channels:
            - |channel identifier|
            - |channel identifier|
            ...
        services:
            - job: |service identifier|
        routes:
            - new: |route identifier|
            - edit: |route identifier|
            - edit_modal: |route identifier|
            - edit_api: |route identifier|
            - read: |route identifier|
        hooks:
            - |hook-name|: |true|false|
            - |hook-name|: |true|false|
            ...
        system:
            navigation:
                settings:
                    - [|Nav item name|, |symfony_route|]
                    ...
                ...
    |module-identifier|:
        ...
```

Example: An activity module's campaignchain.yml file lists the channels the activity belongs to and the Symfony routes to create and edit new activities.

```
modules:
   campaignchain-twitter-update-status:
       display_name: 'Update Status'
       channels:
           - campaignchain/channel-twitter/campaignchain-twitter
       services:
           job: campaignchain.activity.twitter.job.update_status
       routes:
```

[23] http://symfony.com/doc/current/book/routing.html
[24] http://symfony.com/doc/current/book/service_container.html

```
            new: campaignchain_activity_twitter_update_status_new
            edit: campaignchain_activity_twitter_update_status_edit
            edit_modal: campaignchain_activity_twitter_update_status_edit_modal
            edit_api: campaignchain_activity_twitter_update_status_edit_api
        hooks:
            campaignchain-due: true
            campaignchain-duration: false
            campaignchain-assignee: true
```

Module Identifier The module's identifier should be provided as the child of the *modules* parameter. Multiple modules can be specified in this way. The recommended syntax of the module identifier is to use dashes (-) to separate words, which helps to separate it from the parameters which use underscores. Furthermore, the identifier should start with an application or vendor name followed by a string that best captures the purpose of the module.

In sum, the recommended syntax is: <application or vendor name>-<purpose of module>

Example: campaignchain-twitter-update-status

Note: It is important to note that **the module identifier must be unique per module type across bundles**. In other words: In a bundle, only CampaignChain modules of the same type are allowed and the identifier of each module must be unique in all bundles containing the same type of modules.

Parameter *display_name* All modules have to specify the module name that will be displayed in CampaignChain's graphical user interface by providing a string as the value of the *display_name* parameter.

Parameter *services* A module can define the following services to be consumed by CampaignChain.

- job: This service will be called by CampaignChain's scheduler to automatically execute functionality, e.g. publishing a scheduled post to Twitter.

Parameter *routes* Within the *campaignchain.yml* configuration file, CampaignChain recognizes four types of Symfony routes.

- new: The route to invoke when creating a new Channel, Location, Activity, Operation
- edit: The route to invoke when editing an existing Channel, Location, Activity, Operation
- edit_modal: The route to invoke for the pop-up view of the 'edit' route
- edit_api: The route to invoke for the submit action of the 'edit_modal' route
- read: The route where information can be viewed

Parameter *hooks* Hooks can be assigned to a module by specifying the hook's identifier and *true* to activate it or *false* to deactivate it. If a hook is omitted, CampaignChain will regard it as inactive.

Parameter *system* This parameter allows a module to define system-wide configuration options. For example, to add a new navigation item to the settings navigation menu available in the header of CampaignChain's graphical user interface.

Parameters Specific to a Module Type Some module types require certain parameters in the *campaignchain.yml* configuration file to be defined. For example, an Activity module should list at least one related channel module. Similarly, an Operation module must define whether it creates a Location or not. You will find more detailed information in the documentation related to a specific module type.

Channel and Location Modules

This document provides an overview of concepts that developers should know when developing Channel and Location modules.

Note: Every Channel must include at least one Location.

Naming Conventions (Channel)

composer.json

- The *name* parameter should be of the form [application name or vendor name]/channel-[channel name].

  Example: campaignchain/channel-twitter

- The *type* parameter should be ‘campaignchain-channel’.

campaignchain.yml

- The name of a Channel module should follow the convention [application name or vendor name]-[channel name].

  Example: campaignchain-twitter

Naming Conventions (Location)

composer.json

- The *name* parameter should be of the form [application name or vendor name]/location-[channel name]-[bundle purpose].

  Example: campaignchain/location-twitter-status-update

- The *type* parameter should be ‘campaignchain-location’.

campaignchain.yml

- The name of a Location module should follow the convention [application name or vendor name]-[channel name]-[location descriptor].

  Example: campaignchain-twitter-user

Wizards and Routes

CampaignChain provides a set of “wizards” that ease integration of your module into the CampaignChain GUI. The Channel Wizard takes care of redirecting the client browser to the appropriate routes when creating or editing a channel and/or location.

Channel and Location module developers should use the Channel Wizard as a convenient way to attach and persist a new location for a channel. The Channel Wizard can be invoked from within a controller using the service identifier ‘campaignchain.core.channel.wizard’ as shown below.

```php
<?php
// invoke and use channel wizard
$wizard = $this->get('campaignchain.core.channel.wizard');
$wizard->setName($profile->displayName);
$wizard->addLocation($location->getIdentifier(), $location);
$channel = $wizard->persist();
$wizard->end();
```

The route used by the Channel Wizard is obtained from the module configuration in the Channel bundle's *campaignchain.yml* file:

```
modules:
   campaignchain-linkedin:
       display_name: LinkedIn
       routes:
           new: campaignchain_channel_linkedin_create
```

Location Module and Service

The Channel Wizard's *addLocation()* method should be passed a Location object representing the location to be added to the channel. CampaignChain's Location service can be used to retrieve the correct Location module, using the Location bundle name and Location module identifier. The location service is available using the identifier 'campaignchain.core.location'.

```php
<?php
$locationService = $this->get('campaignchain.core.location');
$locationModule = $locationService->getLocationModule(
  'campaignchain/location-linkedin', 'campaignchain-linkedin-user');
```

In this example, the Location service finds the Location module named 'campaignchain-linkedin-user' in the bundle named 'campaignchain/location-linkedin'.

Channel Authentication

CampaignChain provides an OAuthBundle (based on HybridAuth) which can be used for OAuth-based authentication with online channels. The client can be accessed as a Symfony service using the service identifier 'campaignchain.security.authentication.client.oauth.application'.

```php
<?php
$oauthApp = $this->get(
  'campaignchain.security.authentication.client.oauth.application');
$application = $oauthApp->getApplication(self::RESOURCE_OWNER);

if(!$application){
   return $oauthApp->newApplicationTpl(self::RESOURCE_OWNER,
     $this->applicationInfo);
}
else {
   return $this->render(
       'CampaignChainChannelLinkedInBundle:Create:index.html.twig',
       array(
           'page_title' => 'Connect with LinkedIn',
           'app_id' => $application->getKey(),
       )
   );
}
```

The client's *getApplication()* method retrieves any existing channel credentials (that were previously configured) from the CampaignChain database. In case no such credentials exist (such as the first time a location is created), the *getApplicationTpl()* method generates a Web form for the user to input the required data.

CampaignChain also provides an OAuth authentication client via the 'campaignchain.security.authentication.client.oauth.authentication' identifier. The client's *authenticate()* method can be used to perform authentication against the remote service.

```php
<?php
$oauth = $this->get(
  'campaignchain.security.authentication.client.oauth.authentication');
$status = $oauth->authenticate(self::RESOURCE_OWNER,
  $this->applicationInfo);
```

Note: CampaignChain's OAuthBundle is an optional bundle to ease authentication with third-party services. Developers are free to implement their own authentication client, or use third-party clients as needed.

Channel Icon

Each Channel module must provide a channel icon image in PNG format with size 16x16 pixels. The image file must reside in the *your-project/src/your-bundle-namespace/Resources/public/images/icons/16x16/* folder of the bundle and the image's file name should match the descriptive string used at the end of the bundle name.

Example: The bundle named 'campaignchain/channel-google' would have its icon reside at your-project/src/Acme/CampaignChain/Channel/GoogleBundle/Resources/public/images/icons/16x16/google.png.

Activity and Operation Modules

This document provides an overview of concepts that developers should know when developing Activity and Operation modules.

Note: Every Activity must include at least one Operation and at least one Job.

Naming Conventions (Activity)

composer.json

- The *name* parameter should be of the form [application name or vendor name]/activity-[bundle purpose].

  Example: campaignchain/activity-twitter

- The *type* parameter should be 'campaignchain-activity'.

campaignchain.yml

- The name of an Activity module should follow the convention [application name or vendor name]-[channel name]-[bundle purpose].

  Example: campaignchain-twitter-update-status

Naming Conventions (Operation)

composer.json

- The *name* parameter should be of the form [application name or vendor name]/operation-[bundle purpose].

  Example: campaignchain/operation-twitter

- The *type* parameter should be 'campaignchain-operation'.

campaignchain.yml

- The name of an Operation module should follow the convention [application name or vendor name]-[channel name]-[operation descriptor].

  Example: campaignchain-twitter-update-status

Linking Activities and Channels

The Activity bundle's *campaignchain.yml* file should contain a *channels* parameter, which specifies the link between the Channel and the Activity. The *channels* parameter should be of the form [channel bundle name]/[channel module name]

Example: 'acme/channel-linkedin/acme-linkedin' refers to the Channel bundle 'acme/channel-linkedin' and the Channel module within it named 'acme-linkedin'.

Wizards and Routes

CampaignChain provides a set of "wizards" that ease integration of your module into the CampaignChain GUI. The Activity Wizard takes care of presenting the user with a form that lists available campaigns, channels and operations. Based on the user's selection in the form, the Activity Wizard is able to retrieve and display the available Operations for the selected Channel and link it to the selected Campaign.

The Activity Wizard can be invoked from within a controller using the service identifier 'campaignchain.core.activity.wizard' as shown below.

```php
<?php
// invoke and use activity wizard
$wizard = $this->get('campaignchain.core.activity.wizard');
$campaign = $wizard->getCampaign();
$activity = $wizard->getActivity();
```

The routes and display name used by the Activity Wizard are obtained from the module configuration in the Activity bundle's *campaignchain.yml* file:

```yaml
modules:
  campaignchain-linkedin-share-news-item:
      display_name: 'Share News'
      channels:
          - campaignchain/channel-linkedin/campaignchain-linkedin
      routes:
          new: campaignchain_activity_linkedin_share_news_item_new
          edit: campaignchain_activity_linkedin_share_news_item_edit
          edit_modal: campaignchain_activity_linkedin_share_news_item_edit_modal
          edit_api: campaignchain_activity_linkedin_share_news_item_edit_api
      hooks:
          campaignchain-due: true
```

Activities with a Single Operation

If an Activity has only one Operation, this should be made explicit by calling the Activity object's setEqualsOperation() method, as shown below:

```php
<?php
$wizard = $this->get('campaignchain.core.activity.wizard');
$activity = $wizard->getActivity();
$activity->setEqualsOperation(true);
```

Operation Form

When a user defines a new operation, CampaignChain renders a form with fields appropriate to that operation. This form must be included within the Operation module. The easiest way to create this form is by using Symfony's Form component and FormBuilder interface to define a Form object, as shown below:

```php
<?php
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolverInterface;
use Symfony\Component\DependencyInjection\ContainerInterface;

class ShareNewsItemOperationType extends AbstractType
{

    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('message', 'text', array(
                'property_path' => 'message',
                'label' => 'Message',
                'attr' => array(
                    'placeholder' => 'Add message...',
                    'max_length' => 200
                )
            ));

        $builder
            ->add('submitUrl', 'text', array(
                'property_path' => 'linkUrl',
                'label' => 'URL of page being shared',
                'attr' => array(
                    'placeholder' => 'Add URL...',
                    'max_length' => 255
                )
            ));

        // ... and so on //

    }
}
```

This Form object can then be used within controller action methods to create or edit a new operation, as shown below:

```php
<?php
$activityType = $this->get('campaignchain.core.form.type.activity');
$shareNewsItemOperation = new ShareNewsItemOperationType(
   $this->getDoctrine()->getManager(), $this->get('service_container')
);
$operationForms[] = array(
   'identifier' => self::OPERATION_IDENTIFIER,
   'form' => $shareNewsItemOperation,
   'label' => 'LinkedIn Message',
);
$activityType->setOperationForms($operationForms);

$form = $this->createForm($activityType, $activity);
$form->handleRequest($request);

if ($form->isValid()) {
      // process input
}
```

Operation Module and Service

The Activity object's *addOperation()* method should be passed an Operation object representing the operation to be added to the activity. CampaignChain's Operation service can be used to retrieve the correct Operation module, using the Operation bundle name and module identifier. The Operation service is available using the identifier 'campaignchain.core.operation'.

```
<?php
$operationService = $this->get('campaignchain.core.operation');
$operationModule = $operationService->getOperationModule(
  'campaignchain/operation-linkedin', 'campaignchain-linkedin-share-news-item'
);

$operation = new Operation();
$operation->setName($activity->getName());
$operation->setActivity($activity);
$activity->addOperation($operation);
```

In this example, the Operation service finds the Operation module named 'campaignchain-linkedin-share-news-item' in the bundle named 'campaignchain/operation-linkedin'.

Jobs

Every Operation module should include a Job, which actually executes the operation. This Job should implement the JobServiceInterface, which mandates an *execute()* method that is called when the job is executed. The Job is invoked by the CampaignChain scheduler when an Operation becomes due; it can also be invoked manually to execute an operation immediately.

Hooks

To process the hooks associated with an Activity, CampaignChain makes a Hook service available, via the service name 'campaignchain.core.hook'. Call this service's *processHooks()* method to process the hooks for an Activity, as shown below:

```
<?php
$hookService = $this->get('campaignchain.core.hook');
$activity = $hookService->processHooks(
  self::BUNDLE_NAME, self::MODULE_IDENTIFIER, $activity, $data
);
```

Operations and Locations

An Operation can create a new Location as its end result and/or it can include CTAs that point to Locations. For example, a LinkedIn news sharing Operation would create a new Location - a Linkedin status message that is directly accessible via a unique URL.

When you create a new Activity within CampaignChain, your Operation should also create a Location entry. At the time the Activity is created, the Location entry will necessarily be incomplete as the URL to the Location will not be known.

Once the Operation is executed, the Job that executes it must update the Location with the URL. It must also change the Location's status from 'STATUS_UNPUBLISHED' to 'STATUS_ACTIVE'.

It's important to define the *owns_location* parameter in the Operation module's *campaignchain.yml* file as shown below:

```
modules:
  campaignchain-linkedin-share-news-item:
      display_name: 'Share News'
      owns_location: true
      services:
          job: campaignchain.operation.linkedin.job.share_news_item
```

Call to Action (CTA) Tracking

This section describes the inner workings of CampaignChain's Call to Action Tracking. The provided information is useful for anyone developing Operation modules with CampaignChain or for those configuring third-party channels to work with CampaignChain.

What is a CTA?

> In web design, a CTA is a banner, button, or some type of graphic or text on a website meant to prompt a user to click it and continue down a conversion funnel. It is an essential part of inbound marketing as well as permission marketing in that it actively strives to convert a user into a lead and later into a customer.
>
> —Wikipedia[25]

In CampaignChain, a CTA is essentially a URL that appears in a HTML href link or form action. It appears within an Operation and each Operation can contain 0 to many CTAs. For example, a tweet could include various links that CampaignChain treats as CTAs, while a Google Ad would contain only 1 link as the CTA.

Tracking ID

CampaignChain leverages two types of IDs for its CTA tracking:

- *CTA Tracking ID*: Each CTA has a unique ID assigned by CampaignChain per URL that is included in an Operation. If you read about just the Tracking ID in the documentation, then it's referring to the CTA Tracking ID.

[25] http://en.wikipedia.org/wiki/Call_to_action_%28marketing%29

- *Channel Tracking ID*: For each Channel that has been connected with CampaignChain, a unique ID will be generated. This ID must be provided in the tracking code that is being included in a Channel. When the tracking code is activated, CampaignChain checks whether the provided Channel Tracking ID exists and whether the tracking code has been executed from a URL that actually resides within the Channel.

Tracking Process

To make CTA tracking work, a Channel that is connected with CampaignChain provides the relevant information to CampaignChain for tracking the CTA path that is part of a conversion funnel.

High-level View From a 30,000-feet perspective, this is how the tracking process works:

1. A link or form inside an Operation acts as the initial CTA at the beginning of a marketing funnel. The initial CTA contains a unique Tracking ID which allows CampaignChain to trace back the link to the respective Actions and Media (Campaign, Activity, Operation, Location, etc.). For example, a Twitter post that links to a landing page within a website.
2. All subsequent Locations inside a connected Channel ping CampaignChain and let it know through the Tracking ID that the respective Location was referred by a CTA.

The communication between CampaignChain and a Channel is achieved through JavaScript included in the Channel that posts the CTA information to CampaignChain and optionally also through REST-based communication between CampaignChain and the Channel. Depending on the depth of integration between CampaignChain and a Channel, there are 3 different types of tracking (described in subsequent sections).

In-depth Flow Description The single steps of the CTA tracking process are as follows:

1. The Operation's content gets parsed for links right before execution.

2. If the Operation contains 1 or more links, the following happens:

2.1. A unique Tracking ID gets assigned to each URL, no matter if an Operation contains the same URLs multiple times.

Note: An Operation could well contain the same URL multiple times, For example, a banner image on a landing page could point to the same URL in its key visual as well as a button that is part of the banner. When analyzing the effectiveness of the banner image, you want to know whether the key visual or the button caused more clicks. That's why each of them gets treated as a unique CTA with its own Tracking ID, although they have the same URL.

2.2. The Tracking ID gets appended to the URLs found inside the Operation.

2.2.1. If it is a full URL, then append the Tracking ID and replace the original URL with a shortened URL (CampaignChain uses Bit.ly[26] by default).

2.2.2. If the URL is already shortened, expand it, append the Tracking ID and replace the original shortened URL with a new shortened URL.

2.3. For each link, an entry is made in the CTA table with the Tracking ID and the related Operation as well as the original URL (full and short, if the latter was provided).

3. CampaignChain executes the Operation that now contains the new short URLs with the Tracking ID, e.g. it publishes a status update on Twitter that contains a link to a landing page.

[26] http://dev.bitly.com

4. When someone activates a CTA, e.g. clicks a link in a Tweet published by CampaignChain, the URL points to a Location. If that Location is part of a Channel that includes the tracking code and is connected with CampaignChain, then the following happens:

4.1. The tracking code checks whether the URL that pointed to the current page includes the Tracking ID. If yes, then it proceeds. If not, then it exits.

4.2. If the Tracking ID exists, the Tracking code sends this information to CampaignChain: Channel Tracking ID, CTA Tracking ID, URL of current Location, URL of target Location and additional information useful for monitoring.

4.3. CampaignChain checks whether the Channel Tracking ID is valid, i.e. if the Channel sending the tracking data is actually connected with CampaignChain.

4.3.1. If yes, then it performs some validity checks on the data, most notably whether the Tracking ID exists within CampaignChain, and finally saves the tracking data for monitoring purposes.

4.3.2. If no, then it will not save the data and instead notify the admin of an error (most likely, the Tracking Code has been included in a Channel that has not been connected with CampaignChain yet or this is a Denial of Service attack).

4.4. While CampaignChain processes the tracking data, the tracking code in the Channel appends the Tracking ID to the target URL (if another one does not exist yet, because the target URL is part of a new Operation) or saves it in a cookie. It then redirects the visitor to the target Location.

Note: Passing on the Tracking ID enables CampaignChain to do two things:

- Understand, whether e.g. the visitor browses a website away from a landing page before coming back to it and activating a CTA that leads to another Operation.
- Track the effectiveness of Operations across Channels.

Types of Tracking

To track CTAs, different types of tracking are used with CampaignChain to monitor the inbound marketing funnel.

CampaignChain-to-Channel (unidirectional)

- *Integration level:* Useful if CTA is under control, but not the Channel.
- *Example:* We can add a Tracking ID to a link that will be published on Twitter, but we cannot install something on Twitter to establish a connection between Twitter and CampaignChain to exchange information.
- *Tracking ID:* The Tracking ID must be included in the CTA. It is important, because it helps to distinguish between Campaigns and Activities if e.g. the same Landing Page is being used as a CTA target within the same Campaign various times or in different campaigns.
- *Pros:* Simple to implement by adding the Tracking ID to the URL of the CTA.
- *Cons:* Ideally, CampaignChain would be in control of the Operation (e.g. posting to Twitter from within CampaignChain). If not possible, then a user would have to manually append the Tracking ID.

Channel-to-CampaignChain (unidirectional)

- *Integration level:* The channel sends information about the Operation, Location and CTA to CampaignChain.
- *Example:* A JavaScript snippet included in Wordpress sends information to CampaignChain about a link's URL that was clicked inside a blog post, as well as the URL of the blog entry, etc.

- *Tracking ID:* At least the Tracking ID of the initial CTA should be available. Then CampaignChain is able to match the CTA's URL provided by the Channel with the Campaign and Activity it belongs to. Information about the source and target Location is also provided by the Channel for CampaignChain to easily map the related URLs to the Locations inside CampaignChain.
- *Pros:* This approach has the security advantage that the third-party application is in control of the communication towards CampaignChain.
- *Cons:* There must be a mechanism inside the Channel that ensures that at least the Tracking ID of the initial CTA is being carried on to the target Location.

CampaignChain-to-Channel (bidirectional)

- *Integration level:* CampaignChain and the Channel are tightly integrated when it comes to creating Operations and Locations, thus providing maximum communication between the two when it comes to CTA tracking.
- *Example:* A landing page has been created within Wordpress. With CampaignChain connected to Wordpress (e.g. through a REST API), CampaignChain grabs the content of the Wordpress page, parses it, stores the CTAs the page includes and makes the page public at the scheduled time. As in the unidirectional Channel-to-CampaignChain approach, a JavaScript snippet inside Wordpress sends information to CampaignChain once the CTA gets activated.
- *Tracking ID:* CampaignChain can pass all Tracking IDs for the CTAs in a Location to the Channel to be appended to each respective URL inside a Location for more granular tracking.
- *Pros:* The tighter coupling allows for more granular tracking, i.e. it is possible for CampaignChain to identify not just a Location, but also the Operation that includes a triggered CTA. Also, this approach has the performance advantage that the Channel as well as CampaignChain can handle the tracking more efficiently, because both are aware of all relevant information.
- *Cons:* Creating the tighter integration requires a higher investment in terms of time and money.

Development Environment

CampaignChain, Inc. provides a live development environment which provides readily available online channels to test CampaignChain modules against along with sample data that automatically sets up CampaignChain to be used with the development environment.

Name

The fictitious company featured in the development environment is called *Amariki*. This word was invented by Emilia, the younger daughter of Sandro Groganz, who initiated the CampaignChain project and is the founder of CampaignChain, Inc.

Yet, no one figured out whether it means anything in one of the languages on earth.

Online Channels

The following online channels have been set up for developers:

- http://wordpress.amariki.com (default Wordpress installation with test data)
- https://twitter.com/AmarikiTest1
- https://twitter.com/AmarikiTest2
- Facebook test user *Amariki Test One*

- Facebook test user *Amariki Test Two*
- Linkedin test user
- http://www.slideshare.net/amariki_test
- Bit.ly user *amariki*
- MailChimp user *amariki*
- GoToWebinar test user

To use these channels when developing modules for CampaignChain, you can load ready-made sample data into CampaignChain as described below.

Sample Data

By default, CampaignChain ships with the following sample data packages if installed with the `--stability=dev` option through Composer[27].

- amariki/data-test[28]
- amariki/data-demo[29]

Learn how to load sample data (page 40) in CampaignChain.

Note: Please contact partner@campaignchain.com to retrieve a `credentials.yml` file that works out-of-the-box with above data packages and the development environment.

Test Emails

We have test email accounts for above test online channels. For example, as an account email to receive notifications from Twitter or a recipient email to send MailChimp newsletters to.

Note: Please contact partner@campaignchain.com to apply for access to the email inbox where we collect all incoming test emails and to retrieve a list of available test emails.

Sample Data

It is possible to load sample data into CampaignChain, which eases the development process.

Warning:

- Don't use sample data in production environments, because they might be buggy or contain confidential data.
- All existing data in your CampaignChain might be wiped out. Make sure you backup your CampaignChain database if you are not sure whether the sample data you load is safe.

[27] https://getcomposer.org
[28] https://github.com/Amariki/data-test
[29] https://github.com/Amariki/data-demo

Packages

Sample data for CampaignChain are being made available as Composer packages[30] and can be installed through Composer with a command such as:

```
$ composer require amce/mydata
```

If you installed CampaignChain through composer with the `--stability=dev` option, these two default sample data packages are already available:

- amariki/data-test[31]
- amariki/data-demo[32]

The default sample data leverages the `live development environment`, i.e. the various test accounts and instances CampaignChain, Inc. set up at Twitter, Facebook, MailChimp, wordpress.amariki.com and other online channels.

Take a look at above packages if you'd like to create your own sample data. Some hints:

- The data is being stored as Fixtures[33].
- To have CampaignChain load your sample data, put a file `data.yml` into your bundle located at `Resources/data/campaignchain/`.
- It's good practice to give the super user the user name `admin` and password `test`.

Credentials

We recommend that you put all passwords, App keys, secrets, access tokens, token secrets, refresh tokens and other security sensitive information into a dedicated `credentials.yml` file. Reason being that you might want to share the sample data with others, but not the related credentials.

The aforementioned default sample data packages ship with a template file for credentials called `credentials.yml.tpl`. To create your own file, follow these instructions:

1. Rename `credentials.yml.tpl` to `credentials.yml`.
2. Register your CampaignChain instance as an app for the various Channels listed as `resourceOwner` in `credentials.yml`, such as Twitter and Facebook. If you've never done that, go to http://hybridauth.sourceforge.net/userguide.html and click on a provider (e.g. Twitter). On the respective page, there's a section called "Registering application". Proceed as described there.
3. Now connect a new Location in CampaignChain (e.g. a Twitter stream). When done, look up the respective tokens in the database table `campaignchain_security_authentication_client_oauth_token` and put them into `credentials.yml`.

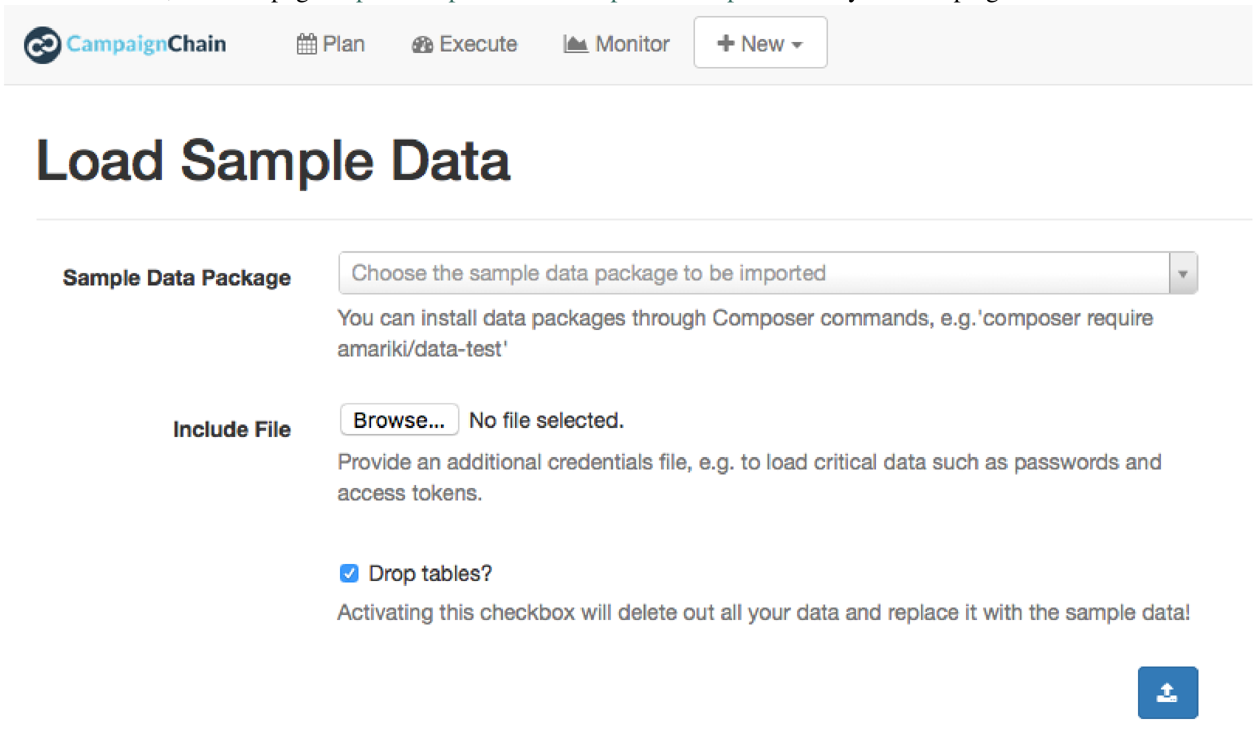
Note: If you would like to retrieve a `credentials.yml` file for the default sample data packages that works out-of-the-box, check out how to apply for access to the live development environment (page 39).

[30] https://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup
[31] https://github.com/Amariki/data-test
[32] https://github.com/Amariki/data-demo
[33] https://github.com/nelmio/alice

Usage

If `campaignchain.env` is set to `true` in `app/config/parameters.yml`, then you will be able to load sample data into CampaignChain in its Graphical User Interface or through a command.

To use the GUI, visit the page http://example.com/development/sample-data of your CampaignChain installation.



Load Sample Data

Sample Data Package: Choose the sample data package to be imported

You can install data packages through Composer commands, e.g.'composer require amariki/data-test'

Include File: Browse... No file selected.

Provide an additional credentials file, e.g. to load critical data such as passwords and access tokens.

☑ Drop tables?

Activating this checkbox will delete out all your data and replace it with the sample data!

1. Make sure you have a working `credentials.yml` file - see above.
2. Load the page http://example.com/development/sample-data of your CampaignChain installation.
3. There, pick the package of choice in the field "Data file" and select `credentials.yml` as the Include File. Activate the checkbox "Drop tables?" to start with a clean slate.
4. Click "Upload" and good luck :)
5. Log into CampaignChain with user `admin` and password `test` (unless otherwise specified by the package).

In addition to the user interface, you could also load sample data by using the command line. Issue the following command in the root of your CampaignChain installation to load the test data along with its credentials:

```
$ php app/console campaignchain:fixture vendor/amariki/data-test/Resources/data/campaignchain/data.y
```

Recovery

Should the sample data upload not work, you can try two things:

1. Fix the sample data and reload the browser window where you tried to upload the sample data.
2. If 1. does not work, install CampaignChain from scratch.

Contributing to CampaignChain

If you would like to add new functionality or fix a bug in CampaignChain, you are more than welcomed!

In general, we follow the best practices of open-source collaboration and developed some of our own - which is mostly due to the fact that CampaignChain is highly modular.

Find below some guidelines and we'd appreciate if you followed them. Should you have questions, please contact us[34].

Symfony

We closely follow the contribution guidelines of the Symfony project. Please make yourself acquainted with:

- Coding standards[35]
- Code conventions[36]

GitHub

We use GitHub[37] to develop the CampaignChain Community Edition. It allows us to collaborate with developers world-wide.

Please follow these guidelines:

Issues Always create an issue for the task you are working on at https://github.com/CampaignChain/campaignchain-ee/issues.

Commit Message The Git commit message should point to the issue you are working on. The syntax is:

```
CampaignChain/campaignchain#{issue-number} {issue-title}
```

In PHPStorm[38], you can use below definition for the commit message field when configuring the GitHub server to retrieve issues within PHPStorm:

```
CampaignChain/{project}#{number} {summary}
```

Pull Requests Never commit directly to the `master` branch of CampaignChain or its modules. Instead, please *create branches* (page 44). That way we can first test your contributions in the branch, ask you for changes, and then merge them with the `master` branch.

What's great about GitHub is that it let's you easily create pull requests[39] and then comment on it, even on single lines of the code[40].

So, once you are done with your changes and they are all committed to your branch, please create a pull request and we'll review it.

[34] http://www.campaignchain.com/contact/
[35] http://symfony.com/doc/current/contributing/code/standards.html
[36] http://symfony.com/doc/current/contributing/code/conventions.html
[37] http://www.github.com
[38] https://www.jetbrains.com/phpstorm/
[39] https://help.github.com/articles/creating-a-pull-request/
[40] https://help.github.com/articles/commenting-on-differences-between-files/

Note: The description of a pull request should include the link to the related issue at the very top. This ensures that the pull request is showing up when viewing the issue.

Branching

Whenever you create a new feature for CampaignChain or want to contribute a bug fix, please create a new branch of the respective CampaignChain module(s) or Symfony bundles you are working on.

Also, make sure you create a new branch of the CampaignChain/campaignchain[41] application as well.

How to Create a Branch? If you are not familiar with Git and branching, please read the introduction Basic Branching and Merging[42].

The name of the branch should include `campaignchain-` and the number of the related issue, such as `campaignchain-42`.

Inline Alias in composer.json You can define in the composer.json file of the application[43], which branched modules/bundles you'd like to use for development. Just list them in the `require-dev` section and define the branch as a Composer inline alias[44].

Note: You must prefix our branch name with‘‘dev-‘‘.

Here's an example how to do it for your custom module/bundle:

```
"require-dev": {
    "acme/my-bundle": "dev-my-branch as dev-master"
},
```

If you are working on existing CampaignChain modules, e.g. to fix a bug, create a branch and define it as an inline alias. All Composer version constraints[45] defined elsewhere will be overridden.

```
"require-dev": {
    "campaignchain/core": "dev-my-branch as dev-master"
},
```

Modify Sample Data Please branch the sample data repositories as well and then modify them *sample data* (page 45) accordingly by changing existing fixtures or by adding those which allow others to test new features.

Of course, the branch name of the sample data should be the same as the branch name of the modules/bundles you are modifying, so that we know which branch to use when loading the sample data for testing.

Branching Model We structure the flow of our development às follows:

- The `master` branch holds the latest stable code.
- New features are being developed in separate branches.

[41] https://github.com/CampaignChain/campaignchain
[42] https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging
[43] https://github.com/CampaignChain/campaignchain/blob/master/composer.json
[44] https://getcomposer.org/doc/articles/aliases.md#require-inline-alias
[45] https://getcomposer.org/doc/articles/versions.md

- Release branches hold the code of the tagged `master` branch.

Sample Data

When you are developing with CampaignChain, sample data will be available automatically. Learn how to import sample data to the system (page 40).

To allow testing of your changes by others, please keep all the publicly available *sample data packages* (page 41) in sync with your branch.

License

All code developed for the CampaignChain Community Edition is available under the Apache License[46]. We ask all contributors to assign new code to the same license.

Please add a `LICENSE` file with the content of the Apache License[47] into the root of new packages that you want to be included as part of the CampaignChain Community Edition.

The below license block has to be present at the top of every file.

In PHP before the namespace:

```
/*
 * Copyright 2016 CampaignChain, Inc. <info@campaignchain.com>
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

In `.yml` configuration files, at the very top:

```
# Copyright 2016 CampaignChain, Inc. <info@campaignchain.com>
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

In TWIG files at the very top:

[46] http://www.apache.org/licenses/LICENSE-2.0
[47] http://www.apache.org/licenses/LICENSE-2.0

```
{#
Copyright 2016 CampaignChain, Inc. <info@campaignchain.com>

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
#}
```

In CSS files at the very top:

```
/*
Copyright 2016 CampaignChain, Inc. <info@campaignchain.com>

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/
```

Credits

If you use third-party intellectual property, you must make sure that you are allowed to do so. Please add a `NOTICE` file in the root directory of a new module/bundle where you credit the copyright holders. See for example the NOTICE file of the core bundle[48].

Documentation

For any changes or bug fixes, please consider amending the CampaignChain Documentation[49]

Error Codes

This document provides information about error codes returned by CampaignChain.

All codes are defined in the ErrorCode class[50].

48 https://github.com/CampaignChain/core/blob/master/NOTICE
49 https://github.com/CampaignChain/campaignchain-docs
50 https://github.com/CampaignChain/core/blob/master/Exception/ErrorCode.php

Error Code	Description
1001	An error in the PHP code occurred.
1002	The Operation cannot be executed in the Location.
1003	The Activity cannot be executed in the Location.
1004	The Milestone cannot be executed in the Location.
1005	The connection to the REST API failed.

Hands-on tutorials that teach you how to use CampaignChain in concrete scenarios.

3.1.3 Developer Cookbook

Connect A New Online Channel

The tutorial will walk you through the process of adding support for a new online channel - in this case, the online professional network XING[51] - to CampaignChain by creating and programming the necessary modules and packaging them into a Symfony bundle.

The new XING modules will include functionality to connect to a user's XING activity stream and post updates to it.

Assumptions and Prerequisites

- You have a Symfony environment[52] meeting all the system requirements.
- You have a working `CampaignChain development installation`.
- You have a good understanding of the OAuth authentication process.
- You have a good understanding of the XING REST API. Learn more about the API[53].
- You have an application registered with XING and have obtained the necessary keys. Register your application[54].
- You have an account registered with XING, or you have access to the CampaignChain live development environment (page 39) (optional, for testing).

Overview

To connect a new online channel through CampaignChain, here are the steps you will follow:

1. *Generate Channel and Location modules* (page 48)
2. *Create Channel and Location controllers and views* (page 48)
3. *Add a channel icon* (page 57)

To enable activities and operations for the new channel, here are the steps you will follow:

1. *Generate Operation and Activity modules* (page 57)
2. *Understand the API exposed by the channel you're connecting* (page 58)
3. *Create entities and entity managers for your Activities and Operations* (page 59)
4. *Create input forms and connect them to your entities* (page 63)

[51] http://www.xing.com
[52] http://symfony.com/doc/current/book/installation.html
[53] https://dev.xing.com/docs
[54] https://dev.xing.com/applications/dashboard

5. *Create an API client and operation processor* (page 65)
6. *Define an Activity handler* (page 69)
7. *Collect metrics and create a report processor* (page 77)

Connect Channels and Locations

1. Generate Channel and Location modules The first step is to create Channel and Location modules for the XING channel. In this case, we'll assume the organization name is Acme, and use this organization name for the module namespaces.

The easiest way to generate these modules is with the CampaignChain module generator (included when you install with Composer and the option *–dev*). The commands below walk you through the process of generating a new Channel module. Note that you're safe using the defaults for all interactive prompts except for certain items shown below.

```
$ php app/console campaignchain:generate:module
Module type: Channel
Vendor name: Acme
Module name: Xing
Module name suffix:
Module display name: XING
Module description: This module connects to XING
Author name: Acme Inc.
Author's email address: info@acme.example.com
Package license: Apache-2.0
Package description: XING module for CampaignChain
Package website URL: http://acme.example.com
```

Symfony will now produce a new bundle containing stub code and files, in the location *your-project/src/Acme/Channel/XingBundle*. The name of the bundle will be *AcmeChannelXingBundle*.

Channels go hand-in-hand with Locations. Typically, a Location in a Channel refers to a specific user's activity stream or profile. In this tutorial, the Location will be the user's XING news stream.

Follow the steps above to generate a similar bundle for a Location in the Channel, remembering to specify the "Module type" as "Location" this time. The new bundle will be created at *your-project/src/Acme/Location/XingBundle* with the name *AcmeLocationXingBundle*.

Note: Most of the code you see in the next few sections is automatically generated for you by the CampaignChain module generator. You only need to copy and paste the parts that are specific to the channel you're trying to connect (in this example, XING).

2. Add Controllers and Views The Channel module takes care of creating a new Location and handling authentication between CampaignChain and the channel. The Channel module's auto-generated *campaignchain.yml* file specifies the routes and hooks used by its modules. Here's what the file looks like:

```
# src/Acme/Channel/XingBundle/campaignchain.yml

modules:
    acme-xing:
        display_name: Xing
        description: This module connects to Xing
        routes:
            new: acme_channel_xing_create
        hooks:
            default:
```

Notice the name of the Symfony route for creating a new channel. The corresponding URL and controller is defined in the module's auto-generated *routing.yml* file, as shown below:

```
# src/Acme/Channel/XingBundle/Resources/config/routing.yml

acme_channel_xing_create:
    pattern: /channel/xing/create
    defaults: { _controller: AcmeChannelXingBundle:Xing:create }
```

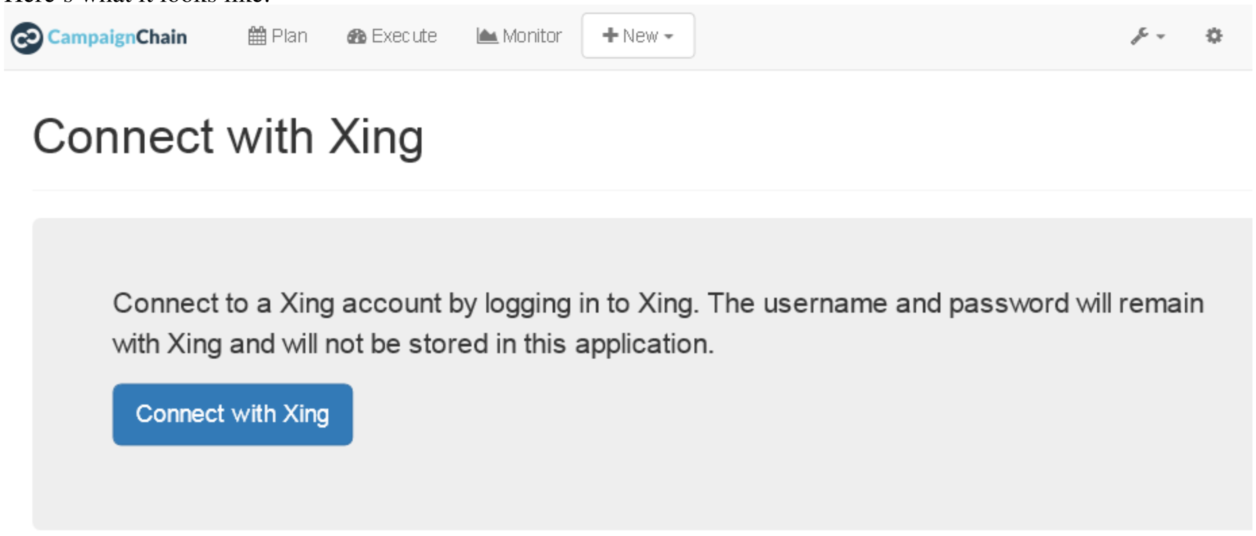
To begin, define this controller and action:

```
<?php
// src/Acme/Channel/XingBundle/Controller/XingController.php

namespace Acme\Channel\XingBundle\Controller;
use CampaignChain\CoreBundle\Entity\Location;
use Acme\Location\XingBundle\Entity\XingUser;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Session\Session;

class XingController extends Controller
{
    const RESOURCE_OWNER = 'Xing';

    const LOCATION_BUNDLE = 'acme/location-xing';

    const LOCATION_MODULE = 'acme-xing';

    private $applicationInfo = array(
        'key_labels' => array('key', 'Consumer key'),
        'secret_labels' => array('secret', 'Consumer secret'),
        'config_url' => 'https://dev.xing.com/applications/dashboard',
        'parameters' => array(),
        'wrapper' => array(
            'class'=>'Hybrid_Providers_XING',
            'path' => 'vendor/hybridauth/hybridauth/additional-providers/hybridauth-xing/Providers/X
        ),
    );

    public function createAction()
    {
        $oauthApp = $this->get('campaignchain.security.authentication.client.oauth.application');
        $application = $oauthApp->getApplication(self::RESOURCE_OWNER);

        if (!$application) {

            return $oauthApp->newApplicationTpl(self::RESOURCE_OWNER, $this->applicationInfo);
        }

        return $this->render(
            'AcmeChannelXingBundle:Create:index.html.twig',
            array(
                'page_title' => 'Connect with Xing',
                'app_id' => $application->getKey(),
            )
        );
    }
```

The *createAction()* method wraps CampaignChain's OAuth module and renders a splash page asking the user to connect to the XING account by providing credentials and granting permission to CampaignChain to access user data. This page is rendered with the view script shown below:

```
 # src/Acme/Channel/XingBundle/Resources/views/Create/index.html.twig

{% extends 'CampaignChainCoreBundle:Base:base.html.twig' %}

{% block body %}
    <div class="jumbotron">
        <p>Connect to a Xing account by logging in to Xing. The username and password will remain wi
        <p><a class="btn btn-primary btn-lg" role="button" onclick="popupwindow('{{ path('acme_chann
    </div>

 {% endblock %}
```

Here's what it looks like:



Connect with Xing

Connect to a Xing account by logging in to Xing. The username and password will remain with Xing and will not be stored in this application.

Clicking the "Connect now" button in the above view requests a 'login' route. Define this route as below:

```
# src/Acme/Channel/XingBundle/Resources/config/routing.yml

acme_channel_xing_login:
    pattern:  /channel/xing/create/login
    defaults: { _controller: AcmeChannelXingBundle:Xing:login }
```

Next, write a corresponding controller action to use the credentials entered by the user, attempt authentication and if successful, add the location to the CampaignChain database for later use.

To simplify this task, CampaignChain provides a Location service and a Channel Wizard which together encapsulate most of the functionality you will need. The code below illustrates the typical process:

```
<?php
// src/Acme/Channel/XingBundle/Controller/XingController.php

namespace Acme\Channel\XingBundle\Controller;
use CampaignChain\CoreBundle\Entity\Location;
use Acme\Location\XingBundle\Entity\XingUser;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Session\Session;

class XingController extends Controller
{

  public function loginAction(Request $request)
  {
      $oauth = $this->get('campaignchain.security.authentication.client.oauth.authentication');
      $status = $oauth->authenticate(self::RESOURCE_OWNER, $this->applicationInfo);
      $profile = $oauth->getProfile();

      if(!$status){
          $this->addFlash(
              'warning',
              'A location has already been connected for this Xing account.'
          );

          return $this->render(
              'AcmeChannelXingBundle:Create:login.html.twig',
              array(
                  'redirect' => $this->generateUrl('campaignchain_core_channel')
              )
          );
      }

      $repository = $this->getDoctrine()->getManager();
      $repository->getConnection()->beginTransaction();

      $wizard = $this->get('campaignchain.core.channel.wizard');
      $wizard->setName($profile->displayName);

      // Get the location module.
      $locationService = $this->get('campaignchain.core.location');
      $locationModule = $locationService->getLocationModule(self::LOCATION_BUNDLE, self::LOCATION_MOI

      $location = new Location();
      $location->setIdentifier($profile->identifier);
      $location->setName($profile->displayName);
      $location->setImage($profile->photoURL);
      $location->setLocationModule($locationModule);

      $wizard->addLocation($location->getIdentifier(), $location);

      try {
          $channel = $wizard->persist();
          $wizard->end();

          $oauth->setLocation($channel->getLocations()->first());

          $user = new XingUser();
          $user->setLocation($channel->getLocations()->first());
          $user->setIdentifier($profile->identifier);
          $user->setDisplayName($profile->displayName);
          $user->setFirstName($profile->firstName);
          $user->setLastName($profile->lastName);
          $user->setProfileImageUrl($profile->photoURL);

          if (isset($profile->emailVerified)) {
            $user->setEmail($profile->emailVerified);
          } else {
            $user->setEmail($profile->email);
          }

          $repository->persist($user);
          $repository->flush();

          $repository->getConnection()->commit();

      } catch (\Exception $e) {
          $repository->getConnection()->rollback();
          throw $e;
      }

      $this->addFlash(
          'success',
          'The Xing location <a href="#">'.$profile->displayName.'</a> was connected successfully.'
      );

      return $this->render(
          'AcmeChannelXingBundle:Create:login.html.twig',
          array(
              'redirect' => $this->generateUrl('campaignchain_core_channel')
          )
      );

  }
}
```

The first few lines of the *loginAction()* action method use CampaignChain's OAuth module, which in turn uses HybridAuth, to authenticate against the remote service. If authentication is successful, the OAuth object's *getProfile()* method returns the profile of the authenticated user. This location now needs to be added to CampaignChain's database.

To accomplish this, the action method first creates a new Channel Wizard object, which is a convenience object that makes it easy to connect the new location to the channel and save it to CampaignChain's database. The Channel Wizard is invoked as a Symfony service. The Channel Wizard is also assigned a name using its *setName()* method; this could be a fixed name, or based on input entered by the user (although you'd need to provide a form field in the view to accept this input).

The whole point of logging in is to authorize CampaignChain to connect a Location. So, the action method then calls CampaignChain's Location service to identify the Location module. The Location module's name and unique module identifier play a critical role in helping the Channel Wizard correctly identify and store the Location so that CampaignChain can generate routes for the Location.

The method initializes a new Location object using the information from the returned user profile, and attaches this Location object to the Channel using the Channel Wizard's *addLocation()* method. The information about the new Location is saved to the database using the Channel Wizard's *persist()* method.

Since every Location is typically associated with a user profile or activity stream, it makes sense to also store additional information about the user in the CampaignChain database. The typical properties you'd want to store are the user identifier, first name, last name, email address, profile URL and profile image URL, plus any properties specific to the channel you're connecting. In this example, this information is encapsulated in a XingUser entity, with properties and getter/setter methods for the user identifier, first name, last name, email address, XING profile URL and XING profile image URL.

The XingUser entity logically belongs in the Location module and can be seen below. Entity records are stored in the *acme_location_xing_user* table in the CampaignChain database and each record has a 1:1 relationship with CampaignChain's core Location entity.

```php
<?php
// src/Acme/Location/XingBundle/Entity/XingUser.php

 namespace Acme\Location\XingBundle\Entity;

 use Doctrine\ORM\Mapping as ORM;
 use CampaignChain\CoreBundle\Util\ParserUtil;

 /**
  * @ORM\Entity
  * @ORM\Table(name="acme_location_xing_user")
  */
 class XingUser
 {
     /**
      * @ORM\Column(type="integer")
      * @ORM\Id
      * @ORM\GeneratedValue(strategy="AUTO")
      */
     protected $id;

     /**
      * @ORM\OneToOne(targetEntity="CampaignChain\CoreBundle\Entity\Location", cascade={"persist"})
      */
     protected $location;

     /**
      * @ORM\Column(type="string", length=255, unique=true)
      */
     protected $identifier;

     /**
      * @ORM\Column(type="string", length=255)
      */
     protected $displayName;

     /**
      * @ORM\Column(type="string", length=255, nullable=true)
      */
     protected $firstName;

     /**
      * @ORM\Column(type="string", length=255, nullable=true)
      */
     protected $lastName;

     /**
      * @ORM\Column(type="string", length=255, nullable=true)
      */
     protected $email;

     /**
      * @ORM\Column(type="string", length=255, nullable=true)
      */
     protected $profileUrl;
```

```
    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    protected $profileImageUrl;

    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set location
     *
     * @param \CampaignChain\CoreBundle\Entity\Location $location
     * @return User
     */
    public function setLocation(\CampaignChain\CoreBundle\Entity\Location $location = null)
    {
        $this->location = $location;
        return $this;
    }

    /**
     * Get location
     *
     * @return \CampaignChain\CoreBundle\Entity\Location
     */
    public function getLocation()
    {
        return $this->location;
    }

    /**
     * Set identifier
     *
     * @param string $identifier
     * @return LocationBase
     */
    public function setIdentifier($identifier)
    {
        $this->identifier = $identifier;
        return $this;
    }

    /**
     * Get identifier
     *
     * @return string
     */
    public function getIdentifier()
    {
        return $this->identifier;
    }
    /**
     * Set displayName
     *
     * @param string $displayName
     * @return User
     */
    public function setDisplayName($displayName)
    {
        $this->displayName = $displayName;
        return $this;
    }

    /**
     * Get displayName
     *
     * @return string
     */
    public function getDisplayName()
    {
        return $this->displayName;
    }

    /**
     * Set firstName
     *
     * @param string $firstName
     * @return User
     */
    public function setFirstName($firstName)
    {
        $this->firstName = $firstName;
        return $this;
    }

    /**
     * Get firstName
     *
     * @return string
     */
    public function getFirstName()
    {
        return $this->firstName;
    }

    /**
     * Set lastName
     *
     * @param string $lastName
     * @return User
     */
    public function setLastName($lastName)
    {
        $this->lastName = $lastName;
        return $this;
    }

    /**
     * Get lastName
     *
     * @return string
     */
    public function getLastName()
    {
        return $this->lastName;
    }

    /**
     * Set email
     *
     * @param string $email
     * @return User
     */
    public function setEmail($email)
    {
        $this->email = $email;
        return $this;
    }

    /**
     * Get email
     *
     * @return string
     */
    public function getEmail()
    {
        return $this->email;
    }

    /**
     * Set profileUrl
     *
     * @param string $profileUrl
     * @return User
     */
    public function setProfileUrl($profileUrl)
    {
        $this->profileUrl = ParserUtil::sanitizeUrl($profileUrl);
        return $this;
    }

    /**
     * Get profileUrl
     *
     * @return string
     */
    public function getProfileUrl()
    {
        return $this->profileUrl;
    }

    /**
     * Set profileImageUrl
     *
     * @param string $profileImageURL
     * @return User
     */
    public function setProfileImageUrl($profileImageUrl)
    {
        $this->profileImageUrl = $profileImageUrl;
        return $this;
    }

    /**
     * Get profileImageUrl
     *
     * @return string
     */
    public function getProfileImageUrl()
    {
        return $this->profileImageUrl;
    }

}
```

3. Add a Channel Icon Every Channel module should include a channel icon image, for easy identification within the CampaignChain GUI. In most cases, the channel you're trying to connect to will provide a logo image, so all that's really needed is to resize it to 3 different sizes (16x16, 24x24 and 32x32 pixels) and save it in PNG format.

Note: Remember to read the channel's terms of use for its images, ensure that your usage of the image is compliant and provide an image credit, link and/or attribution as needed.

For the XING Channel module created in this tutorial, the 16x16 channel icon image should be saved to *your-project/src/Acme/Location/XingBundle/Resources/public/images/icons/16x16/xing.png*. Other image sizes should be saved similarly. The name of the image ('xing') should match the descriptive string used in the module name ('acme-xing')

At this point, your Channel and Location modules are complete.

Define Activities and Operations

With the Channel and Location defined, the next step is to define the Activities and Operations possible. To keep things simple, we'll assume that only a single Activity is required: sharing news on the user's XING news stream. This will be accomplished using XING's REST API, which makes it possible to add posts to a user's news stream.

1. Generate Operation and Activity modules The first step here is again to create modules for the Activity and Operation. Remember that every Activity must have at least one Operation. In this case, since only one Operation is needed, the Activity is equal to the Operation. This is important to know in advance, as it's critical to proper functioning and it's also part of the information requested by the module generator.

Since Activity and Operation modules are linked, it's generally recommended that you create the Operation module first and the Activity module later. Use the CampaignChain module generator as before, and be aware that you will be asked for additional information for Activity and Operation modules, as shown below:

```
$ php app/console campaignchain:generate:module
Module type: Operation
Vendor name: Acme
Module name: Xing
Module name suffix: message
Module display name: Post message on XING
Module description: This module posts a message on XING
Does the operation own its location?: true
Metrics for the operation: Comments, Likes
Author name: Acme Inc.
Author's email address: info@acme.example.com
Package license: Apache-2.0
Package description: XING module for CampaignChain
Package website URL: http://acme.example.com
```

The new bundle will be created at *your-project/src/Acme/Operation/XingBundle* with the name *AcmeOperationXing-Bundle*.

Next, create the corresponding Activity module, as below. Note that the module name suffix is left empty on purpose for this tutorial.

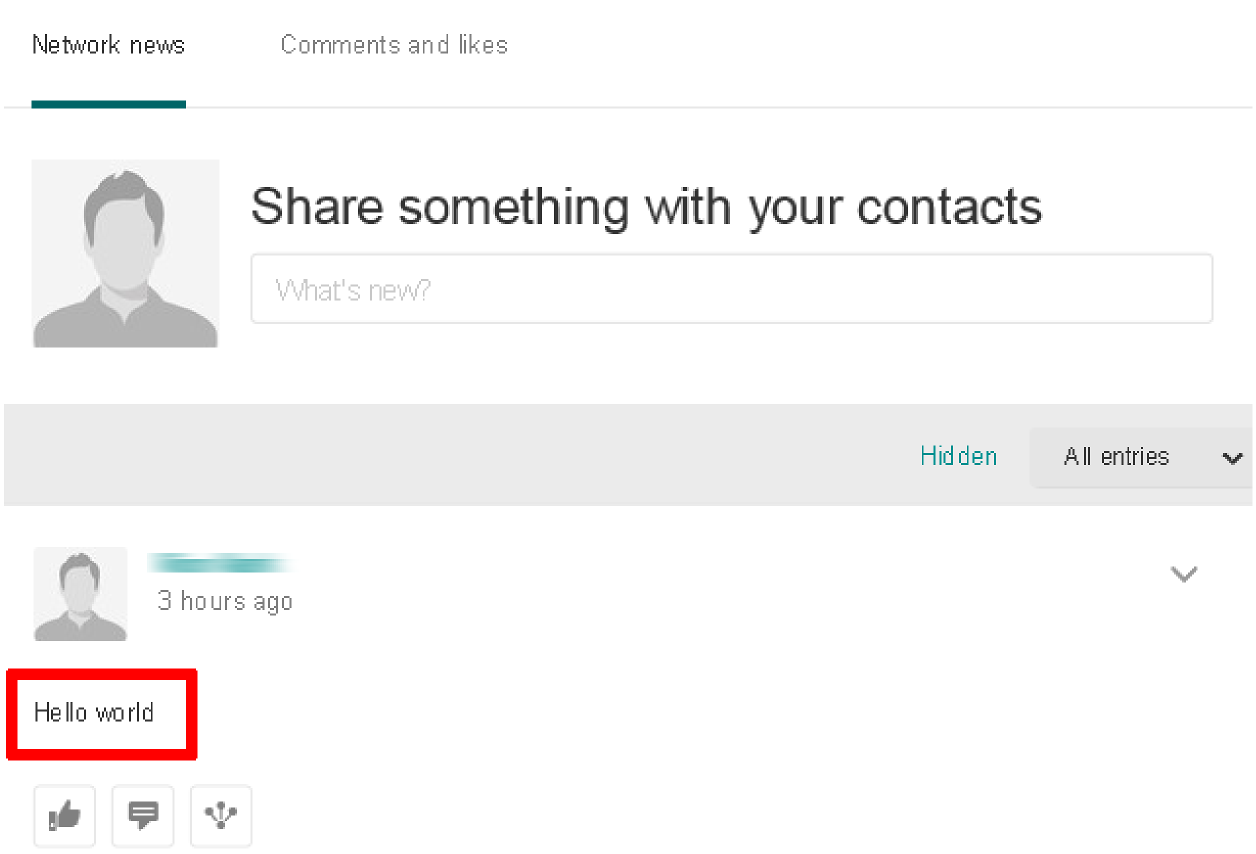
```
$ php app/console campaignchain:generate:module
Module type: Activity
Vendor name: Acme
Module name: Xing
Module name suffix:
Module display name: Post message on XING
Module description: This module posts a message on XING
Does the Activity module execute in a single Channel?: yes
Channels for the activity: acme/channel-xing/acme-xing
Hooks for the activity: campaignchain-due, campaignchain-assignee
Location parameter name: campaignchain.location.acme.xing
Does the Activity equal an Operation?: true
Operation parameter names for the Activity: campaignchain.operation.acme.xing.message
Author name: Acme Inc.
Author's email address: info@acme.example.com
Package license: Apache-2.0
Package description: XING module for CampaignChain
Package website URL: http://acme.example.com
```

Note that the module and module name suffix used in the Operation module should be correctly referenced in the Activity module's Operation parameter name.

The new bundle will be created at *your-project/src/AcmeActivity/XingBundle* with the name *AcmeActivityXingBundle*.

Note: Most of the code you see in the next few sections is automatically generated for you by the CampaignChain module generator. You only need to copy and paste the parts that are specific to the channel you're trying to connect (in this example, XING).

2. Understand the XING API Once the modules are created, let's look more closely at the message posting operation to be implemented. Review the image below, which displays a typical item in a XING user's news stream.



As you can see, a XING news item is a simple text message. The most efficient way to post such a message to a XING user's stream programmatically is with the XING API. Using this API involves sending an authenticated POST request to the API endpoint https://api.xing.com/v1/users/:id/status_message, and transmitting the message in the body of the POST request. The XING documentation has an example and more information[55].

Fortunately, you don't need to worry about the nitty-gritty of creating, transmitting and handling POST requests and responses. CampaignChain internally uses Guzzle[56] and so, you can simply invoke Guzzle's *post()* method to transmit a POST request and process a POST response. Here's an example of how it would work:

```php
<?php

  $client = new GuzzleHttp\Client(['base_uri' => 'https://api.xing.com/v1/']);
  $request = $client->post(
      'users/123456/status_message',
      array(),
      array('id' => '123456', 'message' => 'Hello world')
  );
  $response = $request->send();
```

Obviously, you still need an input form in CampaignChain for the user to enter the message. And, as one of CampaignChain's core capabilities is the ability to schedule activities and operations ahead of time, you'll need to store newly-created messages in the CampaignChain database, and implement a job to transmit them to XING at the appropriate time. The following sections will guide you through these tasks.

[55] https://dev.xing.com/docs/post/users/:id/status_message
[56] http://docs.guzzlephp.org/en/latest/quickstart.html

3. Create An Entity and Entity Manager The first step is to create a XingMessage entity representing a XING message, and a service manager to work with that entity. A stub entity should have been auto-generated for you already at *your-project/src/Acme/Operation/XingBundle/Entity/XingMessage.php*. Simply update it to reflect the information you wish to save for a message, as below:

```php
<?php

// src/Acme/Operation/XingBundle/Entity/XingMessage.php

namespace Acme\Operation\XingBundle\Entity;

use CampaignChain\CoreBundle\Entity\Meta;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ORM\Table(name="acme_operation_xing_message")
 */
class XingMessage extends Meta
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    /**
     * @ORM\OneToOne(targetEntity="CampaignChain\CoreBundle\Entity\Operation", cascade={"persist"})
     */
    protected $operation;

    /**
     * @ORM\Column(type="text", length=420)
     */
    protected $message;

    /**
     * @ORM\Column(type="text", length=255, nullable=true)
     */
    protected $url;

    /**
     * @ORM\Column(type="text", length=255, nullable=true)
     */
    protected $messageId;

    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set operation
     *
     * @param \CampaignChain\CoreBundle\Entity\Operation $operation
     * @return Status
     */
    public function setOperation(\CampaignChain\CoreBundle\Entity\Operation $operation = null)
    {
        $this->operation = $operation;

        return $this;
    }

    /**
     * Get operation
     *
     * @return \CampaignChain\CoreBundle\Entity\Operation
     */
    public function getOperation()
    {
        return $this->operation;
    }

    /**
     * Set message
     *
     * @param string $message
     * @return XingMessage
     */
    public function setMessage($message)
    {
        $this->message = $message;

        return $this;
    }

    /**
     * Get message
     *
     * @return string
     */
    public function getMessage()
    {
        return $this->message;
    }

    /**
     * Set URL
     *
     * @param string $url
     * @return XingMessage
     */
    public function setUrl($url)
    {
        $this->url = $url;

        return $this;
    }

    /**
```

```
     * Get URL
     *
     * @return string
     */
    public function getUrl()
    {
        return $this->url;
    }

    /**
     * Set message id
     *
     * @param string $messageId
     * @return XingMessage
     */
    public function setMessageId($messageId)
    {
        $this->messageId = $messageId;

        return $this;
    }

    /**
     * Get message id
     *
     * @return string
     */
    public function getMessageId()
    {
        return $this->messageId;
    }
}
```

As you can see, the entity includes properties corresponding to those expected by the XING API (in this case, the message text and the unique message identifier on XING), as well as some properties needed by CampaignChain.

You will also need an entity service manager, which will retrieve an instance of the entity by its identifier. Here's the code, which should be saved to *your-project/src/Acme/Operation/XingBundle/EntityService/XingMessage.php*.

```
<?php

// src/Acme/Operation/XingBundle/EntityService/XingMessage.php

namespace Acme\Operation\XingBundle\EntityService;

use Doctrine\ORM\EntityManager;
use CampaignChain\CoreBundle\EntityService\OperationServiceInterface;
use CampaignChain\CoreBundle\Entity\Operation;

class XingMessage implements OperationServiceInterface
{
    /**
     * @var EntityManager
     */
    protected $em;

    public function __construct(EntityManager $em)
    {
        $this->em = $em;
    }

    public function getMessageByOperation($id)
    {
        $message = $this->em
          ->getRepository('AcmeOperationXingBundle:XingMessage')
          ->findOneByOperation($id);

        if (!$message) {
            throw new \Exception(
                'No message found by operation id '.$id
            );
        }

        return $message;
    }

    public function cloneOperation(Operation $oldOperation, Operation $newOperation)
    {
        $message = $this->getMessageByOperation($oldOperation);
        $clonedMessage = clone $message;
        $clonedMessage->setOperation($newOperation);

        $this->em->persist($clonedMessage);
        $this->em->flush();
    }

    public function removeOperation($id)
    {
        try {
            $operation = $this->getMessageByOperation($id);

            $this->em->remove($operation);
            $this->em->flush();
        } catch (\Exception $e) {
        }
    }
}
```

The *getMessageByOperation()* method takes care of retrieving a specific message using its unique identifier in the database.

This is also a good point to update the Operation module's list of exposed services to include the new entity service manager. To do this, update the file at *your-project/src/Acme/Operation/XingBundle/Resources/config/services.yml* and add the service identifier to it, as shown below. Remember to leave the existing auto-generated service identifiers as is.

```
# src/Acme/Operation/XingBundle/Resources/config/services.yml

parameters:

services:
    campaignchain.operation.acme.xing.message:
        class: Acme\Operation\XingBundle\EntityService\XingMessage
        arguments: [ @doctrine.orm.entity_manager ]
```

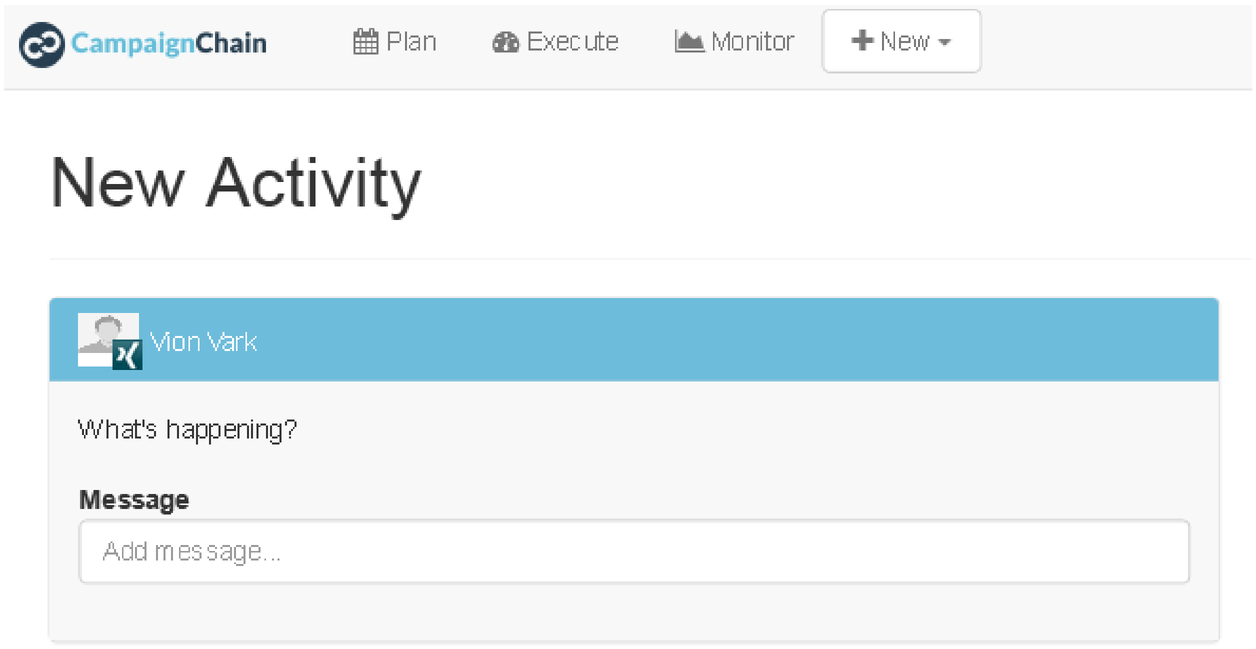
4. Create an Input Form for Entity Data With the entity created, the next step is to provide an input form that will be rendered by the CampaignChain user interface. This form will be used when setting up a new XING message, and the fields in the form must therefore correspond with the properties of the XingMessage entity.

The easiest way to create the form is by using Symfony's Form component and FormBuilder interface. The following code, which should be saved to the auto-generated file at *your-project/src/Acme/Operation/XingBundle/Form/Type/XingMessageOperationType.php*, illustrates how to do this.

```
<?php

// src/Acme/Operation/XingBundle/Form/Type/XingMessageOperationType.php

namespace Acme\Operation\XingBundle\Form\Type;

use CampaignChain\CoreBundle\Form\Type\OperationType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\TextType;

class XingMessageOperationType extends OperationType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('message', TextType::class, [
                'property_path' => 'message',
                'label' => 'Message',
                'attr' => [
                    'placeholder' => 'Add message...',
                    'max_length' => 420
                ]
            ]);
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $defaults = [
            'data_class' => 'Acme\Operation\XingBundle\Entity\XingMessage',
        ];

        if ($this->content) {
            $defaults['data'] = $this->content;
        }
        $resolver->setDefaults($defaults);
    }

    public function getBlockPrefix()
    {
        return 'acme_operation_xing_message';
    }
}
```

The main work here is done by the *buildForm()* method, which takes care of creating the necessary form fields, and the *setDefaultOptions()* method, which links the data entered into the form with the XingMessage entity created earlier.

Here's an example of what the form looks like when rendered:



New Activity

What's happening?

Message

Add message...

5. Create an API Client and Operation Processor In the previous steps, you enabled the user to enter details of a new XING message into a form and have that data saved to the CampaignChain database. The next step is to build an operation processor, which can check periodically for scheduled messages, authenticate against the XING API as needed, and post those messages to the user's stream.

To accomplish this task, it is necessary to create an HTTP client object which will handle authentication between CampaignChain on the one hand, and the XING API on the other hand. Since CampaignChain already comes with an OAuth client, you can use this client's built-in functionality to take care of most of the heavy lifting.

To do this, go back to your Channel module and add the following XingClient object to it, at the location *your-project/src/Acme/Channel/XingBundle/REST/XingClient.php*.

```php
<?php

// src/Acme/Channel/XingBundle/REST/XingClient.php

namespace Acme\Channel\XingBundle\REST;

use Symfony\Component\HttpFoundation\Session\Session;
use Guzzle\Http\Client;
use Guzzle\Plugin\Oauth\OauthPlugin;
use Guzzle\Http\Exception\ClientErrorResponseException;
use Guzzle\Http\Exception\ServerErrorResponseException;
use Guzzle\Http\Exception\BadResponseException;
use CampaignChain\Security\Authentication\Client\OAuthBundle\EntityService\ApplicationService;
use CampaignChain\Security\Authentication\Client\OAuthBundle\EntityService\TokenService;

class XingClient
{
    const RESOURCE_OWNER = 'Xing';
    const BASE_URL    = 'https://api.xing.com/v1';

    /**
```

```
     * @var ApplicationService
     */
    protected $applicationService;

    /**
     * @var TokenService
     */
    protected $tokenService;

    /**
      * XingClient constructor.
      * @param ApplicationService $applicationService
      * @param TokenService $tokenService
      */
    public function __construct(ApplicationService $applicationService, TokenService $tokenService)
    {
        $this->applicationService = $applicationService;
        $this->tokenService = $tokenService;
    }

    public function connectByActivity($activity)
    {
        return $this->connectByLocation($activity->getLocation());
    }

    public function connectByLocation($location)
    {
        $application = $this->applicationService->getApplication(self::RESOURCE_OWNER);
        $token = $this->tokenService->getToken($location);

        return $this->connect($application->getKey(), $application->getSecret(), $token->getAccessTok
    }

    public function connect($appKey, $appSecret, $accessToken, $tokenSecret)
    {
        try {
            $client = new Client(self::BASE_URL.'/');
            $oauth  = new OauthPlugin(array(
                'consumer_key'    => $appKey,
                'consumer_secret' => $appSecret,
                'token'           => $accessToken,
                'token_secret'    => $tokenSecret,
            ));

            return $client->addSubscriber($oauth);
        } catch (ClientErrorResponseException $e) {
            $request = $e->getRequest();
            $response = $e->getResponse();
            print_r($request);
            print_r($response);
        } catch (ServerErrorResponseException $e) {
            $request = $e->getRequest();
            $response = $e->getResponse();
            print_r($response);
        } catch (BadResponseException $e) {
            $request = $e->getRequest();
            $response = $e->getResponse();
            print_r($response);
        } catch(Exception $e){
          print_r($e->getMessage());
        }
    }
}
```

The two important values set in this client are the constants at the top: the RESOURCE_OWNER constant specifies the owning channel, which is then used to retrieve the keys and secrets needed for an authenticated API connection, and the BASE_URL constant specifies the base URL for all API requests.

You will also need to update the Channel module's list of exposed services to include the new client. To do this, update the file at *your-project/src/Acme/Channel/XingBundle/Resources/config/services.yml* with the following information.

```
# src/Acme/Channel/XingBundle/Resources/config/services.yml

parameters:

services:
    acme.channel.xing.rest.client:
        class: Acme\Channel\XingBundle\REST\XingClient
        arguments: ["@campaignchain.security.authentication.client.oauth.application", "@campaignchai
```

You'll notice that this client object merely takes care of connecting and authenticating against the XING API. It doesn't actually take care of creating and sending a POST request to the XING API. That task is handled by a separate Job object, which should have been auto-generated within your Operation module at *your-project/src/Acme/Operation/XingBundle/Job/XingMessage.php*.

```
<?php

// src/Acme/Operation/XingBundle/Job/XingMessage

namespace Acme\Operation\XingBundle\Job;

use CampaignChain\CoreBundle\Entity\Action;
use Doctrine\ORM\EntityManager;
use CampaignChain\CoreBundle\Entity\Medium;
use CampaignChain\CoreBundle\Job\JobActionInterface;
use Symfony\Component\HttpFoundation\Response;
use CampaignChain\CoreBundle\EntityService\CTAService;
use CampaignChain\Security\Authentication\Client\OAuthBundle\EntityService\TokenService;

class XingMessage implements JobActionInterface
{
    /**
     * @var EntityManager
     */
    protected $em;

    /**
     * @var TokenService
     */
    protected $tokenService;

    /**
     * @var XingClient
     */
    protected $xingClient;

    public function __construct(EntityManager $em, TokenService $tokenService, XingClient $xingClient
    {
        $this->em = $em;
        $this->tokenService = $tokenService;
        $this->xingClient = $xingClient;
    }

    public function execute($operationId)
    {
        $message = $this->em
            ->getRepository('AcmeOperationXingBundle:XingMessage')
            ->findOneByOperation($operationId);

        if (!$message) {
            throw new \Exception('No message found for an operation with ID: '.$operationId);
        }

        $ctaService = $this->container->get('campaignchain.core.cta');
        $message->setMessage(
            $ctaService->processCTAs($message->getMessage(), $message->getOperation(), CTAService::FO
        );

        $activity = $message->getOperation()->getActivity();
        $identifier = $activity->getLocation()->getIdentifier();
        $token = $this->tokenService->getToken($activity->getLocation());

        $connection = $this->xingClient->connectByActivity($message->getOperation()->getActivity());

        $request = $connection->post('users/' . $identifier . '/status_message', array(), array('id'
        $response = $request->send();
        $messageEndpoint = $response->getHeader('location');
        $messageId = basename($messageEndpoint);
        $messageUrl = 'https://www.xing.com/feedy/stories/' . strtok($messageId, '_');
        $message->setUrl($messageUrl);
        $message->setMessageId($messageId);

        $message->getOperation()->setStatus(Action::STATUS_CLOSED);
        $location = $message->getOperation()->getLocations()[0];
        $location->setIdentifier($messageId);
        $location->setUrl($messageUrl);
        $location->setName($message->getOperation()->getName());
        $location->setStatus(Medium::STATUS_ACTIVE);

        $this->em->flush();

        $this->message = 'The message "'.$message->getMessage().'" with the ID "'.$messageId.'" has b

        return self::STATUS_OK;
    }
}
```

A Job object is always part of an Operation module and it is called as necessary to perform the corresponding operation. It should implement the JobActionInterface, which mandates an *execute()* method which is called when the job is executed.

If you look into the *execute()* method above, you'll see that it begins by retrieving the required message from the CampaignChain database (using the message identifier). It then uses CampaignChain's CTA service and *processCTAs()* method to inspect all URLs in the message body and replace those URLs that match a Location with short URLs for tracking.

The next step is to invoke the XingClient created earlier from the service manager and uses the client to authenticate against the XING API. The method uses the client's inherited *post()* method to transmit a POST request to the API endpoint https://api.xing.com/v1/users/ID/status_message containing the user's identifier on XING and the message content. If successful, the response will contain a Location header containing the URL to the posted message. It's now easy enough to extract the message identifier from this and create a new Location record pointing to it in the CampaignChain database. This Location can later be used in CampaignChain's Call-to-Action tracking.

At the same time, a new XingMessage record is also created to store the message URL and unique message identifier on XING. This message identifier is particularly important, as it will later be used to collect statistics about the number of likes and comments received on the message.

The final steps are to update the status of the operation in the CampaignChain database and present a success message to the user.

You also need to update the Activity module's list of exposed services to include the new job. To do this, update the file at *your-project/src/Acme/Activity/XingBundle/Resources/config/services.yml* so it now looks like the following.

```
# src/Acme/Activity/XingBundle/Resources/config/services.yml

parameters:
# Parameters for the CampaignChain Activity modules in this Symfony bundle
    campaignchain.activity.acme.xing:
        bundle_name: "acme/activity-xing"
        module_identifier: "acme-xing"
        location: %campaignchain.location.acme.xing%
        equals_operation: true
        operations:
            - %campaignchain.operation.acme.xing.message%
        handler: "campaignchain.activity.controller.handler.acme.xing"

services:
    # The Symfony service evoking the default controller of the CampaignChain
    # core package
    campaignchain.activity.controller.acme.xing:
        class: CampaignChain\CoreBundle\Controller\Module\ActivityModuleController
        calls:
            - [setContainer, ["@service_container"]]
            - [setParameters, ["%campaignchain.activity.acme.xing%"]]
    # The CampaignChain controller handler where the Activity's GUI and data
    # is being processed.
    campaignchain.activity.controller.handler.acme.xing:
      class: Acme\Activity\XingBundle\Controller\XingHandler
      arguments:
          - "@doctrine.orm.entity_manager"
          - "@session"
          - "@templating"
          - "@campaignchain.operation.acme.xing.message"
          - "@campaignchain.job.operation.acme.xing.message"
```

6. Define an Activity Handler The Activity module specifies the routes for creating and editing Operations. This implies that the Activity should define four routes:

- A route to create a new Activity ('new')
- A route to edit an existing Activity ('edit')
- A route to display the details of an existing Activity ('read')
- A route to edit an existing Activity in the campaign timeline's pop-up/lightbox view ('edit_modal')

- A route for the submit action of the pop-up/lightbox view in the campaign timeline ('edit_api')

These routes are automatically generated for you in the Activity module's *campaignchain.yml* file, as shown below:

```
# src/Acme/Activity/XingBundle/campaignchain.yml

modules:
    campaignchain-xing:
        display_name: Post message on Xing
        description: This module posts a message on Xing
        channels:
            - acme/channel-xing/acme-xing
        routes:
            new: acme_activity_xing_new
            edit: acme_activity_xing_edit
            edit_modal: acme_activity_xing_edit_modal
            edit_api: acme_activity_xing_edit_api
            read: acme_activity_xing_read
        hooks:
            default:
                campaignchain-due: true
                campaignchain-assignee: true
```

The corresponding controller and action names are also auto-generated in the Activity module's *routing.yml* file:

```
# src/Acme/Activity/XingBundle/campaignchain.yml

acme_activity_xing_new:
    pattern: /activity/xing/new
    defaults: { _controller: campaignchain.activity.controller.acme.xing:newAction }

acme_activity_xing_edit:
    pattern: /activity/xing/{id}/edit
    defaults: { _controller: campaignchain.activity.controller.acme.xing:editAction }

acme_activity_xing_edit_modal:
    pattern: /modal/activity/xing/{id}/edit
    defaults: { _controller: campaignchain.activity.controller.acme.xing:editModalAction }

acme_activity_xing_edit_api:
    pattern: /api/private/activity/xing/byactivity/{id}/edit
    defaults: { _controller: campaignchain.activity.controller.acme.xing:editApiAction }
    options:
        expose: true

acme_activity_xing_read:
    pattern: /activity/xing/{id}
    defaults: { _controller: campaignchain.activity.controller.acme.xing:readAction }
```

Normally, you'd need to create views and controllers for the routes above. However, CampaignChain offers you a simpler approach: an ActivityHandler which contains methods to retrieve, create and process the data of an Activity. A stub ActivityHandler will already be produced for you by the CampaignChain module generator; all you need to do is fill out the stub methods with the appropriate code for your module.

Here's the code for the XingHandler:

```
<?php

// src/Acme/Activity/XingBundle/Controller/XingHandler
```

```
namespace Acme\Activity\XingBundle\Controller;

use CampaignChain\CoreBundle\Controller\Module\AbstractActivityHandler;
use Symfony\Component\Form\Form;
use CampaignChain\CoreBundle\Entity\Location;
use CampaignChain\CoreBundle\Entity\Campaign;
use CampaignChain\CoreBundle\Entity\Activity;
use CampaignChain\CoreBundle\Entity\Operation;
use Doctrine\ORM\EntityManager;
use Symfony\Bundle\TwigBundle\TwigEngine;
use Symfony\Component\HttpFoundation\Session\Session;

use Acme\Operation\XingBundle\Entity\XingMessage;
use Acme\Operation\XingBundle\EntityService\XingMessage as XingMessageService;
use Acme\Operation\XingBundle\Job\XingMessage as XingMessageJob;

/**
 * Class XingHandler
 * @package Acme\Activity\XingBundle\Controller\Module
 */
class XingHandler extends AbstractActivityHandler
{
    protected $em;
    protected $session;
    protected $templating;
    protected $contentService;
    protected $job;
    private $message;

    public function __construct(
        EntityManager $em,
        Session $session,
        TwigEngine $templating,
        XingMessageService $contentService,
        XingMessageJob $job
    )
    {
        $this->em = $em;
        $this->session = $session;
        $this->templating = $templating;
        $this->contentService = $contentService;
        $this->job = $job;
    }

    /**
     * When a new Activity is being created, this handler method will be called
     * to retrieve a new Content object for the Activity.
     *
     * Called in these views:
     * - new
     *
     * @param Location $location
     * @param Campaign $campaign
     * @return null
     */
    public function createContent(Location $location = null, Campaign $campaign = null)
    {
        return null;
    }

    /**
     * Overwrite this method to return an existing Activity Content object which
     * would be displayed in a view.
     *
     * Called in these views:
     * - edit
     * - editModal
     * - read
     *
     * @param Location $location
     * @param Operation $operation
     * @return null
     */
    public function getContent(Location $location, Operation $operation)
    {
        return $this->contentService->getMessageByOperation($operation);
    }

    /**
     * Implement this method to change the data of an Activity as per the form
     * data that has been posted in a view.
     *
     * Called in these views:
     * - new
     *
     * @param Activity $activity
     * @param $data Form submit data of the Activity.
     * @return Activity
     */
    public function processActivity(Activity $activity, $data)
    {
        return $activity;
    }

    /**
     * Modifies the Location of the Activity.
     *
     * Called in these views:
     * - new
     *
     * @param Location $location The Activity's Location.
     * @return Location
     */
    public function processActivityLocation(Location $location)
    {
        return $location;
    }

    /**
     * After a new Activity was created, this method makes it possible to alter
     * the data of the Content's Location (not the Activity's Location!) as per
     * the data provided for the Content.
     *
     * Called in these views:
     * - new
     *
     * @param Location $location Location of the Content.
     * @param $data Form submit data of the Content.
     * @return Location
     */
    public function processContentLocation(Location $location, $data)
    {
        return $location;
    }

    /**
     * Create or modify the Content object from the form data.
     *
     * Called in these views:
     * - new
     * - editApi
     *
     * @param Operation $operation
     * @param $data Form submit data of the Content.
     * @return mixed
     */
    public function processContent(Operation $operation, $data)
    {
        try {
            if(is_array($data)) {
                $message = $this->contentService->getMessageByOperation($operation);
                $message->setMessage($data['message']);
            } else {
                $message = $data;
            }
        } catch (\Exception $e){
            // message has not been created yet, so do it from the form data.
            $message = $data;
        }
        return $message;
    }

    /**
    * Define custom template rendering options for the new view in this method
    * as an array. Here's a sample of such an array:
    *
    * array(
    *     'template' => 'foo_template::edit.html.twig',
    *     'vars' => array(
    *         'foo1' => $bar1,
    *         'foo2' => $bar2
    *         )
    *     );
    *
    * Called in these views:
    * - new
    *
    * @param Operation $operation
    * @return null
    */
    public function getNewRenderOptions(Operation $operation = null)
    {
        return null;
    }
    /**
     * Overwrite this method to define how the Content is supposed to be
     * displayed.
     *
     * Called in these views:
     * - read
     *
     * @param Operation $operation
     * @return mixed
     */
    public function readAction(Operation $operation)
    {
        $message = $this->contentService->getMessageByOperation($operation);

        return $this->templating->renderResponse(
            'CampaignChainOperationXingBundle::read_message.html.twig',
            array(
                'page_title' => $operation->getActivity()->getName(),
                'operation' => $operation,
                'location' => $operation->getActivity()->getLocation(),
                'activity' => $operation->getActivity(),
                'message' => $message,
                'show_date' => true,
            ));
    }

    /**
     * The Activity controller calls this method after the form was submitted
     * and the new activity was persisted.
     *
     * @param Activity $activity
     * @param $data
     */
    public function postFormSubmitNewEvent(Activity $activity, $data)
    {
    }

    /**
     * This event is being called after the new Activity and its related
     * content was persisted.
     *
     * Called in these views:
     * - new
     *
     * @param Operation $operation
     * @param Form $form
     * @param $content The Activity's content object.
     * @return null
     */
    public function postPersistNewEvent(Operation $operation, Form $form, $content = null)
    {
        $this->publishNow($operation, $form, $content);
        $this->em->persist($content);
        $this->em->flush();
    }

    /**
     * This event is being called before the edit form data has been submitted.
     *
     * Called in these views:
     * - edit
     *
     * @param Operation $operation
     * @return null
     */
    public function preFormSubmitEditEvent(Operation $operation)
    {
        return null;
    }

    /**
     * This event is being called after the edited Activity and its related
     * content was persisted.
     *
     * Called in these views:
     * - edit
     *
     * @param Operation $operation
     * @param Form $form
     * @param $content The Activity's content object.
     * @return null
     */
    public function postPersistEditEvent(Operation $operation, Form $form, $content = null)
    {
        $this->publishNow($operation, $form, $content);
    }

    /**
     * Define custom template rendering options for the edit view in this method
     * as an array. Here's a sample of such an array:
     *
     * array(
     *     'template' => 'foo_template::edit.html.twig',
     *     'vars' => array(
     *         'foo1' => $bar1,
     *         'foo2' => $bar2
     *         )
     *     );
     *
     * Called in these views:
     * - edit
     *
     * @param Operation $operation
     * @return null
     */
    public function getEditRenderOptions(Operation $operation)
    {
        return null;
    }

    /**
     * This event is being called before the editModal form data has been
     * submitted.
     *
     * Called in these views:
     * - editModal
     *
     * @param Operation $operation
     * @return null
     */
    public function preFormSubmitEditModalEvent(Operation $operation)
    {
        return null;
    }

    /**
     * Define custom template rendering options for editModal view as array.
     *
     * Called in these views:
     * - editModal
     *
     * @see AbstractActivityHandler::getEditRenderOptions()
     * @param Operation $operation
     * @return null
     */
    public function getEditModalRenderOptions(Operation $operation)
    {
        return null;
    }

    /**
     * Let's a handler implementation define whether the Content should be
     * displayed or processed in a specific view or not.
     *
     * Called in these views:
     * - new
     * - edit
     * - editModal
     * - editApi
     *
     * @param $view
     * @return bool
     */
    public function hasContent($view)
    {
        return true;
    }

    private function publishNow(Operation $operation, Form $form)
    {
        if (!$form->get('campaignchain_hook_campaignchain_due')->has('execution_choice') || $form->ge
            return false;
        }

        $this->job->execute($operation->getId());
        $content = $this->contentService->getMessageByOperation($operation);
        $this->session->getFlashBag()->add(
            'success',
            'The message was published.'
        );

        return true;
    }
}
```

The ActivityHandler includes a number of methods, designed to let you customize the handling of key events in the CampaignChain workflow. You'll find an explanation in the comments above each method, but let's quickly look through the key methods here:

- The constructor injects the key dependencies needed for the handler, including the entity service manager and the job processor.
- The *getContent()* method returns an existing XingMessage entity, for use in an edit or display view.
- The *processContent()* method processes the input submitted in the Operation form and creates a new XingMessage entity. It is also invoked to process modifications to an existing XingMessage entity.
- The *readAction()* method retrieves an existing XingMessage entity and sets template variables from its properties for display.
- The *postPersistNewEvent()* and *postPersistEditEvent()* methods are called after a XingMessage entity is saved to the database. In this example, both methods invoke the *publishNow()* method.
- The *publishNow()* method checks when the Activity is scheduled for and, if set to "now", retrieves the job via the service manager and invokes its *execute()* method to post the message to the XING API.

Note: The Activity and Operation modules use CampaignChain's base views, and it is not necessary to create new views unless you specifically wish to override the base views.

7. Collect Metrics and Create a Report Processor Most online channels expose an API to collect metrics on posted data, such as the number of favorites, retweets, likes or comments. One of CampaignChain's key features is the ability to aggregate this data and generate reports to help end-users evaluate the success of a campaign.

To do this, it is necessary to define a second Job, this one responsible for periodically connecting to the service's API, collecting metrics and saving them to the CampaignChain database. These metrics are then used to generate reports through the CampaignChain interface.

The CampaignChain module generator will have got you started by creating a stub Job for this purpose within your Operation module, at *your-project/src/Acme/Operation/XingBundle/Job/XingMessageReport.php*. You'll notice that this stub file already includes variables for the two metrics, likes and comments, which you specified in Step 1 when generating the Operation module.

Update this file with the following code:

```php
<?php

// src/Acme/Operation/XingBundle/Job/XingMessageReport.php

namespace Acme\Operation\XingBundle\Job;

use CampaignChain\CoreBundle\Entity\SchedulerReportOperation;
use CampaignChain\CoreBundle\Job\JobReportInterface;
use Doctrine\ORM\EntityManager;

class XingMessageReport implements JobReportInterface
{
    const OPERATION_BUNDLE_NAME = 'acme/operation-xing';
    const METRIC_LIKES      = 'Likes';
    const METRIC_COMMENTS = 'Comments';

    protected $em;
    protected $container;
    protected $message;
    protected $operation;

    public function __construct(EntityManager $em, $container)
    {
        $this->em = $em;
        $this->container = $container;
    }

    public function getMessage(){
        return $this->message;
    }

    public function schedule($operation, $facts = null)
    {
        $scheduler = new SchedulerReportOperation();
        $scheduler->setOperation($operation);
        $scheduler->setInterval('1 hour');
        $scheduler->setEndAction($operation->getActivity()->getCampaign());
        $this->em->persist($scheduler);

        $facts[self::METRIC_LIKES] = 0;
        $facts[self::METRIC_COMMENTS] = 0;

        $factService = $this->container->get('campaignchain.core.fact');
        $factService->addFacts('activity', self::OPERATION_BUNDLE_NAME, $operation, $facts);
    }

    public function execute($operationId)
    {
        $operationService = $this->container->get('campaignchain.core.operation');
        $operation = $operationService->getOperation($operationId);

        $message = $this->em
                        ->getRepository('AcmeOperationXingBundle:XingMessage')
                        ->findOneByOperation($operationId);

        if (!$message) {
            throw new \Exception('No message found for an operation with ID: '.$operationId);
        }

        $activity = $message->getOperation()->getActivity();
        $messageId = $message->getMessageId();

        $client = $this->container->get('acme.channel.xing.rest.client');
        $connection = $client->connectByActivity($activity);

        $request = $connection->get('activities/' . $messageId, array());
        $response = $request->send()->json();

        if(!count($response)){
            $likes = 0;
            $comments = 0;
        } else {
            $likes = $response['activities'][0]['likes']['amount'];
            $comments = $response['activities'][0]['comments']['amount'];
        }

        // Add report data.
        $facts[self::METRIC_LIKES] = $likes;
        $facts[self::METRIC_COMMENTS] = $comments;

        $factService = $this->container->get('campaignchain.core.fact');
        $factService->addFacts('activity', self::OPERATION_BUNDLE_NAME, $operation, $facts);

        $this->message = 'Added to report: likes = '.$likes.', comments = '.$comments.'.';

        return self::STATUS_OK;
    }

}
```

If you look into the *execute()* method above, you'll see that it begins by retrieving the required message from the CampaignChain database (using the message identifier). It then uses the message object's *getMessageId()* method to retrieve the unique identifier for that message on XING.

Next, it invokes the XingClient created earlier from the service manager and uses the client to authenticate against the XING API.

It then uses the client's inherited *get()* method to transmit a GET request to the API endpoint https://api.xing.com/v1/activities/ID, passing the message identifier to XING. If successful, the response will include complete details for the specified message, including the number of likes and comments it received. It's now easy enough to extract these metrics from the response and save them to the CampaignChain database using the Fact service. The XING documentation explains this API method in more detail[57].

The *execute()* method does the work of collecting data from the XING API and saving it to the CampaignChain database...but how is it invoked? That's where the *schedule()* method comes in it creates a new SchedulerReportOperation that runs periodically for a specified operation. In this case, the scheduler is configured to collect data every hour, and run for as long as the campaign associated with the operation.

Since the scheduler is associated with an Operation, it makes sense to invoke the *schedule()* method when the operation is created. So, update the *your-project/src/Acme/Operation/XingBundle/Job/XingMessage.php* file and add the lines below to its *execute()* method, to be executed after the message is successfully posted:

```
<?php

  // src/Acme/Operation/XingBundle/Job/XingMessage

  public function execute($operationId)
  {
      // ... snip
      $location->setStatus(Medium::STATUS_ACTIVE);

      // schedule data collection for report
      $report = $this->container->get('campaignchain.job.report.acme.xing.message');
      $report->schedule($message->getOperation());

      $this->em->flush();
      // ... snip
  }
```

Conclusion

At this point, your modules are all complete. Once you add your modules to CampaignChain through the module installer, or reset your system to automatically register them, you should be able to connect to a new XING location

[57] https://dev.xing.com/docs/get/activities/:id

and begin posting news to it. Try it out for yourself and see how easy it is!

Creating Update Routines for Your Module

When you `update a CampaignChain installation`, the automatic update functionality takes the update routines provided with each CampaignChain module and applies them to the database schema and user-generated data (e.g. data in the database or in the file system).

When developing your own modules, it is best practice to include such update routines as soon as something changes in your module that affects the database schema or user-generated data.

CampaignChain ships with commands that support developers in creating and testing such update routines.

Prerequisites

Automatic updates only work if you installed CampaignChain as explained in the README.md file[58].

To verify whether this is the case, please check whether the following tables exist in your CampaignChain database and that they are populated with data:

- *campaignchain_update_schema*
- *campaignchain_update_data*

Warning: If you installed CampaignChain prior to July 23rd, 2016, then the automatic update functionality is not included. Please make a new installation from scratch. If you need help with migrating your data, please contact us.

http://www.campaignchain.com/contact

Schema Update

Whenever you add or modify an entity class[60] in your CampaignChain module, it affects the database. To keep track of changes and allow for automatic update of a CampaignChain installation, there are commands that ship with CampaignChain:

- *campaignchain:schema:diff*: Automatically generates a PHP class that includes SQL commands that define how to modify the database schema as per your changes.
- *campaignchain:update:schema*: Apply the changes to the database schema.

Given that our schema update functionality is based on the Doctrine Migration Bundle[61], you could also use the following command:

- *doctrine:migrations:status*
- *doctrine:migrations:migrate*

Warning: CampaignChain's update functionality extends that of the Doctrine Migration Bundle to manage update routines contained in multiple modules. Hence, you should not use other Doctrine Migration commands unless you understand the inner workings of the Doctrine Migration Bundle in relation to CampaignChain's module system.

[58] https://github.com/CampaignChain/campaignchain/blob/master/README.md#installation
[60] http://symfony.com/doc/current/book/doctrine.html#creating-an-entity-class
[61] http://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html

Note: As of now, schema updates are only available for MySQL.

Generate a Schema Diff Whenever you add or modify an entity class in a module, you should run the *diff* command for that specific module. This ensures that the generated *diff* file includes only the changes for that single module.

Note: It is best practice to provide update routines per CampaignChain module.

To create a *diff* file, execute the following command in the root of your CampaignChain installation:

```
$ php app/console campaignchain:schema:diff
```

You will be asked to provide the Composer package name[62], so that the command knows where to put the *diff* file. For example: *acme/my-package*. Once you provided the name, the command will place a schema update file in the directory */Resources/udpate/schema* of your module. The name of the file is *Version*.php*.

Inside the schema update file, you will find the SQL commands for up- or downgrading. For example:

```
$this->addSql('RENAME TABLE acme_my_bundle_activity_sensors TO acme_my_bundle_activity_iot_devices');
```

If you are working on the same module for a longer time, it becomes cumbersome to always provide the package name. To make life easier, you can configure a default package name in *app/config/parameters.yml* with:

```
campaignchain_update:
    diff_package: acme/my-package
```

Test the Schema Update To test your schema update routines, run this command in the root of your CampaignChain installation:

```
$ php app/console campaignchain:update:schema
```

You should now see the results of the applied changes:

```
Migrating up to 20160723191847 from 20160717050911

  ++ migrating 20160723191847

     -> RENAME TABLE acme_my_bundle_activity_sensors TO acme_my_bundle_activity_iot_devices

  ++ migrated (0.05s)

  ------------------------

  ++ finished in 0.05s
  ++ 1 migrations executed
  ++ 1 sql queries
```

If you would like to manually change something in the SQL and then test the schema update routine again, first downgrade your installation to the previous version, so that it does not include your module's updates. Issue this command to find the previous version:

```
$ php app/console doctrine:migrations:status
```

[62] https://getcomposer.org/doc/04-schema.md#name

In the displayed information, find the entry for the *Previous Version*:

```
>> Previous Version:                                   2016-07-17 05:09:11 (20160717050911)
```

Roll back your CampaignChain installation by issuing this command and the version number provided in above entry:

```
$ php app/console doctrine:migrations:migrate 20160717050911
```

Once you confirmed to downgrade, you run CampaignChain's schema update command again to apply the changes:

```
$ php app/console campaignchain:update:schema
```

Data Update

If changes in your module affect user-generated data, then you should use the data update functionality available with CampaignChain. Data updates are similar to schema updates, but they require more manual work.

Create the Data Update Class Start by defining the data update PHP class, which is supposed to implement the data update interface and to reside in the */Resources/update/data* folder of your module.

```
<?php
// /Resources/update/data/UpdateAddIdToName.php

namespace Acme\MyBundle\Resources\update\data;

use CampaignChain\UpdateBundle\Service\DataUpdateInterface;
use Doctrine\ORM\EntityManager;
use Symfony\Component\Console\Style\SymfonyStyle;

class UpdateAddIdToName implements DataUpdateInterface
{
    private $entityManager;

    public function __construct(EntityManager $entityManager)
    {
        $this->entityManager = $entityManager;
    }

    public function getVersion()
    {
        return 20160723191847;
    }

    public function getDescription()
    {
        return [
            'Adding the device ID to its name to make it globally unique',
        ];
    }

    public function execute(SymfonyStyle $io = null)
    {
        $iotDevices = $this->entityManager
            ->getRepository('AcmeMyBundle:IotDevice')
            ->findAll();

        if (empty($iotDevices)) {
            $io->text('There is no IoT device to process');

            return true;
        }

        foreach ($iotDevices as $iotDevice) {
            $iotDevice->setName($iotDevice->getId().'-'.$iotDevice->getName());
            $this->entityManager->persist($iotDevice);
        }

        $this->entityManager->flush();

        return true;
    }

}
```

In the *__construct()* method, you can inject Symfony services or parameters that the data update class needs to execute. In our case, we pass the Doctrine entity manager to the constructor.

The *getVersion()* method is supposed to return the version string of this update, which consists of the year, month, day, hour, minute and second. The version allows the CampaignChain updater to execute your data update in the right order in relation to other data update scripts.

In the *getDescription()* method, you can provide multiple lines of information as array values, which will be displayed in the shell while your data update routine gets executed.

The actual work is being done in the *execute()* method. This is where you can change data in the database, move or rename files, connected to other applications to retrieve data, etc.

Define the Data Update Service Essentially, data updates are tagged Symfony services[63]. Hence, the next step is to define the service for above data update routine.

In the */Resources/config/services.yml* file of your module, add the following lines:

```
acme_my_bundle.update.add_id_to_name:
    class: Acme\MyBundle\Resources\update\data\UpdateAddIdToName
    arguments:
        - '@doctrine.orm.default_entity_manager'
    tags:
        - { name: campaignchain.update.data }
```

What you see here is that we define a service name *acme_my_bundle.update.add_id_to_name*. The best practice is to start with a string that resembles the Composer package name of your module, followed by *.update.* and then a string that resembles the name of the data update class.

Next, we define the class path of the data update class.

The *arguments* parameter is where Symfony services and parameters can be passed to the constructor of the data update class.

Finally, you must provide the tag *campaignchain.update.data*, because only then will your data update class be included in CampaignChain's updater.

Testing the Data Update Routine To test the data update routine for your module, issue this command in the route of your CampaignChain installation:

[63] http://symfony.com/doc/current/components/dependency_injection/tags.html

```
$ php app/console campaignchain:update:data
```

Note: Currently, downgrading data updates is not possible.

Checklist for Developing Modules

- Make sure the GUI uses only Bootstrap components, CSS and JS. Where applicable, use the Braincrafted Boostrap components[64].
- Add license information and copyright notice on top of each file in your module.
- Did you remove all unnecessary files which have been automatically created by the module generator (e.g. *Controller/Default.php*, *Resources/doc*)?
- Test against the sample data and first of all, make sure that it gets loaded properly.
- Update the sample data if applicable and in a respective feature or release branch.
- Make sure each module ships with schema and data update routines.
- Reset the system and install again to test whether the new feature breaks anything.
- Please contact us[65] to add newly released modules to the modules store[66].

3.2 Administrators

Documentation that explains how to install and configure CampaignChain.

3.2.1 Community Edition Administrator Handbook

Configuration

Overview

CampaignChain knows two types of configuration options:

- App-wide parameters
- Bundle-specific parameters

App-wide Parameters Parameters that configure the app-wide framework are prefixed with `campaignchain.` and defined in two files:

- *app/config/parameters.yml* contains the parameters that should be considered during installation under the `parameters` key.
- In *app/config/config_campaignchain.yml*, you will find other app-wide parameters (again under the `parameters` key), which contain default values that should be fine for most installations of CampaignChain.

For example, in *parameters.yml*, there's this option to define the route of the tracking script:

[64] http://bootstrap.braincrafted.com/components.html
[65] http://www.campaignchain.com/contact
[66] https://github.com/store-campaignchain-com/store-campaignchain-com.github.io

```
parameters:
    campaignchain.env: prod
    campaignchain.tracking.js_route: /tracking.js
```

Bundle-specific Parameters Each bundle or module might define its own configuration options. Should you want to override the default values, we recommend to set your values in *app/config/config_campaignchain_bundles.yml* under an alias for each bundle, e.g. `campaignchain_core`.

For example:

```
campaignchain_core:
    env: prod
    tracking:
        id_name: cctid
        js_mode: prod
        js_class: CCTracking
        js_init: cc
```

You can look up the available options in the *Resources/config/config.yml* file for each bundle.

Themes

The look & feel of CampaignChain can be configured as follows.

Skin The colors of the CampaignChain theme can be changed through skins. The skins available are the same skins as for AdminLTE[67].

You can specify the skin in *app/config/parameters.yml*:

```
campaignchain.theme.skin: skin-black
```

Layout CampaignChain lets you configure various parts of the layout. For example, whether you want to show the header and footer of the graphical user interface. This comes handy if you want to omit them to embed CampaignChain through an iFrame.

The standard configuration of CampaignChain supports two layouts:

- default
- embedded

You can find the respective configuration in the *config.yml* file of the core bundle:

```
campaignchain_core:
    theme:
        layouts:
            default:
                show_header: true
                show_footer: true
            embedded:
                show_header: false
                show_footer: false
```

[67] https://almsaeedstudio.com/themes/AdminLTE/documentation/index.html#layout

To enable a specific layout, call a page in CampaignChain with the *campaignchain-layout* URL parameter. For example:

http://www.mycampaignchain.com/?campaignchain-layout=embedded

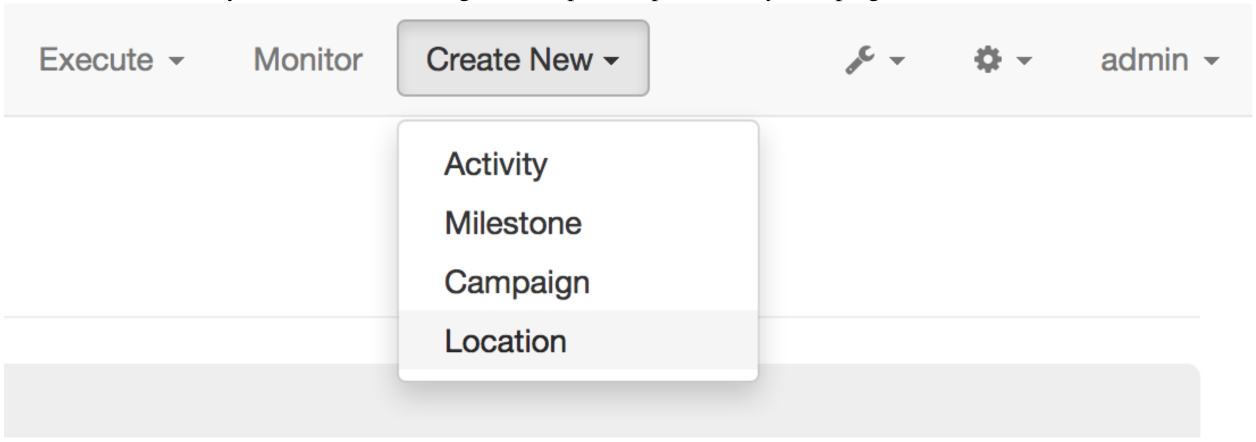
The setting will be applied to all subsequent pages. If you want to go back to the default view, simply switch by calling:

http://www.mycampaignchain.com/?campaignchain-layout=default

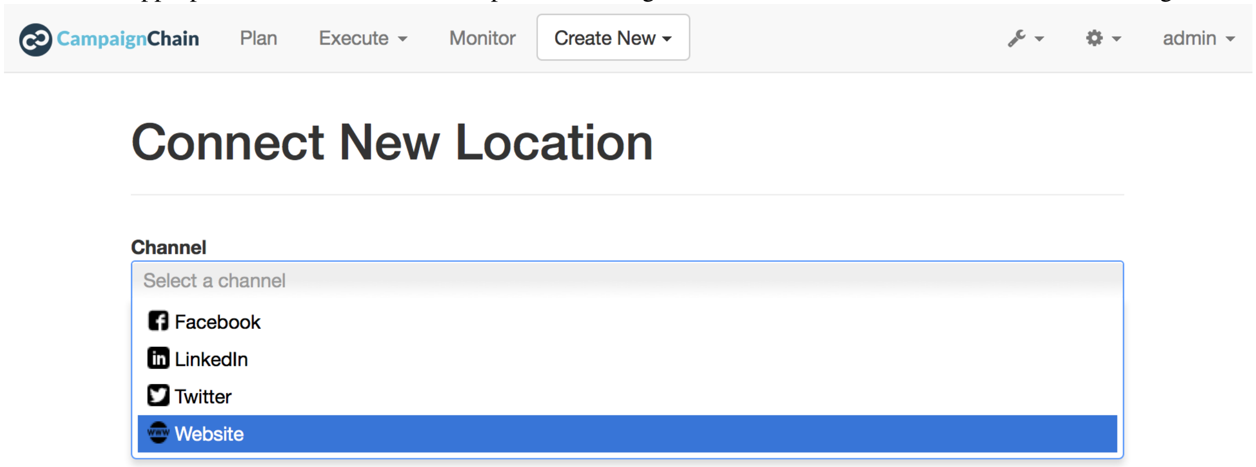
Call to Action Configuration

To enable *CTA tracking* (page 18) for a Channel, follow these steps:

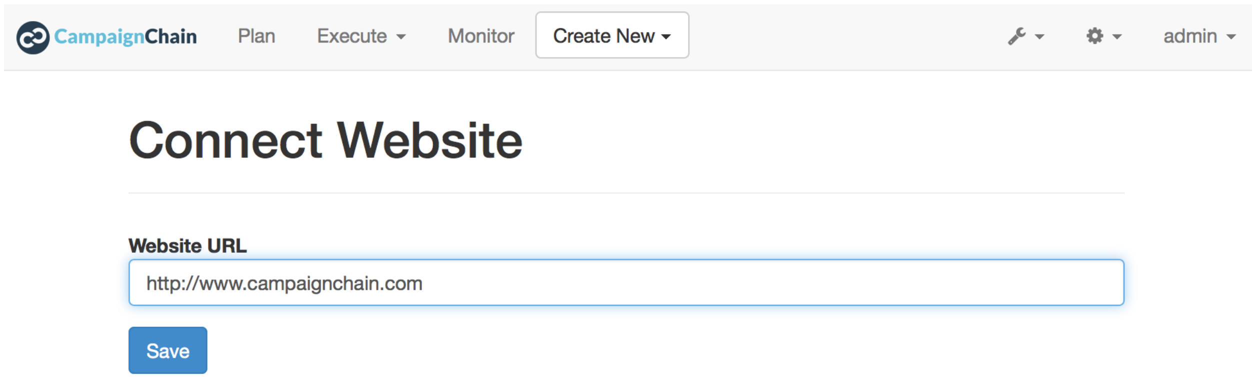
1. Connect Trackable Channel In the CampaignChain header menu, click the *Create New* button and select a *Location* that allows you to add the tracking JavaScript code provided by CampaignChain.



Choose the appropriate Location from the drop-down list, e.g. *Website* to include a Website into CTA tracking.



Fill in the required data to connect the Channel. For example, provide the base URL of a Website (you can omit adding pages of the Website).



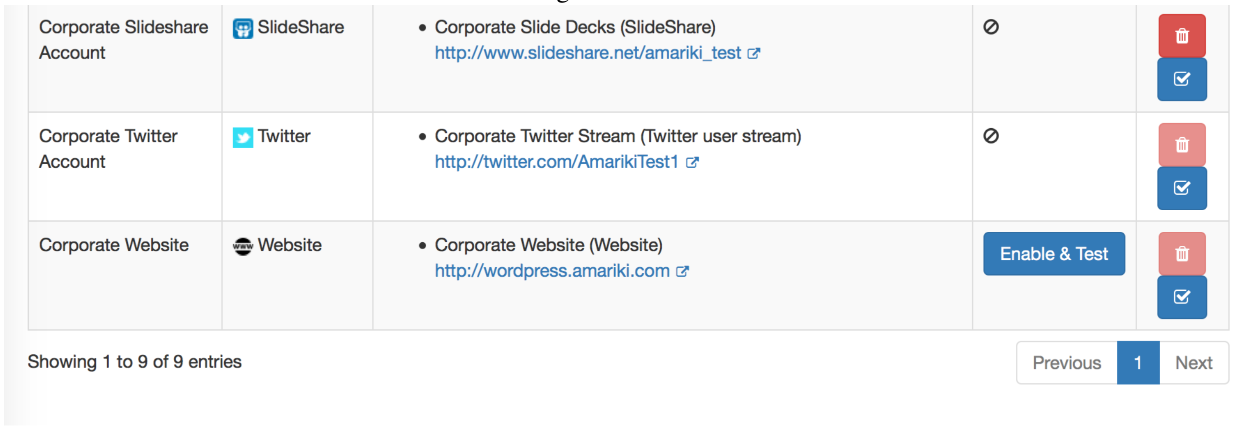
Connect Website

Website URL

http://www.campaignchain.com

2. Include Tracking Code Once you're done with connecting a trackable Channel, CampaignChain will display a list of connected Channels to you. The list of Channels can also be reached via the settings icon in the header.

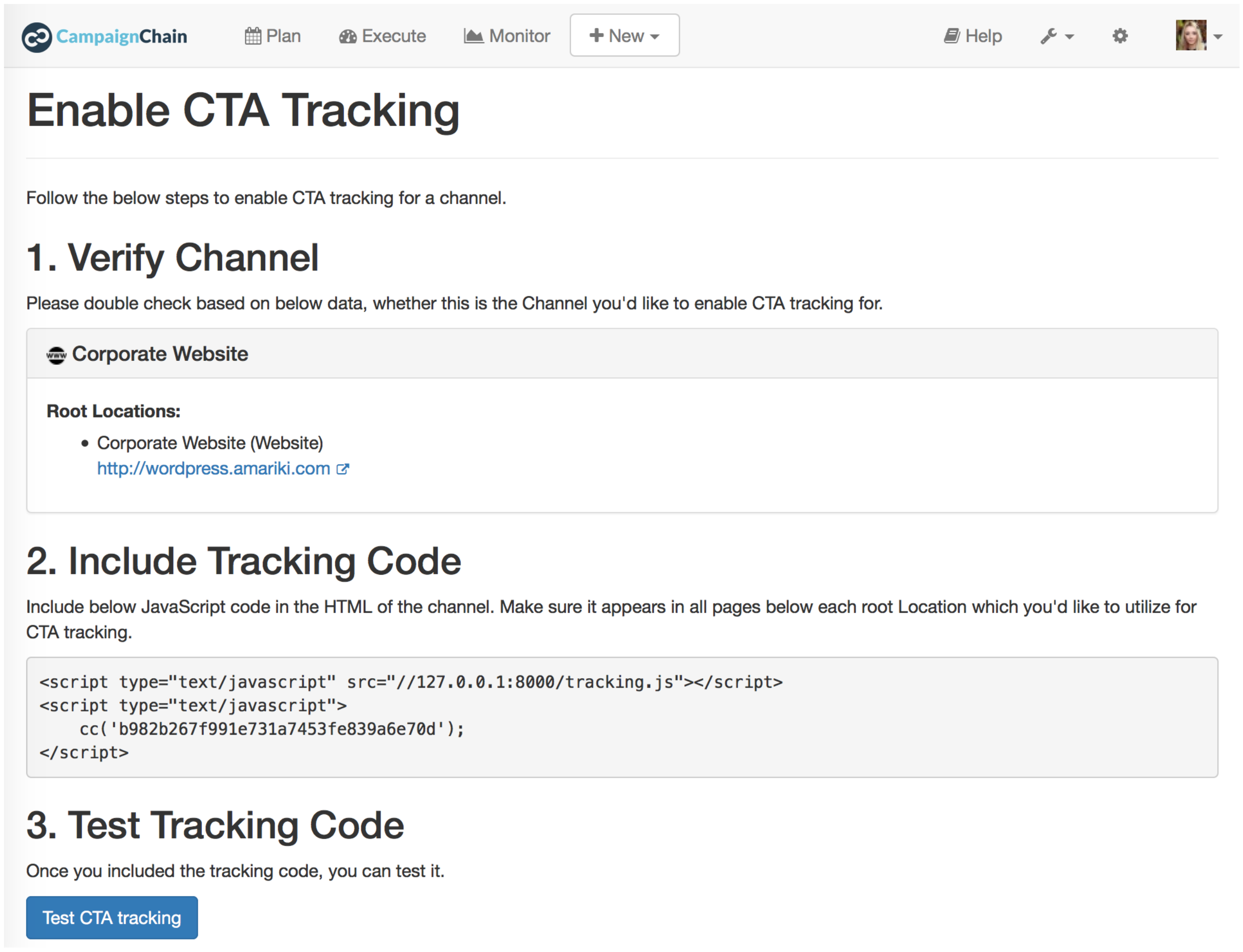
This list will include a button to enable CTA tracking for trackable Channels.



Corporate Slideshare Account	SlideShare	• Corporate Slide Decks (SlideShare) http://www.slideshare.net/amariki_test	⊘	
Corporate Twitter Account	Twitter	• Corporate Twitter Stream (Twitter user stream) http://twitter.com/AmarikiTest1	⊘	
Corporate Website	Website	• Corporate Website (Website) http://wordpress.amariki.com	Enable & Test	

Showing 1 to 9 of 9 entries

Clicking on such button will display a page with instructions to include the tracking snippet.



Enable CTA Tracking

Follow the below steps to enable CTA tracking for a channel.

1. Verify Channel

Please double check based on below data, whether this is the Channel you'd like to enable CTA tracking for.

Corporate Website

Root Locations:

- Corporate Website (Website) http://wordpress.amariki.com

2. Include Tracking Code

Include below JavaScript code in the HTML of the channel. Make sure it appears in all pages below each root Location which you'd like to utilize for CTA tracking.

```
<script type="text/javascript" src="//127.0.0.1:8000/tracking.js"></script>
<script type="text/javascript">
    cc('b982b267f991e731a7453fe839a6e70d');
</script>
```

3. Test Tracking Code

Once you included the tracking code, you can test it.

Test CTA tracking

2.1 Directly within HMTL Source Code If you own the Channel you plan to include and you also have full control to directly manipulate the HTML source, then you can include the JavaScript snippet provided on the "Enable CTA Tracking" page.

Include the snippet by adding the code to your Channel, ideally right before the closing body element (i.e. *</body>* element) and make sure that it appears on all pages of the Channel.

2.2 Use Google Tag Manager In case you can't access the Channel or instead of waiting months for someon else to update the code of a Channel, Google Tag Manager (GTM) comes to the rescue. It lets you launch new tags with just a few clicks. GTM supports container snippets, a small piece of JavaScript or non-JavaScript code that it includes into your pages.

Google Tag Manager allows you to create a new *Custom HTML Tag* at the GTM Web interface[68]. Paste the CampaignChain tracking code into the *HMTL* section. Make sure the event is fired on all pages. The last step is to publish your new tag in GTM.

3. Anonymizing or Branding the Tracking Script If you want to hide from visitors to your CTA-tracked Channel, that you are using CampaignChain or you want to custom brand the tracking code, then we have configuration options for you.

One is in the *app/config/parameters.yml* file:

[68] https://tagmanager.google.com

```
parameters:
    campaignchain.tracking.js_route: /tracking.js
```

The URL of the tracking script itself can be changed with *campaignchain.tracking.js_route*. There you defined the path aka URI to the script relative to the base URL where CampaignChain is installed.

> **Warning:** We recommend to not change the route after you started using CampaignChain. If you have to for whatever reason, be aware that it affects all Channels that include the tracking snippet. They would have to adjust the path to the tracking script in said snippet.

The other relevant configuration options can be found in *app/config/config_campaignchain_bundles.yml*:

```
campaignchain_core:
    tracking:
        id_name: cctid
        js_mode: prod
        js_class: CCTracking
        js_init: cc
```

With *campaignchain_core.tracking.id_name*, you can define the name of the URL parameter which CampaignChain attaches to links pointing to a connected channel. Make sure the name you choose is short and as unique as possible, to avoid that it collides with other parameters that might already be in the URL.

> **Warning:** Never change the `id_name` after you started using CampaignChain, because previous tracking data might get lost and there are unforeseeable side-effects with upcoming changes to the functionality.

The name of the JavaScript class that appears inside the tracking script can be customized with the *campaign-chain_core.tracking.js_class* parameter.

Finally, *campaignchain_core.tracking.js_init* allows you to define the name of the JavaScript function that is being called to pass the Channel ID in the tracking code.

> **Warning:** As with the route, we recommend to not change it after you started using CampaignChain, for the very same reasons.

When accessed through the Symfony production environment (i.e. */app.php*), then the tracking script will automatically be minimized/obfuscated. This is how it looks then with the default configuration parameters explained above:

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a)))+((c=c%a)>35?String.fromChar
```

OAuth Apps

Create new OAuth Apps To allow CampaignChain to post to Twitter, Facebook and some more Channels, it must be able to access the REST APIs of such Channels. Hence, CampaignChain must be registered as an App with these Channels to receive an App Key and App Secret.

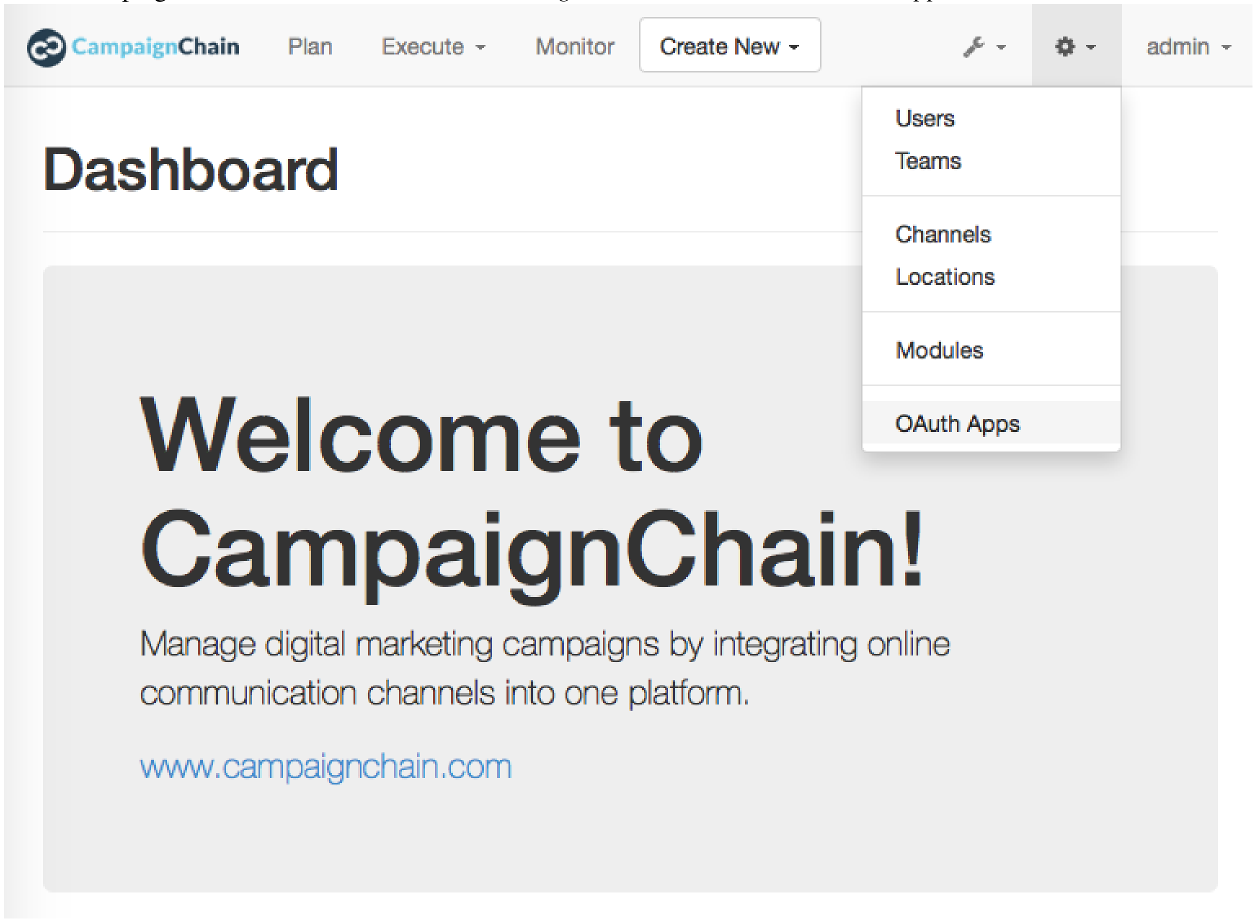
There are How-Tos available for each Channel:

- *Twitter* (page 116)
- *Facebook* (page 117)
- *Linkedin* (page 119)
- *Google Analytics* (page 120)
- *MailChimp* (page 121)

- *GoToWebinar* (page 122)
- *SlideShare* (page 124)

Edit existing OAuth App credentials The App Keys and Secrets of Channels which have already been connected with CampaignChain, can be edited as follows:

In the CampaignChain header menu, click the *Settings* icon and select *OAuth Client Apps*.



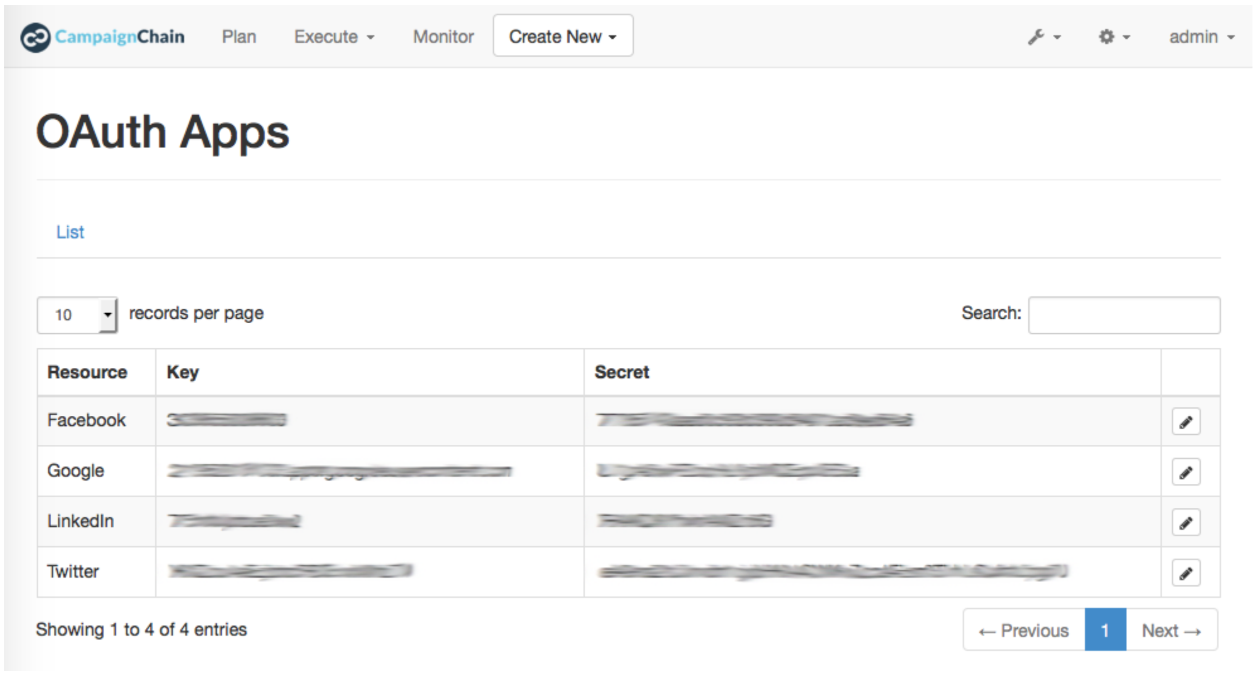
Dashboard

Welcome to CampaignChain!

Manage digital marketing campaigns by integrating online communication channels into one platform.

www.campaignchain.com

In the list of OAuth Apps, pick the entry you'd like to edit by clicking on the *Edit* icon.



OAuth Apps

List

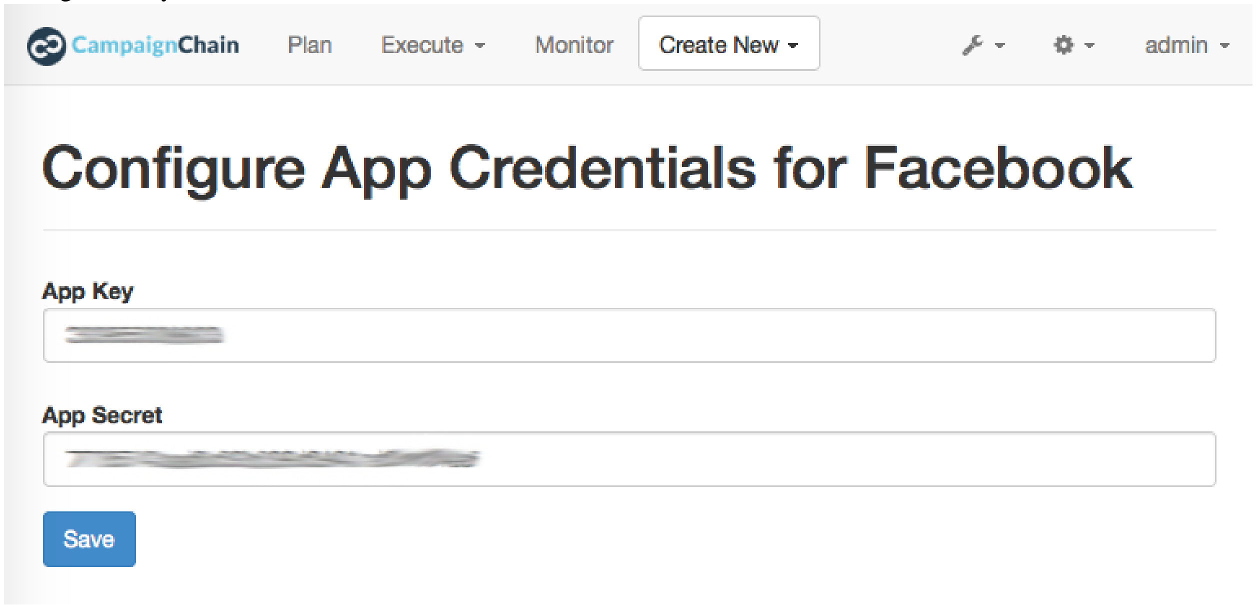
10 records per page

Search:

Resource	Key	Secret
Facebook	[redacted]	[redacted]
Google	[redacted]	[redacted]
LinkedIn	[redacted]	[redacted]
Twitter	[redacted]	[redacted]

Showing 1 to 4 of 4 entries

Change the Key and Secret field and click *Save*.



Configure App Credentials for Facebook

App Key

[redacted]

App Secret

[redacted]

System Requirements

To run CampaignChain, your system must meet the following requirements:

- Composer (https://getcomposer.org/)
- PHP 5.4 or better
- PHP's JSON, PDO and intl extensions enabled
- PHP's `system()` function must work
- MySQL 5.5 or better

- Java 1.5 or better

Part IV

CampaignChain Enterprise Edition

The CampaignChain Enterprise Edition is a commercial product. It is based on the CampaignChain Community Edition and provides additional features, such as easy deployment for development or production environments, as well as automated updates to simplify the maintenance of CampaignChain instances.

CHAPTER 4

CampaignChain Enterprise Edition

The CampaignChain Enterprise Edition is a commercial product for in-house IT teams or service providers who would like to build custom solutions. With the Custom Edition, they can benefit from easier maintenance, additional functionality, as well as services and support from the vendor.

The Enterprise Edition is based on the CampaignChain Community Edition. It provides additional end-user features and simplifies deployment for development or production environments.

4.1 Administrator Handbook for EE

4.1.1 Installation

Platform.sh

This is a step-by-step guide on how to deploy CampaignChain Enterprise Edition on Platform.sh.

CampaignChain Enterprise Edition can be installed out-of-the-box on Platform.sh, either for production or development.

What's great about Platform.sh is that you can hook it up with GitHub to automatically re-build and update CampaignChain on a Platform.sh server as soon as changes to the code have been pushed to GitHub.

Development or Production

You can configure Platform.sh so that CampaignChain will be built and deployed either for development or production use.

When in production mode, CampaignChain will be installed when you first create the Platform.sh environment and afterwards, update routines will be executed automatically upon every push to the GitHub repository which keep the database up-to-date.

When in development mode, CampaignChain will be installed from scratch every time a GitHub push happens. Additionally, sample data will be imported to CampaignChain automatically. This means that all data created manually in the database will be lost.

Configuring Platform.sh and GitHub

First, you create a new Platform.sh project by following these steps:

1. Select a Platform.sh plan[69]. Select "Development" if you would like to use Platform.sh as a live development environment or another plan if you'd like to use it for a production site.
2. Add an environment variable[70] to the Platform.sh environment with the name *campaignchain.env* and either the value *"dev"* for developing with CampaignChain, or *"prod"* to run CampaignChain EE for production use. Don't forget to put the value in quotes. Activate the *JSON value* checkbox.
3. To allow Platform.sh to access the private GitHub repositories which are included in the distribution, proceed as follows: Copy the SSH deploy key[71], then log into GitHub as the Platform.sh machine user. You can find the username and password in the credentials file you received with CampaignChain EE. Then add the Platform.sh deploy key, which you just copied, as a new deploy key of the machine user[72].

Deploying on Platform.sh

To sync the CampaignChain EE GitHub repository with Platform.sh, install the Platform.sh Command Line Interface[73] on your computer.

Then execute the following command. Replace *[project-id]* with the ID of your Platform.sh project and *[github-token]* with your personal access token[74] for GitHub:

```
platform integration:add --type=github --project=[project-id]
--token=[github-token]
--repository=CampaignChain/campaignchain-ee --build-pull-requests=true
--fetch-branches=true
```

[69] https://accounts.platform.sh/platform/buy-now
[70] https://docs.platform.sh/administration/web/configure-environment.html#variables
[71] https://docs.platform.sh/development/private-repository.html#pull-code-from-a-private-git-repository
[72] https://github.com/settings/keys
[73] https://docs.platform.sh/overview/cli.html#how-do-i-get-it
[74] https://docs.platform.sh/administration/integrations/github.html#1-generate-a-token

Part V

Users

Tips & tricks for marketers to make the best use of CampaignChain.

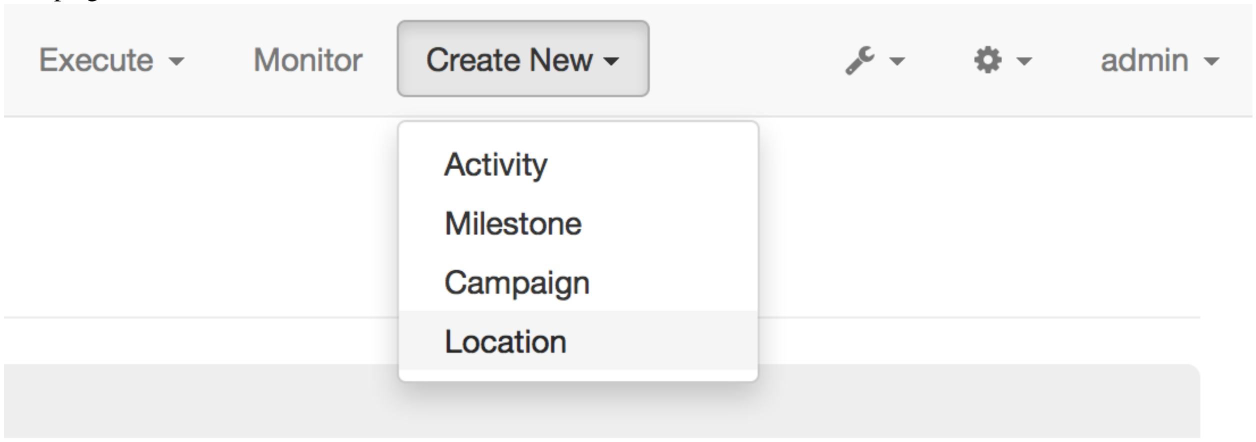
User Manual

5.1 Get Started

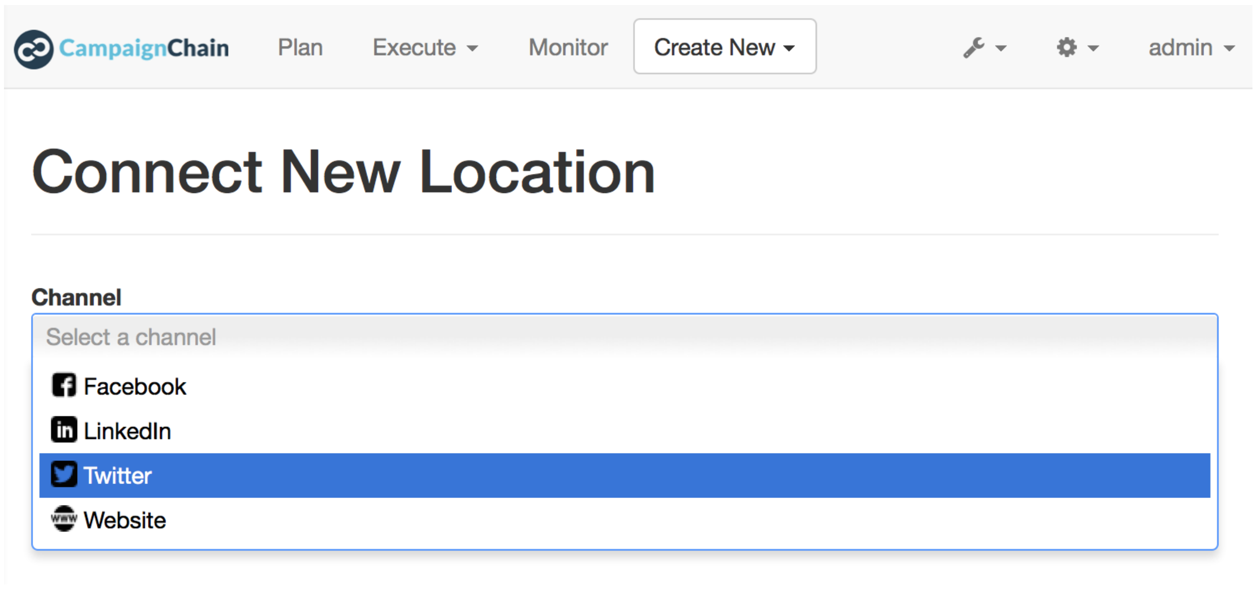
This is a brief step-by-step guide on how to create your first marketing Campaign and Activity within CampaignChain. You will learn how to connect Twitter as a Channel and how to post a Tweet on the stream of the related Twitter user account.

5.1.1 1. Connect to a Location

To show you how to connect a new Location, we'll use Twitter as an example. To connect a Twitter Location with CampaignChain, click the *Create New* button in the header menu and choose *Location*.

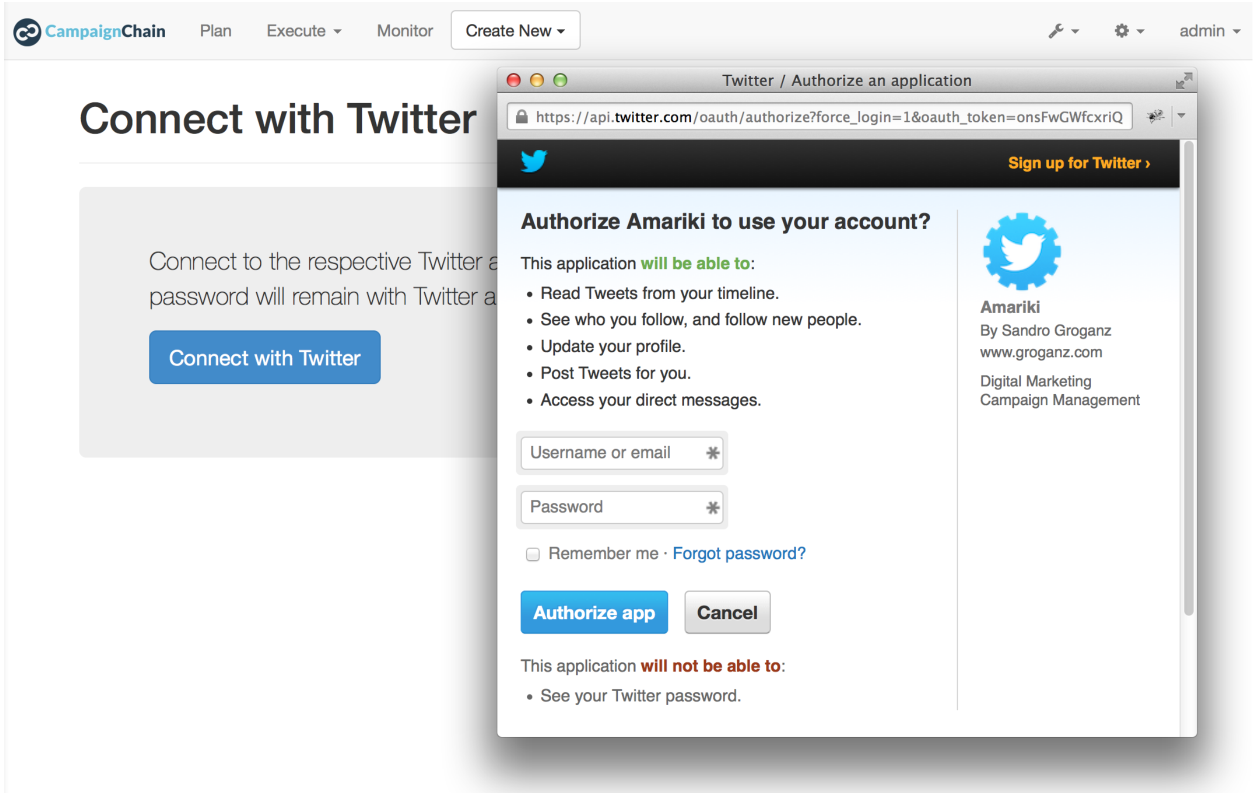


On the next screen, select Twitter as the Channel and click *Next*.



Note: Should you now see the *Provide Application Credentials* screen, then please see *how to create new OAuth Apps* (page 89).

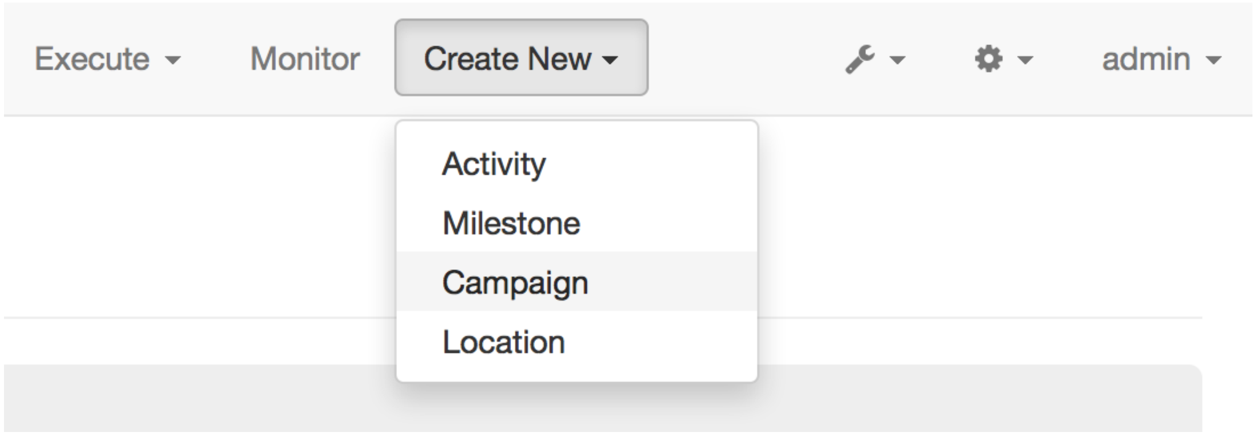
When clicking the button *Connect with Twitter*, the login screen for Twitter will be displayed to you. Please enter your Twitter user name and your Twitter password.


CampaignChain
Plan
Execute
Monitor
Create New
admin
Connect with Twitter
Connect to the respective Twitter a
password will remain with Twitter a
Connect with Twitter
Twitter / Authorize an application
https://api.twitter.com/oauth/authorize?force_login=1&oauth_token=onsFwGWfcxriQ
Sign up for Twitter ›
Authorize Amariki to use your account?
This application will be able to:
• Read Tweets from your timeline.
• See who you follow, and follow new people.
• Update your profile.
• Post Tweets for you.
• Access your direct messages.
Username or email
Password
Remember me · Forgot password?
Authorize app
Cancel
This application will not be able to:
• See your Twitter password.
Amariki
By Sandro Groganz
www.groganz.com
Digital Marketing
Campaign Management

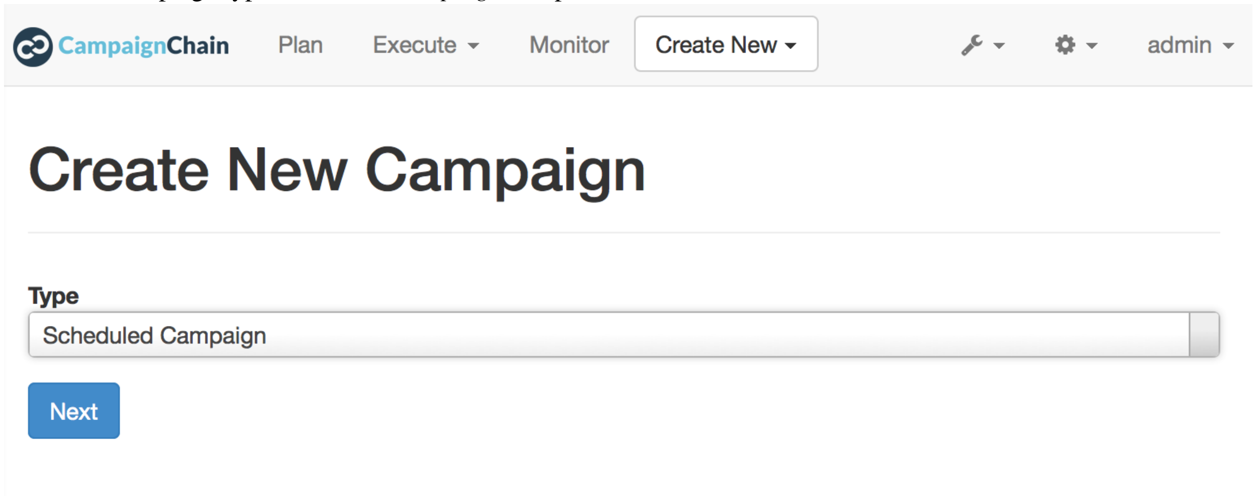
If Twitter accepted your credentials, the stream of the Twitter user you logged in as will now be available as a Channel Location within CampaignChain.

5.1.2 2. Create a Campaign

An Activity such as posting on Twitter can only be created from within a Campaign. Click the *Create New* button in the header and choose *Campaign*.



Select the campaign type *Scheduled Campaign* and proceed with *Next*.



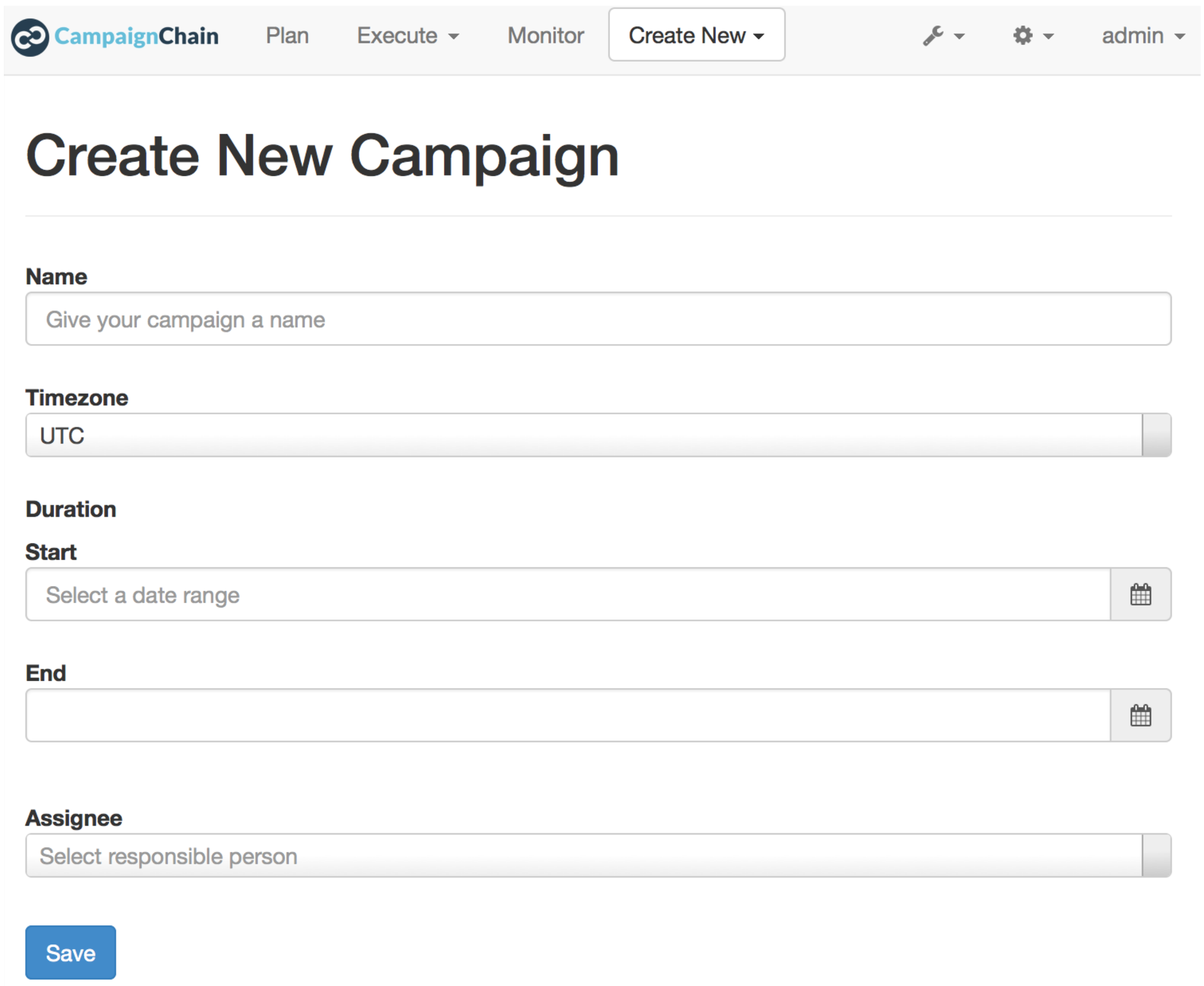
Create New Campaign

Type
Scheduled Campaign

Fill in the fields to populate your new Campaign with data, such as:

- *Name*: An arbitrary name of your Campaign, e.g. "Launch of new product"
- *Timezone*: The timezone of the Campaign. For international marketing teams, the best choice is *UTC*.
- *Duration*: Pick the start and end date of your Campaign.
- *Assignee*: The person in your team responsible for the Campaign.

Click *Save* and your first Campaign will be created.



Create New Campaign

Name

Give your campaign a name

Timezone

UTC

Duration

Start

Select a date range

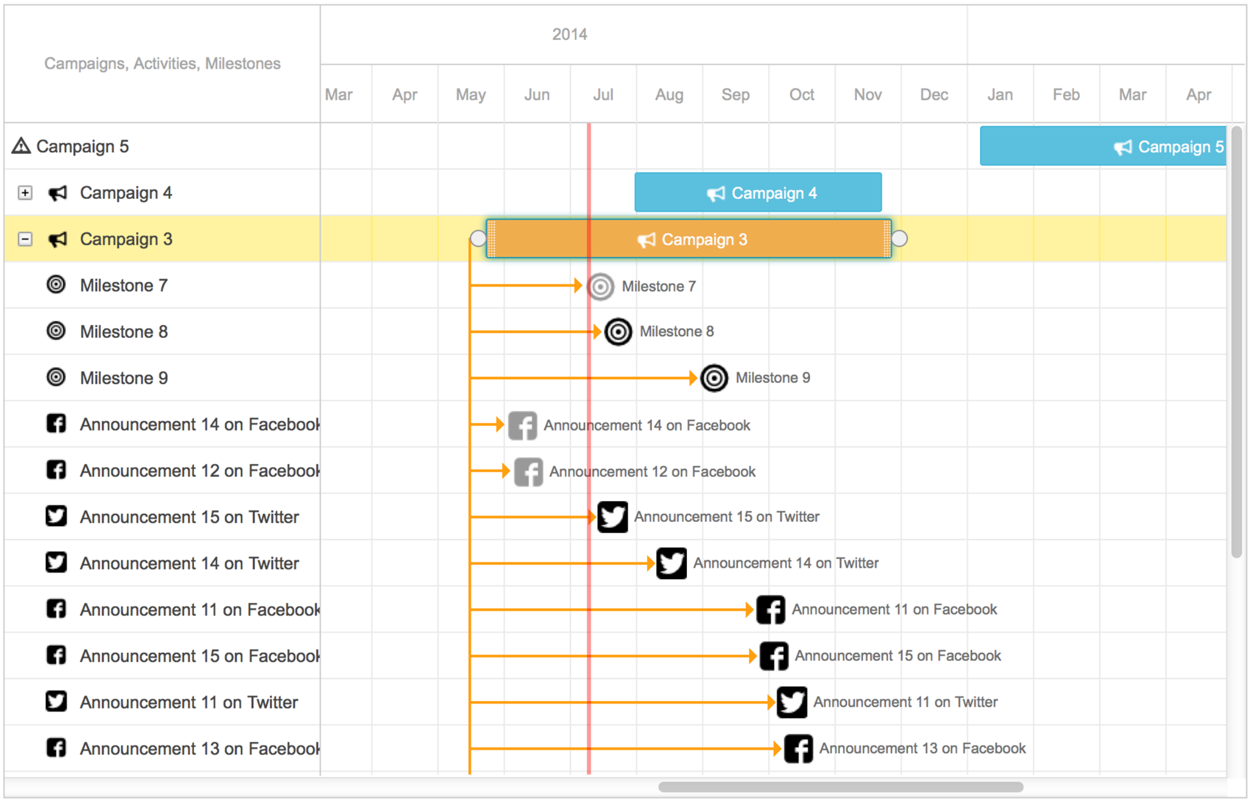
End

Assignee

Select responsible person

Save

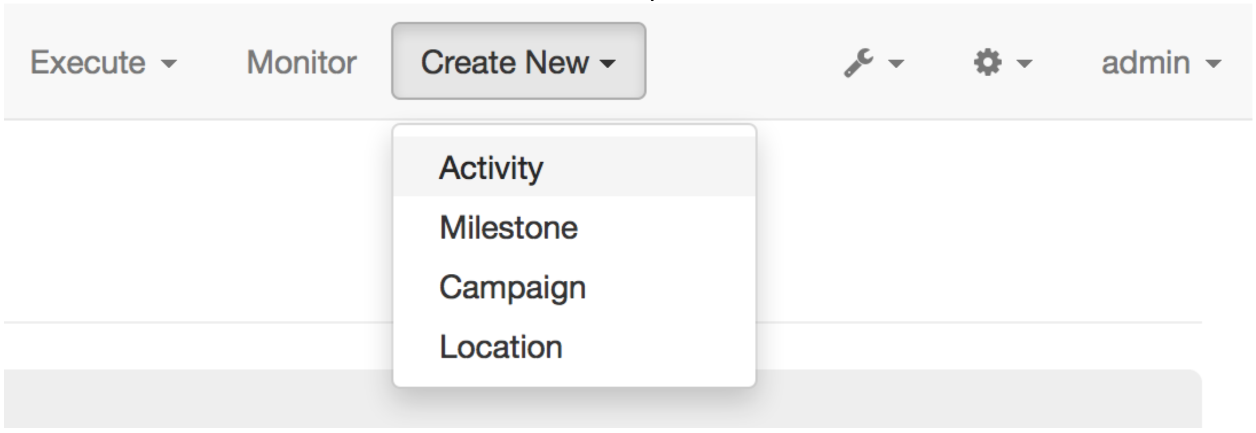
If you now click *Plan* in the header navigation, you will see your new Campaign in the Timeline.


Campaigns, Activities, Milestones
2014
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
Campaign 5
Campaign 4
Campaign 3
Milestone 7
Milestone 8
Milestone 9
Announcement 14 on Facebook
Announcement 12 on Facebook
Announcement 15 on Twitter
Announcement 14 on Twitter
Announcement 11 on Facebook
Announcement 15 on Facebook
Announcement 11 on Twitter
Announcement 13 on Facebook

5.1.3 3. Create an Activity

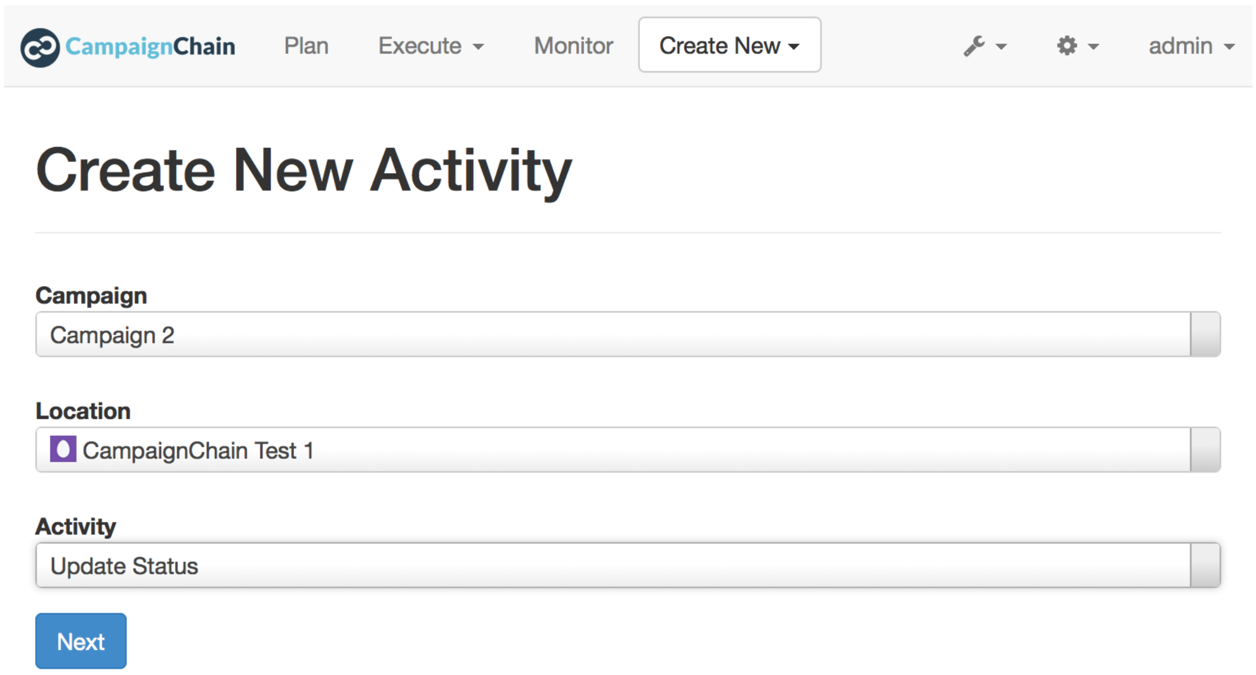
Now you are ready to create your fist Activity, which will be posting a status update on Twitter.

Click the *Create New* button in the header and choose *Activity*.



In the next screen, select your newly created Campaign and in the *Location* field, pick the Twitter user stream you just connected to.

Once you have selected the Location, a new field will appear which allows you to select the Activity you want to perform within the Location. Here, choose *Update Status* and click *Next*.



Create New Activity

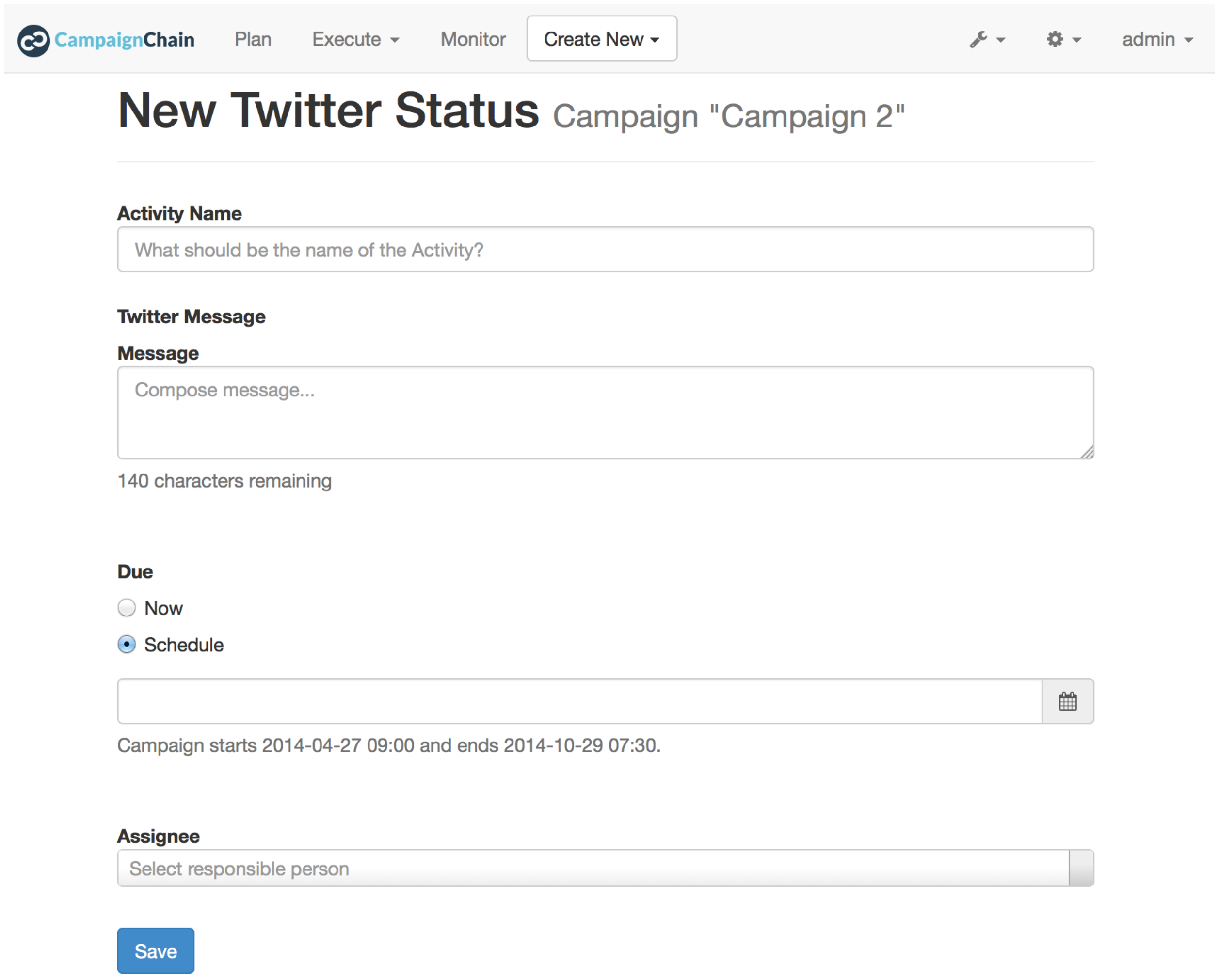
Campaign
Campaign 2

Location
CampaignChain Test 1

Activity
Update Status

A form will appear and prompt you to insert the following data:

- *Activity Name*: An arbitrary name that will be used within CampaignChain. For example, "Initial announcement".
- *Twitter Message*: This is the text that will appear on Twitter, e.g. "Try our new product, it's awesome: http://www.example.com/newproduct"
- *Due*: Here, you can schedule the tweet to be posted at a specific date and time.
- *Assignee*: Define who is responsible for taking care of this Tweet.



New Twitter Status Campaign "Campaign 2"

Activity Name

What should be the name of the Activity?

Twitter Message

Message

Compose message...

140 characters remaining

Due

Now [✓ Schedule]

Campaign starts 2014-04-27 09:00 and ends 2014-10-29 07:30.

Assignee

Select responsible person

Save

That's it! If you now click *Plan* again, you will see the new Activity as part of your new Campaign.

Part VI

Packages

Find the documentation for single packages included in CampaignChain below.

CHAPTER 6

Packages

This part of the documentation covers some of the modules included in or available for CampaignChain.

6.1 CampaignChain Twitter

This documentation covers the standard Twitter[75] functionality available by default in the CampaignChain Community Edition.

6.1.1 Features

- Connect a Twitter stream to CampaignChain with username and password
- Assign a Twitter message to a campaign
- Include up to 4 pictures in a Tweet
- Schedule a Tweet to automatically post it on Twitter at a given date and time
- Assign a responsible person to a Tweet
- Track the customer journey from an included link to the Location of a connected Channel (e.g. a Website page)
- Once a Tweet was posted, regularly collect statistical data about likes and re-tweets
- Ensure that the same Twitter content won't be posted within 24 hours to avoid a duplicate status message error

6.1.2 Packages

The standard Twitter functionality is being provided by CampaignChain, Inc. through these packages:

- campaignchain/channel-twitter[76]
- campaignchain/location-twitter[77]
- campaignchain/activity-twitter[78]
- campaignchain/operation-twitter[79]

[75] https://twitter.com/
[76] https://github.com/CampaignChain/channel-twitter
[77] https://github.com/CampaignChain/location-twitter
[78] https://github.com/CampaignChain/activity-twitter
[79] https://github.com/CampaignChain/operation-twitter

Although these modules are based on Symfony bundles[80], they do not work independently of CampaignChain.

6.1.3 Installation

The above modules are included in the Community Edition by default.

6.1.4 Configuration

Before you can post on Twitter from within CampaignChain, an OAuth app must be created in Twitter:

1. Go to https://dev.twitter.com/apps and create a new application.
2. Fill out any required fields such as the application name and description.
3. Put your website domain in the Website field.
4. Provide the host name of your CampaignChain instance as the Callback URL (e.g. http://mydomain.com). Make sure that you specify the correct scheme (*http* or *https*). If you omitted the Callback URL, the app won't work with CampaignChain.
5. The Callback URL's parts must be identical with the *router.request_context.host* and *router.request_context.scheme* parameters defined in the *app/config/parameters.yml* configuration file.
6. Once you have registered the app, *connect to a Location* (page 103) choosing Twitter as the Channel.
7. When the *Provide Application Credentials* screen comes up, go back to https://dev.twitter.com/apps, select your app and visit the *Keys and Access Tokens* tab to copy and paste the *Consumer Key* and the *Consumer Secret* and insert it in the *Provide Application Credentials* form.

6.1.5 Issues

Please post reports, questions, suggestions, etc. at https://github.com/CampaignChain/campaignchain/issues.

6.2 CampaignChain Facebook

This documentation covers the standard Facebook[81] functionality available by default in the CampaignChain Community Edition.

6.2.1 Features

- Connect a Facebook user or page stream to CampaignChain with username and password
- Assign a Facebook message to a campaign
- Include 1 picture in the Facebook message
- Schedule a message to automatically post it on Facebook at a given date and time
- Assign a responsible person to a Facebook message
- Track the customer journey from an included link to the Location of a connected Channel (e.g. a Website page)
- Once a Facebook update was posted, regularly collect statistical data about likes and comments

[80] http://symfony.com/doc/current/bundles.html
[81] https://www.facebook.com

- Ensure that the same Facebook content won't be posted within 24 hours to avoid a duplicate status message error

6.2.2 Packages

The standard Facebook functionality is being provided by CampaignChain, Inc. through these packages:

- campaignchain/channel-facebook[82]
- campaignchain/location-facebook[83]
- campaignchain/activity-facebook[84]
- campaignchain/operation-facebook[85]

Although these modules are based on Symfony bundles[86], they do not work independently of CampaignChain.

6.2.3 Installation

The above modules are included in the Community Edition by default.

6.2.4 Configuration

Before you can post on Facebook from within CampaignChain, an OAuth app must be created in Facebook:

1. Go to https://developers.facebook.com/apps and create a new application by clicking *Create New App*.
2. Fill out any required fields such as the application name and description.
3. Put your website domain in the *Site Url* field.
4. Provide the host name of your CampaignChain instance as the Callback URL (e.g. http://mydomain.com). Make sure that you specify the correct scheme (*http* or *https*).
5. The Callback URL's parts must be identical with the *router.request_context.host* and *router.request_context.scheme* parameters defined in the *app/config/parameters.yml* configuration file.
6. Once you have registered the app, *connect to a Location* (page 103) choosing Facebook as the Channel.
7. When the *Provide Application Credentials* screen comes up, go back to https://developers.facebook.com/apps, select your app and visit the *Settings* page to copy and paste the *App ID* and the *App Secret* and insert it in the *Provide Application Credentials* form.

Should you want CampaignChain to post to Facebook streams which are *not* owned by the same Facebook account which owns your OAuth App, then you must have your OAuth App approved by Facebook. To do so, click on *App Review* on the left menu of your Facebook OAuth App.

The following Facebook login permissions[87] will have to be approved:

- business_management
- email
- manage_pages

[82] https://github.com/CampaignChain/channel-facebook
[83] https://github.com/CampaignChain/location-facebook
[84] https://github.com/CampaignChain/activity-facebook
[85] https://github.com/CampaignChain/operation-facebook
[86] http://symfony.com/doc/current/bundles.html
[87] https://developers.facebook.com/docs/facebook-login/permissions

- public_profile
- publish_actions
- publish_pages
- read_insights
- user_friends

6.2.5 Issues

Please post reports, questions, suggestions, etc. at https://github.com/CampaignChain/campaignchain/issues.

6.3 CampaignChain Linkedin

This documentation covers the standard Linkedin[88] functionality available by default in the CampaignChain Community Edition.

6.3.1 Features

- Connect a Linkedin user or company page to CampaignChain with username and password
- Assign a Linkedin message to a campaign
- Include 1 picture in a Linkedin status update
- Schedule a Linkedin message to automatically post it on Linkedin at a given date and time
- Assign a responsible person to a Linkedin update
- Track the customer journey from an included link to the Location of a connected Channel (e.g. a Website page)

Note: To let CampaignChain regularly collect statistical data about likes and comments of Linkedin messages, you would have to apply to become a Linkedin partner[89]. Otherwise, CampaignChain won't have access to the respective REST API endpoints.

6.3.2 Packages

The standard Linkedin functionality is being provided by CampaignChain, Inc. through these packages:

- campaignchain/channel-linkedin[90]
- campaignchain/location-linkedin[91]
- campaignchain/activity-linkedin[92]
- campaignchain/operation-linkedin[93]

[88] https://www.linkedin.com
[89] https://developer.linkedin.com/partner-programs
[90] https://github.com/CampaignChain/channel-linkedin
[91] https://github.com/CampaignChain/location-linkedin
[92] https://github.com/CampaignChain/activity-linkedin
[93] https://github.com/CampaignChain/operation-linkedin

Although these modules are based on Symfony bundles[94], they do not work independently of CampaignChain.

6.3.3 Installation

The above modules are included in the Community Edition by default.

6.3.4 Configuration

Before you can post on Linkedin from within CampaignChain, an OAuth app must be created in Linkedin:

1. Go to https://www.linkedin.com/secure/developer (or https://www.linkedin.com/secure/developer?newapp=) and create a new application.
2. Fill out any required fields such as the application name and description.
3. Once you created the App, go to the *Authentication* menu entry.
4. Under *Default Application Permissions*, enable all permissions, i.e. *r_basicprofile*, *r_emailaddress*, *rw_company_admin*, *w_share*.
5. Under *OAuth 2.0*, provide the host name of your CampaignChain instance as the *Authorized Redirect URL* (e.g. http://mydomain.com). Make sure that you specify the correct scheme (*http* or *https*).
6. The Authorized Redirect URL's parts must be identical with the *router.request_context.host* and *router.request_context.scheme* parameters defined in the *app/config/parameters.yml* configuration file.
7. Once you saved all changes on the *Authentication* page, click on the *JavaScript* menu entry. There, add the host name of your CampaignChain installation as a *Valid SDK Domain*.
8. After you saved the changes to the *JavaScript* page, *connect to a Location* (page 103) choosing Linkedin as the Channel.
9. When the *Provide Application Credentials* screen comes up, go back to the *Authentication* page of your Linkedin App to copy and paste the *Client ID* and the *Client Secret* and insert it in the *Provide Application Credentials* form.

6.3.5 Issues

Please post reports, questions, suggestions, etc. at https://github.com/CampaignChain/campaignchain/issues.

6.4 CampaignChain Google Analytics

This documentation covers the standard Google Analytics[95] functionality available by default in the CampaignChain Community Edition.

6.4.1 Features

- Connect a Google Analytics account to CampaignChain with username and password
- Pull data from Google Analytics to relate the Website traffic during a campaign statistics of other Channels (e.g. likes of all Facebook posts).

[94] http://symfony.com/doc/current/bundles.html
[95] https://www.google.com/analytics/

6.4.2 Packages

The standard Google Analytics functionality is being provided by CampaignChain, Inc. through these packages:

- campaignchain/channel-google-analytics[96]
- campaignchain/location-google-analytics[97]

Although these modules are based on Symfony bundles[98], they do not work independently of CampaignChain.

6.4.3 Installation

The above modules are included in the Community Edition by default.

6.4.4 Configuration

Before you can use Google Analytics data within CampaignChain, an OAuth app must be created in Google:

1. Go to https://code.google.com/apis/console/ and create a new project if you did not do so yet.
2. Go to https://console.developers.google.com/apis/library, search for these APIs and enable them: *Analytics API*, *Google+ API*
3. Go to https://console.developers.google.com/apis/credentials, click on *Create credentials* and select *OAuth client ID*.
4. On the next page, select *Web application*.
5. Under *Authorized JavaScript origins*, enter the domain of your CampaignChain installation, e.g. http://mycampaignchain.com.
6. Provide this URL as an *Authorized redirect URI*: http://mycampaignchain.com/security/authentication/client/oauth/login?hauth.do
   Make sure that you specify the correct scheme (*http* or *https*).
7. The Callback URL's parts must be identical with the *router.request_context.host* and *router.request_context.scheme* parameters defined in the *app/config/parameters.yml* configuration file.
8. Once you have created the app, *connect to a Location* (page 103) choosing Google Analytics as the Channel.
9. When the *Provide Application Credentials* screen comes up, go back to https://console.developers.google.com/apis/credentials, select your app and copy and paste the *Client ID* and the *Client Secret* and insert it in the *Provide Application Credentials* form.

6.4.5 Issues

Please post reports, questions, suggestions, etc. at https://github.com/CampaignChain/campaignchain/issues.

6.5 CampaignChain MailChimp

This documentation covers the standard MailChimp[99] functionality available by default in the CampaignChain Community Edition.

[96] https://github.com/CampaignChain/channel-google-analytics
[97] https://github.com/CampaignChain/location-google-analytics
[98] http://symfony.com/doc/current/bundles.html
[99] http://mailchimp.com

6.5.1 Features

- Connect a MailChimp account to CampaignChain with username and password
- Assign a MailChimp email campaign into a CampaignChain campaign
- Schedule a MailChimp newsletter to be distributed by MailChimp at a given date and time
- Assign a responsible person to a MailChimp newsletter within CampaignChain
- Track the customer journey from an included link to the Location of a connected Channel (e.g. a Website page)
- Once a newsletter was posted, regularly collect statistical data about opens, clicks, etc.

6.5.2 Packages

The standard MailChimp functionality is being provided by CampaignChain, Inc. through these packages:

- campaignchain/channel-mailchimp[100]
- campaignchain/location-mailchimp[101]
- campaignchain/activity-mailchimp[102]
- campaignchain/operation-mailchimp[103]

Although these modules are based on Symfony bundles[104], they do not work independently of CampaignChain.

6.5.3 Installation

The above modules are included in the Community Edition by default.

6.5.4 Configuration

Before you can manage a MailChimp email campaign from within CampaignChain, an OAuth app must be created in MailChimp:

1. Go to https://admin.mailchimp.com/account/oauth2/ and register an app.
2. Fill out any required fields such as the application name and description.
3. Provide the host name of your CampaignChain instance as the *Redirect URL* (e.g. http://mydomain.com). Make sure that you specify the correct scheme (*http* or *https*).
4. The Callback URL's parts must be identical with the *router.request_context.host* and *router.request_context.scheme* parameters defined in the *app/config/parameters.yml* configuration file.
5. Once you have registered the app, *connect to a Location* (page 103) choosing MailChimp as the Channel.
6. When the *Provide Application Credentials* screen comes up, go back to https://admin.mailchimp.com/account/oauth2/, select your app and copy and paste the *Client ID* and the *Client Secret* and insert it in the *Provide Application Credentials* form.

[100] https://github.com/CampaignChain/channel-mailchimp
[101] https://github.com/CampaignChain/location-mailchimp
[102] https://github.com/CampaignChain/activity-mailchimp
[103] https://github.com/CampaignChain/operation-mailchimp
[104] http://symfony.com/doc/current/bundles.html

6.5.5 Issues

Please post reports, questions, suggestions, etc. at https://github.com/CampaignChain/campaignchain/issues.

6.6 CampaignChain GoToWebinar

This documentation covers the standard GoToWebinar[105] functionality available by default in the CampaignChain Community Edition.

6.6.1 Features

- Connect a GoToWebinar account to CampaignChain with username and password
- Assign a GoToWebinar Webinar to a campaign within CampaignChain
- (Re-)schedule a Webinar at a given date and time
- Within CampaignChain, assign a responsible person to a Webinar
- Collect statistics about registrations prior to a Webinar and afterwards the number of participants

6.6.2 Packages

The standard GoToWebinar functionality is being provided by CampaignChain, Inc. through these packages:

- campaignchain/channel-citrix[106]
- campaignchain/location-citrix[107]
- campaignchain/activity-gotowebinar[108]
- campaignchain/operation-gotowebinar[109]

Although these modules are based on Symfony bundles[110], they do not work independently of CampaignChain.

6.6.3 Installation

The above modules are included in the Community Edition by default.

6.6.4 Configuration

Before you can manage a GoToWebinar Webinar from within CampaignChain, an OAuth app must be created in Citrix (the company behind GoToWebinar):

1. Go to https://developer.citrixonline.com/user/me/apps and add a new app.
2. Fill out any required fields such as the application name and description and select *GoToWebinar* as the *Product API*.

[105] https://www.gotomeeting.com/webinar
[106] https://github.com/CampaignChain/channel-citrix
[107] https://github.com/CampaignChain/location-citrix
[108] https://github.com/CampaignChain/activity-gotowebinar
[109] https://github.com/CampaignChain/operation-gotowebinar
[110] http://symfony.com/doc/current/bundles.html

3. Provide the host name of your CampaignChain instance as the *Application URL* (e.g. http://mydomain.com). Make sure that you specify the correct scheme (*http* or *https*).
4. The Callback URL's parts must be identical with the *router.request_context.host* and *router.request_context.scheme* parameters defined in the *app/config/parameters.yml* configuration file.
5. Once you have added the app, *connect to a Location* (page 103) choosing GoToWebinar as the Channel.
6. When the *Provide Application Credentials* screen comes up, go back to https://developer.citrixonline.com/user/me/apps, select your app and visit the *Keys* tab to copy and paste the *Consumer Key* and the *Consumer Secret* and insert it in the *Provide Application Credentials* form.

Note: If you already have an account for http://developer.citrixonline.com, you would still have to create a new separate account for https://www.gotomeeting.com/webinar should you not already have one.

6.6.5 Issues

Please post reports, questions, suggestions, etc. at https://github.com/CampaignChain/campaignchain/issues.

6.7 CampaignChain SlideShare

This documentation covers the standard SlideShare[111] functionality available by default in the CampaignChain Community Edition.

6.7.1 Features

- Connect a SlideShare account to CampaignChain with username and password
- Assign a SlideShare slide deck to a campaign
- Schedule to make an already uploaded and private slide deck publicly available on SlideShare at a given date and time
- Within CampaignChain, assign a responsible person to a slide deck
- Once a slide deck was made public, regularly collect statistical data about Views, Favorites, Downloads, Comments

6.7.2 Packages

The standard SlideShare functionality is being provided by CampaignChain, Inc. through these packages:

- campaignchain/channel-slideshare[112]
- campaignchain/location-slideshare[113]
- campaignchain/activity-slideshare[114]
- campaignchain/operation-slideshare[115]

[111] http://www.slideshare.net
[112] https://github.com/CampaignChain/channel-slideshare
[113] https://github.com/CampaignChain/location-slideshare
[114] https://github.com/CampaignChain/activity-slideshare
[115] https://github.com/CampaignChain/operation-slideshare

Although these modules are based on Symfony bundles[116], they do not work independently of CampaignChain.

6.7.3 Installation

The above modules are included in the Community Edition by default.

6.7.4 Configuration

Before you can post on SlideShare from within CampaignChain, an OAuth app must be created in SlideShare:

1. Apply for an API key at http://www.slideshare.net/developers/applyforapi.
2. Fill out any required fields such as the application name and description.
3. Once you received the API key via email, *connect to a Location* (page 103) choosing SlideShare as the Channel.
4. When the *Provide Application Credentials* screen comes up, insert the API key in the *Provide Application Credentials* form.

Warning: If a SlideShare user was newly created with a Linkedin account, then please follow this workaround to make sure CampaignChain can access the SlideShare REST API: Explicitly set the SlideShare password by changing it from the Linkedin password to a new one at https://www.slideshare.net/ordnas/edit_manage_accounts. Only then, the new password for SlideShare will allow CampaignChain to access the SlideShare REST API.
Also, when connecting the SlideShare account to CampaignChain, please be aware of the following: The SlideShare username is not the same as the Linkedin username. For Linkedin, the username might be identical with the account's email, while the SlideShare username can be found in the SlideShare profile URL. For example, *amariki* in http://www.slideshare.net/amariki.

6.7.5 Issues

Please post reports, questions, suggestions, etc. at https://github.com/CampaignChain/campaignchain/issues.

6.8 CampaignChain Update

The update feature is a package that is included in CampaignChain by default and can also be used independently from CampaignChain as a regular Symfony bundle.

The update package offers you the ability to include update routines in multiple separate CampaignChain modules or Symfony bundles. Essentially, it extends the DoctrineMigrationsBundle[117].

6.8.1 Installation

CampaignChain

The ‘‘campaignchain/update‘ Composer package is included in any CampaignChain distribution by default. It will be installed automatically.

[116] http://symfony.com/doc/current/bundles.html
[117] https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html

Symfony

First, install the package with

```
$ composer require campaignchain/update "dev-master"
```

If everything worked, the update package can now be found at `vendor/campaignchain/update`.

Finally, be sure to enable the bundle in `AppKernel.php` by including the following:

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = array(
        //...
        new CampaignChain\UpdateBundle\CampaignChainUpdateBundle(),
    );
}
```

6.8.2 Configuration

You can configure the default package name and the target directories for the schema and update files in your `config.yml`. The examples below are the default values:

```
campaignchain_update:
    diff_package: campaignchain/core
    bundle:
        schema_dir: /Resources/update/schema
        data_dir: /Resources/update/data
```

6.8.3 Usage

Learn how to use the update package in the tutorial on creating update routines (page 80).

6.8.4 Issues

Please post reports, questions, suggestions, etc. at https://github.com/CampaignChain/update/issues.

Part VII

Contributing to Documentation

Learn how to contribute to this documentation, about its license, and more.

CHAPTER 7

About this Documentation

7.1 Documentation Standards

7.1.1 Symfony Documentation Standards

CampaignChain documentation follows the documentation standards of the Symfony project[118].

7.1.2 Additional Conventions

CampaignChain documentation also follows these additional conventions.

1. When referring to an CampaignChain entity, you should start the entity name with a capital letter. For example, you should use "Channel module" instead of "channel module". This makes it clear that you are referring to a specific entity and not a generic channel, location, activity, operation, hook or milestone concept.
2. All file names and directory paths are italicized. You should always include a trailing slash on directory paths. For example, *public/images/icons/16x16/linkedin.png* refers to a file name and *your-project/src/* refers to a directory path.
3. When referring to key or parameters names in configuration files, those names should be italicized. However, when referring to the corresponding values, you should enclose those values in single quotes. For example, the parameter *foo* and the value 'bar'.
4. CampaignChain documentation favors using :: shorthand for PHP code blocks and code-block:: syntax for other code blocks. For example:

```
::
  <?php
  // PHP code block
  $wizard = $this->get('campaignchain.core.channel.wizard');
  $wizard->setName($profile->displayName);
  $wizard->addLocation($location->getIdentifier(), $location);
  $channel = $wizard->persist();
  $wizard->end();

.. code-block::yaml

  # YAML code block
  acme_campaignchain_channel_linkedin_create:
```

118 http://symfony.com/doc/current/contributing/documentation/standards.html

```
        pattern:  /channel/linkedin/create
        defaults: { _controller: AcmeCampaignChainChannelLinkedInBundle:LinkedIn:create }
```

5. CampaignChain documentation calls out pieces of key information using note:: and tip:: specific admonitions. For example:

```
    .. note::
       This is something important.
```

6. Images should be created in LibreOffice Draw[119]. Please provide the original *.odg* files in the */images/* directory along with the related PNG file.

7.2 Documentation License

The CampaignChain documentation is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License[120].

You are free:

- to *Share* — to copy, distribute and transmit the work;
- to *Remix* — to adapt the work.

Under the following conditions:

- *Attribution* — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work);
- *Share Alike* — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

With the understanding that:

- *Waiver* — Any of the above conditions can be waived if you get permission from the copyright holder;
- *Public Domain* — Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license;
- *Other Rights* — In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
  - The author's moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.
- *Notice* — For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

This is a human-readable summary of the Legal Code (the full license)[121].

[119] http://www.libreoffice.org/discover/draw/
[120] http://creativecommons.org/licenses/by-sa/3.0/
[121] http://creativecommons.org/licenses/by-sa/3.0/legalcode

Part VIII

Miscellanea

CHAPTER 8

Glossary

Activity Every location allows one or more Activities which can be undertaken. For example, creating a new post is an example of an Activity for a blog Channel.

Campaign Campaigns are a series of marketing operations utilizing one or more communication channels. Campaigns have Milestones and Activities.

Channel Campaigns use online channels, which are the pathways by which campaign content reaches its audience. Common examples of Channels include websites, blogs and social networks like Facebook and LinkedIn.

CTA Call to Action

GUI Graphical User Interface

Hook A Hook is a reusable component that provide common functionality and can be used across modules to configure Campaigns, Milestones, Channels, Locations, Activities and Operations.

Location Every Channel includes one or more Locations, which allow granular publishing of campaign content. For example, for a Twitter Channel, the Twitter stream is a Location.

Milestone Milestones are key events or reference points during a Campaign. For example, the campaign go-live date could be a Milestone, and a press tour could be a second Milestone.

Module A module is a pre-packaged set of functionality. In CampaignChain, modules are developed as Symfony bundles, with additional configuration that allows CampaignChain to load them into its system.

Operation Every Activity must always have at least one Operation. For example, posting on Twitter is one Activity which equals the Operation.

Tracking ID Each Call to Action has a unique Tracking ID assigned by CampaignChain which enables CampaignChain to match a URL clicked in a Channel to a Location specified inside CampaignChain.