



Developer

[API Home](#)

Your Services

No services yet!

[+ Create a new service.](#)

Full API Reference

Below you'll find in-depth textual descriptions of terminology around our developer interface. You probably **do not** want to start by reading this all the way through. Rather, use it as a reference when you run into trouble [when creating your service](#).

Service Summary ¶

The Service Summary page is the screen you are presented with immediately after creating a new service. Fields include `Title`, `Description`, and `Image`.

- `Title` should be the short human-readable version of your service. Try to keep under 16 characters. It shows up in many places across the site.
- `Description` is a short few-sentence description of what your service primary does. It shows up on our [Zapbook](#).
- `Image` is used everywhere across our site. For this reason, please supply a large, square, transparent background, and tightly cropped PNG image. We will scale and size the image appropriately.

Remember, a little polish here goes a long way...

Documentation

[Overview](#)[The Perfect API](#)[Examples](#)[Campfire \(JSON & API Keys\)](#)[GitHub \(JSON & OAuthV2\)](#)

Actions ¶

Actions answer the question: *What should my users be able to create via Zapier?* They are things like:

- Create ToDo Task (EG: Basecamp)
- Create Chat Message (EG: Campfire or Hipchat)

[Hacker News \(HTML Scraper\)](#)[Scripting API](#)[About](#)[Specification](#)[Examples](#)[Debugging](#)[Reference](#)[Service Summary](#)[Actions](#)[Action Fields](#)[Action Fields \(Custom\)](#)[Activation](#)[Auth Fields](#)[Auth Mappings](#)[Authentication](#)[Datetimes](#)[Deduplication](#)[Logged Requests](#)[Migration](#)[OAuth](#)[Scripting](#)[Triggers](#)[Trigger Data](#)[Trigger Sample Results](#)[Trigger Fields](#)[Polling](#)[Prefill](#)[Status](#)[Variable Syntax](#)[Static Webhooks](#)[Subscription Webhooks](#)

- [Create Issue](#) (EG: [GitHub](#) or [Pivotal Tracker](#))

You can think of Actions as `POST`s or writes, or the creation of a resource.

You can define your actions via your services's dashboard. When you create a new action, you'll be prompted with multiple options:

Label

This is a human readable label a user would see when browsing the directory and adding your service. Make it short but descriptive.

Example: `Create Issue`, `Create Ticket` or `Create Note`

Key

This is a field only really used internally for both [prefill](#) and [scripting](#) references. Needs to be at least 2 characters long, start with an alpha, and only contain `a-z`, `A-Z`, `0-9` or `_`.

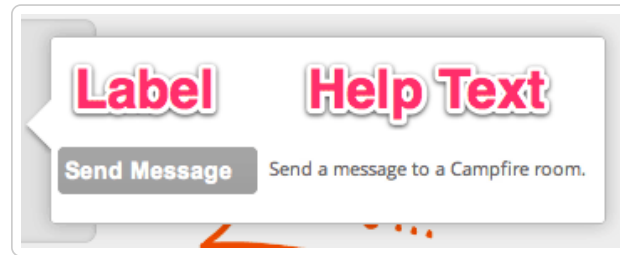
Example: `create_issue`, `ticket` or `newNote`

Help Text

This some human readable explanatory text, usually something that clarifies what the action does.

Example: `Creates a new issue in a selected repository.`

Demo: *the user will see **Label** and **Help Text** like below:*



URL Route

Define the URL route we'd hit to get a list of results, optionally using our [variable syntax](#) where [auth fields](#) and [action fields](#) will be injected.

Example: `http://api.example.com/v2/clients.json` or
`http://{{account}}.example.com/api/v1/projects.json`

Custom Action Fields URL Route

This allows you to dynamically define action fields that are user set (IE: custom fields). They get passed into the POST'ed JSON just like normal action fields (or into our [Scripting API](#)).

Example: `http://api.example.com/v2/fields.json` or
`http://{{account}}.example.com/api/v1/fields.json`

Important

Usually you'll want to leave this checked, but if you don't we'll hide that action behind an "uncommon" link when a user selects their action. Mainly this is helpful for removing actions that are there for breadth but are rarely used.

Action Fields ¶

Action Fields answer the question: *What details can a user provide when creating an Action?* These are things like:

- Title (EG: Note Title in Evernote)
- Description (EG: Issue Description in Github)
- Parent Object (span relationships via [prefill](#))
 - Repo for an Issue (EG: Github)
 - Notebook for a Note (EG: Evernote)
- Message Body (EG: Chat Body in Campfire or Hipchat)

Note: each action should have at least **one** action field. It really makes no sense to POST nothing to an endpoint.

Also: you can dynamically load custom action fields by inspecting a custom field endpoint of your own.

When you create a new action field, you'll be offered up a myriad of options which are explained below:

Key

A key for you and your API's consumption. This is available for [variable syntax](#) in the Action URL field as well as being added to the POSTed JSON (optional). Needs to be at least 2 characters long, start with an alpha, and only contain `a-z`, `A-Z`, `0-9` or `_`.

We'll take double underscores and convert them to nested dictionaries before we POST JSON.

Example: `room` or `project__title` (converts to `{"project": {"title": "some value"}}`)

Label

A human readable Label shown in the UI as a user works to complete an Action.

Example: `Room` or `Title`

The diagram shows a form field with three main components highlighted by red arrows and labels:

- Label:** Points to the top text input field.
- Help Text:** Points to the text "This is Help Text" below the label.
- Default:** Points to the "Default Value" text input field.

Other visible elements include the text "This is a Label (required)" above the help text, a green checkmark to the right of the default value field, and a "Continue" button at the bottom right.

Help Text

Human readable description of an action field, useful for describing some detail you couldn't list in the Label.

Example: or

Default

A default value that is preloaded in the form field as a user works to create the action.

Type

The type we will try to coerce to on the backend. Fails silently to ensure that tasks aren't dropped on coercion errors.

Required

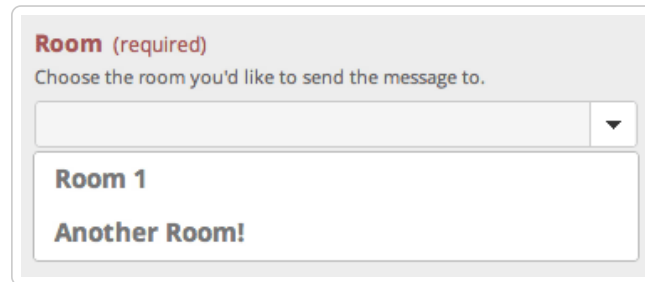
If checked a user will not be able to continue without entering some value.

Prefill

Use an existing [Trigger](#) to load in values for selection, using the machine readable value your API consumes (like `id` or `hash`) while showing a human readable version to the user (like `name` or `itemName`).

Refer to our [prefill docs](#) for a more in depth explanation.

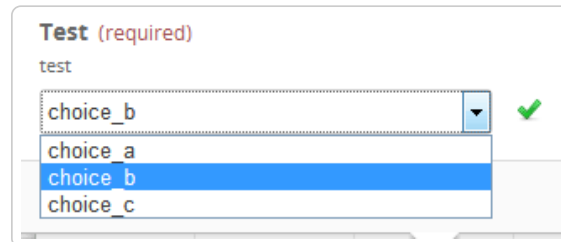
Example: `TRIGGERKEY.id.name` or `TRIGGERKEY.hash.itemName`



Choices

A comma separated string that will be turned into a select field for limiting the choices a user can provide to an action field.

Example: `choice_a,choice_b,choice_c` or `Yesterday, Today, Tomorrow`



Send in POST by default?

If checked, we'll include the user provided value in the POSTed JSON on the provided key (or nested key if `__` are present). If you utilize the `ACTIONKEY_pre_post` method via the [Scripting API](#) you can get complete control over the output JSON beyond simple exclusion/inclusion.

Action Fields (Custom) ¶

A natural extension of normal hard coded [action fields](#) are dynamic action fields, or custom fields. These are extremely common for contacts in CRMs or ticket fields in helpdesk software. All you need to do to enable custom fields is:

- Provide a `Custom Action Fields URL Route` for the actions in question.
- Ensure the URL route returns data in the below format, or manipulate to fit with our [Scripting API](#).

```
[
  {
    "type": "unicode",
    "key": "json_key",
    "label": "Pretty Label", // optional
    "help_text": "Helps to explain things to users.", // optional
    "important": true // optional, default false
  }
]
```

Activation ¶

When you are ready to publish your service to the world, you have two options: `Invite only` or `Global`.

Draft

The default status. Visible only to you.

Invite Only

Only visible to you and the users you invite. This is perfect if you have a service on Zapier that you'd like to keep out of wider circulation. To invite users you can use our promotion tools.

Pending

The waiting period before being globally enabled.

Global

Most vendors who add themselves want to submit for global activation, as this allows you to be placed in both the Zapbook and show up in our standard list for all users to see. Once you submit, expect a response within 24 hours.

After you've been approved you'll also have full access to our promotional tools, including exclusive Zapier coupons for your users, some SEO lovin' and invitation templates.

Authentication ¶

Your API probably has some kind of authentication needs in order for us to talk to it on behalf of a user. This usually comes in one of two flavors: `API Keys` or `OAuth`.

API Keys

Typically, you'll provide your users with an API Key inside your service somewhere. Many times these are provided on the user's settings or accounts page. These keys can be given to Zapier by the user so that we may make authenticated requests to access that user's information on their behalf.

Define any data which you Zapier must always include with each request as [auth fields](#). Eg. `api_key`, `account_name`, etc.

OAuth

We support OAuth version 1a and a subset of the version 2 spec. For details, check out the guide for the version your service supports:

- [Version 1](#)
- [Version 2](#)

Auth Fields ¶

Auth fields are simply what you need from the user to get started authenticating their API calls. For example, take a dashboard set up like:

Auth Fields *(Basic Auth)*

Auth Fields are fields asked for when the user adds your app's account. Commonly `account_name` and `api_key`.

Label	Key	Required	Manage
Account Name	<code>account_name</code>	✓	edit delete
API Key	<code>api_key</code>	✓	edit delete

[+ Add a new auth field.](#) [✎ Edit your auth configuration.](#)

You can see what it will be translated too when a user tries to authenticate your service:

Add a new Campfire account ×

Give this authorization a name (optional)
This is for your use only. For example: "Joe's Basecamp Account".

OR

Account Name (required)
Your account name, also serves as your subdomain.

https:// .campfirenow.com/

API Key (required)
You can find this in your settings page of Campfire, just copy it here.

Continue

Additionally, these user provided variables will be passed in with the `bundle` for usage in the [Scripting API](#), for example, the above screenshots would translate to:

```
{
  "account_name": "myaccount",
  "api_key": "123456789"
}
```

You can use [Auth Mappings](#) to put your Auth Fields to use in Basic Auth, Headers and query strings.

Let's look at each of the settings you define for a single Auth Field.

Key

This is somewhat arbitrary but you should pick a good, short key name because you'll need to include it elsewhere in the developer interface as `{{key}}` using the [variable syntax](#). Needs to be at least 2 characters long, start with

an alpha, and only contain `a-z`, `A-Z`, `0-9` or `_`.

Example: (you may not need all of these)

- `api_key` an API Key which you give to your users somewhere inside your service. Allows Zapier to authenticate on their behalf.
- `account_name` an account name which you use to build the API Request URL (perhaps as a subdomain: `https://{{account_name}}.example.com/api/v1/`)

Label

This will show up just above the text input field as a human-readable name. You don't need to be super descriptive here; you can provide more details and information about this field in the Help Text field.

Example: API Key, Account Name

The screenshot shows a configuration window for a form field. It includes a text input field at the top, followed by a label 'This is a Label (required)' with a red arrow pointing to it from the word 'Label'. Below the label is a text area for 'This is Help Text' with a red arrow pointing to it from the words 'Help Text'. Underneath is a text input field for 'Default Value' with a red arrow pointing to it from the word 'Default'. A green checkmark is visible to the right of the 'Default Value' field. At the bottom right is a 'Continue' button. The entire interface is enclosed in a light gray border with a vertical scrollbar on the right.

Required

Pretty easy concept; most all auth fields should be required. You really shouldn't ask for non-essential information as an auth field.

Help Text

Smaller text which appears under the Label of the field. You can type longer directions here. A common pattern for Auth Field Help Text is to tell the user where in your interface they can find their API Key.

Example: The API Key for your account, you can find this by going to your Basecamp Profile page, clicking settings, then API Keys.

Default

If you specify a default value, the actual field will be pre-filled with this value when the user goes to add a new authentication for your service. It is strictly set on the front-end only. If the user clears out a Required auth field which had a default set, they will still receive an error.

Most auth fields will not have a default.

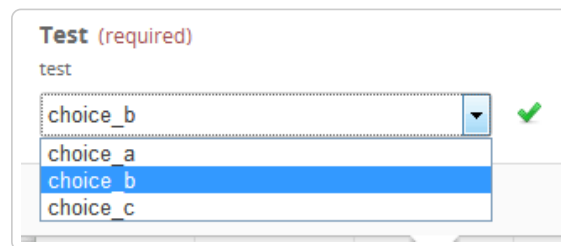
Example: `choice_b` (if you use the choices feature detailed below)

Choices

You can create a dropdown-style select box by giving us a comma-separated list of choices. Note that we don't have labels for choices so you may want to make your comma-separated list have human readable names. This works well in conjunction with default values above.

Example:

- `choice_a,choice_b,choice_c`
- `Version 1, Version 2, Version 3`

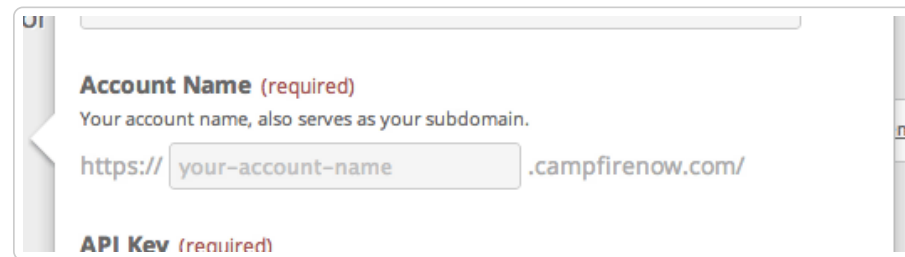


The screenshot shows a configuration window for a 'Test (required)' field. The field name is 'test'. Below it is a dropdown menu with 'choice_b' selected. The dropdown list is open, showing three options: 'choice_a', 'choice_b', and 'choice_c'. A green checkmark is visible to the right of the dropdown menu.

Subdomain Format

If you need the user to supply an account name (or similar) which gets translated into a subdomain (or similar) URL format, we recommend you take advantage of this feature. It helps guide the user with what to type in.

Example: `https://,{{input}},.campfirenow.com/` will create:

A screenshot of a web form interface. At the top, it says "Account Name (required)" in red. Below that, it says "Your account name, also serves as your subdomain." in a smaller font. There is a text input field containing "your-account-name". To the right of the input field, the text ".campfirenow.com/" is displayed. Below the input field, it says "API Key (required)" in red. The form is enclosed in a light gray border with a vertical scrollbar on the right.

Note: don't forget the comma's!

Auth Mappings ¶

Auth mappings tell us how to interpret what your user provides as usable Basic Auth, Headers and query strings. Let's use an example where there are two auth fields available to us: `account_name` and `api_key`.

Basic auth: where `api_key` is the username and password is ignored:

```
{
  "username": "{{api_key}}",
  "password": "x"
}
```

Or maybe, where `account_name` is the username and `api_key` is the password:

```
{
  "username": "{{account_name}}",
  "password": "{{api_key}}"
}
```

Which will automatically create and use a header like:

```
Authorization: Basic WkFQSUVSIExPVkVTIFlPVTPyT1hP
```

Header auth, where there are two headers called `X-Account-Name` and `X-API-Key`:

```
{
  "X-Account-Name": "{{account_name}}",
  "X-API-Key": "{{api_key}}"
}
```

Either of which will automatically create and use headers like:

```
X-Account-Name: myfancyaccount
X-API-Key: 0123456789
```

Query strings, where there is a parameter called simply `user_key`:

```
{
  "user_key": "{{api_key}}"
}
```

Which will append `?user_key=0123456789` to the end of a URL.

Datetimes ¶

For triggers, please supply any date-related fields in an RFC standard of your choosing. We do our best to parse a variety of datetime formats.

For actions, datetimes are supplied in the format in which they came from the trigger service. An example: a user has a Zap between Wufoo and Mailchimp. The Wufoo "new entry" trigger supplies a unix epoch timestamp field. If the user maps the datetime field from Wufoo into Mailchimp, it will be supplied to Mailchimp as a unix epoch timestamp.

Deduplication ¶

Note: we do the deduplication automatically for you, but you must be mindful of how it works. The important bits:

Provide a unique `id` key.

Sort reverse-chronologically by time created.

An unfortunate artifact of [polling](#) for new data is that we must deduplicate the results we get back from your API. After all, we don't want to trigger an action multiple times when a newly created item in your API exists in multiple distinct polls.

For example, say your endpoint returns a list of to-dos:

```
[
  {
    "id": 7,
    "created": "Mon, 25 Jun 2012 16:41:54 -0400",
    "list_id": 1,
    "description": "integrate our api with zapier",
    "complete": false
  },
  {
    "id": 6,
    "created": "Mon, 25 Jun 2012 16:41:45 -0400",
```

```
"list_id": 1,  
"description": "get published in zapier library",  
"complete": false  
}  
]
```

Every time a new Zap is created or unpaused, we make a call to your API to populate our deduplication mechanism. We will cache and store each `id` field in our database. After this, we will poll at an interval (based on customer's plan) looking for changes against this cached list of `id` s.

Now let's say we created a new to-do:

```
[  
  {  
    "id": 8,  
    "created": "Mon, 25 Jun 2012 16:42:09 -0400",  
    "list_id": 1,  
    "description": "re-do our api to support webhooks",  
    "complete": false  
  },  
  {  
    "id": 7,  
    "created": "Mon, 25 Jun 2012 16:41:54 -0400",  
    "list_id": 1,  
    "description": "integrate our api with zapier",  
    "complete": false  
  },  
  {  
    "id": 6,  
    "created": "Mon, 25 Jun 2012 16:41:45 -0400",  
    "list_id": 1,  
    "description": "get published in zapier library",  
    "complete": false  
  }  
]
```

The to-do with `id` equal to `8` will be seen as a new item. That particular JSON object will then be passed through to the action in the user's Zap.

Because of the way our deduplication works, your API must be able to return results in reverse-chronological order by date created. [Wufoo is a great example](#) of flexible result ordering.

One other requirement is the `id` field should always be supplied and always be unique among objects of the same type.

Logged Requests ¶

We log the requests your service makes to aid in debugging. We collect every request, successful or not, during the development stage. As soon as your service is approved, we only log requests that have status codes of `>= 300`.

The logged request list on your service dashboard should look like:

App Requests

A list of all development requests and live failed requests (status code `>= 300`).

When	Method	Status	URL	More
8 minutes ago	POST	401	<code>https://zapier1.campfirenow.com/room/517632/speak.json</code>	details
8 minutes ago	GET	401	<code>https://zapier1.campfirenow.com/rooms.json</code>	details
9 minutes ago	POST	201	<code>https://zapier1.campfirenow.com/room/517632/speak.json</code>	details
13 minutes ago	POST	302	<code>https://zapier1.campfirenow.com/room/{room}/speak.json</code>	details
30 minutes ago	GET	200	<code>https://zapier1.campfirenow.com/rooms.json</code>	details

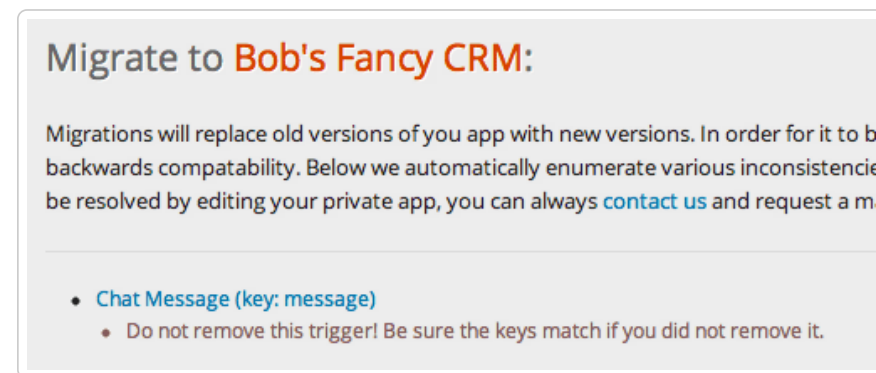
[See more app requests.](#)

Clicking details will give you lots more information, like `headers`, `query strings`, `POST` and response data.

Note: we also offer rudimentary Javascript exceptions messages for our [scripting API](#). We plan on offering more details along with tracebacks soon.

Migration ¶

Once your service is live, you'll probably want to make changes to it and add features or fix bugs. While you cannot make changes to a live service, you can **clone** a service and then make changes to the clone. Once you've tested and confirmed your changes work fine, you can migrate the service. When you start the migration, you'll get a page like so:



In the screenshot you can see an error for **Chat Message: (key: message)**, with the error text of *Do not remove this trigger! Be sure the keys match if you did not remove it.* This simply means you can't add a required field to a trigger or action that is in use.

Usually you can just make some changes and ensure there is a symmetry between old and new versions of the service. If there is a breaking changes that resolves a bug, please [contact us](#) and we'll perform a manual migration.

If you must introduce a breaking change, simply **hide** the old trigger or action and create a brand new replacement. That way your old users can continue using the previous version but new users won't see it.

OAuth v1 ¶

OAuth v1 is a popular authentication mechanism found on the web. Our official support is for v1 "revision a".

Initial Setup:

Start by selecting `OAuth v1` as your auth type, and then provide the following OAuth data in JSON:

```
{
  // required fields, you must provide these!
  "consumer_key": "7SMek7zT2LC",
  "consumer_secret": "UxGcBTSeUF4Gy5LxWbYCf7SghOtPMe0dBSEp",
  "request_token_url": "https://example.com/oauth/request",
  "authorization_url": "https://example.com/oauth/authorize",
  "access_token_url": "https://example.com/oauth/access",

  // optional field
  "access_token_placement": "header" // is the default, or "querystring"

  // sometimes an oauth endpoint returns extra information
  // this allows you to catch and store these and use in
  // our variable syntax. EG: {{user_id}}
  "requested_fields": ["user_id", "host_name"],
}
```

We automatically create the required `oauth_token` and `oauth_token_secret` auth fields for you.

1. Requesting Temporary Credentials From Your Service:

When a user goes to add a new account for your service, we make a request behind-the-scenes to ask your service for a request token and secret. Your service should respond back with a valid `oauth_token` and `oauth_token_secret`. Additionally, our request contains an `oauth_callback` URL that your service will use in a later step.

2. Redirecting the User to Your Service:

With token in hand, we redirect the user to the `authorization_url` you provided.

```
302 <authorize_url>?oauth_token=<oauth_token>
```

3. Redirecting the User Back to Zapier:

After the user clicks allow, your service redirects the user to `oauth_callback` (from Step 1), along with an `oauth_verifier` parameter in the querystring.

```
302 <oauth_callback>?oauth_verifier=<oauth_verifier>
```

4. Exchanging Temporary Credentials for Token Credentials:

We use that `oauth_verifier` to make a POST to the `access_token_url` you provided.

```
POST <access_token_url> \  
-H Accept: application/json \  
-d 'consumer_key=<consumer_key>&  
    consumer_secret=<consumer_secret>&  
    oauth_verifier=<oauth_verifier>&  
    oauth_token=<oauth_token>&  
    oauth_token=<oauth_token_secret>
```

While we ask for a JSON response, the response can be JSON, XML or querystring encoded. The values your service returns are a new (authorized) pair of `oauth_token` and `oauth_token_secret`. For example, any of the below are valid responses:

```
Content-Type: application/json  
  
{  
  "oauth_token": "1234567890abcdef",  
  "oauth_token_secret": "1234567890abcdef",  
}
```

```
Content-Type: application/xml  
  
<AnyRootElem>  
  <oauth_token>1234567890abcdef</oauth_token>
```

```
<oauth_token_secret>1234567890abcdef</oauth_token_secret>
</AnyRootElem>
```

Content-Type: text/plain

oauth_token=1234567890abcdef&oauth_token_secret=1234567890abcdef

5. Using the Token Credentials for Good:

We sign all requests using your secret and the new `oauth_token` and `oauth_token_secret`. We're talking standard HMAC-SHA1, RSA is not supported. However, you can use our [Scripting API](#) to modify request parameters (including headers) as you see fit.

Example:

Try looking at [Bitbucket's](#) documentation for an example of how OAuth v1a can be implemented.

OAuth v2 ¶

While the official OAuth v2 spec offers lots of options, the flow we respect matches [GitHub's](#) and [Facebook's](#) implementation of the `authorization_code` flow:

Initial Setup:

First, you'll need to select `OAuth v2` or `OAuth v2 (w/refresh)` as your auth type, and then provide the following OAuth data in JSON:

```
{
  // required fields, you must provide these!
  "client_id": "7SMek7zT2LC",
  "client_secret": "UxGcBTSeUF4Gy5LxWbYCF7SghOtPMe0dBSEp",
  "authorization_url": "https://example.com/oauth/authorize",
  "access_token_url": "https://example.com/oauth/access",
```

```
// optional fields & options below
"scope": "read,write,update,delete",
"refresh_token_url": "https://example.com/oauth/refresh",

// sometimes an oauth endpoint returns extra information
// this allows you to catch and store these and use in
// our variable syntax. EG: {{user_id}}
"requested_fields": ["user_id", "host_name"],
"access_token_placement": "header" // is the default, or "querystring"
}
```

We'll automatically create the required `access_token` and `refresh_token` (optional) auth fields for you.

1. Redirecting the User to Your Service:

We'll redirect the user to an `authorize_url` (provided by you), along with a querystring which contains a `redirect_uri` (provided by us), `scope` (optional, provided by you) and `client_id` (provided by you).

```
302 <authorize_url>?client_id=<client_id>&scope=<scope>&redirect_uri=https://zapier.com/dashboa
```

2. Redirecting the User Back to Zapier:

After the user clicks allow, you should redirect the user to the `redirect_uri` along with a `code` parameter in the querystring.

```
302 <redirect_uri>?code=<code>
```

We use that `code` to make a POST to the `access_token_url` (provided by you) along with the `code`, matching `redirect_uri` (provided by us), `client_id` (provided by you), and `client_secret` (provided by you) as a form encoded request.

```
POST <access_token_url> \
-H Accept: application/json \
```

```
-d 'client_id=<client_id>&
  client_secret=<client_secret>&
  code=<code>&
  grant_type=authorization_code&
  redirect_uri=https://zapier.com/dashboard/auth/oauth/return/AppIDAPI/'
```

While we ask for a JSON response, the response can be JSON, XML or querystring encoded as where the value you return is `access_token`. For example, any of the below are valid responses:

Content-Type: application/json

```
{
  "access_token": "1234567890abcdef",
  "refresh_token": "1234567890abcdef" // optional
}
```

Content-Type: application/xml

```
<AnyRootElem>
  <access_token>1234567890abcdef</access_token>
  <refresh_token>1234567890abcdef</refresh_token> <!-- optional -->
</AnyRootElem>
```

Content-Type: text/plain

```
access_token=1234567890abcdef&refresh_token=1234567890abcdef
```

3. Authorizing Requests to Your Service:

By default, we append the `access_token` to the querystring for all requests, as well as in an Authorization header, however, you can use our [Scripting API](#) to modify request parameters as you see fit (IE: place the `access_token` in a special header). This is called the `bearer` token type.

An example of the default with `access_token_placement` set to `header`:

```
GET https://api.example.com/v2/tickets.json \  
-H Authorization: Bearer 1234567890abcdef
```

And now with `access_token_placement` set to `queryset`:

```
GET https://api.example.com/v2/tickets.json?access_token=1234567890abcdef
```

4. Refreshing Tokens: (optional)

Optionally, if your service supports refresh tokens, we natively support that.

You'll need to select `OAuth v2 (w/refresh)` and provide us the `refresh_token_url` in addition to the previous requirements. When we encounter a `401` status code, we'll attempt to refresh like so:

```
POST <refresh_token_url> \  
-H Accept: application/json \  
-d 'client_id=<client_id>&  
    client_secret=<client_secret>&  
    grant_type=refresh_token&  
    refresh_token=<refresh_token>
```

We will only attempt to refresh an access token when we encounter a `401` status code from your API.

If your API does not signal expired tokens with a `401`, use our [Scripting API](#) to manually set the status code header to `401` upon conditions of your choosing.

The resulting `access_token` will be updated in our system and the previous, failed call will be attempted once more.

Some Examples:

We highly recommend looking at each of [GitHub's](#), [Facebook's](#) and [Podio's](#) (with refresh tokens) authentication documentation for some great examples of how OAuth V2 can be implemented. Our system is designed to match their industry standard implementation pattern.

Scripting ¶

See our full [Scripting API](#) document.

Triggers ¶

Triggers answer the question: *When can my users trigger an integration in Zapier?* They are things like:

- New Contact (EG: Highrise or Salesforce)
- New Email (EG: Gmail or IMAP)
- New Issue (EG: GitHub or Pivotal Tracker)

You can think of a Trigger as a `GET` or read.

Basically, they're things that happen inside your API which cause a user defined action to happen. Additionally, your triggers supply data about what happened. The user can map that data into the action.

For example, say your service has a "New Ticket Opened" trigger. We will detect when this trigger happens by [polling](#). The new data we trigger off of might look like this:

```
{
  "id": 123456,
  "owner_id": 654,
  "date_created": "Mon, 25 Jun 2012 16:41:54 -0400",
  "description": "Add our service to Zapier"
}
```

These key/values are available for users to map into the [action](#) as they see fit.

Viewing triggers for your service and creating new ones can be done through our developer interface:

Triggers *(reads/GET)*

Triggers are the read side (left side) of our zap creator.

Label	Key	Important	Pre-JS	Post-JS	Manage
Room	trigger_2	✓	⊖	⊖	edit delete

Field Types: *When triggers need extra meta data to perform.*

[+ Add a new trigger field.](#)

[+ Add a new trigger.](#)

On the overview screen shown above, you can see each trigger listed (there is only one in the screenshot). Each trigger can have [trigger fields](#). Trigger fields act like required filters for your trigger (Eg. which project to watch for new issues?).

Label, Key, and Important are defined when creating your new trigger. Pre-JS and Post-JS are relevant if you decide to use our [Scripting API](#).

When adding a new trigger, you're presented several fields. Let's define what each of those do:

Label

Human readable, short name of the trigger. Shown in various places in our interface.

Example: or

Key

This is a field only really used internally for both [prefill](#) and [scripting](#) references. Needs to be at least 2 characters long, start with an alpha, and only contain , , or .

Example: `create_issue`, `ticket` or `newEmailLabel`

Help Text

A longer description of what this trigger actually watches for.

Example: `Triggered when a new email occurs with a label of your choice.`

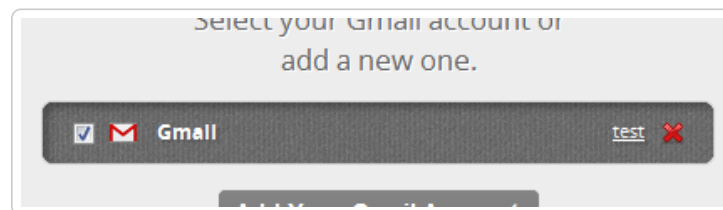
Important

Ocassionally, you'll have unimportant triggers which are solely used to drive things [like prefill fields](#). If you mark a trigger as un-important (by not checking the box), we will hide the trigger behind a link which says "Show other, unimportant triggers". The user can still pick these if they really want to but it is hidden by default.

Note: if you have no important triggers, we will not hide any of them by default.

Make Test Trigger

When a user adds an authentication for your service, they have the ability to "test" it, shown in the screenshot below. When the user clicks test, we will attempt to make a call to one of your triggers. So long as the response status code is `2XX` the test will be successful. You can only mark one trigger as the test call.



Polling: URL Route

Where in your API can [we poll](#) for this trigger? Your URL route can include [variables](#) (from [auth fields](#) and [trigger fields](#)), as well as query parameters. Query parameters are helpful for telling your API to sort things in reverse-chronological order, [a requirement we have](#).

Important: this endpoint should return a *list* of JSON objects.

Example: `https://example.com/api/v1/users/{id}/contacts.json?sort=desc&on=date_created`

Webhook: Subscription Event Name

A part of our [subscription webhooks](#), this lets you namespace the name of event that this trigger identifies with.

Webhook: Static Webhook

A part of our [static webhooks](#), this simply toggles the ability to receive webhooks and disables polling.

Webhook: Static Webhook Directions

A part of our [static webhooks](#), when static webhooks are in use this lets you define specialized directions for setup. It also supports [Markdown](#) so you can provide links or other specialized HTML formatting.

Trigger Data ¶

Trigger data is the result of a successful [Trigger](#). You can manipulate trigger data with our [scripting API](#), both before it enters our system (via the `TRIGGERKEY_post_poll` method, which returns an array of trigger data) or use it to supplement your POST requests (via the `ACTIONKEY_pre_write` method (*though this is unlikely due to the fact that you never know what trigger data you might get!*)).

An example of Trigger Data might be something like:

```
{
  "id": 123456,
  "owner_id": 654,
  "date_created": "Mon, 25 Jun 2012 16:41:54 -0400",
  "title": "Hello world!",
```

```
"description": "It's a beautiful day!"
}
```

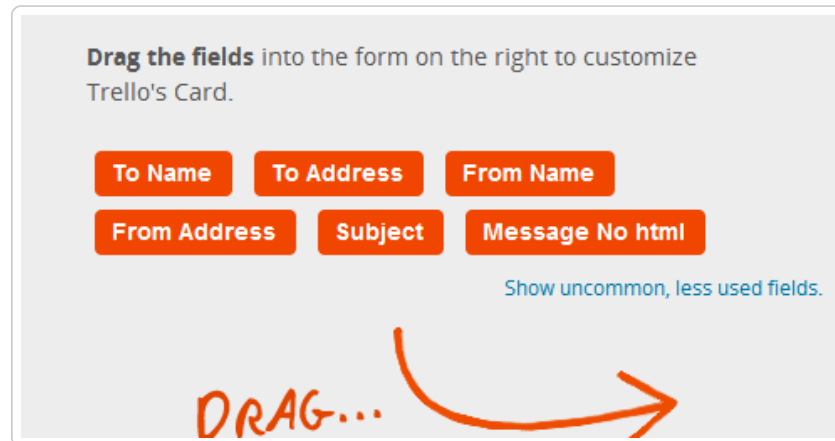
Again, if you are working on the [Action](#) side, you *never* know what trigger data you might get. It could be data from an RSS feed, or maybe a todo item. We'll apply the trigger data to the [Actions Fields](#) according to how the user wants it, so you'll likely just use that.

Trigger Sample Results ¶

While your trigger URL Route should be returning a list, we want you to paste a sample JSON dictionary of a single item in that list. We will use this sample to show the user what keys are available in your API.

Here is the sample JSON for something like a new email message, and how it shows up in our customer-facing Editor:

```
{
  "to_name": "Mike Knoop",
  "to_address": "mike@zapier.com",
  "from_name": "Bryan Helmig",
  "from_address": "bryan@zapier.com",
  "subject": "Testing out this new Zap!",
  "message": {
    "no_html": "Let me know it it works.",
    "html": "<div>Let me know if it works.</div>",
  }
}
```



By default, we can handle flat dictionaries and dictionaries within dictionaries (via our ☐ delimiter in keys). If you have more complex data structures, you can [use our scripting interface](#) to pre-parse your inbound raw data into a format we can handle (flat dictionary or nested).

Trigger Fields ¶

Trigger Fields answer the question: *How can a user filter Triggers?* Almost exclusively these are [prefill](#), but some real world examples are:

- Search Term (EG: Twitter Tweet Search)
- Label (EG: Gmail Inbox)
- Parent Object (span relationships via [prefill](#))
 - Repo for an Issue (EG: Github)
 - Notebook for a Note (EG: Evernote)

When you attempt to add a trigger field, you'll be prompted to provide the following fields:

Key

A key for your consumption. It is available for [variable syntax](#) in the Trigger URL field (as well as in the [Scripted API](#)). Needs to be at least 2 characters long, start with an alpha, and only contain `a-z`, `A-Z`, `0-9` or `_`.

Example: `project` or `search_term`

Label

A human readable Label shown in the UI as a user works to complete a trigger.

Example: `Room` or `Title`

A screenshot of a Zapier trigger configuration form. The form has a light gray background. At the top, there is a text input field. Below it, the text "This is a Label (required)" is displayed in a bold, dark gray font. Underneath that, the text "This is Help Text" is displayed in a smaller, lighter gray font. Below the help text, there is another text input field containing the text "Default Value". To the right of this field is a green checkmark icon. At the bottom right of the form is a dark gray button with the text "Continue" in white. Three red arrows with white outlines point to specific parts of the form: one points to the top text input field, another points to the "This is Help Text" line, and a third points to the "Default Value" text input field. The words "Label", "Help Text", and "Default" are written in large, bold, red, stylized font with white outlines near their respective arrows.

Help Text

Human readable description of a trigger field, useful for describing some detail you couldn't list in the Label.

Example: `Choose which project to watch for new messages.` or `Define a search term for finding Tweets.`

Default

A default value that is preloaded in the form field as a user works to create the trigger.

Type

The type we will try to coerce to on the backend. Fails silently to ensure that triggers aren't dropped on coercion errors.

Required

If checked a user will not be able to continue without entering some value.

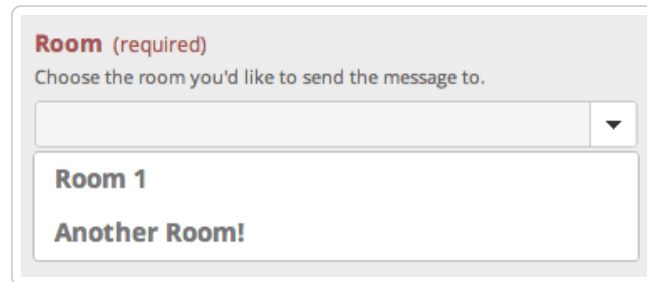
Prefill

Use an existing [Trigger](#) to load in values for selection, using the machine readable value your API consumes (like `id` or `hash`) while showing a human readable version to the user (like `name` or `itemName`).

In our experience, most trigger fields are prefills.

Refer to our [prefill docs](#) for a more in depth explanation.

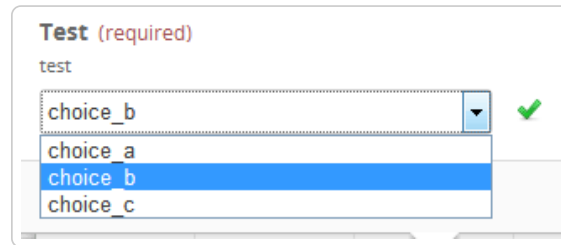
Example: `TRIGGERKEY.id.name` or `TRIGGERKEY.hash.itemName`



Choices

A comma separated string that will be turned into a select field for limiting the choices a user can provide to a trigger field.

Example: `choice_a,choice_b,choice_c` or `Yesterday, Today, Tomorrow`



Test (required)

test

choice_b

choice_a

choice_b

choice_c

Polling ¶

Polling is the process of repeatedly hitting the same endpoint looking for new data. We don't like doing this (its wasteful), vendors don't like us doing it (again, its wasteful) and users dislike it (they have to wait a maximum interval to trigger on new data). However, it is the one method that is ubiquitous, so we support it.

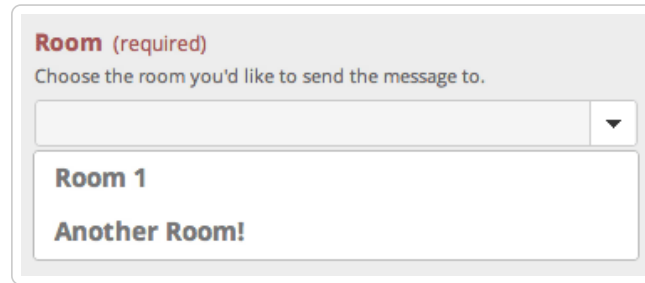
It is also closely tied into how we handle [deduplication](#).

That said, we now support [webhooks](#), so if you like to make your users and servers happy, implement webhooks!

Prefill ¶

In order to access or create resources that have parents, the client must provide a unique identifier to designate the relationship. Since people don't speak in auto-incremented ID's, it is necessary that we offer a simple way to select that identifier using human readable handles. That is what we call a **prefill**.

What a user see's as a prefill:



In order for a prefill to work, we must have a working **trigger** set up for the parent object. This allows to query for that prefill, grab the identifier and a human readable representation and offer the user a drop down selector. Once they make a selection, we store the identifier for use later when reading or writing child objects.

The prefill syntax is split into three parts:

- Which trigger is this prefill referring to?
- Which piece of data is the unique identifier?
- Which piece of data is a human readable representation?

We combine those into one prefill field: `TRIGGERKEY.identifier_key.human_readable_key`.

For example, let's say we have a trigger called Project which offers the following data:

```
{
  "id": 4287243,
  "owner_id": 4632,
  "date_created": "Mon, 25 Jun 2012 16:41:54 -0400",
  "title": "My secret project!",
  "description": "It's a Facebook for dogs!"
}
```

And we want to let users read just the messages from that project as a trigger. As we're making our **message trigger** we'd need to create a `project` **trigger field** with the follow prefill value:

Example: `my_trigger.id.title`

Note: `my_trigger` is a fictional key for the Project trigger, you'll set your specific trigger key when you create your trigger. `id` is the identifier from the above project data. The human readable representation's key

`title` is located there as well.

A prefill is simply a handy artifact on top of [trigger fields](#) and [action fields](#), nothing more. The identifier is passed through just as if they'd entered it in manually.

Status ¶

Your new service can have several statuses as indicated in gray text next to the service name in the left-hand column:

- `draft` *service is only visible to you, service is editable.*
- `pending` *service is only visible to you, service has been submitted for public release, service is not editable.*
- `public` *service is visible to everyone, service is approved, service is not editable.*

Currently, the process for moving your service through the various stages involves [emailing us](#). We hope to have a formal process soon.

Variable Syntax ¶

If you've used [Django](#) or [Mustache](#) (or many, many different templating engines), you are probably well versed in the variable syntax we use. This also happens to be the exact same syntax our users use to map [Trigger Data](#) into [Action Fields](#) (though they get a drag and drop interface to do it).

Given the following `context` (think [auth fields](#) or [trigger data](#)):

```
{
  "id": 123456,
  "owner": "Larry",
  "title": "Hello world!",
  "description": "It's a beautiful day!"
}
```

And the following `template` (think [auth mapping](#) or [action fields](#)):

```
{
  "talking_user": "{{owner}}",
  "chat_message": "{{title}} and of course {{description}}"
}
```

We will provide the following `output`:

```
{
  "talking_user": "Larry",
  "chat_message": "Hello world! and of course It's a beautiful day!"
}
```

You can simulate our syntax with the follow Javascript (suitable for our [Scripting API](#)) using the wonderful Underscore:

```
// we set up templating for you in the scripting API...
_.templateSettings = {
  interpolate : /\{\{(.+)\}\}/g
};
// ...so this is illustration purposes only

function templatize(template, context) {
  // template should be an object
  // context should be an object
  for (key in template) {
    template[key] = _.template(template[key])(context);
  }
  return template;
}
```

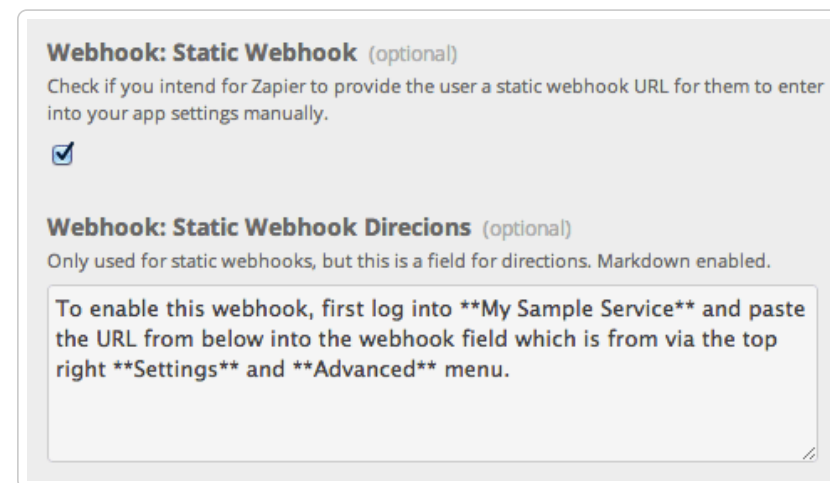
Static Webhooks ¶

Static webhooks are a very simple version of webhooks wherein the user of your service will have to take action to enable a webhook. Usually, this is done copying a url that Zapier provides during the Zap setup process (like `https://zapier.com/hooks/catch/123/n/456789/`). We would prefer you use [subscription webhooks](#) but that usually requires extra code and a full REST API.

To get a taste of how static webhooks work, check out our built-in [generic webhook service](#).

Setup the Webhooks

To get started simply create a new trigger on your service, complete the few required fields and at the bottom pay attention to the form fields imaged below:



The screenshot shows a user interface for setting up a static webhook. It contains two main sections:

- Webhook: Static Webhook (optional)**
Check if you intend for Zapier to provide the user a static webhook URL for them to enter into your app settings manually.
A checkbox is checked.
- Webhook: Static Webhook Direcions (optional)**
Only used for static webhooks, but this is a field for directions. Markdown enabled.
A text box contains the following instructions: "To enable this webhook, first log into **My Sample Service** and paste the URL from below into the webhook field which is from via the top right **Settings** and **Advanced** menu."

Those are the relevent important bits, as they'll provide to your user an interface very similar to the one imaged below:

1 Choose My Sample Ser

To enable this webhook, first log into **My Sample Service** and paste the URL from below into the webhook field which is from via the top right **Settings** and **Advanced** menu.

`https://zapier.com/hooks/catch/123/n/456789/`

Using the Webhooks

For as long as the zap exists, requests made to that URL will be associated with your service and the user. That means requests made during the setup process (while the zap is paused) will still get stored, making it easier for users to debug. However, requests will **not** trigger any actions until the zap is unpaused.

Also, it is important to know that by default, we do our best to inspect and pull out all relevant information in a request. If you use our [scripting API](#) to further refine the hook data you'll also see it passed in as `bundle.cleaned_request`. For example, a request might have a `cleaned_request` similar to below:

```
POST https://zapier.com/hooks/catch/123/n/456789/?hello=world \
-H Content-Type: application/x-www-form-urlencoded \
-d 'some_json=%7B%22foo%22%3A%22hooray!%22%7D&some_xml=%3Cbar%3Eyay!%3C%2Fbar%3E'
```

```
{
  "queryset": {
    "hello": "world",
  },
  "some_json": {
    "foo": "hooray!"
  },
  "some_xml": {
    "bar": "yay!"
  }
}
```

```
}  
}
```

Of course, more simplistic examples like straight JSON or XML are handled as you'd expect.

Catching Single Objects vs. Lists

Some webhooks are single objects, while some are batched as lists of objects for efficiency reasons. Though Zapier supports both, we highly recommend using the [scripting API](#) if you are dealing with batched lists. We do our best via `cleaned_request` to interpret lists where possible, but if your list is hidden in a subkey, we won't capture it properly.

For example, a generically supported list:

```
POST https://zapier.com/hooks/catch/123/n/456789/?hello=world \  
-H Content-Type: application/json \  
-d '[{"name": "bryan", "age": 27}, {"name": "mike", "age": 23}]'
```

Would result in two triggers...

```
{  
  "name": "bryan",  
  "age": 27  
}
```

...and...

```
{  
  "name": "mike",  
  "age": 23  
}
```

Notice that the querystring `hello=world` is ignored? That is because we assumed the data list was more important and we didn't want to maim the data contained therein. Check out an example where we would ignore a subkeyed list:

```
POST https://zapier.com/hooks/catch/123/n/456789/?hello=world \
-H Content-Type: application/json \
-d '{"data":[{"name":"bryan","age":27}, {"name":"mike","age":23}]}'
```

Because the list isn't at the root, by default, we'll interpret this as a single object. That means we'll only trigger once for this object:

```
{
  "queryset": {
    "hello": "world",
  },
  "some_json": [
    {
      "name": "bryan",
      "age": 27
    },
    {
      "name": "mike",
      "age": 23
    }
  ]
}
```

This is where you'd need to break out our [scripting API](#) to do some refinement, check out the examples we provide in the [scripting examples](#).

Payload Subscription Webhooks ¶

Below is our simple standard to stop the [polling](#) madness™. This is a suggested pattern for you to implement. It was chosen because there are many examples of REST APIs providing resources around GETting, POSTing and DELETEing webhooks.

If you read on to our Notification Subscription Webhooks, you'll notice some similarities. The first difference is **step 2**. Payload Subscription Webhooks simply dump the data when hooking, wherein Notification Subscription

Webhooks offer up a `resource_url` to *retrieve* the object in real time, which adds an extra step.

Before: Your Checklist *(an todo overview)*

- enumerate the events you'd like to be available as triggers
- create a `subscribe` REST endpoint
- when each event happens, loop over and notify each active subscription (or batch events of same type)
- respect `410` responses on notifications and remove subscriptions
- create a `unsubscribe` REST endpoint

Read on for a step by step through the endpoints involved...

Step 1: Subscribe *(a call from Zapier to your service)*

```
POST https://your-app.com/<subscribe_endpoint> \
-H Authenticated: Somehow \
-H Content-Type: application/json \
-d '{"subscription_url": "https://zapier.com/<unique_subscription_url>",
    "event": "user_created"}'
```

This endpoint reuses whatever auth standard you have across the rest of your API (IE: Basic Auth, API key, OAuth2, etc...). We'd send along a **unique, generated** subscription URL and the event we'd like to subscribe to. These three items would be persisted on your backend (the authenticated user, `subscription_url`, and event).

Alternatively, you can modify this request to call a more generic REST endpoint (say `/webhooks`) via our `pre_subscribe` scripting API method.

On a successful subscribe, return a `201` status code. **Normally, response data is discarded**, but you can store data about the webhook you just created with the `post_subscribe` scripting API method.

Note: subscription URLs should be unique. Return a `409` status code if this criteria isn't met (IE: there is a uniqueness conflict).

Step 2: Hooking *(a call from your service to Zapier)*

```
POST https://zapier.com/<unique_subscription_url> \  
-H Content-Type: application/json \  
-d <json payload>
```

This hook could provide any amount of data or payload in either JSON or XML (or form-encoded). Just like static webhooks, we'll do our best to parse the response.

On a successful hook, we'll return a `200` status code, content is irrelevant.

Important: if Zapier responds with a `410` status code you should immediately remove the subscription to the failing hook (unsubscribe). Additionally, excessive failures (multiple `4xx` or `5xx` failures) can be handled at your discretion. It is the clients responsibility to resubscribe if needed after failures.

Step 3: Unsubscribe *(a call from Zapier to your service)*

```
POST https://your-app.com/<unsubscribe_endpoint> \  
-H Content-Type: application/json \  
-d '{"subscription_url": "https://zapier.com/<unique_subscription_url>"}'
```

No authentication should be needed for unsubscribing a URL (as the auth tokens may expire and we won't have a way to remove a webhook). Therefore, the subscription URL should effectively be unique (this should be enforced by your service). This also allows us to clean up unrecognized webhooks, if that were to ever happen.

However, if your unsubscribe endpoint is different from our spec, you can use our `pre_unsubscribe` scripting API method to customize this call. You can also use any `bundle.subscribe_data` returned from `post_subscribe` to further refine your unsubscribe call.

On a successful unsubscribe, return a `200` status code, content is irrelevant.

Notification Subscription Webhooks ¶

Below is our simple standard to stop the [polling](#) madness™. This is a suggested pattern for you to implement. It was chosen because it is simple and requires minimal work to add to an existing REST API.

Our goal is a simple, hook based alert system that revolves around existing API REST tech. A consumer would subscribe to a stream of events, but the hooks that send an event wouldn't provide the actual data, just an (optionally time-sensitive one-time use) URL which is also protected by the standard auth scheme already implemented by your service.

The big benefit is consumers wouldn't need to poll for changes, but could instead wait for hooks, then grab the underlying data one at a time. Additionally, producers could provide real-time updates with fewer devoted resources (compared to polling).

Though we highly recommend this native pattern, you can override all the methods below with our [scripting API](#). This means you can bend our pattern to match an existing pattern of your own.

Before: Your Checklist *(an todo overview)*

- enumerate the events you'd like to be available as triggers
- create a `subscribe` REST endpoint
- when each event happens, loop over and notify each active subscription (or batch events of same type)
- respect `410` responses on notifications and remove subscriptions
- create a `unsubscribe` REST endpoint

Read on for a step by step through the endpoints involved...

Step 1: Subscribe *(a call from Zapier to your service)*

```
POST https://your-app.com/<subscribe_endpoint> \  
-H Authenticated: Somehow \  
-H Content-Type: application/json \  
-d '{"subscription_url": "https://zapier.com/<unique_subscription_url>",  
    "event": "user_created"}'
```

This endpoint reuses whatever auth standard you have across the rest of your API (IE: Basic Auth, API key, OAuth2, etc...). We'd send along a **unique, generated** subscription URL and the event we'd like to subscribe to. These three

items would be persisted on your backend (the authenticated user, subscription_url, and event).

Alternatively, you can modify this request to call a more generic REST endpoint (say `/webhooks`) via our `pre_subscribe` scripting API method.

On a successful subscribe, return a `201` status code. **Normally, response data is discarded**, but you can store data about the webhook you just created with the `post_subscribe` scripting API method.

Note: subscription URLs should be unique. Return a `409` status code if this criteria isn't met (IE: there is a uniqueness conflict).

Step 2: Hooking (or alerting) *(a call from your service to Zapier)*

```
POST https://zapier.com/<unique_subscription_url> \
-H Content-Type: application/json \
-d '{"resource_url": "https://your-app.com/<resources_url>"}'
```

All this hook would do is provide a resource URL to get the data. Could be a one time use URL or a reversed out URL to the standard REST resource.

On a successful hook, we'll return a `200` status code, content is irrelevant.

Important: if Zapier responds with a `410` status code you should immediately remove the subscription to the failing hook (unsubscribe). Additionally, excessive failures (multiple `4xx` or `5xx` failures) can be handled at your discretion. It is the clients responsibility to resubscribe if needed after failures.

Step 3: Consuming *(a call from Zapier to your service)*

```
GET https://your-app.com/<resources_url> \
-H Authenticated: Somehow \
-H Content-Type: application/json
```

This is an old fashioned REST hit to a standard endpoint, it requires the same authentication as the rest of the API. Can return a list or a single object.

If you don't want to perform this step, look at our Payload Subscription Webhooks above!

Step 4: Unsubscribe *(a call from Zapier to your service)*

```
POST https://your-app.com/<unsubscribe_endpoint> \  
-H Content-Type: application/json \  
-d '{"subscription_url": "https://zapier.com/<unique_subscription_url>"}'
```

No authentication should be needed for unsubscribing a URL (as the auth tokens may expire and we won't have a way to remove a webhook). Therefore, the subscription URL should effectively be unique (this should be enforced by your service). This also allows us to clean up unrecognized webhooks, if that were to ever happen.

However, if your unsubscribe endpoint is different from our spec, you can use our `pre_unsubscribe` scripting API method to customize this call. You can also use any `bundle.subscribe_data` returned from `post_subscribe` to further refine your unsubscribe call.

On a successful unsubscribe, return a `200` status code, content is irrelevant.

Helper Library

We will provide a Django/TastyPie application to do this. There is talk of a Ruby Gem as well. If you are interested in writing (and open-sourcing) a general wrapper against this standard, [ping us](#).



[Reviews](#) | [Announcements](#) | [Service Directory](#) | [Developers](#) | [Blog](#) | [About](#) | [Support](#) | [Privacy](#)



Site copyright © 2013 Zapier, Inc. All Rights Reserved.

