waliki Documentation

Release 0.8.1

Martín Gaitán

Mar 26, 2017

Contents

Waliki is an extensible wiki app for Django with a Git backend.

> **Attention:** It's in an early development stage. I'll appreciate your feedback and help.

home https://github.com/mgaitan/waliki/

demo http://waliki.pythonanywhere.com

documentation http://waliki.rtfd.org

twitter @Waliki_ // @tin_nqn_

group https://groups.google.com/forum/#!forum/waliki-devs

license BSD

At a glance, Waliki has these features:

- File based content storage.
- UI based on Bootstrap and CodeMirror
- Version control and concurrent edition for your content using git
- An extensible architecture through plugins
- reStructuredText or Markdown support, configurable per page (and it's easy to add extensions)
- A very simple *per slug* ACL system
- A nice attachments manager (that respects the permissions over the page)
- Realtime collaborative edition via togetherJS
- Wiki content embeddable in any django template (as a "dummy CMS")
- Few helpers to migrate content (particularly from MoinMoin, using moin2git)
- It works with Python 2.7, 3.4 or PyPy in Django 1.8, 1.9 (and 1.10, most probably)

It's easy to create a site powered by Waliki using the preconfigured project which is the same code that motorize the demo.

Waliki was inspired in Github's wikis, but it tries to be a bit smarter than many others git backed wiki engines at handling changes: instead of a hard *"newer wins"* or *"page blocking"* approaches, Waliki uses git's merge facilities on each save. So, if there was another change during an edition and git can merge them automatically, it's done and the user is notified. If the merge fails, the last edition is still saved but the editor is reloaded asking the user to fix the conflict.

CHAPTER 1

Getting started

Install it with pip:

```
$ pip install waliki[all]
```

Or the development version:

```
$ pip install https://github.com/mgaitan/waliki/tarball/master
```

Add `waliki` and the optionals plugins to your INSTALLED_APPS:

```
INSTALLED_APPS = (
    ...
    'waliki',
    'waliki.git',           # optional but recommended
    'waliki.attachments',   # optional but recommended
    'waliki.pdf',           # optional
    'waliki.search',        # optional, additional configuration required
    'waliki.slides',        # optional
    'waliki.togetherjs',    # optional
    ...
)
```

Include `waliki.urls` in your project's `urls.py`. For example:

```
urlpatterns = patterns('',
    ...
    url(r'^wiki/', include('waliki.urls')),
    ...
)
```

Sync your database:

```
$ python manage.py migrate
```

Tip: Do you already have some content? Put it in your `WALIKI_DATA_DIR` (or set it to the actual path) and run:

```
$ python manage.py sync_waliki
```

Do you want everybody be able to edit your wiki? Set:

```
WALIKI_ANONYMOUS_USER_PERMISSIONS = ('view_page', 'add_page', 'change_page')
```

in your project's settings.

CHAPTER 2

Contribute

This project is looking for contributors. If you have a feature you'd like to see implemented or a bug you'd liked fixed, the best and fastest way to make that happen is to implement it and submit it back upstream for consideration. All contributions will be given thorough consideration.

Everyone interacting in the Waliki project's codebases, issue trackers and mailing lists is expected to follow the PyPA Code of Conduct.

CHAPTER 3

Why *Waliki* ?

Waliki is an Aymara word that means *all right*, *fine*. It sounds a bit like *wiki*, has a meaningful sense and also plays with the idea of using a non-mainstream language[1] .

And last but most important, it's a humble tribute to the president Evo Morales and the Bolivian people.

Contents:

Installation

Install via pip

To install the latest stable release, run the following:

```
$ pip install waliki
```

By default, Waliki uses reStructuredText as its markup, so docutils and other required dependencies are retrieved. If you prefer a Markdown only wiki, install it as it follows:

```
$ pip install waliki[markdown]
```

Alternatively, if you want to install every dependency, use:

```
$ pip install waliki[all]
```

Configure `settings.INSTALLED_APPS`

Add `waliki` and optional plugins to your INSTALLED_APPS:

[1] *wiki* itself is a hawaiian word

```
INSTALLED_APPS = (
    ...
    'waliki',
    'waliki.git'    # optional
    ...
)
```

Attention: To enable `waliki.git` you need Git installed in your system. In Debian/Ubuntu:

```
$ sudo apt-get install git
```

Sync database

Although Waliki stores page content as flat files, it uses a model to store page titles, slugs and other fields.

Create this model table using:

```
$ python manage.py syncdb
```

Include url patterns

Include the waliki urls in your project `urls.py`. For example:

```
urlpatterns = patterns('',
    ...
    url(r'^wiki/', include('waliki.urls')),
    ...
)
```

Waliki will handle the inclusion of installed plugins urls automatically.

Settings

Waliki follows a convention over configuration paradigm, defining sensible defaults for every constant.

You can override any settings in your project's `settings.py` file

WALIKI_DATA_DIR
: Waliki's content path. By default it's `<project_root>/waliki_data`

  You can set it to an absolute path. Ensure the path exists and it's writable from your web server

WALIKI_INDEX_SLUG
: The slug of the index page. Default is `home`

WALIKI_ANONYMOUS_USER_PERMISSIONS
: The tuple of permissions given to not authenticated users. Default is `('view_page',)` Check *The access control system* for further details. If there is no `change_page` permission, the anonymous user is redirected to the login page when try to edit the page.

WALIKI_LOGGED_USER_PERMISSIONS
: The tuple of permissions given to any authenticated user. Default is `('view_page', 'add_page', 'change_page')`. Check *The access control system* for further details.

WALIKI_AVAILABLE_MARKUPS
A list tha define the enabled markups. Default is `['reStructuredText', 'Markdown']`. Available markups are `reStructuredText`, `Markdown` and `Textile`

WALIKI_DEFAULT_MARKUP
The default markup for new pages. Default `WALIKI_AVAILABLE_MARKUPS[0]`

WALIKI_SLUG_PATTERN
Pattern used in urls to match any page related view, wich is also de filename of the file that store the page content. Default is `'[a-zA-Z0-9-_\/]+'`.

WALIKI_SLUGIFY_FUNCTION
String pointing to a callable that receive a text and return and slug. Default `'waliki.utils.get_slug'`

If you override it, ensure that `your_get_slug(any_valid_slug) == any_valid_slug`

WALIKI_SANITIZE_FUNCTION
New in version 0.6.

String pointing to a callable that receive html and return and return a sanitized version of it. Default `'waliki.utils.sanitize'`, which just removes `<script>` tags.

You can define a more sofisticated version using bleach or lxml's Cleaner

WALIKI_MARKUPS_SETTINGS
Dictionary of keywords arguments to extend or override the ones passed for each markup class. By default, this is the dictionary used

```
{'reStructuredText': {
    # check http://docutils.sourceforge.net/docs/user/config.html
    'settings_overrides': {
        'initial_header_level': 2,
        'record_dependencies': True,
        'stylesheet_path': None,
        'link_stylesheet': True,
        'syntax_highlight': 'short',
        'halt_level': 5,
    },
    'writer': HTML5Writer(),
    'writer_name': 'html5',
    },
'Markdown': {
        'extensions': ['wikilinks', 'headerid'],
        'extension_configs': {
            'wikilinks': {'build_url': get_url},
            'headerid': {'level': 2},
        }
    }
}
```

WALIKI_BREADCRUMBS
New in version 0.6.

If `True`, show a breadcrumbs with links to "parent" pages. Default is `False`

WALIKI_PDF_INCLUDE_TITLE
Apply if *PDF plugin* is installed.

As the title is not part of the file content but stored in the database, it should be given to rst2pdf. Default is `False`

WALIKI_PDF_RST2PDF_BIN
Apply if *PDF plugin* is installed.

A custom binary path to rst2pdf. E.g. '/usr/bin/rst2pdf'

WALIKI_CODEMIRROR_SETTINGS
A dictionary (converted to json) used to configure Codemirror. The default is:

```
{'theme': 'mbo', 'autofocus': True, 'lineNumbers': True}
```

WALIKI_RENDER_403
If `True`, raise an HTTP 403 (Forbidden error) if an authenticated user is not allowed to edit a page. Default is `True`.

WALIKI_PAGINATE_BY
The numbers of items per page in paginated lists, for example "what changed". Default is `20`.

WALIKI_COMMITTER_EMAIL
If *The "Git" backend* is enabled and anonymous editios allowed, this is the git's committer email used. Default is `waliki@waliki.pythonanywhere.com`.

WALIKI_COMMITTER_NAME
Analog to *WALIKI_COMMITTER_EMAIL*. Default is `Waliki`

WALIKI_CACHE_TIMEOUT
The maximum expiration time for a page cache, in seconds. Default is `60*60*24` (i.e. 1 day)

WALIKI_ATTACHMENTS_DIR
If *Plugin "Attachments"* is enabled, this is the path where uploaded files are stored.

By default it's `<project_root>/waliki_attachments`. Ensure the path exists and it's writable by your web server.

WALIKI_UPLOAD_TO_PATTERN
The pattern used in the path relative to *WALIKI_ATTACHMENTS_DIR* to store uploaded files. It's interpolated with the following dictionary:

```
    {'slug': instance.page.slug,
    'page_id': getattr(instance.page, 'id', ''),
    'filename': filename,
    'filename_extension': os.path.splitext(filename)[1]}

Default is ``'%(slug)s/%(filename)s'``
```

WALIKI_RST_DIRECTIVES
List of string poiting to modules with `register_directive()` function that register extra reStructuredText Directives. Default is `['waliki.directives.embed']`

Check embed.py as an example.

WALIKI_RST_TRANSFORMS
List of string poiting to reStructuredText extra Transforms classes to be applied

Check transforms.py as an example.

Default is `['waliki.directives.transforms.Emojis']`

WALIKI_USE_MATHJAX
If `True`, load Mathjax's assets from the official CDN service Default is `False`. Check the *faq* for details.

Autolinks

One of the most important features in any wiki system is the simplicity to create and link pages. Waliki has a simple support for internal links

Autolinks in reStructuredText

There is autolinking support for restructuredtext with a very simple trick: if you don't explicitly define the target in a link (a word ending with and undercore like `this_`), it will automatically point to an internal wiki page (even if it doesn't exist yet).

So, just define `somewhere_` and link the page with the slug *somewhere*

Waliki find every link with an undefined target, and interpret it as an internal link

Autolinks in Markdown

Waliki enables the Markdown's WikiLinks extension by default.

It uses `[[Somewhere]]` to link the page with slug *somewhere*.

Builtin plugins

The "Git" backend

The Git backend is a regular plugin but, probably, it is what adds the most interesting features to Waliki.

Basically, `waliki.git` converts your content folder in a git repository, and makes a commit on each edition.

With this simple logic you'll get:

- History of changes (who, when, what)
- Diff: compare any version and see what was added or removed
- Smart concurrent edition handling: don't lock editions, merge them!
- View and restore old revisions (without losing the history)
- Simple stats: how many lines were added or removed. (go to the history page to see it in action!)
- Backup (pushing your repo to a remote place)
- Edit your content outside the web using the editor of your preference!
- Webhook/s (pull changes from a remote repository)

To install it, add `'waliki.git'` after `'waliki'` in your `settings.INSTALLED_APPS`.

Tip: This plugin is optional, but strongly recommended.

This extension uses the `git` command line machinary wrapped via the wonderful sh package. Although it could have a performance impact compared with a python git library, my experience is that pygit2 is a bit complex to use and GitPython doesn't work with Python 3.

The *pull* webhook

`waliki.git` has a webhook endpoint that receives an HTTP POST requests (without parameters) to pull and sync content from a remote repository:

```
POST http://yoursite.com[/<waliki_prefix>]/_hooks/pull/<remote name>
```

This is useful to sync your wiki whenever a repository is pushed to. For example when you push to github.

When a `POST` event arrives to the webhook url, Waliki programatically run the command `sync_waliki` that pulls the code from the remote name `<remote name>`, and then syncs the database adding or deleting `Page` instances as needed.

Of course, the `remote` name should be registered in your waliki data repo. See the man page for `git remote` to do this.

To keep your remote synced with the changes done via the web editor, you should push back frequently, setting a cron job or a similar tool.

Plugin "Attachments"

This plugin allows to upload files to a page and include it in the text as an embedded image or a link.

Using the standard template, it extends the `edit` page with a new button 'Attachments'

In old-fashioned browsers without FormData support (i.e. Internet Explorer < 10) the ajaxified upload form (including the uploading progress bar) degrades to a popup window.

Setup

It requires django-sendfile as an extra requirement. Install it via pip:

```
$ pip install django-sendfile
```

then adds `waliki.attachments` and `sendfile` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = (
     ...
     'waliki',
     'waliki.attachments',
     ...
     'sendfile',
     ...
 )
```

Permissions

The permissions over attachment are inherited from the container page.

- To view or download an attachment, the user needs the permission `view_page` over the page's slug to which the attachment belongs.
- To upload a new attachment, `change_page` is required.
- and to delete, `delete_page` is required.

In order to serve the attached files using your web server, but still checking permission, Waliki uses django-sendfile , which wraps the different techniques (for different webserver) to do this.

You need to define the `SENDFILE_BACKEND` to use. For a basic configuration set:

```
SENDFILE_BACKEND = 'sendfile.backends.simple'
```

Attention: by default, Waliki uploads attachments to the path `<MEDIA_ROOT>/waliki_attachments/<page_slug>/<filename>`. Override the function `WALIKI_UPLOAD_TO` in your settings if you need another structure.

Read the django-sendfile documentation for specific instructions if you use Nginx or Apache.

Plugin "Get as PDF"

This a very simple plugin that leverages on rst2pdf to get a PDF version of a page.

It registers a new *extra_page_action view* in the dropdown menu as *"Get as PDF"*. For obvious reasons, it only appears in reStructuredText pages.

Get it working on python 3

rst2pdf is a python2 software, so if we are running Waliki with python3 is not possible to run rst2pdf inside the virtualenv. To get it working you simply have to install rst2pdf as an OS package (apt-get install rst2pdf or pacman -S python2-rst2pdf) and then add *WALIKI_PDF_RST2PDF_BIN* to your waliki settings file detailing the rst2pdf binary path. For example:

```
WALIKI_PDF_RST2PDF_BIN='/usr/bin/rst2pdf'
```

Tip: It should be trivial to write new plugins that add support to other converter tools like any `rst2*` or Pandoc to convert from markdown.

Plugin "View as slides"

This plugin leverages on Hovercraft to view a page as a presentation. It only works for restructuredText)

It registers a new *extra_page_action view* in the dropdown menu as *"View as slides"*. For obvious reasons, it only appears in reStructuredText pages.

To install it, add `'waliki.slides'` after `'waliki'` in your `settings.INSTALLED_APPS`.

Real-time collaboration via together.js

The TogetherJS plugin enables real-time collaborative editing of a page. It builds off of Mozilla's TogetherJS library.

To install it, add `'waliki.togetherjs'` after `'waliki'` in your `settings.INSTALLED_APPS`.

The only two things this plugin does is to setup `together.js` and add a button in the toolbar to initialize a collaborative session.

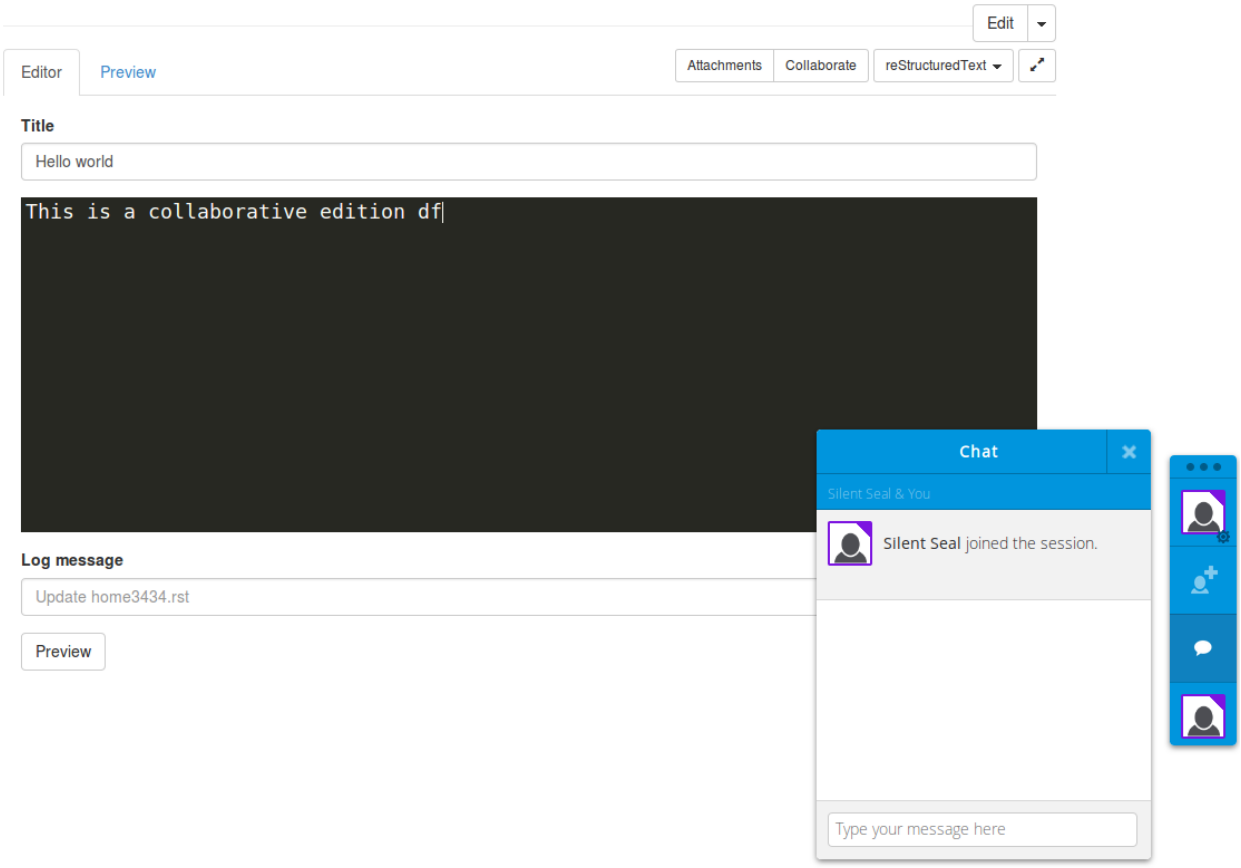

Fig. 3.1: A collaborative session of two users editing a page

The default configuration turns off some togetherJS's features like the audio chat. It's defined in `waliki/togetherjs_edit_extra_script.html` and you can override it in your project to extend or replace some config variables.

Currently, it looks like this:

```
{% load i18n %}

  <script type="text/javascript">

    var TogetherJSConfig_toolName = "{% trans "Collaborate" %}";
    var TogetherJSConfig_dontShowClicks = true;
    var TogetherJSConfig_disableWebRTC = true;
    var TogetherJSConfig_getUserName = function (){
        {% if request.user.is_authenticated %}
          return '{{ request.user.username }}';
        {% else %}
          return null;
        {% endif %}
    };

  </script>

  <script src="https://togetherjs.com/togetherjs-min.js"></script>
```

Write a plugin

Waliki has a very little core designed to be extensible with **plugins**.

At the moment, there are a few plugins built in but you can create a new one very easily.

A plugin is a normal Django app, with a file named `waliki_plugin.py` that defines a subclass of `BasePlugin`.

As an example, see the *waliki.git.waliki_plugin.py*.

```
from django.utils.translation import ugettext_lazy as _
from waliki.plugins import BasePlugin, register


class GitPlugin(BasePlugin):

    slug = 'git'
    urls_page = ['waliki.git.urls']
    extra_page_actions = {'all': [('waliki_history', _('History'))]}
    navbar_links = (('waliki_whatchanged', _('What changed')),)

register(GitPlugin)
```

What can a plugin do?

In the first place, it's important to remark that a waliki plugin **is a django app**, so you can do with them anything an app can do: define new models, add or override templates, connect signals, etc.

Tip: Moreover, you can override a waliki core view! It's possible because the urls registered by plugins take precedence over the core ones.

In addition, you can register "actions" (views that receive a page slug as parameter).

The field `extra_page_actions` is a list of tuples `('url_name', 'link text')`, where each `url_name` is reversed passing the page's `slug` as parameter. These actions appear in the dropdown of the * "Edit"* button.

Analogously, `extra_edit_actions` add "buttons" (links) to the editor toolbar.

Extending templates with entry points

Another thing a plugin can do is to extend the core templates. It leverages in the template tag `entry_point`.

Wherever a tag `{% entry_point 'name' %}` is present, this templatetag will look for templates named `waliki/<plugin_slug>_name` for each plugin registered and it will include those found.

For example, the block `{% block content %}` in `edit.html` ends like this:

```
    ...

    {% entry_point 'edit_content' %}
{% endblock content %}
```

At that point, the template `waliki/attachments_edit_content.html` (and any other template with the `waliki/<plugin_slug>_edit_content.html`) will be appended , using a standard include that receives the whole context.

Tip: you can search the code to know every template entry point available

Waliki signals

In addition to the built-in model signals, your plugin can connect receivers functions to the signals that Waliki sends when few actions happen. At the moment, there is one:

- `page_saved` is sent just after saving a page. The parameters are the `page` instance, the `user` who edited the page, and the optional `message`. For example, Git extensions uses it to make a commit with the new comment.

Note: Of course, you can add any new signals you need!

Boxes: Waliki as a (dummy) CMS

The templatetag `waliki_box` allows to display the body of a wiki page as a portion (a "box") of a webpage, and a rapid inline edition if the user has the right permission.

The templatetag receives the page's slug as unique parameter:

```
{% waliki_box "page/slug" %}
```

Usage example

Consider a view that renders a template:

```
urlpatterns = patterns('',
    ...
    url(r'^boxes-example/', TemplateView.as_view(template_name="boxes_example.html")),
    ...
)
```

Where `boxes_example.html` is as following:

```
{% extends "base.html" %}
{% load waliki_tags %}

{% block body %}
    <h1>Waliki boxes example</h1>

        <div class="row" style="margin-top: 50px">
            <div class="col-sm-8">
                {% waliki_box "boxes/left" %}
            </div>
            <div class="col-sm-4">
                {% waliki_box "boxes/right" %}
            </div>
        </div>
        <hr>
        <div class="row">
            <div class="col-lg-12">
                {% waliki_box "boxes/footer/"|add:request.user.username %}
            </div>
        </div>
    </div>

{% endblock %}
```

You can see this example live in the demo site. Note that the demo site applies a *ACL rule* to limit the edition under the *namespace* `boxes` to authenticated users.

Login to edit the boxes!

Context dependent boxes

As you can see in the code of the template, the last box is specifically for you, because it will render `boxes/footer/<your_username>`.

This technique can be used, for example, to display a different content for different languages:

```
{% waliki_box "my-content/"|add:request.LANGUAGE_CODE %}
```

This will display `my-content/en` for English, `my-content/es` for Spanish, etc.

The access control system

Waliki has a very simple *"per slug"* Access Control List system built-in, that allows to control who has access to view, add, change or delete pages (and possible other permissions and objects) in your wiki.

It's inspired in django-guardian and leverages on `django.contrib.auth`.

You can define your ACL policies defining default permissions for anonymous and logged users in your settings (`WALIKI_ANONYMOUS_USER_PERMISSIONS` and `WALIKI_LOGGED_USER_PERMISSIONS`) and instances of the model `ACLRule`` that stores:

- which permissions the rule gives
- to which groups and/or users
- limited to which slug

So, here is how it works:

- Access controlled views have a decorator `acl.permission_required()` that asks the user for one or more permissions **in that specific slug** to access the view.
- The decorator checks if there is an ACL rule with the requested permission/s that apply to this slug.
- If there is a rule and the user is in the rule's allowed users (because it was explicitly assigned or because it belongs to a group assigned to the rule), then the user will be able to access
- If there isn't a matching rule, check Waliki's defaults permissions
- Lastly, check standard user's *per model* permissions.

Example

Suppose you want this policy:

- Anonymous users can view any page except the ones under the slug *intranet*. Anonymous users can't edit pages.
- Identified users are allowed to see and edit any page, even the ones under the slug *intranet*, but they aren't allowed to edit the page with slug *home* (the homepage) nor to delete any pages
- The user *john* and any user from the group *editors* can edit the home
- Only superusers can delete pages.

So, first, by default, anonymous users only have `view_page` permission, and logged in users can also edit but not delete. In your settings:

```
WALIKI_ANONYMOUS_USER_PERMISSIONS = ('view_page', )
WALIKI_LOGGED_USER_PERMISSIONS = ('view_page', 'add_page', 'change_page')
```

Note: Note that, in this case, those are the Waliki's defaults permissions, so, you wouldn't need to set them. Check *WALIKI_ANONYMOUS_USER_PERMISSIONS* and *WALIKI_LOGGED_USER_PERMISSIONS* for further details.

Then go to the admin an create the following rules:

- One rule for the slug **intranet** with the permissions `view_page`, `add_page` and `change_page`. In "Apply to" select *Any authenticated user*
- Add a rule for the homepage: slug *home* (or the slug defined in `WALIKI_INDEX_SLUG`), with the permission `add_page` and `change_page`, apply to *Any user/group explicitly defined*, and add the user *jhon* and the group *editors* respectively.
- Lastly, add a rule for the permission `delete_page` and apply it to *Any superusers*

Checking permissions in your plugins

If you are writing your own plugin, you can use the ACL reusing the view decorator. For example:

```
from waliki.acl import permission_required

@permission_required('view_page')
def your_read_only_view(request, slug):
    ...

@permission_required(['change_page', 'add_page'])
def your_read_write_view(request, slug):
    ...
```

> **Attention:** When a view requires more than one permission, at least one rule with **all those permissions** should apply to the user.
>
> For example, if the rule *A* gives to *user1* the permission `change_page` and the rule *B* gives to *user1* the permission `delete_page`, *user1* is still not allowed to request a view that requires both `change_page` and `delete_page`.

Also, you can use the low-level helper `acl.check_perms()`:

```
if check_perms(('edit_page'), request.user, page.slug):
    do_something()
```

To check permissions in a template, you can use the templatetag `waliki_tags.check_perms()`

> **Attention:** Make sure you have `django.core.context_processors.request` in your `TEMPLATE_CONTEXT_PROCESSORS` setting to use contextual variables like `request.user`

The format is:

```
{% check_perms "perm1[, perm2, ...]" for user in slug as "context_var" %}
```

or:

```
{% check_perms "perm1[, perm2, ...]" for user in "slug" as "context_var" %}
```

For example (assuming `page` objects are available from *context*)

```
{% load waliki_tags %}

{% check_perms "delete_page" for request.user in page.slug as "can_delete" %}
{% if can_delete %}
    <a id="confirmDelete" class="text-error">Delete</a>
{% endif %}
```

Frequently Asked Questions

Which is the biggest site powered by Waliki? It's the 'Python Argentina Community's wiki <http://python.org.ar/wiki/ >'_ . It was migrated to Waliki from MoinMoin in March 2015.

It has more than 1000 pages and few active users.

Does Waliki scale? May be, but huge wiki site are not the Waliki's target.

My main concern about the Waliki's "scalability" is on how many concurrent users may it support and how slow is to save a page..

The *"git commit per edition"* is cool, I'm happy with the *merge-instead-block* approach for concurrent editions, but it could be a boottleneck for a high traffic wiki.

I guess it could be improved in the future, using libgit2 instead of plain system calls to the git cli.

Can Waliki render math? Sure! Both reStructuredText and Markdown play well with Mathjax. As the Mathjax's assets (javascript file) are huge, it's disable by default. To enable it you need add `waliki.context_processors.settings` to `TEMPLATE_CONTEXT_PROCESSORS` and set `WALIKI_USE_MATHJAX` to `True` in your settings:

```
TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    "waliki.context_processors.settings"
)

WALIKI_USE_MATHJAX = True
```

Does it use some cache? Yes, it uses the builtin django's cache framework . The cache is invalidated automatically when a page is modified. The default cache timeout is 1 day, but you can override it setting `WALIKI_CACHE_TIMEOUT` (in seconds). For example:

```
WALIKI_CACHE_TIMEOUT = 3600    # 1 hour cache
```

Can I have *user pages*? Of course, but this feature isn't built-in. As a *user page* is a regular page editable exclusively by an user, you can hook a simple code to create an `ACLRule` instance to the signal raised when a new `User` is signed up.

Check the code of this implementation in the Python Argentina's site.

A tale on Waliki's history

In june of 2013 I tweeted this:

When the master Roberto Alsina offered to convert Alva in a wiki system someday I asked him this:

That was the beginning. I found and forked a tiny wiki system based on Flask and sent a first pull request where I explained a few of my motivations:

> I've researched for a while, looking for a simple, yet usable, wiki engine with support for reStructuredText, since it's in what I write fluently and what I can use to render with Sphinx (de facto standard documentation tool in the python ecosystem), rst2pdf or whatever
>
> Few wikis like MoinMoin or the builtin wiki in Trac have this feature, but processing a "block of restructuredtext" in the middle of another core markup.
>
> Even when this project uses markdown, it's so simple to refactor this "render function" as a config option.

During those days, I started to take a lot of design decisions and committed a bunch of code, so my fork diverged too much from Alex's wiki. That's how the original version of Waliki was born.

Waliki reborn

I was happy with the result but blocked to continue. As a newbie in Flask, each attempted step was a challenge, and I was not sure whether to be faithful to the conventions: my brain was too *djangonized*.

Moreover, the Python Argentina web project needed a wiki engine based on Django.

So, I decided to redo it, taking as much code and as many ideas as possible. It took some time, but it's here :).

Changelog

0.8.1 (2017-03-26)

- Fixed compatibiltiy with Django 1.10 (thangs to Martí Bosch)
- Fixed #125
- Upgraded demo's setting to recent format
- Added missing migration

0.7 (2016-12-19)

- Fix compatibility with Django 1.9.x and Markup 2.x (thanks to Oleg Girko for the contribution)

0.6 (2015-10-25)

- Slides view use the cache. Fix #81
- Implemented an RSS feed listing lastest changes. It's part of #32
- Added a configurable "sanitize" function.
- Links to attachments doesn't relay on IDs by default (but it's backaward compatible). #96
- Added an optional "breadcrumb " hierarchical links for pages. #110
- Run git with output to pipe instead of virtual terminal. #111

0.5 (2015-04-12)

- Per page markup is now fully functional. It allows to have a mixed rst & markdown wiki. Fixed #2
- Allow save a page without changes in a body. Fixed #85
- Fixed #84, that marked deleted but no commited after a move
- Allow to choice markup from new page dialog. #82
- Fix wrong encoding for raw of an old revision. #75

0.4.2 (2015-03-31)

- Fixed conflict with a broken dependecy

0.4.1 (2015-03-31)

- Marked the release as beta (instead of alpha)
- Improves on setup.py and the README

0.4 (2015-03-31)

- Implemented views to add a new, move and delete pages
- Implemented real-time collaborative editing via together.js (#33)
- Added pagination in *what changed* page
- Added a way to extend waliki's docutils with directives and transformation for
- A deep docs proofreading by chuna
- Edit view redirect to detail if the page doesn't exist (#37)
- waliki_box fails with missing slug #40
- can't view diffs on LMDE #60
- fix typos in tutorial #76 (martenson)
- Fix build with Markups 0.6. #63 (loganchien)
- fixed roundoff error for whatchanged pagination #61 (aszepieniec)
- Enhance slides #59 (loganchien)
- Fix UnicodeDecodeError in waliki.git.view. #58 (loganchien)

0.3.3 (2014-11-24)

- Tracking page redirections
- fix bugs related to attachments in *sync_waliki*
- The edition form uses crispy forms if it's installed
- many small improvements to help the integration/customization

0.3.2 (2014-11-17)

- Url pattern is configurable now. By default allow uppercase and underscores
- Added `moin_migration_cleanup`, a tool to cleanup the result of a moin2git import
- Improve git parsers for *page history* and *what changed*

0.3.1 (2014-11-11)

- Plugin *attachments*
- Implemented *per namespace* ACL rules
- Added the `waliki_box` templatetag: use waliki content in any app
- Added `entry_point` to extend templates from plugins

- Added a webhook to pull and sync change from a remote repository (Git)
- Fixed a bug in git that left the repo unclean

0.2 (2014-09-29)

- Support concurrent edition
- Added a simple ACL system
- `i18n` support (and locales for `es`)
- Editor based in Codemirror
- Migrated templates to Bootstrap 3
- Added the management command `waliki_sync`
- Added a basic test suite and setup Travis CI.
- Added "What changed" page (from Git)
- Plugins can register links in the nabvar (`{% navbar_links %}`)

0.1.2 / 0.1.3 (2014-10-02)

- "Get as PDF" plugin
- rst2html5 fixes

0.1.1 (2014-10-02)

- Many Python 2/3 compatibility fixes

0.1.0 (2014-10-01)

- First release on PyPI.

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/mgaitan/waliki/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.

- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Waliki could always use more documentation, whether as part of the official waliki docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/mgaitan/django-waliki/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *django-waliki* for local development.

1. Fork the *django-waliki* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/waliki.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv waliki
$ cd waliki/
$ pip install -r requirements-test.txt
$ pip install -e .[all]
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

   Pull Request Guidelines

   Before you submit a pull request, check that it meets these guidelines:

   1. The pull request should include tests.
   2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
   3. The pull request should work for Python 2.6, 2.7, and 3.3, and for PyPy. Check https://travis-ci.org/mgaitan/waliki/pull_requests and make sure that the tests pass for all supported Python versions.

   Tips

   To run a subset of tests:

```
$ python runtests.py
```

Credits

Development Lead

- Martín Gaitán <gaitan@gmail.com>

Contributors

- Manuel Kaufmann (@humitos)

Index

C

W