Google Summer of Code 2022

Make Wagtail editor guide a stand-alone project

Overview

Abstract

The primary goal of this project is to pull out the current **Editor's Guide** and convert it into a stand-alone project. Furthermore the guide should be improved in various directions such that it targets an audience which does not necessarily have a technical background. Also adding a functionality that allows the guide to be translated will be a key part of the project.

Drawbacks of the current **Editor's Guide**

- The key drawback of the current guide is that it is highly technical for a user who is using Wagtail's editor for content-editing, permissions-management etc.
- Moreover, sending non-technical users to the technical documentation as the **Editor's Guide** is a subset of it can be a bit overwhelming as they do not have a technical background.
- One more thing that should be kept in mind is that new users can find using Wagtail complicated and therefore reference to the guide could also be added to the components of Wagtail itself.
- If someone wants to give feedback on the current guide, there is no direct way for it except submitting an issue on Github (which can be

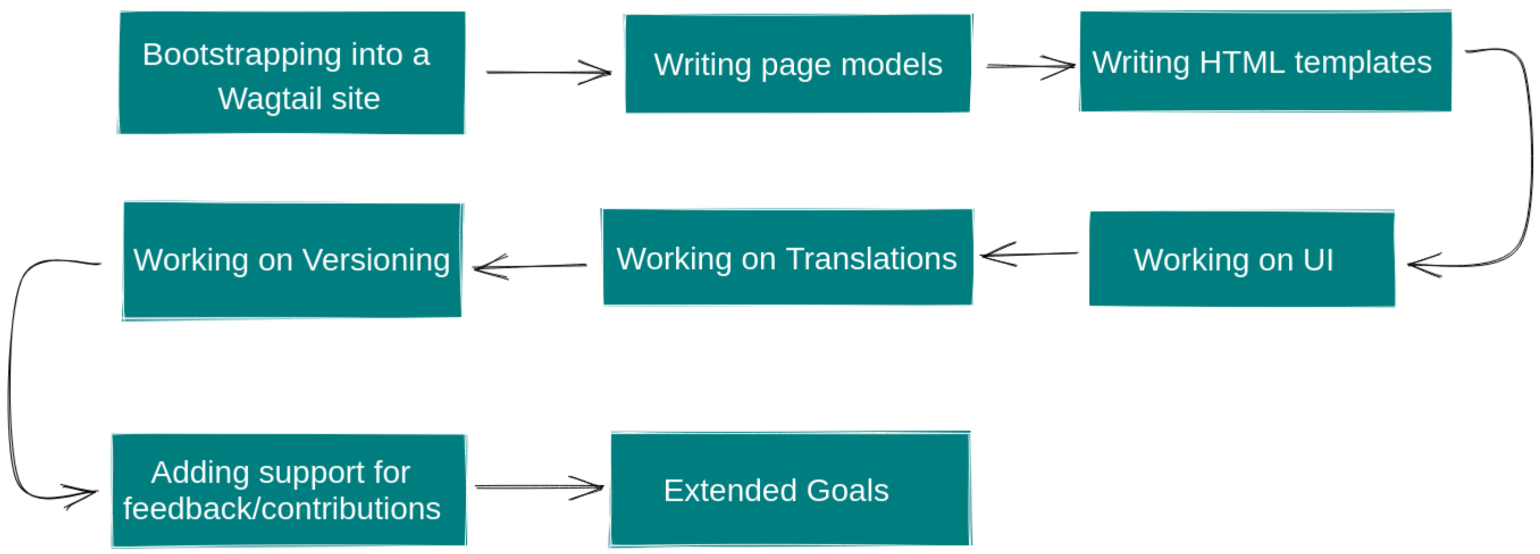
overwhelming for non-technical people and it also requires a Github account) or joining slack and giving their feedback there.

- Also the current guide does not have the translation functionality for international users.

Benefits to the community:

- The new **Editor's Guide** will enable non-technical users to understand Wagtail in a precise and simple manner which will encourage more users to use Wagtail.
- Also as the interactive nature of the new guide may allow more and more users to share their experience and feedback, the community can constantly work on possible improvements.
- As it will support translation, more people will be able to access it.
- It will overall enhance the quality of Wagtail and hence its community.

Approach



Basic Workflow

Bootstrapping into a Wagtail site:

Bootstrapping into new package:

Cookiecutter can be used with cookiecutter-wagtail-package template to bootstrap into a package with a directory structure as follows:

```
wagtail-editor's-guide/
 ├ .circleci/
 ├ .github/
 ├ wagtail_editor's_guide/
 ├ .coveragerc
 ├ .eslintrc.js
 ├ .gitignore
 ├ .nvmrc
 ├ .prettierrc.toml
 ├ CHANGELOG.md
 ├ CONTRIBUTING.md
 ├ LICENSE
 ├ MANIFEST.in
 ├ README.md
 ├ SECURITY.md
 ├ package.json
 ├ setup.py
 ├ testmanage.py
 ├ tox.ini
 ├ tsconfig.json
 └ webpack.config.js
```

Technicalities:

The approach will be simple. Firstly, to install dependencies like cookiecutter, we can use either docker or a virtual environment. For simplicity in the beginning let's use `venv` (virtual environment). After activating the environment, we will install all the dependencies and then bootstrap into a package with structure similar to above.

Initializing Wagtail:

After creating the package, we will initialize Wagtail into the project.

Technicalities:

We will start/initialize Wagtail in the “wagtail_editor’s_guide” folder itself by running `wagtail start editors_guide` and install all the dependencies from `requirements.txt` file and quickly get the vanilla site running.

Pulling out static files from current Editor:

Current documentation uses the Sphinx document generator. The task would be to pull all the content containing files like RST/Md, images etc. and transfer them to the new Wagtail-based guide.

Technicalities:

This is a simple step, hardly even requiring any coding. It can be accomplished by simply copy-pasting the files into a temporary folder on a local computer.

Writing Page Models:

Writing-modes:

An ideal documentation needs to include and be structured around its four different functions: *tutorial*, *how-to*, *explanation* and *reference*. Each of them requires a distinct mode of writing. This division makes it obvious to both author and reader what material, and what kind of material, goes where. It tells the author how to write, and what to write, and where to write it.

Technicalities:
We will create 4 page types for 4 writing-modes. For each page type we will create a new app by running `python manage.py startapp <app name>` and include the app in the `settings.py` file. We will create a total of 4 apps for accomplishing the task. Finally the directory structure will look like this:

```
wagtail-editor's-guide/
 ├ .circleci/
 ├ .github/
 ├ wagtail_editor's_guide/
 |   ├ editors_guide/
 |   ├ explanation/
 |   ├ how_to/
 |   ├ reference/
 |   ├ tutorial/
 |   ├ home/
 |   ├ migrations/
 |   ├ search/
 |   ├ static/
 |   ├ static_src/
 |   ├ test/
 |   ├ .dockerignore
 |   ├ Dockerfile
 |   ├ __init__.py
 |   ├ apps.py
 |   ├ db.sqlite3
 |   ├ manage.py
 |   ├ models.py
 |   ├ requirements.txt
 |   └ wagtail_hooks.py
 ├ .coveragerc
 ├ .eslintrc.js
 ├ .gitignore
 ├ .nvmrc
 ├ .prettierrc.toml
 ├ CHANGELOG.md
 ├ CONTRIBUTING.md
 ├ LICENSE
 ├ MANIFEST.in
 ├ README.md
 ├ SECURITY.md
 ├ package.json
 ├ setup.py
 ├ testmanage.py
 ├ tox.ini
 ├ tsconfig.json
 └ webpack.config.js
```

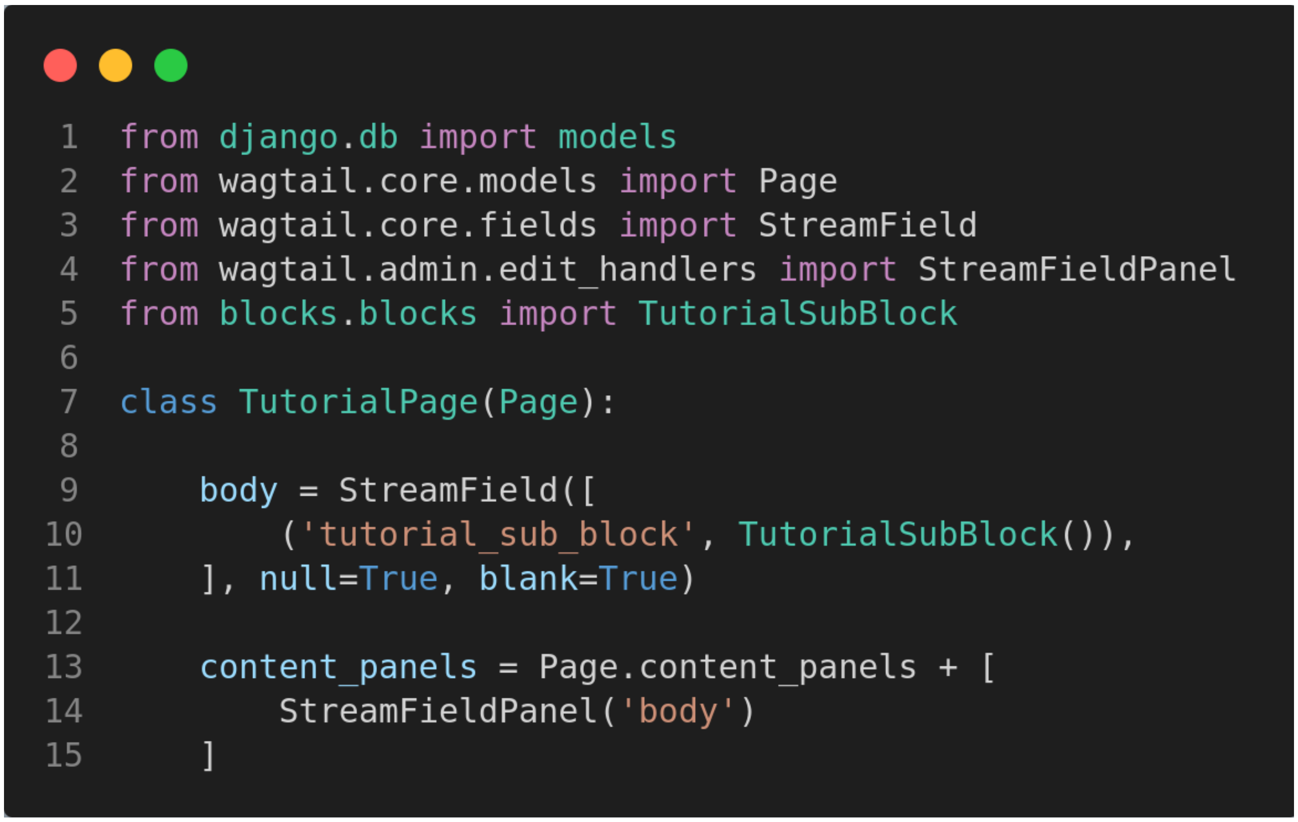
Preparing Page models:

The task is to define page models for all the above mentioned writing-modes in `models.py` in their respective apps.

Technicalities:

Writing page models will be a key part of the project as it will serve as the schema for the content. The structure/content of a page can differ largely depending on the particular topic of that page. Luckily for us, Wagtail provides a tool specifically for solving this type of problem and that is **"Streamfields".** We can divide the content into various types of blocks that Wagtail provides.

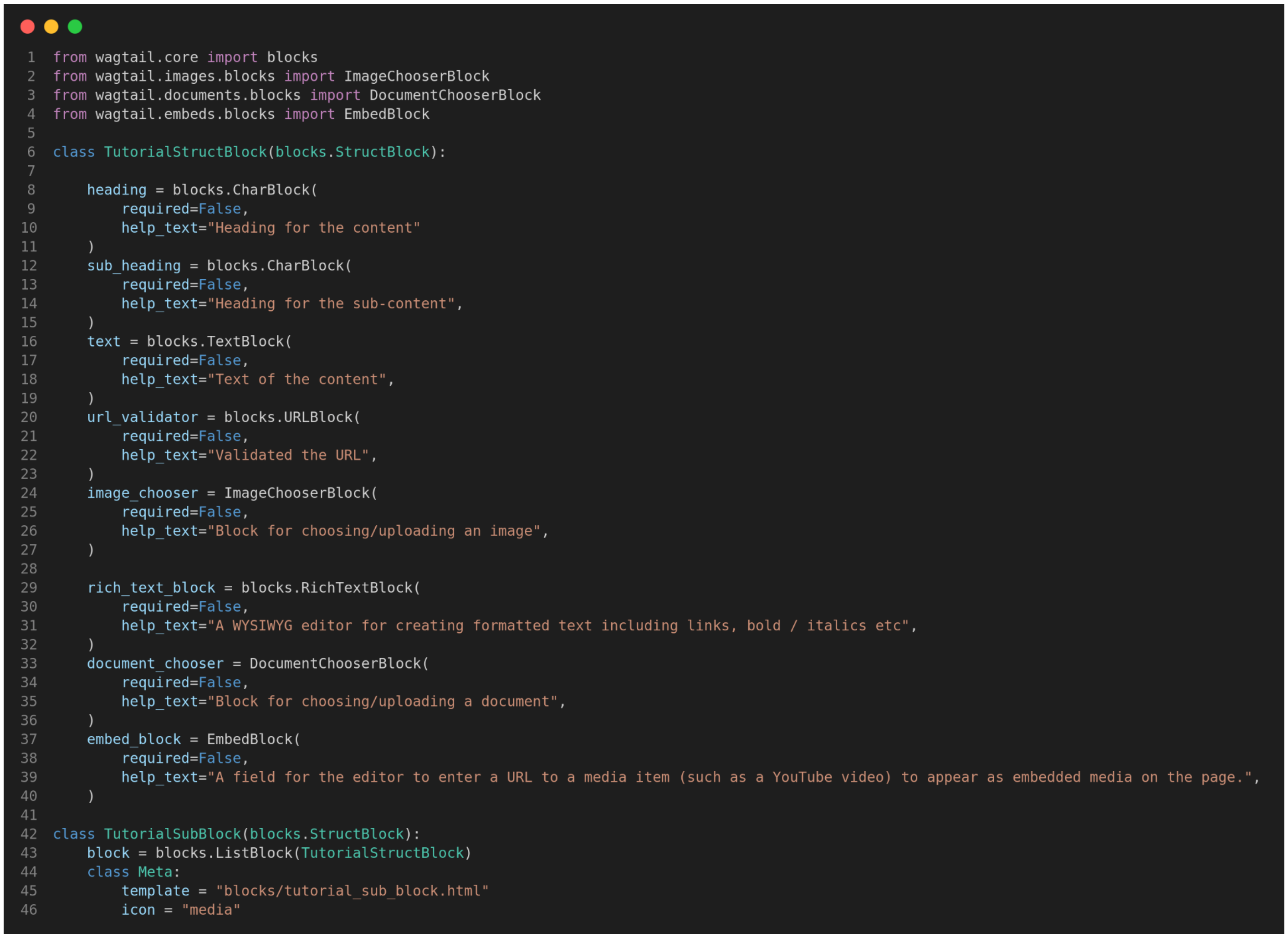
I have created another app "blocks" which contains a `blocks.py` file that stores all the `StructBlocks`. For explanation purposes I have created a *tutorial* type page with structure as follows:



```python
from django.db import models
from wagtail.core.models import Page
from wagtail.core.fields import StreamField
from wagtail.admin.edit_handlers import StreamFieldPanel
from blocks.blocks import TutorialSubBlock

class TutorialPage(Page):

    body = StreamField([
        ('tutorial_sub_block', TutorialSubBlock()),
    ], null=True, blank=True)

    content_panels = Page.content_panels + [
        StreamFieldPanel('body')
    ]
```

tutorial/models.py

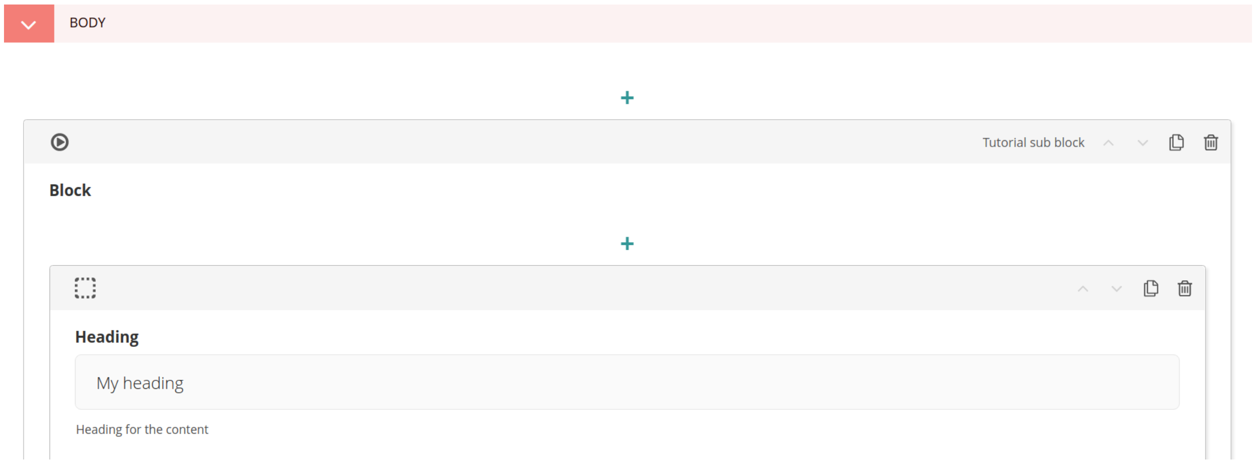
In this TutorialPage model there is a body Streamfield which contains a Struckblock 'tutorial_sub_block' which is imported from *blocks.py* from blocks app. "TutorialSubBlock" is a StructBlock consisting of a ListBlock of "TutorialStructBlock" which is also a StructBlock. There are a total of 8 blocks namely *heading, sub_heading, text, url_validator, image_chooser, rich_text_block, document_chooser* and *embed_block* in TutorialStructBlock. ListBlock allows the editor to add as many instances of a block as they like. We are considering that any content page can be divided into these struct-blocks as each struct-block provides almost all necessary fields required to write content. Every field's required parameter is set to False, therefore the blocks can be used with flexibility. The definition of TutorialSubBlock and TutorialStructBlock is shown below:



```python
from wagtail.core import blocks
from wagtail.images.blocks import ImageChooserBlock
from wagtail.documents.blocks import DocumentChooserBlock
from wagtail.embeds.blocks import EmbedBlock

class TutorialStructBlock(blocks.StructBlock):

    heading = blocks.CharBlock(
        required=False,
        help_text="Heading for the content"
    )
    sub_heading = blocks.CharBlock(
        required=False,
        help_text="Heading for the sub-content",
    )
    text = blocks.TextBlock(
        required=False,
        help_text="Text of the content",
    )
    url_validator = blocks.URLBlock(
        required=False,
        help_text="Validated the URL",
    )
    image_chooser = ImageChooserBlock(
        required=False,
        help_text="Block for choosing/uploading an image",
    )

    rich_text_block = blocks.RichTextBlock(
        required=False,
        help_text="A WYSIWYG editor for creating formatted text including links, bold / italics etc",
    )
    document_chooser = DocumentChooserBlock(
        required=False,
        help_text="Block for choosing/uploading a document",
    )
    embed_block = EmbedBlock(
        required=False,
        help_text="A field for the editor to enter a URL to a media item (such as a YouTube video) to appear as embedded media on the page.",
    )

class TutorialSubBlock(blocks.StructBlock):
    block = blocks.ListBlock(TutorialStructBlock)
    class Meta:
        template = "blocks/tutorial_sub_block.html"
        icon = "media"
```

blocks/blocks.py

The icon set in Meta class is temporary and can be changed whenever we want. There is other data apart from icons that can be set in Meta according to the requirements of the page.
Below I have attached screenshots of the Tutorial Page editor in admin:


BODY

Tutorial sub block

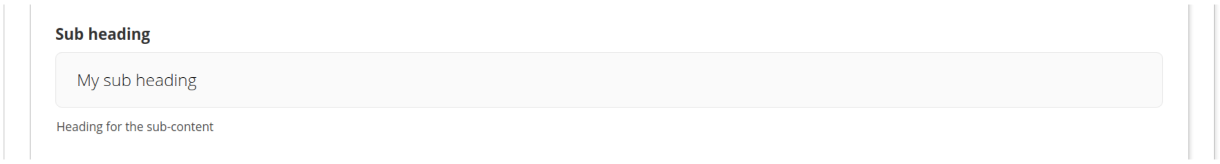
Block

Heading

My heading

Heading for the content

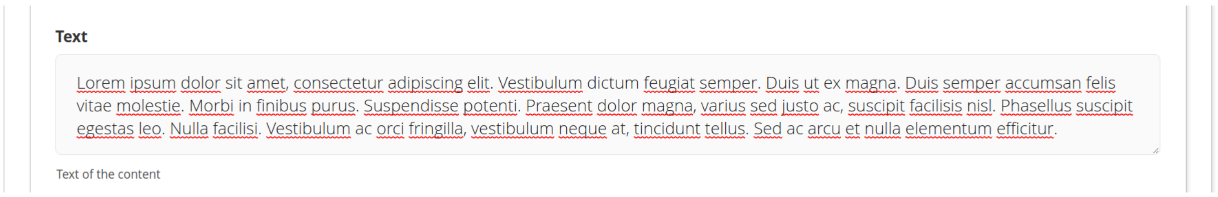
Heading


Sub heading

My sub heading

Heading for the sub-content

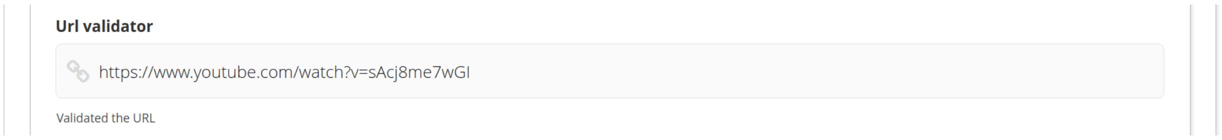
Subheading


Text

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum dictum feugiat semper. Duis ut ex magna. Duis semper accumsan felis vitae molestie. Morbi in finibus purus. Suspendisse potenti. Praesent dolor magna, varius sed justo ac, suscipit facilisis nisl. Phasellus suscipit egestas leo. Nulla facilisi. Vestibulum ac orci fringilla, vestibulum neque at, tincidunt tellus. Sed ac arcu et nulla elementum efficitur.

Text of the content

Text


Url validator

https://www.youtube.com/watch?v=sAcj8me7wGI

Validated the URL

URL

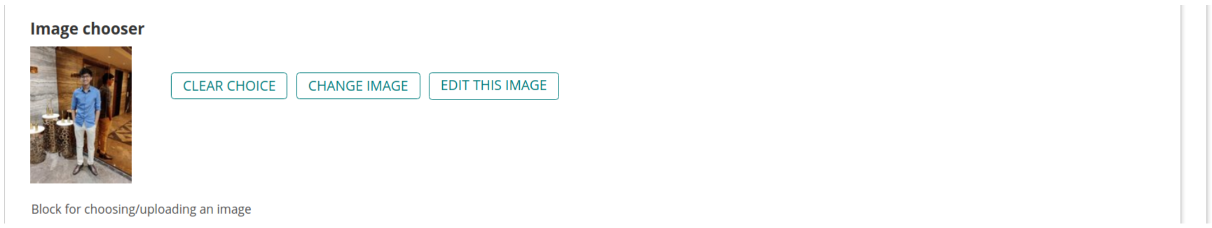
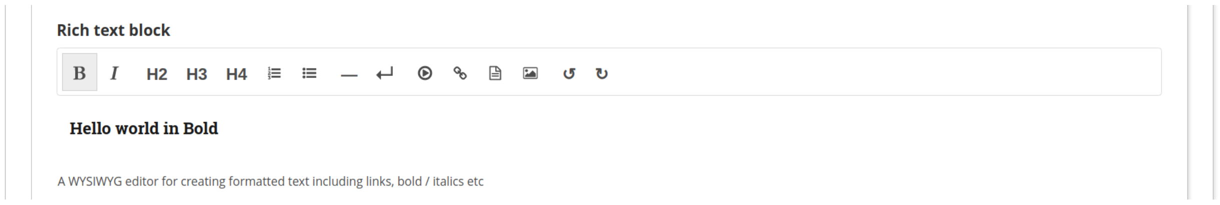


Image chooser

Block for choosing/uploading an image

Image

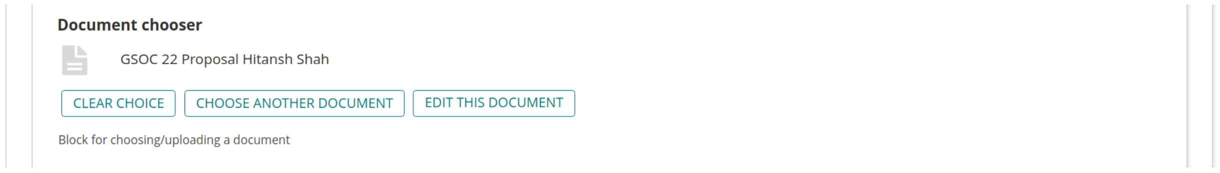


Rich text block

Hello world in Bold

A WYSIWYG editor for creating formatted text including links, bold / italics etc

Rich Text Field



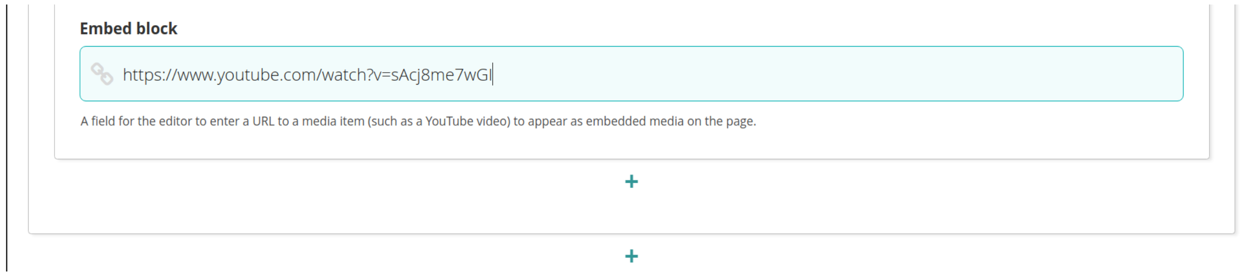
Document chooser

GSOC 22 Proposal Hitansh Shah

Block for choosing/uploading a document

Document

(Note that Document chooser block may or may not have any application in this page. I have put it just for demo purpose)



Embed block

https://www.youtube.com/watch?v=sAcj8me7wGI

A field for the editor to enter a URL to a media item (such as a YouTube video) to appear as embedded media on the page.

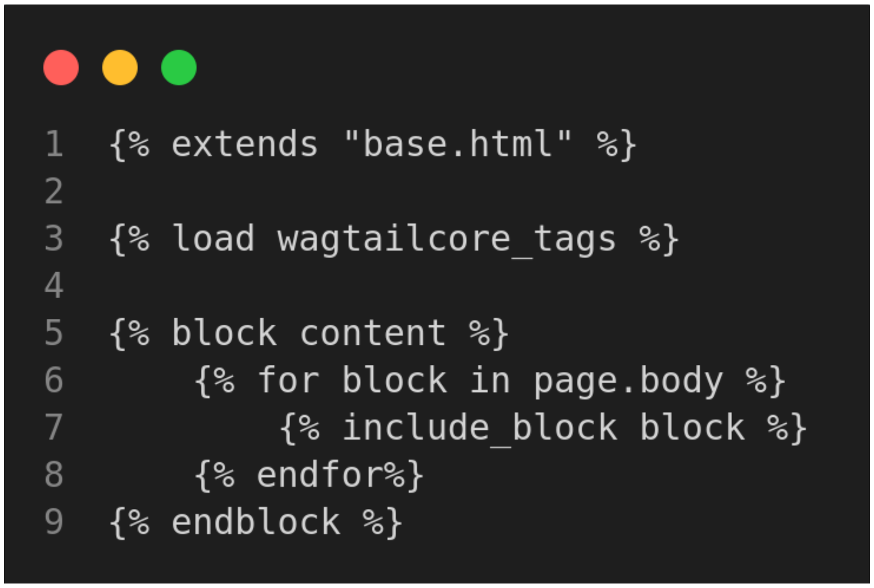
Embed Block

The above shown block was just one in the list, we can create as many instances of it as we want. Furthermore all of the instances are still inside a parent StructBlock and we can create as many parent StructBlocks too. This structure provides flexibility with templates. The structure/models may still have scope to improve and we will find that out eventually as we start adding content to it.

Writing HTML Templates:

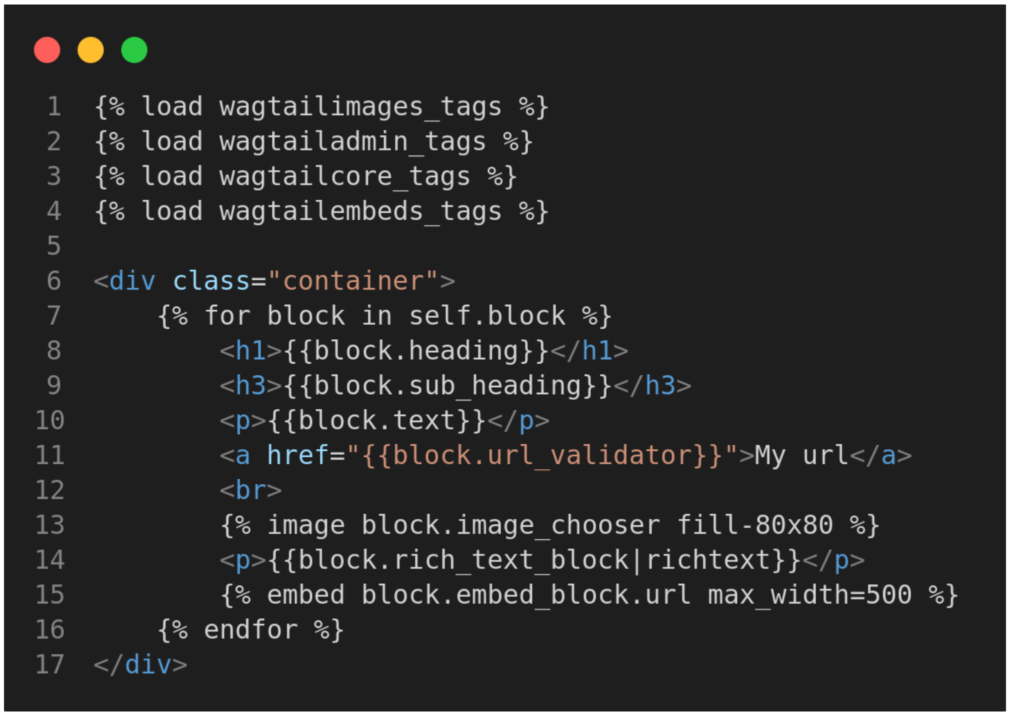
The task will be to prepare HTML templates for respective pages. Here the primary goal will be to extract the content from the blocks of the page and display it in an ordered and structured manner.

I have made basic templates for displaying all the blocks on the page. The design part will be dealt with in the UI section. Currently I have placed all the templates inside the templates directory of the main app with name <app_name>/<template_name>.html and they all extend “base.html”. I have currently written templates for tutorial/models.TutorialPage and blocks/blocks.TutorialSubBlock. I am attaching the code snippets for both the templates:



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

{% load wagtailcore_tags %}

{% block content %}
    {% for block in page.body %}
        {% include_block block %}
    {% endfor%}
{% endblock %}
```

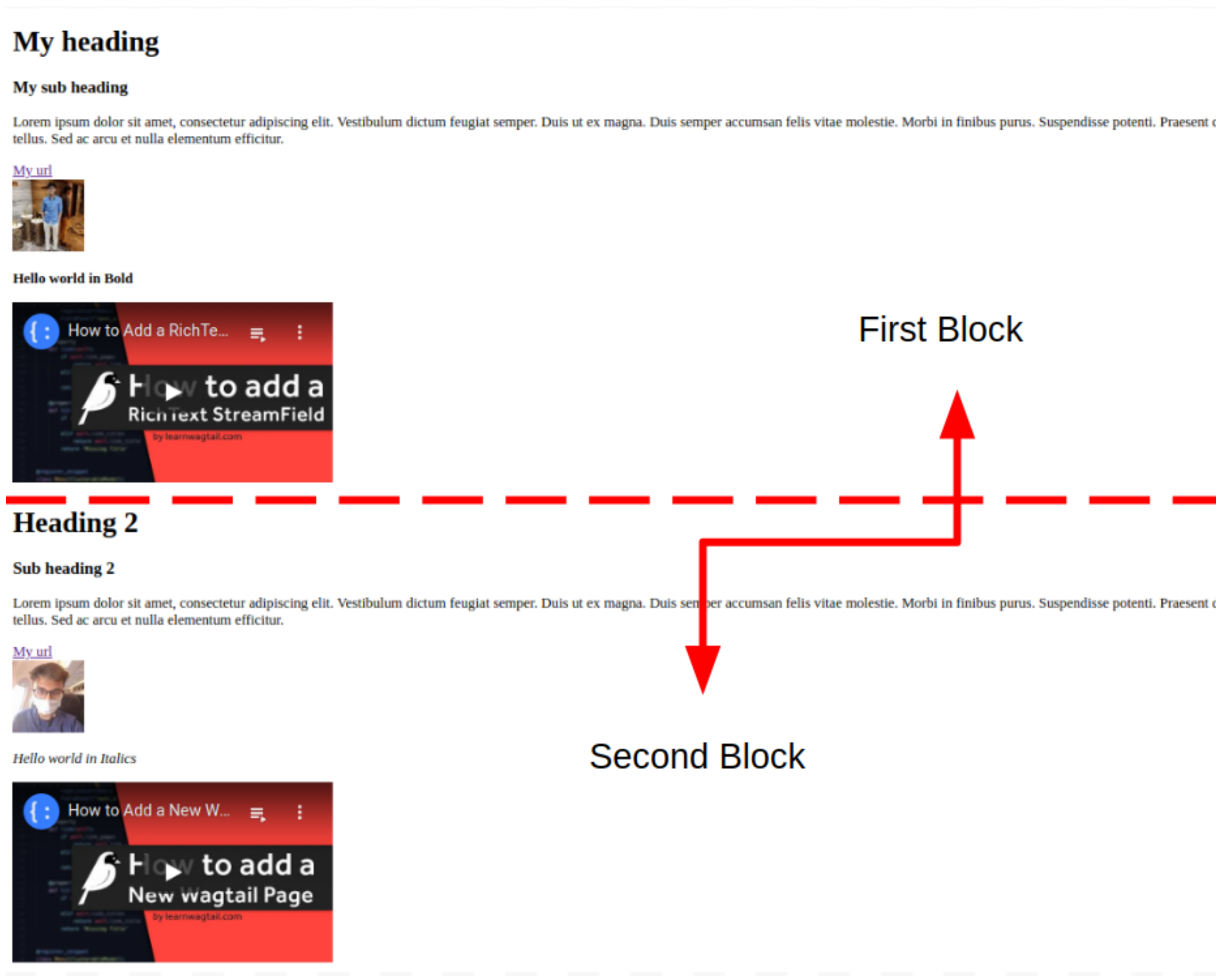
tutorial/models.TutorialPage



```
{% load wagtailimages_tags %}
{% load wagtailadmin_tags %}
{% load wagtailcore_tags %}
{% load wagtailembeds_tags %}

<div class="container">
    {% for block in self.block %}
        <h1>{{block.heading}}</h1>
        <h3>{{block.sub_heading}}</h3>
        <p>{{block.text}}</p>
        <a href="{{block.url_validator}}">My url</a>
        <br>
        {% image block.image_chooser fill-80x80 %}
        <p>{{block.rich_text_block|richtext}}</p>
        {% embed block.embed_block.url max_width=500 %}
    {% endfor %}
</div>
```

blocks/blocks.TutorialSubBlock

The result till now is described below in the photo:



My heading

My sub heading

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum dictum feugiat semper. Duis ut ex magna. Duis semper accumsan felis vitae molestie. Morbi in finibus purus. Suspendisse potenti. Praesent c
tellus. Sed ac arcu et nulla elementum efficitur.

My url

Hello world in Bold

Heading 2

Sub heading 2

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vestibulum dictum feugiat semper. Duis ut ex magna. Duis semper accumsan felis vitae molestie. Morbi in finibus purus. Suspendisse potenti. Praesent c
tellus. Sed ac arcu et nulla elementum efficitur.

My url

Hello world in Italics

In the above picture, I have used 2 blocks of content. Similarly several blocks can be used as per the content size and structure. All the fields in every block are non-required (meaning they can be left blank). This provides flexibility to the content editor as they can add multiple blocks even if they want to use only a few fields of some blocks.

Working on UI

Extracting theme from MKDocs

I plan to use MKDocs Material Design. Currently they have their own documentation system integrated with their theme. However I feel that it will be easy to extract the templates and static files from their Github repository.

Wiring up and testing the theme in another project environment

Once the files are extracted, I plan to build a simple static web page using the extracted template files and see how it works. I plan to install Wagtail in this project so that I can test using Wagtail template tags. Once everything is set up, I will start modifying the CSS styles of the theme to match with wagtail.org and docs.wagtail.org.

Integrating the theme in the new Editor's guide

This step involves integrating the theme in our main project. Since every template will extend "base.html", this file will be the key to templates workflow. Next I will start bringing in the static files and connecting them to the appropriate templates.

Alternatives

The above approach could be messy and time consuming. Therefore, I will also try to find any suitable free to use and easy to integrate Bootstrap themes in the meantime.

Working on Translations

Internationalization of the new guide is a key part of the project. Since it will be built on top of Wagtail, we can use Wagtail's inbuilt translation tools along with Transifex to provide easy interface and work-flow management to translators.

Enabling Internationalization in the project

As the project is built on Wagtail, we can follow this Wagtail's tutorial to enable internationalization and configuring all the required settings and options.

Integrating UI service for translation

Wagtail provides Wagtail-localize and Simple-Translation to serve this purpose. I suggest using Wagtail-Localize as it provides an easy integration plus it has some more features compared to Simple-Translation and also the content editor can directly translate pages in the Wagtail admin.

Localization in Code strings

We can use Django's functionality to enable internationalization and localization of code strings.

Crowd-sourcing

If we want to allow other people to be able to contribute in translations, then we can integrate **Transifex** into the project. It provides an easy to use and clean UI as well as a proper workflow for translators.

Working on Versioning

The new guide should support version control. There should be an option to view older versions of the guide. Basically we want to take snapshots of the database and store them according to their version. Hence, when choosing a version, the content from that version of the database will be loaded everywhere in the site.

Approach

I suggest using Django reversion API. It is a PIP package and it provides various tools to build a version controlled application. Its documentation is pretty clear and straightforward and it is not too complicated to implement. It involves registering models with django-reversion via a couple of methods and then creating revisions of the models. Revisions are basically changes done to the model instances grouped together as a single unit. We can then load revisions with the APIs it provides.

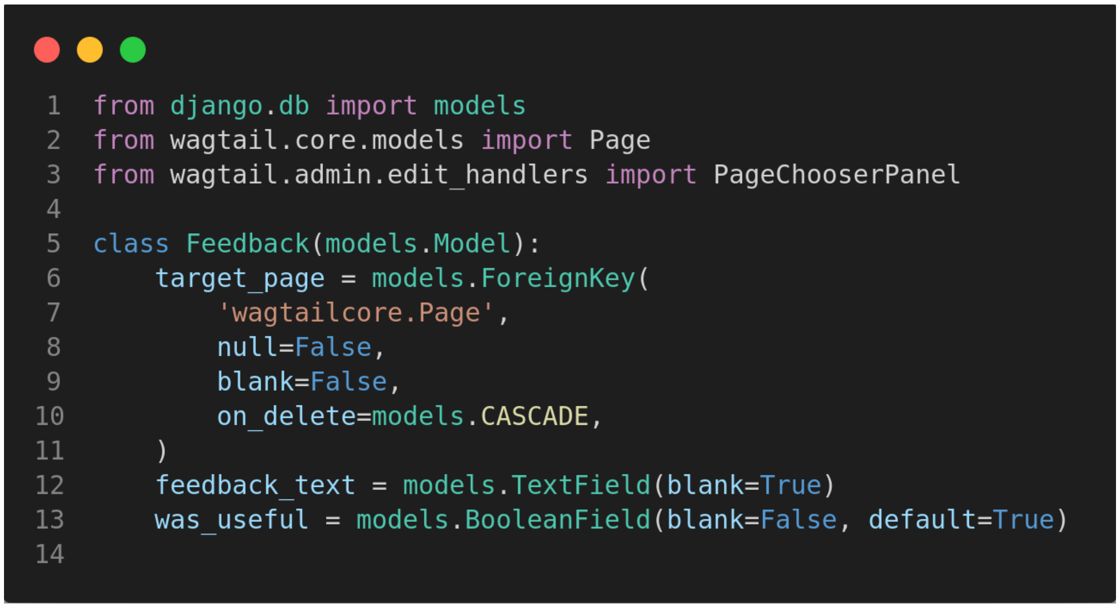
The above approach is the most doable and precise thing I found about versioning. I intend to research more on this in the Pre-Gsoc period and test it on a test application. If this does not serve the purpose or proves to be unworkable, I am always ready to try different approaches.

Feedback Mechanism

We need to enable users/readers to give feedback on the documentation. This way we can maintain a record of what improvements we need to make in order to improve user experience. Moreover, the feedback mechanism should be as simple as possible so that it allows more users to provide feedback.

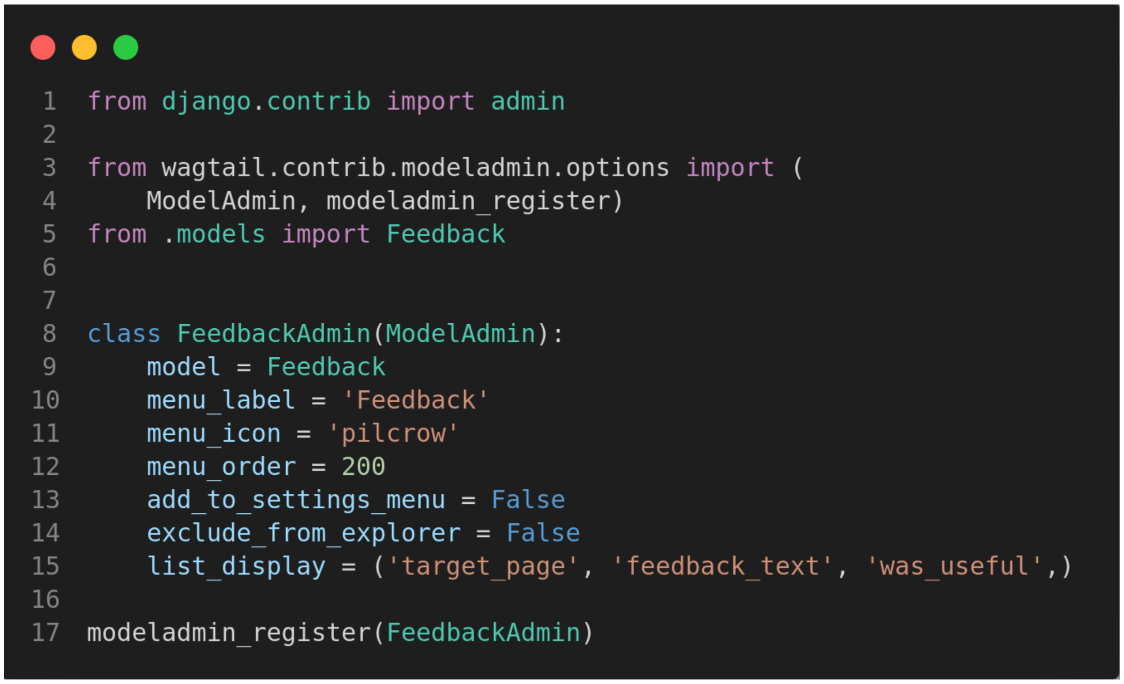
Implementation

We can create a separate app for feedbacks called “feedback”. In that we can define a model “Feedback” which extends Django’s Model. Then we can use Wagtail’s **ModelAdmin** to create and register “FeedbackAdmin” in *admin.py* of “feedback” app. I am attaching the code snippet for this implementation below:



```python
from django.db import models
from wagtail.core.models import Page
from wagtail.admin.edit_handlers import PageChooserPanel

class Feedback(models.Model):
    target_page = models.ForeignKey(
        'wagtailcore.Page',
        null=False,
        blank=False,
        on_delete=models.CASCADE,
    )
    feedback_text = models.TextField(blank=True)
    was_useful = models.BooleanField(blank=False, default=True)

```

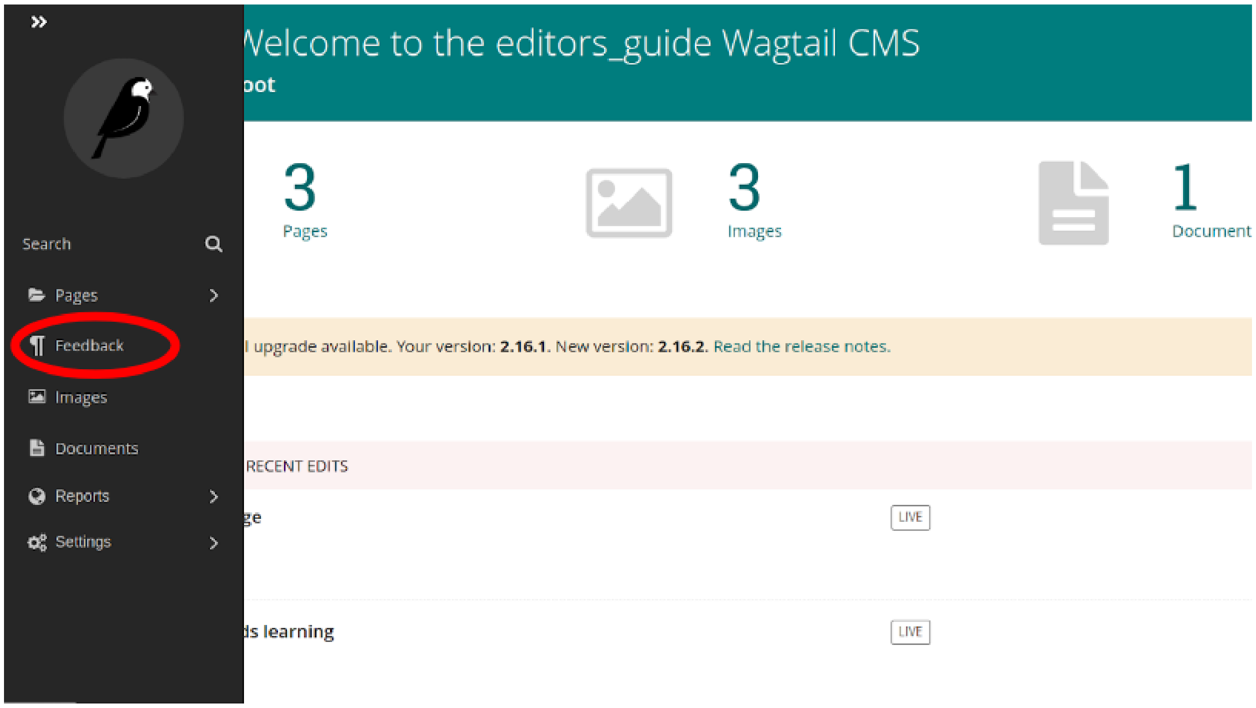
feedback/models.py



```python
from django.contrib import admin

from wagtail.contrib.modeladmin.options import (
    ModelAdmin, modeladmin_register)
from .models import Feedback


class FeedbackAdmin(ModelAdmin):
    model = Feedback
    menu_label = 'Feedback'
    menu_icon = 'pilcrow'
    menu_order = 200
    add_to_settings_menu = False
    exclude_from_explorer = False
    list_display = ('target_page', 'feedback_text', 'was_useful',)

modeladmin_register(FeedbackAdmin)
```

feedback/admin.py

The result of above implementation is show below:



Velcome to the editors_guide Wagtail CMS
oot

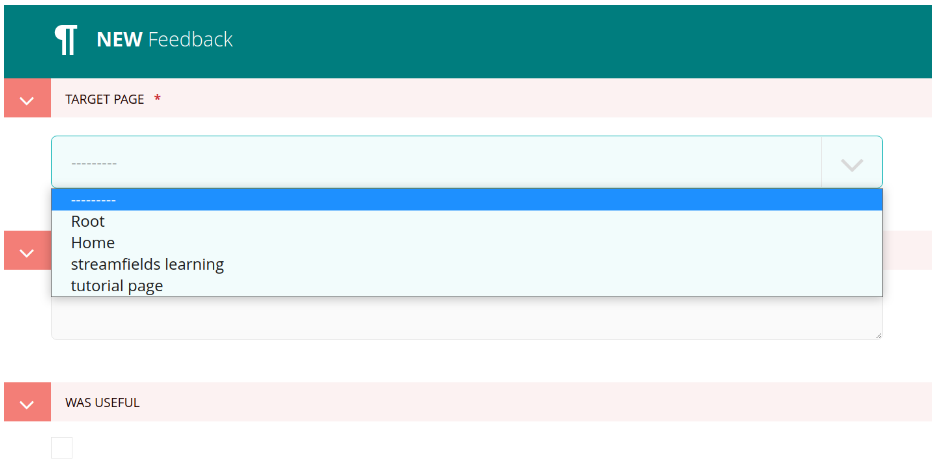
3 Pages | 3 Images | 1 Document

l upgrade available. Your version: **2.16.1**. New version: **2.16.2**. Read the release notes.

RECENT EDITS

ge LIVE

ds learning LIVE

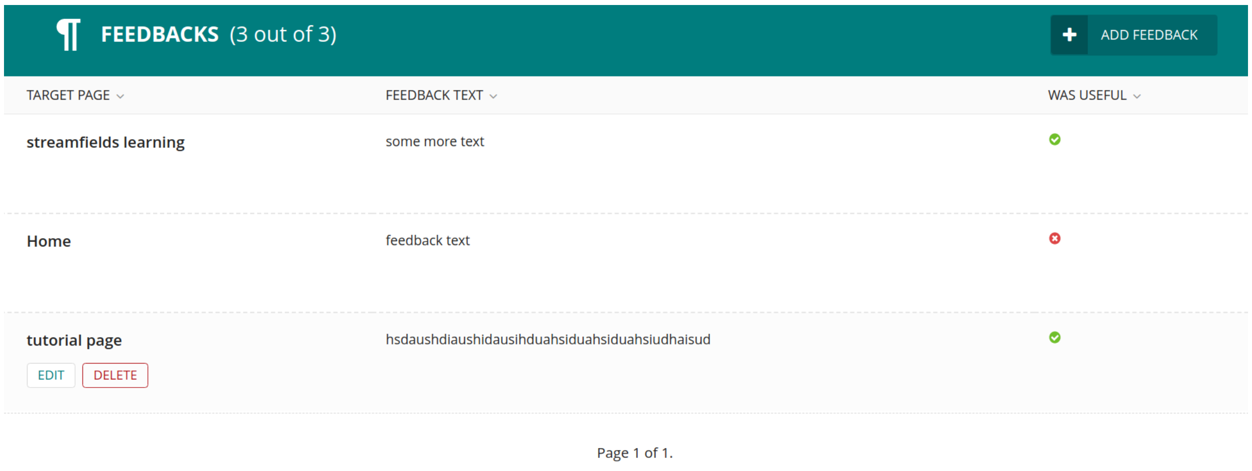


NEW Feedback

TARGET PAGE *

- ---------
- Root
- Home
- streamfields learning
- tutorial page

WAS USEFUL

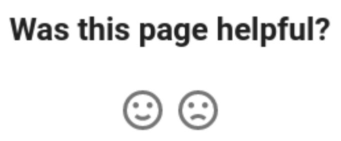


FEEDBACKS (3 out of 3)

TARGET PAGE	FEEDBACK TEXT	WAS USEFUL
streamfields learning	some more text	✓
Home	feedback text	✗
tutorial page	hsdaushdiaushidausihduahsiduahsiduahsiudhaisud	✓

Page 1 of 1.

We can create a “was this page helpful?” just like MKDocs and link it to a form which on submitting will create a new “Feedback”.



If we further want detailed statistics of feedbacks then we can use some external packages like django-admin-tool-stats.

The above approach was just one way to do this. We can also use **Wagtail Forms** for this. Every approach may have its own pros and cons. Maybe we can discuss this a bit more and as always I am ready to try out different approaches and see what suits best.

Contributions

Apart from the feedback, many people are motivated towards improving the guide and contributing to the community. There should be a way to authorize them and allow them to propose their changes. Apart from the coding part, many people would want to contribute to content. Therefore the goal is to provide them an easy workflow to authorize themselves and edit the content.

Workflow

We can use django-allauth for integrating social auth to Django admin. Once the user has got access, they can suggest their changes directly through Wagtail-editor. The suggested changes will go to the moderator for checking and if they approve, the changes would be made.

Also as discussed before, if a contributor is active and seems trustworthy, they can be invited for the role of moderator.

Development

- As discussed before, we will use Git and Github for code, version-control, collaboration, issue tracking etc.
- A CI/CD pipeline will be set up to test and deploy on a regular basis (like Nighty CI maybe).
- Documentation will be written for development and contribution to the project.
- Proper tests will be written for a smooth workflow. Most probably we will use Wagtail and Django's inbuilt testing libraries.

Schedule/Timeline

Pre-GSOC Period	
	April 20, 2022 - May 19, 2022
April 20, 2022 - May 19, 2022	➢ Work on issues ➢ Develop a deeper understanding of Wagtail ➢ Learn concepts like CI/CD thoroughly. ➢ Research more on some topics like versioning, feedback, UI etc. ➢ Add more details to the GSOC project plan.
Community Bonding Period	
	May 19, 2022 - June 12, 2022
May 20, 2022 - June 12, 2022	➢ Talk with mentors about the project details.

	➢ Discuss the workflow of the project with the mentors and continuously take their feedback.
Coding Period	
June 13, 2022 - September 19, 2022	
June 13, 2022 - June 25, 2022	➢ Planning ➢ Writing models ➢ Transferring some content for testing.
June 26, 2022 - July 10, 2022	➢ Discussing and testing themes with mentors. ➢ Preparing design outline after discussion. ➢ Preparing templates
July 11, 2022 - July 24, 2022	➢ Enabling internationalization in the project ➢ Integrating UI service for translations. ➢ Localization in Code strings ➢ Working on crowdsourcing
Phase 1 Evaluations (July 25, 2022 - July 29, 2022)	
Work Period (July 25, 2022, - Sep 04, 2022)	
July 25, 2022 - August 4, 2022	➢ Exploring tools and options like Django-reversion for version control. ➢ Implementing a version control system with a proper workflow.
August 5, 2022 - August 20, 2022	➢ Working on feedback mechanism ➢ Working on adding social auth for contributions
August 21, 2022 - End of coding period	➢ Working on remaining tests (I intend to write tests in parallel from the start) ➢ Building CI/CD pipeline ➢ Work on bugs (if any). ➢ Working on suggested improvements ➢ Writing documentation for setting up project locally (maybe, depending on mentors) ➢ If time remains then working on extended goals.

About Me

I am Hitansh Shah, a sophomore pursuing Mathematics and Computing from Indian Institute of Technology Varanasi (BHU) and I am expected to graduate in the year 2025.

I was introduced to the world of programming in my first year of college. I began to pursue various topics and areas of programming and became a part of the reputed coding club of our college, Club of Programmers (COPS) IIT BHU.

I always try to learn new things and keep experimenting around various fields/subfields to see what best fits my interest. Apart from that, I love to contribute to opensource. I have also made some personal projects involving various tech stacks, ideas and use-case which you can find on my github here.

Questions

- **Why do you want to work on Wagtail?**
  Firstly Wagtail is built on top of Django and I want to polish my Django skills. Secondly, Wagtail has a nice and supportive community which motivates me to work with the community.
- **Why do you want to work on solving this problem?**
  This project involves building a Wagtail site and working with several features such as translations, versions etc. This project provides me with a bunch of learning opportunities that will help me grow my knowledge in various fields. Also Wagtail is a very popular CMS and therefore by working on this project, I can develop a deeper understanding of Wagtail.

- **Preferred languages to communicate with?**
  My first preference will always be English. I am fully capable of reading, writing and talking in English. Apart from that, I am also comfortable in Hindi.
- **Available hours for meeting with mentors?**
  I usually have college classes in the morning and afternoon. Therefore during these days I will most probably be available during 7pm-12am IST (UTC+05:30). However I am having summer vacation from May 12, 2022 - July 20, 2022, therefore I can attend meetings during daytime and afternoon too. There may be some days when I will not be able to attend meetings but I will make sure to let mentors know about this as early as possible.

Experience and contributions

I am proficient in MERN full-stack web development, familiarity with frameworks like NextJS, building REST APIs, Django, python and a little bit of C/C++ and Java.

Contributions in Wagtail

Issue	Pr	Description	Status
#8176	#8177	Splitting the make format and make lint tasks into client and server categories. This will help users to save time when working with only specific files of a category.	Closed. Merged with some tweaks in a8106e5
#8107	#8166	Adding image management commands to purge and regenerate image renditions. Updating docs and adding tests for this command.	Merged

Contributions in other open source organizations

Organization	Repository	PR	Status
Internet Archive	openlibrary	#6157	Merged
Internet Archive	openlibrary	#5996	Merged
Internet Archive	openlibrary	#6045	Closed
Internet Archive/ open-book-genome-project	sequencer	#84	Merged

Contact Information

Github username: Hitansh-Shah

LinkedIn: Hitansh Shah

Slack: Hitansh Shah

Email: shah.hitanshsanjay.mat20@itbhu.ac.in

Alternate Email: hitanshshah123@gmail.com

Contact Number: +91 9624776104

Country: India

Time Zone: IST (UTC+05:30)