The Definitive Guide

Next.js

Wagtail

MichaelYin@Accordbox

Contents

Chapter 1

Introduction

1.1 Objectives

This book will teach you how to build a blog with Next.js, Wagtail CMS and Static Generation, which has good performance and good SEO.

By the end of this course, you will be able to:

1. Understand `Docker` and use `Docker Compose` to do development
2. Create blog models to work with Wagtail.
3. Learn how to write serializer for Django models.
4. Build a REST API for Wagtail CMS
5. Use the `factory` package to help create test data
6. Test the REST API and generate test coverage report
7. Create Next.js project with `create-next-app`
8. Learn React Function Component, and React hooks.
9. Understand how Next.js page route works.
10. Make Wagtail preview work with the Next.js.
11. Build comment system based on `django-contrib-comments` which support `Generic Relations`
12. Use `Tribute.js` to add `Mention` and `Emoji` support to the comment form.
13. Learn to use `SWR` to build lazy load comment list.
14. Test React component using `Jest` and `@testing-library` family of packages.
15. Deploy the Next.js to the Netlify.
16. Deploy the backend API to DigitalOcean

1.2 Tech

- Python 3.8
- Django 3.2
- Wagtail 2.14
- Node 14

- Next.js 11
- Bootstrap 5
- React Function Component, React Hooks
- SWR
- Tribute.js
- Jest

1.3 What is included

1. A PDF ebook which contains about 40 chapters.
2. 20 screenshots and 7 diagrams, all created by me.
3. **The source code (You can get it in the `Extra files`)**

1.4 How to use the source code

1.4.1 Edit /etc/hosts

Below command is for Mac, but you can ask Google for help if you use other OS.

```
$ sudo vi /etc/hosts

# add to the bottom
127.0.0.1 www.local.test
127.0.0.1 api.local.test

# clear DNS cache
$ sudo killall -HUP mDNSResponder
```

```
$ ping www.local.test
```

1.4.2 Backend API

You can use code below to run dev application on your local env.

You need Docker and Docker Compose and you can install it here Get Docker[1]

```
$ cd nextjs_wagtail_project
$ docker-compose up --build
```

Now open a new terminal to import data and change password.

```
$ docker-compose exec web python manage.py load_initial_data
# change password for admin
$ docker-compose exec web python manage.py changepassword admin
```

Now you can check on

- http://api.local.test:8000/cms-admin

[1] https://docs.docker.com/get-docker/

1.4.3 Next.js

It is recommended to use nvm[2] to install `node` on your local env.

```
$ node -v
v14.17.4
```

```
$ cd frontend
$ npm install -g yarn
$ yarn -v
1.22.11
$ yarn install
$ yarn dev
```

Now you can check on

- http://www.local.test:3000/

1.5 Demo

The demo is also online if you want to check.

- Wagtail Demo[3]

1.6 What if you have problem or suggestions

If you meet problem, please check FAQ first (you can find it at the end of the book)

If you want to talk with me, please send email to

michaelyin@accordbox.com

1.7 Changelog

1.7.1 1.0.0

- 2021-08-16: First release
- 2021-08-14: Review done
- 2021-07-30: Draft finished
- 2021-07-03: Start writing

[2] https://github.com/nvm-sh/nvm
[3] https://nextjs-wagtail.accordbox.com/

Chapter 2

Setup Wagtail Project

2.1 Objectives

By the end of this chapter, you should be able to:

1. Create a `Django` project and update the project setting file.
2. Import `Wagtail CMS` and make it work with your Django project.

2.2 Create Django Project

```
$ mkdir nextjs_wagtail_project && cd nextjs_wagtail_project
$ python3 -m venv env
$ source env/bin/activate
```

You can also use other tools such as Poetry[4] or Pipenv[5]

Create *requirements.txt*

```
django==3.2
```

```
(env)$ pip install -r requirements.txt
(env)$ env/bin/django-admin.py startproject nextjs_wagtail_app .
```

You will see structure like this

```
(env)$ tree -L 1
.
├── env
├── manage.py
├── nextjs_wagtail_app
└── requirements.txt
```

2.3 Import Wagtail

Add `Wagtail CMS` to *requirements.txt*.

[4] https://python-poetry.org/
[5] https://pipenv.pypa.io/

```
wagtail==2.14
```

```
(env)$ pip install -r requirements.txt
```

Add the apps to `INSTALLED_APPS` in the *nextjs_wagtail_app/settings.py* file:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'wagtail.contrib.forms',                                    # new
    'wagtail.contrib.redirects',
    'wagtail.embeds',
    'wagtail.sites',
    'wagtail.users',
    'wagtail.snippets',
    'wagtail.documents',
    'wagtail.images',
    'wagtail.search',
    'wagtail.admin',
    'wagtail.core',

    'modelcluster',
    'taggit',
]
```

Add the middleware to `MIDDLEWARE` in the *nextjs_wagtail_app/settings.py* file:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',

    'wagtail.contrib.redirects.middleware.RedirectMiddleware',                  # new
]
```

Add other settings to the bottom of the *nextjs_wagtail_app/settings.py* file:

```
STATIC_URL = '/static/'
STATIC_ROOT = str(BASE_DIR / 'static')

MEDIA_URL = '/media/'
MEDIA_ROOT = str(BASE_DIR / 'media')

WAGTAIL_SITE_NAME = 'My Project'
```

Next, let's edit *nextjs_wagtail_app/urls.py*

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings

from wagtail.core import urls as wagtail_urls
from wagtail.admin import urls as wagtailadmin_urls
```

```
from wagtail.documents import urls as wagtaildocs_urls

urlpatterns = [
    path('admin/', admin.site.urls),

    path('cms-admin/', include(wagtailadmin_urls)),
    path('documents/', include(wagtaildocs_urls)),

    # For anything not caught by a more specific rule above, hand over to
    # Wagtail's serving mechanism
    path('', include(wagtail_urls)),
]

if settings.DEBUG:
    from django.conf.urls.static import static
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

1. Wagtail's admin is on `cms-admin`
2. Do not forget to put `path('', include(wagtail_urls))` at the end of the `urlpatterns`

2.4 Run DevServer

Now, all the config is done, let's run the Wagtail app

```
# migrate db.sqlite3
(env)$ ./manage.py migrate

# runserver
(env)$ ./manage.py runserver

Django version 3.2, using settings 'nextjs_wagtail_app.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

1. Now if you visit http://127.0.0.1:8000/ you will see `Welcome to your new Wagtail site!`.
2. The welcome page is created by `Wagtail migration` and you can check the source code here[6]
3. You will see `db.sqlite3` is created at the project directory, here we did not specify Django to use other db so sqlite is used by default.

2.5 Reference

Integrating Wagtail into a Django project[7]

[6] https://github.com/wagtail/wagtail/blob/v2.10.2/wagtail/core/migrations/0002_initial_data.py#L30
[7] https://docs.wagtail.io/en/latest/getting_started/integrating_into_django.html

Chapter 3

Dockerizing Wagtail App

3.1 Objectives

By the end of this chapter, you should be able to:

1. Understand Docker Compose and the benefits.
2. Use Docker Compose to create and manage Wagtail, Postgres services, and do development.

3.2 Install Docker Compose

> Docker Compose is a tool for defining and running multi-container Docker applications.[47] It uses YAML files to configure the application's services and performs the creation and start-up process of all the containers with a single command.
>
> The docker-compose CLI utility allows users to run commands on multiple containers at once, for example, building images, scaling containers, running containers that were stopped, and more.

First, please download and install Docker Compose[8] if you haven't already done so.

```
$ docker --version
Docker version 19.03.13, build 4484c46d9d

$ docker-compose --version
docker-compose version 1.27.4, build 40524192
```

3.3 Config File Structure

Let's start with our config file structure, this can help you better understand the whole workflow:

```
├── compose
│   └── local
│       └── django
│           ├── Dockerfile
│           ├── entrypoint
│           └── start
├── docker-compose.yml
├── manage.py
```

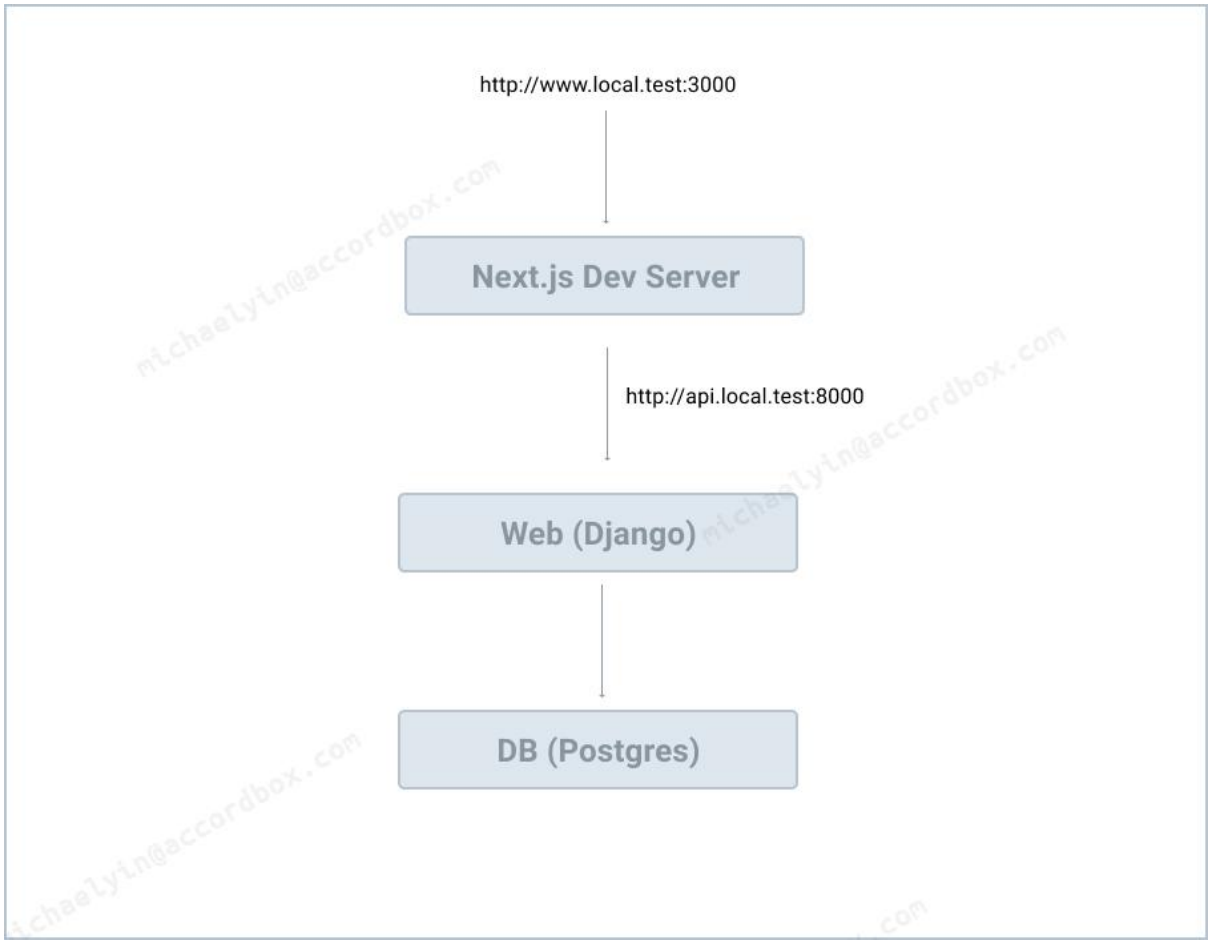
[8] https://docs.docker.com/compose/install/#install-compose

```
├── nextjs_wagtail_app
└── requirements.txt
```

You will see we have config files `docker-compose.yml` and some files in `compose` directory, you do not need to create them for now. I will talk about them with more details in the coming sections.

Note: The config file structure come from cookiecutter-django[9], which is a great project for people who want to learn Django.

3.4 Compose file



Compose file is a YAML file to configure your application's services.

When you run `docker-compose` command, if you do not specify `Compose file`, the default file is `docker-compose.yml`, that is why we create `docker-compose.yml` at root directory of Django project, because it can save us time when typing command during development.

Let's add *docker-compose.yml*

```yaml
version: '3.7'

services:
  web:
    build:
      context: .
      dockerfile: ./compose/local/django/Dockerfile
```

[9] https://github.com/pydanny/cookiecutter-django

```
    image: nextjs_wagtail_app_web
    command: /start
    volumes:
      - .:/app
    ports:
      - 8000:8000
    env_file:
      - ./.env/.dev-sample
    depends_on:
      - db

  db:
    image: postgres:12.0-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment:
      - POSTGRES_DB=wagtail_dev
      - POSTGRES_USER=wagtail_dev
      - POSTGRES_PASSWORD=wagtail_dev

volumes:
  postgres_data:
```

Notes:

1. Here we defined two services, one is `web` (django), the other one is `db`
2. We create a `named docker volume` `postgres_data`, and use it to store the db data, so even db container is deleted, the db data can still exist.

3.5 Environment Variables

We can put env variables in a specific file for easy management.

Let's create `.env` directory, and add `.dev-sample` file

```
DEBUG=1
SECRET_KEY='randome_key'
DJANGO_ALLOWED_HOSTS=*

SQL_ENGINE=django.db.backends.postgresql
SQL_DATABASE=wagtail_dev
SQL_USER=wagtail_dev
SQL_PASSWORD=wagtail_dev
SQL_HOST=db
SQL_PORT=5432
```

Please make sure .env is not excluded in the .gitignore, so it can be added to Git repo

Please note that the db login credential should match environment variables of `db service` in `docker-compose.yml`

Next, let's update `DATABASES`, `SECRET_KEY`, `DEBUG`, and `ALLOWED_HOSTS` *nextjs_wagtail_app/settings.py* to read env variables.

```
import os

SECRET_KEY = os.environ.get("SECRET_KEY", "&nl8s430j^j8l*je+m&ys5dv#zoy)0a2+x1!m8hx290_sx&0gh")

DEBUG = int(os.environ.get("DEBUG", default=1))
```

```
ALLOWED_HOSTS = os.environ.get("DJANGO_ALLOWED_HOSTS", "127.0.0.1").split(" ")

DATABASES = {
    "default": {
        "ENGINE": os.environ.get("SQL_ENGINE", "django.db.backends.sqlite3"),
        "NAME": os.environ.get("SQL_DATABASE", os.path.join(BASE_DIR, "db.sqlite3")),
        "USER": os.environ.get("SQL_USER", "user"),
        "PASSWORD": os.environ.get("SQL_PASSWORD", "password"),
        "HOST": os.environ.get("SQL_HOST", "localhost"),
        "PORT": os.environ.get("SQL_PORT", "5432"),
    }
}
```

3.5.1 Dockerfile

In Docker Compose, we can let it create docker container from existing docker image or custom docker image.

To build custom docker image, we need to provide `Dockerfile`

Please create directory and file like this

```
├── compose
│   └── local
│       └── django
│           ├── Dockerfile
```

Edit *compose/local/django/Dockerfile*

```
FROM python:3.8-slim-buster

ENV PYTHONUNBUFFERED 1
ENV PYTHONDONTWRITEBYTECODE 1

RUN apt-get update \
  # dependencies for building Python packages
  && apt-get install -y build-essential git \
  # psycopg2 dependencies
  && apt-get install -y libpq-dev \
  # Translations dependencies
  && apt-get install -y gettext \
  # Additional dependencies
  && apt-get install -y procps \
  # cleaning up unused files
  && apt-get purge -y --auto-remove -o APT::AutoRemove::RecommendsImportant=false \
  && rm -rf /var/lib/apt/lists/*

# Requirements are installed here to ensure they will be cached.
COPY ./requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

COPY ./compose/local/django/entrypoint /entrypoint
RUN sed -i 's/\r$//g' /entrypoint
RUN chmod +x /entrypoint

COPY ./compose/local/django/start /start
RUN sed -i 's/\r$//g' /start
RUN chmod +x /start

WORKDIR /app
```

```
ENTRYPOINT ["/entrypoint"]
```

Notes:

1. `PYTHONDONTWRITEBYTECODE=1` tell Python to not write bytecode (.pyc) and `__pycache__` directory on local env.
2. `RUN sed -i 's/\r$//g' /entrypoint` is used to process the line endings of the shell scripts, which converts Windows line endings to UNIX line endings.
3. In the above `docker-compose.yml`, we config docker volume `.:/app`, so here we set `WORKDIR /app`. If we edit code on host machine, then the code change can also been seen in `/app` of the docker container.

Next, let's check the `entrypoint` and `start` script.

3.6 Entrypoint

In `docker-compose.yml`, we can use `depends_on` to let `web` service run after `db` service. However, it can not guarantee `web` service start after `db` service is `trully` ready. (Github Issue[10])

So we can add script in entrypoint to solve this problem.

compose/local/django/entrypoint

```
#!/bin/bash

set -o errexit
set -o pipefail
set -o nounset

postgres_ready() {
python << END
import sys

import psycopg2

try:
    psycopg2.connect(
        dbname="${SQL_DATABASE}",
        user="${SQL_USER}",
        password="${SQL_PASSWORD}",
        host="${SQL_HOST}",
        port="${SQL_PORT}",
    )
except psycopg2.OperationalError:
    sys.exit(-1)
sys.exit(0)

END
}
until postgres_ready; do
  >&2 echo 'Waiting for PostgreSQL to become available...'
  sleep 1
done
>&2 echo 'PostgreSQL is available'

exec "$@"
```

[10] https://github.com/docker-library/postgres/issues/146

1. We defined a `postgres_ready` function which is called in loop. The loop would only stop if the `db` service is able to connect.
2. The last `exec "$@"` is used to make the entrypoint a pass through to ensure that Docker container runs the command the user passes in (`command: /start`, in our case). For more, check this Stack Overflow answer[11].

3.7 Start script

Now, let's add start script.

compose/local/django/start

```
#!/bin/bash

set -o errexit
set -o pipefail
set -o nounset

python manage.py migrate
python manage.py runserver 0.0.0.0:8000
```

[11] https://stackoverflow.com/a/39082923/2371995

3.8 Start application

Let's update *requirements.txt* to include `postgres dependency`

```
psycopg2-binary==2.8.6
```

Building the docker images:

```
$ docker-compose build
```

Once the images are build, start the application in `detached mode`:

```
$ docker-compose up -d

# check realtime logs
$ docker-compose logs -f

web_1  | Django version 3.2, using settings 'nextjs_wagtail_app.settings'
web_1  | Starting development server at http://0.0.0.0:8000/
web_1  | Quit the server with CONTROL-C.
```

This will start containers based on the order defined in the `depends_on` option. (`db` first, `web` second)

1. Once the containers are up, the *entrypoint* scripts will execute.
2. Once Postgres is up, the respective *start* scripts will execute. The Django migrations will be applied, and the development server will run. The Django app should then be available.

You can check the docker compose application with this command.

```
            Name                           Command              State           Ports
-------------------------------------------------------------------------------------------
nextjs_wagtail_project_db_1    docker-entrypoint.sh postgres   Up      5432/tcp
nextjs_wagtail_project_web_1   /entrypoint /start              Up      0.0.0.0:8000->8000/tcp
```

Chapter 4

Add Blog Models to Wagtail

4.1 Objectives

By the end of this chapter, you should be able to:

1. Create Django app.
2. Add blog models and understand how it works.
3. Learn how to run code and check data in the Django shell.

4.2 Page structure

Before we start, let's take a look at the page structures, which can help better understand the next sections.

There would be two page type in our project, `BlogPage` and `PostPage`

`BlogPage` would be the index page of the `PostPage`

So the page structures would seem like this.

```
BlogPage
    PostPage1
    PostPage2
    PostPage3
    PostPage4
```

4.3 Create Blog App

Let's create a Django app `blog`

```
$ docker-compose run --rm web python manage.py startapp blog
```

Now you can see Django app `blog` created at the root directory.

```
.
├── blog
├── compose
├── docker-compose.yml
├── manage.py
```

```
├── nextjs_wagtail_app
└── requirements.txt
```

Add `blog` to the `INSTALLED_APPS` of *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    # code omitted for brevity
    'blog',
]
```

Next, let's start adding blog models, there are mainly two types of models we need to add here.

1. Page models (`BlogPage`, `PostPage`)
2. Other models (`Category`, `Tag`)

4.4 Page Models

blog/models.py

```
from django.db import models
from wagtail.core.models import Page
from wagtail.images.edit_handlers import ImageChooserPanel
from wagtail.admin.edit_handlers import FieldPanel


class BlogPage(Page):
    description = models.CharField(max_length=255, blank=True,)

    content_panels = Page.content_panels + [FieldPanel("description", classname="full")]


class PostPage(Page):
    header_image = models.ForeignKey(
        "wagtailimages.Image",
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name="+",
    )

    content_panels = Page.content_panels + [
        ImageChooserPanel("header_image"),
    ]
```

1. When you create page models, please make sure all page classes inherit from the Wagtail `Page` class.
2. Here we add a `description` field to the `BlogPage` and a `header_image` field to the `PostPage`.
3. We should also add edit handlers to the `content_panels` so we can edit the fields in Wagtail admin.

4.5 Category and Tag

To make the blog supports `Category` and `Tag` features, let's add some models.

blog/models.py

```
from django.db import models
from wagtail.snippets.models import register_snippet
from taggit.models import Tag as TaggitTag


@register_snippet
class BlogCategory(models.Model):
    name = models.CharField(max_length=255)
    slug = models.SlugField(unique=True, max_length=80)

    panels = [
        FieldPanel("name"),
        FieldPanel("slug"),
    ]

    def __str__(self):
        return self.name

    class Meta:
        verbose_name = "Category"
        verbose_name_plural = "Categories"


@register_snippet
class Tag(TaggitTag):
    class Meta:
        proxy = True
```

1. Here we created two models, both of them inherit from the `models.Model`, which are standard Django models.
2. `register_snippet` decorator would register them as `Wagtail snippets`, that can make us add/edit/delete the model instances in `snippets` of Wagtail admin.
3. Since Wagtail already has `tag` support built on `django-taggit`, so here we create a proxy-model[12] to declare it as `wagtail snippet`

4.6 Intermediary model

Now page models and snippet models are created. But we still need to create `Intermediary models` so the connections between page and snippet can be stored in the db.

Note: I do not recommend use `ParentalManyToManyField` in Wagtail app even it seems more easy to understand. You can check this Wagtail tip[13] for more details.

```
from modelcluster.fields import ParentalKey
from taggit.models import TaggedItemBase

class PostPageBlogCategory(models.Model):
    page = ParentalKey(
        "blog.PostPage", on_delete=models.CASCADE, related_name="categories"
    )
    blog_category = models.ForeignKey(
        "blog.BlogCategory", on_delete=models.CASCADE, related_name="post_pages"
    )

    panels = [
        SnippetChooserPanel("blog_category"),
```

[12] https://docs.djangoproject.com/en/3.2/topics/db/models/#proxy-models
[13] https://www.accordbox.com/blog/wagtail-tip-1-how-replace-parentalmanytomanyfield-inlinepanel/

```
    ]

    class Meta:
        unique_together = ("page", "blog_category")


class PostPageTag(TaggedItemBase):
    content_object = ParentalKey("PostPage", related_name="post_tags")
```

1. `PostPageBlogCategory` is to store the connection between `PostPage` and `Category`
2. Please remember to use `ParentalKey` instead of `models.ForeignKey` so the Wagtail page `draft` feature can work.
3. `unique_together = ("page", "blog_category")` would add `db constraints` to avoid duplicate records. You can check Django unique_together[14] to learn more.

Next, let's update the `PostPage` model so we can add/edit/remove `Category` and `Tag` for the page in Wagtail admin.

```
from modelcluster.tags import ClusterTaggableManager


class PostPage(Page):
    header_image = models.ForeignKey(
        "wagtailimages.Image",
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name="+",
    )

    tags = ClusterTaggableManager(through="blog.PostPageTag", blank=True)

    content_panels = Page.content_panels + [
        ImageChooserPanel("header_image"),
        InlinePanel("categories", label="category"),
        FieldPanel("tags"),
    ]
```

1. We add `ClusterTaggableManager` and use `through` to specify the intermediary model we just created.
2. And then we add `InlinePanel("categories", label="category")` to the `content_panels`. The `categories` relationship is already defined in `PostPageBlogCategory.page.related_name`
3. The `PostPageBlogCategory.panels` defines the behavior in `InlinePanel`, which means we can set multiple `blog_category` when we create or edit page.

4.7 Source Code

Below is the full code of the *blog/models.py* for reference

```
from django.db import models
from modelcluster.fields import ParentalKey
from modelcluster.tags import ClusterTaggableManager
from taggit.models import Tag as TaggitTag
from taggit.models import TaggedItemBase
from wagtail.admin.edit_handlers import (
```

[14] https://docs.djangoproject.com/en/3.1/ref/models/options/#unique-together

```
    FieldPanel,
    FieldRowPanel,
    InlinePanel,
    MultiFieldPanel,
    PageChooserPanel,
    StreamFieldPanel,
)
from wagtail.core.models import Page
from wagtail.images.edit_handlers import ImageChooserPanel
from wagtail.snippets.edit_handlers import SnippetChooserPanel
from wagtail.snippets.models import register_snippet


class BlogPage(Page):
    description = models.CharField(max_length=255, blank=True,)

    content_panels = Page.content_panels + [FieldPanel("description", classname="full")]


class PostPage(Page):
    header_image = models.ForeignKey(
        "wagtailimages.Image",
        null=True,
        blank=True,
        on_delete=models.SET_NULL,
        related_name="+",
    )

    tags = ClusterTaggableManager(through="blog.PostPageTag", blank=True)

    content_panels = Page.content_panels + [
        ImageChooserPanel("header_image"),
        InlinePanel("categories", label="category"),
        FieldPanel("tags"),
    ]


class PostPageBlogCategory(models.Model):
    page = ParentalKey(
        "blog.PostPage", on_delete=models.CASCADE, related_name="categories"
    )
    blog_category = models.ForeignKey(
        "blog.BlogCategory", on_delete=models.CASCADE, related_name="post_pages"
    )

    panels = [
        SnippetChooserPanel("blog_category"),
    ]

    class Meta:
        unique_together = ("page", "blog_category")


@register_snippet
class BlogCategory(models.Model):
    name = models.CharField(max_length=255)
    slug = models.SlugField(unique=True, max_length=80)

    panels = [
        FieldPanel("name"),
        FieldPanel("slug"),
    ]
```

```
    def __str__(self):
        return self.name

    class Meta:
        verbose_name = "Category"
        verbose_name_plural = "Categories"


class PostPageTag(TaggedItemBase):
    content_object = ParentalKey("PostPage", related_name="post_tags")


@register_snippet
class Tag(TaggitTag):
    class Meta:
        proxy = True
```

4.8 Migrate DB

After we finish the models part, let's migrate our db so relevant tables would be created or migrated.

```
$ docker-compose run --rm web python manage.py makemigrations
$ docker-compose run --rm web python manage.py migrate
```

4.9 Setup The Site

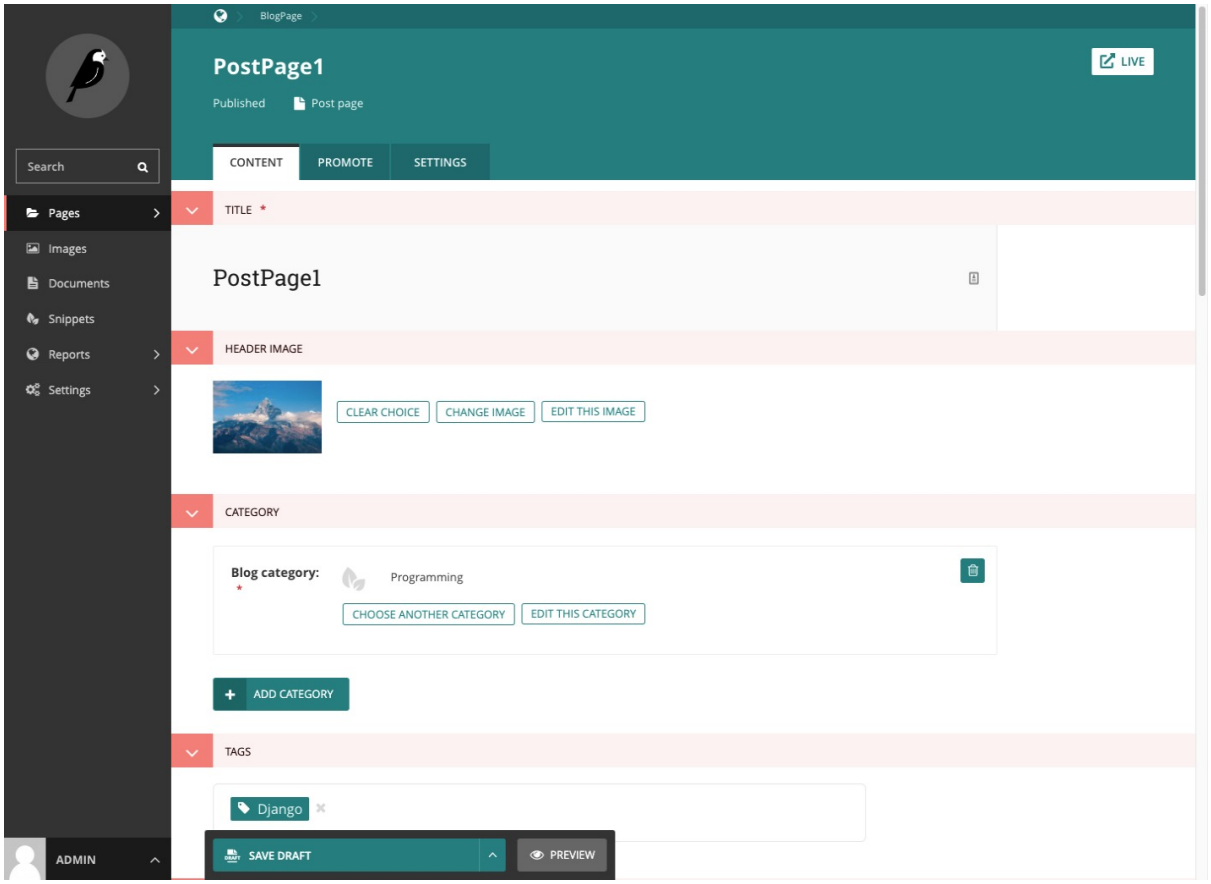
```
# create superuser and password
$ docker-compose run --rm web python manage.py createsuperuser

$ docker-compose up -d

# tail the log
$ docker-compose logs -f
```

1. Login on http://127.0.0.1:8000/cms-admin/
2. Go to http://127.0.0.1:8000/cms-admin/pages/ to create `BlogPage` beside the `Welcome to your new Wagtail site!` page
3. Follow `settings/site` in the sidebar to change the `root page` of the `localhost` site to the `BlogPage` we just created.
4. Go to http://127.0.0.1:8000/cms-admin/pages/ delete the `Welcome to your new Wagtail site!` page
5. Now if you visit http://127.0.0.1:8000/ you will see `TemplateDoesNotExist` exception. This is correct, and we will fix it later.

4.10 Add PostPage

1. Follow `Pages/BlogPage` in the sidebar (not the edit icon)
2. Now the URl would seem like `http://127.0.0.1:8000/cms-admin/pages/3/`
3. Click the `Add child page` button to start adding `PostPage` as children of the `BlogPage`

4. Remember to publish the page after you edit the page.



4.11 Simple Test

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

Now you are in Django shell, and you can run some Python code to quickly check the data and code. This is very useful during the development.

```
>>> from wagtail.core.models import Page

# number 4 is the post page primary key we just created
# you can get it from the url when on the edit page
>>> page = Page.objects.get(pk=4).specific
>>> page.title
'PostPage1'
>>> page.tags.all()
[<Tag: Django>]
>>> page.categories.all()
<QuerySet [<PostPageBlogCategory: PostPageBlogCategory object (1)>]>
>>> page.categories.first().blog_category
<BlogCategory: Programming>
```

4.12 ParentalKey

Here I'd like to talk about a little more about the `ParentalKey` and the difference between with `ForeignKey`

Let's assume you are building a CMS framework which support `preview` feature, and now you have a `live` post page which has category `category 1`

So in the table, the data would seem like this.

```
PostPage: postpage 1 (pk=1)

BlogCategory: category 1 (pk=1)

PostPageBlogCategory (pk=1, blog_category=1, page=1)
```

Now some editor wants to change the page category to `category 2`, and he even wants to `preview` it before publishing it. So what is your solution?

1. You need to create something like `PostPageBlogCategory (blog_category=2, page=1)` in `memory` and **not** write it to `PostPageBlogCategory` table. (Because if you do, it will affect the `live` version of the page)
2. You need to write code to convert the page data, and the `PostPageBlogCategory` to some serialize format (JSON for example), and save it to some `revision` table as the latest revision.
3. On the preview page, fetch the data from the `revision` table and deserialize to a normal page object, and then render it to HTML.

Unfortunately, Django's `ForeignKey` can not work in this case, because it needs `PostPageBlogCategory (blog_category=2, page=1)` to save to db first, so it has `pk`

That is why django-modelcluster[15] is created and `ParentalKey` is introduced.

Now We can solve the above problem in this way.

1. Make the `PostPage` inherit from `modelcluster.models.ClusterableModel`. Actually, Wagtail Page class already did this[16]
2. And define the `PostPageBlogCategory.page` as `ParentalKey` field.
3. So the Wagtail page (`ClusterableModel`) can detect and hold the `PostPageCategory` in memory even it is not created in db yet. (has `null` pk)
4. We can then serialize the page to JSON format (also contains `PostPageBlogCategory` info) and save to `revision` table.
5. Now editor can preview the page without touching the live version.

If you want to dive deeper, try to use code below to check on your local:

```
>>> from wagtail.core.models import PageRevision
# page__pk is the primary key of the page
>>> revision = PageRevision.objects.filter(page__pk=page__pk).first()
>>> revision.content_json
```

So below are tips:

1. If you define some `ForeignKey` relationship with `Page` in `Page` class, for example `PostPage.header_image`, use `ForeignKey`. (This has no the above problem)
2. If you define some `ForeignKey` relationship with `Page` in `other` class, for example, `PostPageCategory.page`, use `ParentalKey`.

[15] https://github.com/wagtail/django-modelcluster
[16] https://github.com/wagtail/wagtail/blob/v2.11.2/wagtail/core/models.py#L721

Chapter 5

StreamField

5.1 Objectives

By the end of this chapter, you should be able to:

1. Understand how `StreamField` works
2. Use `StreamField` as body format of the PostPage.

5.2 What is StreamField

`StreamField` provides a flexible way for us to construct content.

The `StreamField` is a list which contains the `value` and `type` of the sub-blocks (we will see it in a bit). You can use the built-in block shipped with Wagtail or you can create your custom block.

Some blocks can also contain sub-block, so you can use it to create a `complex nested data structure, which is powerful`.

5.3 Block

From my understanding, I'd like to group the Wagtail built-in blocks in this way.

1. Basic block, which is similar with Django model field types[17] For example, `CharBlock`, `TextBlock`, `ChoiceBlock`
2. Chooser Block, which is for `object selection`. For example, `PageChooserBlock`, `ImageChooserBlock`.
3. `StructBlock`, which works like `dict` (Object in `js`), which contains `fixed` sub-blocks.
4. `StreamBlock`, `ListBlock`, which works like `list` (Arrays in `js`), which contains `no-fixed` sub-blocks.

5.4 Body

Next, let's use `StreamField` to define the `PostPage.body`

It is recommended to put blocks in a separate file to keep your model clean.

[17] https://docs.djangoproject.com/en/3.2/ref/models/fields/#field-types

blog/blocks.py

```
from wagtail.core.blocks import (
    BooleanBlock,
    CharBlock,
    ChoiceBlock,
    DateTimeBlock,
    FieldBlock,
    IntegerBlock,
    ListBlock,
    PageChooserBlock,
    RawHTMLBlock,
    RichTextBlock,
    StreamBlock,
    StructBlock,
    StructValue,
    TextBlock,
    URLBlock,
)
from wagtail.core.models import Orderable, Page
from wagtail.embeds.blocks import EmbedBlock
from wagtail.images.api.fields import ImageRenditionField
from wagtail.images.blocks import ImageChooserBlock
from wagtail.snippets.blocks import SnippetChooserBlock


class CustomImageChooserBlock(ImageChooserBlock):
    pass


class ImageText(StructBlock):
    reverse = BooleanBlock(required=False)
    text = RichTextBlock()
    image = CustomImageChooserBlock()


class BodyBlock(StreamBlock):
    h1 = CharBlock()
    h2 = CharBlock()
    paragraph = RichTextBlock()

    image_text = ImageText()
    image_carousel = ListBlock(CustomImageChooserBlock())
    thumbnail_gallery = ListBlock(CustomImageChooserBlock())
```

1. `CustomImageChooserBlock` inherits from `ImageChooserBlock` so we can do some custom work in the future.
2. `ImageText` inherits from `StructBlock`, it has three sub-blocks, we can only set values to `reverse`, `text` and `image`.
3. `BodyBlock` inherits from `StreamBlock`, we can add more than one sub-blocks because `StreamBlock` behaves like list.

Update *blog/models.py*

```
from wagtail.core.fields import StreamField
from .blocks import BodyBlock


class PostPage(Page):
    header_image = models.ForeignKey(
        "wagtailimages.Image",
        null=True,
```

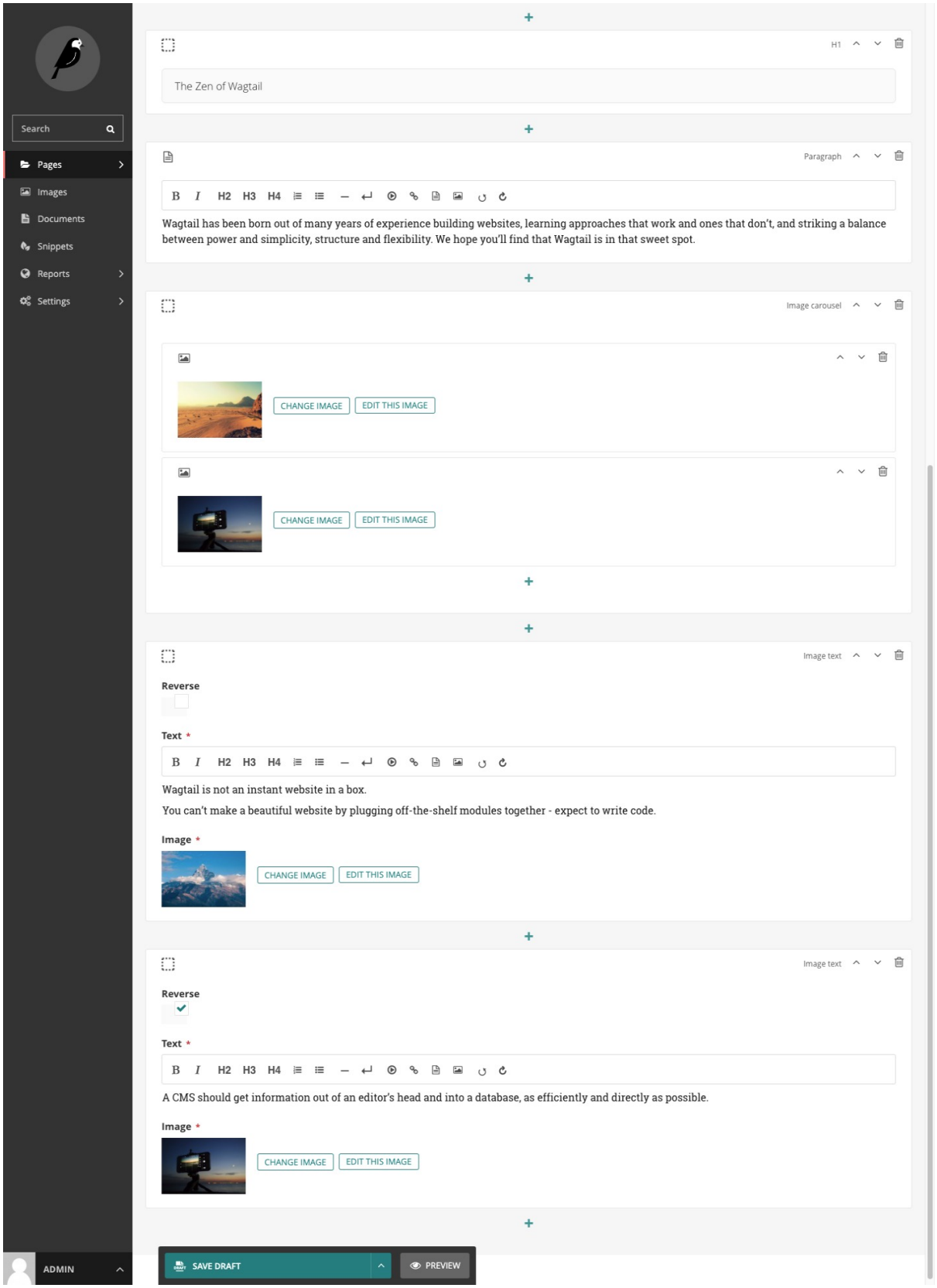
```
        blank=True,
        on_delete=models.SET_NULL,
        related_name="+",
    )

    body = StreamField(BodyBlock(), blank=True)           # new

    tags = ClusterTaggableManager(through="blog.PostPageTag", blank=True)

    content_panels = Page.content_panels + [
        ImageChooserPanel("header_image"),
        InlinePanel("categories", label="category"),
        FieldPanel("tags"),
        StreamFieldPanel("body"),                         # new
    ]
```

1. Import `BodyBlock` from `./blocks`
2. Define body `body = StreamField(BodyBlock(), blank=True)`
3. Remember to update `content_panels` so you can edit in Wagtail admin.

Migrate the db

```
$ docker-compose run --rm web python manage.py makemigrations
$ docker-compose run --rm web python manage.py migrate
```

Now visits http://127.0.0.1:8000/cms-admin/pages/4/edit/ to add some content to the body.



5.5 Dive Deep

Let's run some code to learn more about StreamField and Django shell.

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from wagtail.core.models import Page

>>> page = Page.objects.get(pk=4).specific

>>> page.body.raw_data
[{'type': 'h1', 'value': 'The Zen of Wagtail', 'id': '886b6fb8-472a-47b0-9bef-2317ea04799a'}, {'type
↪': 'paragraph', 'value': '<p>Wagtail has been born out of many years of experience building
↪websites, learning approaches that work and ones that don’t, and striking a balance between power
↪and simplicity, structure and flexibility. We hope you’ll find that Wagtail is in that sweet spot.
↪</p>', 'id': '6b051ad8-d1ba-4dab-84be-1d30b4e272e9'}, {'type': 'image_carousel', 'value': [2, 1],
↪'id': '7231f723-5770-4173-a17e-a86a0b68bac1'}, {'type': 'image_text', 'value': {'reverse': False,
↪'text': '<p>Wagtail is not an instant website in a box.</p><p>You can’t make a beautiful website
↪by plugging off-the-shelf modules together - expect to write code.</p>', 'image': 3}, 'id':
↪'b32bc7bb-0e71-448b-968d-67be3492659f'}, {'type': 'image_text', 'value': {'reverse': True, 'text
↪': '<p>A CMS should get information out of an editor’s head and into a database, as efficiently
↪and directly as possible.</p>', 'image': 1}, 'id': 'ab6f3090-b05d-4c07-a3b0-c105ddf823a6'}]

# let's make the data look more clear
>>> import pprint
>>> pprint.pprint(list(page.body.raw_data))
[{'id': '886b6fb8-472a-47b0-9bef-2317ea04799a',
  'type': 'h1',
  'value': 'The Zen of Wagtail'},
 {'id': '6b051ad8-d1ba-4dab-84be-1d30b4e272e9',
  'type': 'paragraph',
  'value': '<p>Wagtail has been born out of many years of experience building '
           'websites, learning approaches that work and ones that don’t, and '
           'striking a balance between power and simplicity, structure and '
           'flexibility. We hope you’ll find that Wagtail is in that sweet '
           'spot.</p>'},
 {'id': '7231f723-5770-4173-a17e-a86a0b68bac1',
  'type': 'image_carousel',
  'value': [2, 1]},
 {'id': 'b32bc7bb-0e71-448b-968d-67be3492659f',
  'type': 'image_text',
  'value': {'image': 3,
            'reverse': False,
            'text': '<p>Wagtail is not an instant website in a box.</p><p>You '
                    'can’t make a beautiful website by plugging off-the-shelf '
                    'modules together - expect to write code.</p>'}},
 {'id': 'ab6f3090-b05d-4c07-a3b0-c105ddf823a6',
  'type': 'image_text',
  'value': {'image': 1,
            'reverse': True,
            'text': '<p>A CMS should get information out of an editor’s head '
                    'and into a database, as efficiently and directly as '
                    'possible.</p>'}}]
```

1. For basic block, the `value` is usually number and string.
2. For chooser block, the `value` is the `primary key` of the selected object.
3. For `StructBlock`, the `value` is a Python dict
4. For `StreamBlock` and `ListBlock`, the `value` is a Python List.

Chapter 6

Build REST API (Part 1)

6.1 Objectives

By the end of this chapter, you should be able to:

1. Understand what is DRF serializer.
2. Learn how to write serializer for Django models.

6.2 Django REST Framework

Django REST Framework[18] (DRF) is a powerful and flexible tool for building Web APIs for Django project.

There are some basic concepts in DRF

1. routers[19], similar with Django urls.
2. viewsets[20], similar with Django view, which handle request and return response.
3. serializer[21], convert Django model instances to JSON, XML or vice versa.

6.3 Install Django REST Framework

Update *requirements.txt*

```
djangorestframework
```

Note: djangorestframework is also dependency package of Wagtail CMS, but it is better to explicitly add it here.

Update INSTALLED_APPS of *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    # code omitted for brevity

    'wagtail.api.v2',               # new
    'rest_framework',               # new
```

[18] https://www.django-rest-framework.org/
[19] https://www.django-rest-framework.org/api-guide/routers/
[20] https://www.django-rest-framework.org/api-guide/viewsets/
[21] https://www.django-rest-framework.org/api-guide/serializers/

```
    'blog',
]
```

6.4 Serializer

The serializer would serialize the Django model instances to target format.

Create *blog/serializers.py*

```
from rest_framework import serializers

from .models import BlogCategory, PostPage, Tag


class PostPageSerializer(serializers.ModelSerializer):
    class Meta:
        model = PostPage
        fields = (
            "id",
            "slug",
            "title",
        )


class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = BlogCategory
        fields = (
            "id",
            "slug",
            "name",
        )


class TagSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tag
        fields = (
            "id",
            "slug",
            "name",
        )
```

We create three `serializers` here for Django models.

Now, let's run some code in Django shell

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from blog.models import BlogCategory
>>> instance = BlogCategory.objects.first()

>>> from blog.serializers import CategorySerializer
>>> CategorySerializer(instance).data
{'id': 1, 'slug': 'programming', 'name': 'Programming'}

>>> from rest_framework.renderers import JSONRenderer
>>> JSONRenderer().render(CategorySerializer(instance).data)
b'{"id":1,"slug":"programming","name":"Programming"}'
```

As you can see, we can use `serializers` to help us serialize the Django model to JSON format.

6.5 Serializer Field

If we use the above `PostPageSerializer`, we can get `id`, `slug` and `title` fields.

```
>>> from blog.models import PostPage
# number 4 is the post page primary key we just created
>>> instance = PostPage.objects.get(pk=4)

>>> from blog.serializers import PostPageSerializer
>>> PostPageSerializer(instance).data
{'id': 4, 'slug': 'postpage1', 'title': 'PostPage1'}
```

Let's add `tags` to the `Meta.fields` and see what happen.

```
class PostPageSerializer(serializers.ModelSerializer):
    class Meta:
        model = PostPage
        fields = (
            "id",
            "slug",
            "title",
            "tags",                          # new
        )
```

Let's open a new Django shell and check.

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from blog.models import PostPage
# number 4 is the post page primary key we just created
>>> instance = PostPage.objects.get(pk=4)

>>> from blog.serializers import PostPageSerializer
>>> PostPageSerializer(instance).data
{'id': 4, 'slug': 'postpage1', 'title': 'PostPage1', 'tags': <modelcluster.contrib.taggit._
↪ClusterTaggableManager object at 0x7fb1790b5be0>}
```

This is not we want, we need to tell serializers **how to process the data with other Django model**.

6.5.1 TagField

Create *blog/fields.py*

```
from rest_framework.fields import Field


class TagField(Field):
    def to_representation(self, tags):
        try:
            return [
                {"name": tag.name, "slug": tag.slug, "id": tag.id} for tag in tags.all()
            ]
        except Exception:
            return []
```

Update *blog/serializers.py*

```
from .fields import TagField            # new
from .models import BlogCategory, PostPage, Tag


class PostPageSerializer(serializers.ModelSerializer):
    tags = TagField()                   # new

    class Meta:
        model = PostPage
        fields = (
            "id",
            "slug",
            "title",
            "tags",
        )
```

1. We defined a custom `TagField` and overwrite `to_representation` method to define how the data is represented.
2. In `PostPageSerializer`, we declared `tags` with `TagField`

Let's open a new Django shell and check.

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from blog.models import PostPage
# number 4 is the post page primary key we just created
>>> instance = PostPage.objects.get(pk=4)

>>> from blog.serializers import PostPageSerializer
>>> PostPageSerializer(instance).data

{'id': 4, 'slug': 'postpage1', 'title': 'PostPage1', 'tags': [{'name': 'Django', 'slug': 'django',
↪'id': 1}]}
```

As you can see, now the `tag` data looks reasonable.

6.5.2 CategoryField

Let's keep adding `categories` fields using the same way.

Update *blog/fields.py*

```
class CategoryField(Field):
    def to_representation(self, categories):
        try:
            return [
                {
                    "name": category.blog_category.name,
                    "slug": category.blog_category.slug,
                    "id": category.blog_category.id,
                }
                for category in categories.all()
            ]
        except Exception:
            return []
```

Update *blog/serializers.py*

```
from .fields import TagField, CategoryField         # new
from .models import BlogCategory, PostPage, Tag


class PostPageSerializer(serializers.ModelSerializer):
    tags = TagField()
    categories = CategoryField()        # new

    class Meta:
        model = PostPage
        fields = (
            "id",
            "slug",
            "title",
            "tags",
            "categories",             # new
        )
```

Please test again in Django shell to make sure it works without problems.

6.6 StreamField

Actually, Wagtail also use the REST Serializer Field to help serialize some built-in fields.

You can check source code here[22] to learn more.

In the above `serializers.py`, there is a `StreamField` which can help us serialize model `StreamField`, so let's add it to our `PostPageSerializer`

Update *blog/serializers.py*

```
from wagtail.api.v2.serializers import StreamField    # new


class PostPageSerializer(serializers.ModelSerializer):
    tags = TagField()
    categories = CategoryField()
    body = StreamField()                              # new

    class Meta:
        model = PostPage
        fields = (
            "id",
            "slug",
            "title",
            "tags",
            "categories",
            "body",                                   # new
        )
```

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from blog.models import PostPage
# number 4 is the post page primary key we just created
>>> instance = PostPage.objects.get(pk=4)

>>> from blog.serializers import PostPageSerializer
```

[22] https://github.com/wagtail/wagtail/blob/v2.14/wagtail/api/v2/serializers.py

```
>>> import pprint
>>> pprint.pprint(PostPageSerializer(instance).data)

{'body': [{'id': '886b6fb8-472a-47b0-9bef-2317ea04799a',
           'type': 'h1',
           'value': 'The Zen of Wagtail'},
          {'id': '6b051ad8-d1ba-4dab-84be-1d30b4e272e9',
           'type': 'paragraph',
           'value': '<p>Wagtail has been born out of many years of experience '
                    'building websites, learning approaches that work and ones '
                    'that don’t, and striking a balance between power and '
                    'simplicity, structure and flexibility. We hope you’ll '
                    'find that Wagtail is in that sweet spot.</p>'},
          {'id': '7231f723-5770-4173-a17e-a86a0b68bac1',
           'type': 'image_carousel',
           'value': [2, 1]},
          {'id': 'b32bc7bb-0e71-448b-968d-67be3492659f',
           'type': 'image_text',
           'value': {'image': 3,
                     'reverse': False,
                     'text': '<p>Wagtail is not an instant website in a '
                             'box.</p><p>You can’t make a beautiful website by '
                             'plugging off-the-shelf modules together - expect '
                             'to write code.</p>'}},
          {'id': 'ab6f3090-b05d-4c07-a3b0-c105ddf823a6',
           'type': 'image_text',
           'value': {'image': 1,
                     'reverse': True,
                     'text': '<p>A CMS should get information out of an '
                             'editor’s head and into a database, as '
                             'efficiently and directly as possible.</p>'}}],
 'categories': [{'id': 1, 'name': 'Programming', 'slug': 'programming'}],
 'id': 4,
 'slug': 'postpage1',
 'tags': [{'id': 1, 'name': 'Django', 'slug': 'django'}],
 'title': 'PostPage1'}
```

6.7 Image

Let’s keep serializing the `PostPage.header_image` field.

Update *blog/serializers.py*

```
from wagtail.images.api.fields import ImageRenditionField       # new


class PostPageSerializer(serializers.ModelSerializer):
    tags = TagField()
    categories = CategoryField()
    body = StreamField()
    header_image = ImageRenditionField("max-1000x800")          # new

    class Meta:
        model = PostPage
        fields = (
            "id",
            "slug",
            "title",
            "tags",
            "categories",
```

```
            "body",
            "header_image",  # new
        )
```

We use `ImageRenditionField` to control the header_image size

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from blog.models import PostPage
# number 4 is the post page primary key we just created
>>> instance = PostPage.objects.get(pk=4)

>>> from blog.serializers import PostPageSerializer
>>> import pprint
>>> pprint.pprint(PostPageSerializer(instance).data)

{'body': [{'id': '886b6fb8-472a-47b0-9bef-2317ea04799a',
           'type': 'h1',
           'value': 'The Zen of Wagtail'},
          {'id': '6b051ad8-d1ba-4dab-84be-1d30b4e272e9',
           'type': 'paragraph',
           'value': '<p>Wagtail has been born out of many years of experience '
                    'building websites, learning approaches that work and ones '
                    'that don’t, and striking a balance between power and '
                    'simplicity, structure and flexibility. We hope you’ll '
                    'find that Wagtail is in that sweet spot.</p>'},
          {'id': '7231f723-5770-4173-a17e-a86a0b68bac1',
           'type': 'image_carousel',
           'value': [2, 1]},
          {'id': 'b32bc7bb-0e71-448b-968d-67be3492659f',
           'type': 'image_text',
           'value': {'image': 3,
                     'reverse': False,
                     'text': '<p>Wagtail is not an instant website in a '
                             'box.</p><p>You can’t make a beautiful website by '
                             'plugging off-the-shelf modules together - expect '
                             'to write code.</p>'}},
          {'id': 'ab6f3090-b05d-4c07-a3b0-c105ddf823a6',
           'type': 'image_text',
           'value': {'image': 1,
                     'reverse': True,
                     'text': '<p>A CMS should get information out of an '
                             'editor’s head and into a database, as '
                             'efficiently and directly as possible.</p>'}}],
 'categories': [{'id': 1, 'name': 'Programming', 'slug': 'programming'}],
 'header_image': OrderedDict([('url', '/media/images/image_3.max-1000x800.jpg'),
                              ('width', 1000),
                              ('height', 666),
                              ('alt', 'image_3.jpeg')]),
 'id': 4,
 'slug': 'postpage1',
 'tags': [{'id': 1, 'name': 'Django', 'slug': 'django'}],
 'title': 'PostPage1'}
```

Now `header_image` contains useful info about the `header_image` instead of single `pk`

6.8 ImageChooserBlock

If you check the above JSON response carefully, you will see the image field in the StreamField contains image `pk` value instead of image url and other info.

Let's fix it in this section.

If we check https://github.com/wagtail/wagtail/blob/v2.14/wagtail/api/v2/serializers.py#L230

```
class StreamField(Field):

    def to_representation(self, value):
        return value.stream_block.get_api_representation(value, self.context)
```

The `StreamField` will call `get_api_representation` of the child block to get serialized representation.

Update *blog/blocks.py*

```
class CustomImageChooserBlock(ImageChooserBlock):                    # new
    def __init__(self, *args, **kwargs):
        self.rendition = kwargs.pop("rendition", "original")
        super().__init__(**kwargs)

    def get_api_representation(self, value, context=None):
        return ImageRenditionField(self.rendition).to_representation(value)


class ImageText(StructBlock):
    reverse = BooleanBlock(required=False)
    text = RichTextBlock()
    image = CustomImageChooserBlock(rendition="width-800")           # new
```

1. `CustomImageChooserBlock` now accept parameter `rendition` which has default value `original`
2. In `get_api_representation`, we use `ImageRenditionField` to generate new size image.
3. We change `ImageText.image` to `CustomImageChooserBlock(rendition="width-800")`

```
# migrate db
$ docker-compose run --rm web python manage.py makemigrations
$ docker-compose run --rm web python manage.py migrate

# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from blog.models import PostPage
# number 4 is the post page primary key we just created
>>> instance = PostPage.objects.get(pk=4)

>>> from blog.serializers import PostPageSerializer
>>> import pprint
>>> pprint.pprint(PostPageSerializer(instance).data)

{'body': [{'id': '886b6fb8-472a-47b0-9bef-2317ea04799a',
           'type': 'h1',
           'value': 'The Zen of Wagtail'},
          {'id': '6b051ad8-d1ba-4dab-84be-1d30b4e272e9',
           'type': 'paragraph',
           'value': '<p>Wagtail has been born out of many years of experience '
                    'building websites, learning approaches that work and ones '
                    'that don't, and striking a balance between power and '
                    'simplicity, structure and flexibility. We hope you'll '
                    'find that Wagtail is in that sweet spot.</p>'},
          {'id': '7231f723-5770-4173-a17e-a86a0b68bac1',
```

```
          'type': 'image_carousel',
          'value': [OrderedDict([('url', '/media/images/image_2.original.jpg'),
                                 ('width', 1280),
                                 ('height', 853),
                                 ('alt', 'image_2.jpeg')]),
                    OrderedDict([('url', '/media/images/image_1.original.jpg'),
                                 ('width', 1280),
                                 ('height', 853),
                                 ('alt', 'image_1.jpeg')])]},
         {'id': 'b32bc7bb-0e71-448b-968d-67be3492659f',
          'type': 'image_text',
          'value': {'image': OrderedDict([('url',
                                           '/media/images/image_3.width-800.jpg'),
                                          ('width', 800),
                                          ('height', 533),
                                          ('alt', 'image_3.jpeg')]),
                    'reverse': False,
                    'text': '<p>Wagtail is not an instant website in a '
                            'box.</p><p>You can’t make a beautiful website by '
                            'plugging off-the-shelf modules together - expect '
                            'to write code.</p>'}},
         {'id': 'ab6f3090-b05d-4c07-a3b0-c105ddf823a6',
          'type': 'image_text',
          'value': {'image': OrderedDict([('url',
                                           '/media/images/image_1.width-800.jpg'),
                                          ('width', 800),
                                          ('height', 533),
                                          ('alt', 'image_1.jpeg')]),
                    'reverse': True,
                    'text': '<p>A CMS should get information out of an '
                            'editor’s head and into a database, as '
                            'efficiently and directly as possible.</p>'}}],
 'categories': [{'id': 1, 'name': 'Programming', 'slug': 'programming'}],
 'header_image': OrderedDict([('url', '/media/images/image_3.max-1000x800.jpg'),
                              ('width', 1000),
                              ('height', 666),
                              ('alt', 'image_3.jpeg')]),
 'id': 4,
 'slug': 'postpage1',
 'tags': [{'id': 1, 'name': 'Django', 'slug': 'django'}],
 'title': 'PostPage1'}
```

1. Now all the image block in `body` has `url`
2. The image in `image_text` has `800` width
3. The image in `image_carousel` still has `original` width

Chapter 7

Build REST API (Part 2)

7.1 Objectives

By the end of this chapter, you should be able to:

1. Render JSON representation in page `serve` method.
2. Import `BasePageSerializer` to make the code clean and easy to maintain.
3. Learn to use `ViewSet` to expose API endpoint.

7.2 BlogPageSerializer

Update *blog/serializers.py*

```
from .models import BlogCategory, PostPage, Tag, BlogPage      # new


class BlogPageSerializer(serializers.ModelSerializer):     # new
    class Meta:
        model = BlogPage
        fields = (
            "id",
            "slug",
            "title",
            "url",
            "last_published_at",
        )


class PostPageSerializer(serializers.ModelSerializer):
    tags = TagField()
    categories = CategoryField()
    body = StreamField()
    header_image = ImageRenditionField("max-1000x800")

    class Meta:
        model = PostPage
        fields = (
            "id",
            "slug",
            "title",
            "url",                                  # new
            "last_published_at",                    # new
```

```
            "tags",
            "categories",
            "body",
            "header_image",
        )
```

Notes:

1. We add `BlogPageSerializer`, which is for `BlogPage`
2. We add some new fields (url, last_published_at) to the `PostPageSerializer` because we will use them in later chapter.

7.3 BasePage

Update *blog/models.py*

```
from django.utils.module_loading import import_string
from django.http.response import JsonResponse


class BasePage(Page):

    serializer_class = None

    class Meta:
        abstract = True

    def get_component_data(self):
        if not self.serializer_class:
            raise Exception(f'serializer_class is not set {self.__class__.__name__}')

        serializer_class = import_string(self.serializer_class)

        return {
            'page_type': self.__class__.__name__,
            'page_content': serializer_class(self).data
        }

    def get_context(self, request, *args, **kwargs):
        context = super().get_context(request)
        context['page_component'] = self.get_component_data()
        return context

    def serve(self, request, *args, **kwargs):
        context = self.get_context(request, *args, **kwargs)
        return JsonResponse(context['page_component'])


class BlogPage(BasePage):
    serializer_class = "blog.serializers.BlogPageSerializer"


class PostPage(BasePage):
    serializer_class = "blog.serializers.PostPageSerializer"
```

Notes:

1. The `get_component_data` will use `serializer_class` to serializer the page instance.

2. The `page_component` in `context` will contains JSON representation of the page, which has `page_type` and `page_content`
3. In the `serve` method, we return the `context['page_component']` as JSON format.

Next, let's test:

```
$ docker-compose up -d
$ docker-compose logs -f
```

Try to visit:

- http://localhost:8000/
- http://localhost:8000/postpage1/

Or you can use

```
$ curl --header "Content-Type: application/json" http://localhost:8000/
```

As you can see, if we visit the page URL, the relevant JSON data will be returned.

If we want to add more custom data to the JSON response, we can:

1. Edit the `serializer_class` to add more fields.
2. Add some data to the `page_component` in `get_context` method.

7.4 BasePageSerializer

Update *blog/serializers.py*

```
from rest_framework import serializers

from wagtail.images.api.fields import ImageRenditionField
from wagtail.core import fields
from wagtail.api.v2 import serializers as wagtail_serializers

from .fields import TagField, CategoryField
from .models import BlogCategory, PostPage, Tag, BlogPage, BasePage


class BasePageSerializer(serializers.ModelSerializer):                  # new
    serializer_field_mapping = (
        serializers.ModelSerializer.serializer_field_mapping.copy()
    )
    serializer_field_mapping.update(
        {fields.StreamField: wagtail_serializers.StreamField}
    )

    class Meta:
        model = BasePage
        fields = (
            "id",
            "slug",
            "title",
            "url",
            "last_published_at",
        )


class BlogPageSerializer(BasePageSerializer):
    class Meta:
```

```
        model = BlogPage
        fields = BasePageSerializer.Meta.fields


class PostPageSerializer(BasePageSerializer):
    tags = TagField()
    categories = CategoryField()
    header_image = ImageRenditionField("max-1000x800")

    class Meta:
        model = PostPage
        fields = BasePageSerializer.Meta.fields + (
            "tags",
            "categories",
            "body",
            "header_image",
        )
```

Notes:

1. We add `BasePageSerializer` and put all common fields there.
2. Here we add `fields.StreamField` to the `serializer_field_mapping` to make the `BasePageSerializer` can serialize `StreamField` automatically. (`body = StreamField()` is not required anymore)
3. In `BlogPageSerializer` and `PostPageSerializer`, we can add custom fields using `BasePageSerializer.Meta.fields + (...)`

7.5 ViewSet

```
$ mkdir nextjs
$ touch nextjs/__init__.py
$ touch nextjs/api.py

nextjs
├── __init__.py
└── api.py
```

Add `nextjs` to the `INSTALLED_APPS` in *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
  ...

  'blog',
  'nextjs',
]
```

Update *nextjs/api.py*

```
from django.urls import path
from rest_framework import serializers
from wagtail.api.v2.router import WagtailAPIRouter
from wagtail.api.v2.views import BaseAPIViewSet, PagesAPIViewSet
from wagtail.core.models import Page
from blog.serializers import CategorySerializer, TagSerializer
from blog.models import BlogCategory, Tag


api_router = WagtailAPIRouter("nextjs")
```

```
api_router.register_endpoint('pages', PagesAPIViewSet)


class CategoryAPIViewSet(BaseAPIViewSet):
    base_serializer_class = CategorySerializer
    filter_backends = []
    meta_fields = []
    body_fields = ['id', 'slug', 'name']
    listing_default_fields = ['id', 'slug', 'name']
    nested_default_fields = []
    name = 'category'
    model = BlogCategory


api_router.register_endpoint("category", CategoryAPIViewSet)


class TagAPIViewSet(BaseAPIViewSet):
    base_serializer_class = TagSerializer
    filter_backends = []
    meta_fields = []
    body_fields = ['id', 'slug', 'name']
    listing_default_fields = ['id', 'slug', 'name']
    nested_default_fields = []
    name = 'tag'
    model = Tag


api_router.register_endpoint("tag", TagAPIViewSet)
```

Edit *nextjs_wagtail_app/urls.py*

```
from nextjs.api import api_router


urlpatterns = [
    ...
    path('api/v1/nextjs/', api_router.urls),       # new

    # For anything not caught by a more specific rule above, hand over to
    # Wagtail's serving mechanism
    path('', include(wagtail_urls)),
]
```

Notes:

1. We add two ViewSet to `nextjs/api.py`, which inherit from `wagtail.api.v2.views.BaseAPIViewSet`
2. You can also use DRF `viewsets.ModelViewSet` if you like.

Now if you visit http://127.0.0.1:8000/api/v1/nextjs/category/


```
{
    "meta": {
        "total_count": 2
    },
    "items": [
        {
            "id": 1,
            "slug": "programming",
            "name": "Programming"
        },
        {
            "id": 2,
```

```
            "slug": "life",
            "name": "Life"
        }
    ]
}
```

Let's keep adding ViewSet to the *nextjs/api.py*

```
class PageRelativeUrlListSerializer(serializers.Serializer):
    def to_representation(self, obj):
        return {
            "title": obj.title,
            "relative_url": obj.url,
        }


class PageRelativeUrlListAPIViewSet(PagesAPIViewSet):
    """Return all pages and their relative url"""
    model = Page

    def get_serializer(self, qs, many=True):
        return PageRelativeUrlListSerializer(qs, many=many)

    @classmethod
    def get_urlpatterns(cls):
        return [
            path("", cls.as_view({"get": "listing_view"}), name="listing"),
        ]


api_router.register_endpoint("page_relative_urls", PageRelativeUrlListAPIViewSet)
```

Notes:

1. We add `PageRelativeUrlListAPIViewSet`, which inherit from `wagtail.api.v2.views.PagesAPIViewSet`
2. It will return all page title and urls, we will need the data in later chapter.

If we visit http://127.0.0.1:8000/api/v1/nextjs/page_relative_urls/

```
{
    "meta": {
        "total_count": 5
    },
    "items": [
        {
            "title": "BlogPage",
            "relative_url": "/"
        },
        {
            "title": "PostPage1",
            "relative_url": "/postpage1/"
        },
        {
            "title": "PostPage2",
            "relative_url": "/postpage2/"
        },
        {
            "title": "PostPage3",
            "relative_url": "/postpage3/"
        },
        {
```

```
            "title": "PostPage4",
            "relative_url": "/postpage4/"
        }
    ]
}
```

Note:

The data served by `BaseAPIViewSet` has custom pagination class, https://github.com/wagtail/wagtail/blob/v2.14/wagtail/api/v2/pagination.py

Which means it does not respect `DEFAULT_PAGINATION_CLASS` in `REST_FRAMEWORK`.

You can check https://docs.wagtail.io/en/v2.14/advanced_topics/api/v2/usage.html#pagination to learn more.

Chapter 8

Routable Page

8.1 Objective

By the end of this chapter, you should be able to:

1. Understand how to create `Routable page` in Wagtail
2. Make `Caregory` and `Tag` work with `Routable page`

8.2 Router

Wagtail pages are organized following tree structure, as each page in the tree has its own URL path, like so:

```
/
    people/
        nien-nunb/    (http://www.example.com/people/nien-nunb)
        laura-roslin/
    blog/
        post-page-1/
        post-page-2/
```

You can check more on Wagtail doc: Introduction to Trees[23]

> The RoutablePageMixin mixin provides a convenient way for a page to respond on multiple sub-URLs with different views. For example, a blog section on a site might provide several different types of index page at URLs like /blog/category/django/, /blog/tagged/python/, all served by the same page instance.

So here we will make our blog page can handle custom url like `http://127.0.0.1:8000/category/slug/` and `http://127.0.0.1:8000/tag/slug/`

Add `wagtail.contrib.routable_page` to the `INSTALLED_APPS` of *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    # code omitted for brevity

    'wagtail.contrib.forms',
    'wagtail.contrib.redirects',
    'wagtail.embeds',
    'wagtail.sites',
    'wagtail.users',
```

[23] https://docs.wagtail.io/en/v2.13.4/reference/pages/theory.html

```
    'wagtail.snippets',
    'wagtail.documents',
    'wagtail.images',
    'wagtail.search',
    'wagtail.admin',
    'wagtail.core',
    'wagtail.contrib.routable_page',        # new

    # code omitted for brevity
]
```

Update *blog/models.py*

```
from wagtail.contrib.routable_page.models import RoutablePageMixin, route


class BlogPage(RoutablePageMixin, BasePage):
    description = models.CharField(max_length=255, blank=True,)

    content_panels = Page.content_panels + [FieldPanel("description", classname="full")]

    def get_context(self, request, *args, **kwargs):
        context = super().get_context(request, *args, **kwargs)
        context['page_component']['children_pages'] = [
            post.get_component_data()
            for post in self.posts
        ]
        return context

    def get_posts(self):
        return PostPage.objects.descendant_of(self).live()

    @route(r'^tag/(?P<tag>[-\w]+)/$')
    def post_by_tag(self, request, tag, *args, **kwargs):
        self.posts = self.get_posts().filter(tags__slug=tag)
        return self.serve(request)

    @route(r'^category/(?P<category>[-\w]+)/$')
    def post_by_category(self, request, category, *args, **kwargs):
        self.posts = self.get_posts().filter(categories__blog_category__slug=category)
        return self.serve(request)

    @route(r'^$')
    def post_list(self, request, *args, **kwargs):
        self.posts = self.get_posts()
        return self.serve(request)
```

Notes:

1. Update `blog.BlogPage` to make it inherit from both `wagtail.contrib.routable_page.models.RoutablePageMixin` and `BasePage`
2. Please make sure the `RoutablePageMixin` is before the `BasePage`, if not, the router function would fail.
3. We added three routes, the parameters passed in the `route` decorator is a regex expression. If you are new to this, please check Django doc: regular expressions[24]
4. `get_posts` is a common method which return the public `PostPage` of the `BlogPage`. The routes would then filter and set the value to `self.posts`.

[24] https://docs.djangoproject.com/en/3.1/topics/http/urls/#using-regular-expressions

5. The `route` works similar with Django view, here we use `return self.serve(request)` to return the Response back to visitor.

8.3 Context

Let's check *blog/models.py*

```
class BlogPage(RoutablePageMixin, Page):

    # code omitted for brevity

    def get_context(self, request, *args, **kwargs):
        context = super().get_context(request, *args, **kwargs)
        context['page_component']['children_pages'] = [
            post.get_component_data()
            for post in self.posts
        ]
        return context
```

Notes:

1. In the `BasePage.serve` method, we call `self.get_context(request, *args, **kwargs)` to get the `context` object.
2. In `BlogPage.get_context`, we overwrite the method to call `post.get_component_data` to get the JSON representation of the **child page**.
3. Now `children_pages` should also be available in the JSON response.

Let's run our project

```
$ docker-compose up -d --build
$ docker-compose logs -f
```

1. Visit http://127.0.0.1:8000[25]
2. Visit http://127.0.0.1:8000/category/programming/
3. Visit http://127.0.0.1:8000/category/test/
4. Visit http://127.0.0.1:8000/tag/django/
5. Visit http://127.0.0.1:8000/tag/test/

You might need to change the url to test on your local env, after the test, you will see the route is working.

8.4 Reversing route urls

Next, let's see how to generate links for the category and tag.

Since widgets in the sidebar are needed in both `BlogPage` and `PostPage`, let's add relevant code in the `BasePage`.

Let's update *blog/models.py*

```
class BasePage(Page):

    def categories_list(self, context):
        categories = BlogCategory.objects.all()
```

[25] http://127.0.0.1:8000/

```
        blog_page = context['blog_page']
        data = [{
            'name': category.name,
            'slug': category.slug,
            'url': blog_page.url + blog_page.reverse_subpage(
                "post_by_category",
                args=(
                    category.slug,
                )
            )
        } for category in categories]

        return data

    def tags_list(self, context):
        tags = Tag.objects.all()

        blog_page = context['blog_page']
        data = [{
            'name': tag.name,
            'slug': tag.slug,
            'url': blog_page.url + blog_page.reverse_subpage(
                "post_by_tag",
                args=(
                    tag.slug,
                )
            )
        } for tag in tags]

        return data

    def get_context(self, request, *args, **kwargs):
        context = super().get_context(request)
        context['page_component'] = self.get_component_data()

        if 'blog_page' not in context:
            context['blog_page'] = BlogPage.objects.first()
        context['page_component']['categories_list'] = self.categories_list(context)
        context['page_component']['tags_list'] = self.tags_list(context)
        return context
```

Notes:

1. In the `BasePage.get_context`, if `blog_page` is not in the context, we try to get it and add it to context.
2. In `BasePage.tags_list` and `BasePage.categories_list` methods, `blog_page.reverse_subpage` can help us generate the url by using the route rule. It is very similar with Django reverse[26]
3. Now `categories_list` and `tags_list` should also be available in the JSON response.

If you visit http://127.0.0.1:8000[27]

```
{
  "page_type":"BlogPage",
  "page_content":{
    "id":3,
    "slug":"blogpage",
    "title":"BlogPage",
    "url":"/",
    "last_published_at":"2021-08-11T02:44:45.918000Z"
  },
```

[26] https://docs.djangoproject.com/en/3.1/ref/urlresolvers/#reverse
[27] http://127.0.0.1:8000/

```
  "categories_list":[
    {
      "name":"Programming",
      "slug":"programming",
      "url":"/category/programming/"
    },
    {
      "name":"Life",
      "slug":"life",
      "url":"/category/life/"
    }
  ],
  "tags_list":[
    {
      "name":"Django",
      "slug":"django",
      "url":"/tag/django/"
    },
    {
      "name":"Wagtail",
      "slug":"wagtail",
      "url":"/tag/wagtail/"
    },
    {
      "name":"React",
      "slug":"react",
      "url":"/tag/react/"
    }
  ],
  "children_pages":[
  ]
}
```


Notes:

1. `page_type` and `page_content` are added by `blog_page.get_component_data`
2. `categories_list` and `tags_list` are added in `BasePage.get_context`
3. `children_pages` is added in `BlogPage.get_context`

If we want to add more custom data to the JSON response of the page, we can:

1. Edit the `serializer_class` to add more fields.
2. Add some data to the `page_component` in `get_context` method.

Chapter 9

Pagination

9.1 Objectives

By the end of this chapter, you should be able to:

1. Learn how to build `pagination` with Django `Paginator`

9.2 Pagination

Update *blog/models.py*

```
from django.core.paginator import EmptyPage, PageNotAnInteger, Paginator


class BlogPage(RoutablePageMixin, BasePage):

    @route(r'^tag/(?P<tag>[-\w]+)/(?:page-(?P<page_num>\d+)/)?')
    def post_by_tag(self, request, tag, page_num=1, *args, **kwargs):
        self.page_num = int(page_num)
        self.posts = self.get_posts().filter(tags__slug=tag)
        return self.serve(request)

    @route(r'^category/(?P<category>[-\w]+)/(?:page-(?P<page_num>\d+)/)?')
    def post_by_category(self, request, category, page_num=1, *args, **kwargs):
        self.page_num = int(page_num)
        self.posts = self.get_posts().filter(categories__blog_category__slug=category)
        return self.serve(request)

    @route(r'^(?:page-(?P<page_num>\d+)/)?$')
    def post_list(self, request, page_num=1, *args, **kwargs):
        self.page_num = int(page_num)
        self.posts = self.get_posts()
        return self.serve(request)
```

Notes:

1. If you do not understand how `page_num` work in regex url, please check https://docs.djangoproject.com/en/3.2/topics/http/urls/#nested-arguments to learn more
2. For example, `/page-xxx` or `/` can be both handled by `post_list` method.

9.3 Context

Let's update *blog/models.py*

```
class BlogPage(RoutablePageMixin, BasePage):

    def get_context(self, request, *args, **kwargs):
        context = super().get_context(request, *args, **kwargs)

        # https://docs.djangoproject.com/en/3.1/topics/pagination/#using-paginator-in-a-view-
↪function
        per_page = 2
        paginator = Paginator(self.posts, per_page)
        page = self.page_num
        try:
            posts = paginator.page(page)
        except PageNotAnInteger:
            posts = paginator.page(1)
        except EmptyPage:
            posts = paginator.page(1)

        context['page_component']['children_pages'] = [
            post.get_component_data()
            for post in posts
        ]
        context['page_component']['paginator'] = {
            'per_page': per_page,
            'current_page': posts.number,
            'num_pages': posts.paginator.num_pages,
        }
        return context
```

Notes:

1. We use Django `Paginator` to do the pagination operation.
2. We add `paginator` to the `page_component`, which contains meta info of the pagination.
3. So Frontend code can use the meta info to build `pagination component`, and we will learn in later chapter.

If we check http://127.0.0.1:8000/, the data should look like this.

```
{
   "page_type":"BlogPage",
   "page_content":{
      "id":3,
      "slug":"blogpage",
      "title":"BlogPage",
      "url":"/",
      "last_published_at":"2021-08-11T02:44:45.918000Z"
   },
   "categories_list":[
     ...
   ],
   "tags_list":[
     ...
   ],
   "children_pages":[
      {
         "page_type":"PostPage",
         "page_content":{
           ...
         }
```

```
      },
      {
         "page_type":"PostPage",
         "page_content":{
            ...
         }
      }
   ],
   "paginator":{
      "per_page":2,
      "current_page":1,
      "num_pages":2
   }
}
```

Notes:

1. The data in `children_pages` is paginated, the `limit` is `2`
2. The `paginator` contains into which can be used to generate frontend pagination component.

Chapter 10

Full Text Search

10.1 Objective

By the end of this chapter, you should be able to:

1. Understand how Wagtail full text search works.

10.2 Search Backend

Update *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    # code omitted for brevity

    'wagtail.contrib.postgres_search',
]

WAGTAILSEARCH_BACKENDS = {
    'default': {
        'BACKEND': 'wagtail.contrib.postgres_search.backend',
    },
}
```

Notes:

1. We added `wagtail.contrib.postgres_search` to the `INSTALLED_APPS`
2. We config `WAGTAILSEARCH_BACKENDS` to use the `postgres_search.backend`

```
# make sure the app is running
$ docker-compose up -d
$ docker-compose logs -f

$ docker-compose exec web python manage.py migrate
```

Notes:

1. This would create a db table for `postgres search backend` to store the `search index`

10.3 Model

Update *blog/models.py*

```
from wagtail.search import index


class PostPage(BasePage):

    search_fields = Page.search_fields + [
        index.SearchField('title'),
        index.SearchField('body'),
    ]
```

Notes:

1. Here we defined `search_fields` so Wagtail would know which fields need to be processed when indexing.
2. Do not forget to put `from wagtail.search import index` at the top of the file.

After this is done, let's run command to build the index.

```
$ docker-compose up -d
$ docker-compose logs -f

$ docker-compose exec web python manage.py update_index
```

Notes:

1. We need to do this manually when we **first** setup `postgres search backend`
2. The command would extract text from the `search_fields`, and write index to the `index entry` table.
3. By default, when we do db operation (create page, edit page), the `search index` would also be updated. That is why the search function can work with latest data even we do not run `update_index` command.

Let's test in the Django shell.

```
$ docker-compose exec web python manage.py shell
```

```
>>> from blog.models import PostPage

# run search method to do full text search
>>> PostPage.objects.search('wagtail')
<SearchResults [<PostPage: PostPage4>, <PostPage: PostPage3>, <PostPage: PostPage2>, <PostPage:
↪PostPage1>]>
```

Notes:

1. The `postgres search backend` would transform the `search` method to search the Postgres `index entry` table.

10.4 Rest API

In *nextjs/api.py*, we have code

```
api_router.register_endpoint('pages', PagesAPIViewSet)
```

So we can search using Rest API like this

http://127.0.0.1:8000/api/v1/nextjs/pages/?search=wagtail&type=blog.PostPage

```
{
    "meta": {
        "total_count": 4
    },
    "items": [
        {
            "id": 7,
            "meta": {
                "type": "blog.PostPage",
                "detail_url": "http://localhost/api/v1/nextjs/pages/7/",
                "html_url": "http://localhost/postpage4/",
                "slug": "postpage4",
                "first_published_at": "2021-08-11T02:50:53.016000Z"
            },
            "title": "PostPage4"
        },
        {
            "id": 6,
            "meta": {
                "type": "blog.PostPage",
                "detail_url": "http://localhost/api/v1/nextjs/pages/6/",
                "html_url": "http://localhost/postpage3/",
                "slug": "postpage3",
                "first_published_at": "2021-08-11T02:50:39.671000Z"
            },
            "title": "PostPage3"
        },
        {
            "id": 5,
            "meta": {
                "type": "blog.PostPage",
                "detail_url": "http://localhost/api/v1/nextjs/pages/5/",
                "html_url": "http://localhost/postpage2/",
                "slug": "postpage2",
                "first_published_at": "2021-08-11T02:50:32.020000Z"
            },
            "title": "PostPage2"
        },
        {
            "id": 4,
            "meta": {
                "type": "blog.PostPage",
                "detail_url": "http://localhost/api/v1/nextjs/pages/4/",
                "html_url": "http://localhost/postpage1/",
                "slug": "postpage1",
                "first_published_at": "2021-08-11T02:50:23.986000Z"
            },
            "title": "PostPage1"
        }
    ]
}
```


Notes:

1. We can pass `search` querystring to do the full-text search. Here we pass `type` querystring because we only want to search `PostPage`

10.5 Performance Notes:

From Wagtail doc, there are multiple search backends for us to use

1. If the indexed data is huge, please check `Elasticsearch backend`
2. The default `database backend` have performance issue when dealing with big data.
3. If you use `Postgres` and the indexed data is not huge, then `postgres search backend` is the best option.
4. If you use `Mysql`, you can check wagtail-whoosh[28], I am the maintainer of this repo. If it can not work for you, then you might need to use `Elasticsearch backend`

10.6 Filter Meta

Now, we can filter posts on `BlogPage` with `category`, `tag`, let's add the `filter meta info` to the context, so the React component can use it in later chapter.

Update *blog/models.py*

```
class BlogPage(RoutablePageMixin, BasePage):

    def get_context(self, request, *args, **kwargs):
        context = super().get_context(request, *args, **kwargs)

        # https://docs.djangoproject.com/en/3.1/topics/pagination/#using-paginator-in-a-view-
↪function
        per_page = 2
        paginator = Paginator(self.posts, per_page)
        page = self.page_num
        try:
            posts = paginator.page(page)
        except PageNotAnInteger:
            posts = paginator.page(1)
        except EmptyPage:
            posts = paginator.page(1)

        context['page_component']['children_pages'] = [
            post.get_component_data()
            for post in posts
        ]
        context['page_component']['paginator'] = {
            'per_page': per_page,
            'current_page': posts.number,
            'num_pages': posts.paginator.num_pages,
        }
        context['page_component']['filter_meta'] = {                    # new
            'filter_type': getattr(self, 'filter_type', None),
            'filter_term': getattr(self, 'filter_term', None),
        }
        return context

    @route(r'^tag/(?P<tag>[-\w]+)/(?:page-(?P<page_num>\d+)/)?')
    def post_by_tag(self, request, tag, page_num=1, *args, **kwargs):
        self.page_num = int(page_num)
        self.filter_type = 'tag'                                        # new
        self.filter_term = tag

        self.posts = self.get_posts().filter(tags__slug=tag)
        return self.serve(request)

    @route(r'^category/(?P<category>[-\w]+)/(?:page-(?P<page_num>\d+)/)?')
    def post_by_category(self, request, category, page_num=1, *args, **kwargs):
```

[28] https://github.com/wagtail/wagtail-whoosh

```
        self.page_num = int(page_num)
        self.filter_type = 'category'                                   # new
        self.filter_term = category

        self.posts = self.get_posts().filter(categories__blog_category__slug=category)
        return self.serve(request)
```

Notes:

1. We added `filter_term` to make it represent the `category` or `tag`
2. We added `filter_type` to make it represent the filter type.
3. In `get_context` method, we add `filter_meta` dict to the `page_component`, which contains `filter_term` and `filter_type`.

Test http://127.0.0.1:8000/category/programming/

```
{
  "page_type": "BlogPage",
  "page_content": {
    ...
  },
  "categories_list": [
    ...
  ],
  "tags_list": [
    ...
  ],
  "children_pages": [
    ...
  ],
  "paginator": {
    ...
  },
  "filter_meta": {
    "filter_type": "category",
    "filter_term": "programming"
  }
}
```

Chapter 11

UnitTest (Part 1)

11.1 Objectives

By the end of this chapter, you should be able to:

1. Understand the basic workflow of unittest
2. Learn why it is not a good idea to use fixture file to provide test data.
3. Use `factory` packages to generate test data

11.2 Workflow

When writing unittest, you should follow `AAA` pattern.

1. `Arrange`: You would make the test env ready, such as mocking some objects or methods, or create some `test data`.
2. `Act`: Execute the code
3. `Assert`: Check returned value and other objects to make sure everything works as expected.

11.3 Fixture

When you write unittest, you need some test data such as `Site`, `Page`.

The Django doc has talked about using fixture JSON file and load it during unittest (`TestCase.fixtures`), below links can help

1. Django doc[29]
2. create test fixtures for Wagtail[30]

The fixture solution is **easy to understand and get started**, newbie developer can dump data from the DB to JSON file, then load it in unittest and use code to test.

However, there are some drawbacks:

1. The fixture file is hard to edit and maintain over time.
2. The test rely on the fixture file, and the logic of the test seems not that straightforward. (You might need to check the fixture file to figure out the test logic)

[29] https://docs.djangoproject.com/en/3.1/howto/initial-data/#providing-data-with-fixtures
[30] https://www.accordbox.com/blog/how-export-restore-wagtail-site/

3. The fixture file is slow to load.

11.4 Factory packages

To solve the above problem, we better create test data in Python.

`Factory` are some functions that create data for you, you can use Django ORM to build your own factory function like this

```
def create_tag():
    instance = Tag.objects.get_or_create(slug='test', name='test')
    return instance
```

If you write many factory like the above `create_tag`, you will see a lot of `objects.get_or_create`, and you might wonder if there is a way to improve this.

Python has a great community, and you do not need to re-invent the wheel.

`wagtail_factories` and `factory-boy` can help you!

Let's update *app/requirements.txt* to add them.

```
factory-boy==3.2.0
wagtail-factories==2.0.1
```

Since we add new dependency, let's rebuild the image.

```
$ docker-compose build
```

Next, create *blog/factories.py*

```
from factory.django import DjangoModelFactory
from factory import LazyAttribute
from factory.fuzzy import FuzzyText
from django.utils.text import slugify
from blog.models import Tag


class TagFactory(DjangoModelFactory):
    class Meta:
        model = Tag

    name = FuzzyText(length=6)
    slug = LazyAttribute(lambda o: slugify(o.name))
```

Notes:

1. We create a `TagFactory` by inherit the `DjangoModelFactory` from `factory-boy`
2. Use `Meta.model` to set the model the factory would use
3. `name = FuzzyText(length=6)` tell the name would be random strings which has `6` length.

Let's test the factory in Django shell

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from blog.factories import TagFactory
>>> obj = TagFactory.create()
>>> print(obj.name, obj.slug)
ehJBty ehjbty
```

```
# cleanup
>>> obj.delete()

>>> obj = TagFactory.create()
>>> print(obj.name, obj.slug)
kCJuPm kcjupm

# cleanup
>>> obj.delete()
```

As you can see, after we define the `TagFactory`, we can use it quickly create test data for `Tag` models without caring about the implementation details.

`factory-boy` is built on https://github.com/joke2k/faker, if you want the test data more reasonable, please check it.

11.5 Wagtail Factories

`wagtail-factories` provides similar functions for Wagtail built-in models, and it is also built on `factory-boy`

Let's add more factories to *blog/factories.py*

```
from django.utils.text import slugify
from factory import (
    LazyAttribute,
    Sequence,
)
from factory.django import DjangoModelFactory
from factory.fuzzy import (
    FuzzyText,
)
from wagtail_factories import PageFactory

from blog.models import (
    BlogCategory,
    BlogPage,
    PostPageBlogCategory,
    PostPageTag,
    PostPage,
    Tag,
)


class BlogPageFactory(PageFactory):
    class Meta:
        model = BlogPage

    title = Sequence(lambda n: "BlogPage %d" % n)


class PostPageFactory(PageFactory):
    class Meta:
        model = PostPage

    title = Sequence(lambda n: "PostPage %d" % n)


class PostPageBlogCategoryFactory(DjangoModelFactory):
    class Meta:
```

```
        model = PostPageBlogCategory


class BlogCategoryFactory(DjangoModelFactory):
    class Meta:
        model = BlogCategory

    name = FuzzyText(length=6)
    slug = LazyAttribute(lambda o: slugify(o.name))


class PostPageTagFactory(DjangoModelFactory):
    class Meta:
        model = PostPageTag


class TagFactory(DjangoModelFactory):
    class Meta:
        model = Tag

    name = FuzzyText(length=6)
    slug = LazyAttribute(lambda o: slugify(o.name))
```

1. For Wagtail page, the factory class inherit the `PageFactory` from `wagtail_factories`
2. For normal Django model, the factory class inherit the `DjangoModelFactory` from `factory-boy`
3. `title = Sequence(lambda n: "PostPage %d" % n)` would make the page has title like `PostPage 1`, `PostPage 2`, and so on.
4. We can also pass parameters to `create` method to set the value directly.

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> from blog.factories import PostPageFactory
>>> page = PostPageFactory.create()
>>> page.title
'PostPage 0'
# remember to delete the page or Wagtail would raise exception
>>> page.delete()

>>> page = PostPageFactory.create(title='we can also set in this way')
>>> page.title
'we can also set in this way'
# remember to delete the page or Wagtail would raise exception
>>> page.delete()
```

Note: You can use the factory function to generate data on your dev server, but please remember to delete them soon to avoid some weird issues (for example `NoReverseMatch at /cms-admin/` in Wagtail admin).

11.6 Test Serializer

Now, let's start writing unittest.

Edit *blog/tests.py*

```
from django.test import TestCase
from wagtail.core.models import Site
```

```
from blog.factories import (
    BlogCategoryFactory,
    PostPageBlogCategoryFactory,
    BlogPageFactory,
    PostPageTagFactory,
    PostPageFactory,
    TagFactory,
)


from blog.serializers import PostPageSerializer, CategorySerializer, TagSerializer


class TestSerializer(TestCase):
    def setUp(self):
        self.blog_page = BlogPageFactory.create()

        self.site = Site.objects.all().first()
        self.site.root_page = self.blog_page
        self.site.save()

    def test_category_serializer(self):
        # arrange
        category_1 = BlogCategoryFactory.create()

        # act
        data = CategorySerializer(category_1).data

        # assert
        assert data["id"] == category_1.pk
        assert data["name"] == category_1.name
        assert data["slug"] == category_1.slug

    def test_tag_serializer(self):
        tag_1 = TagFactory.create()

        data = TagSerializer(tag_1).data

        assert data["id"] == tag_1.pk
        assert data["name"] == tag_1.name
        assert data["slug"] == tag_1.slug
```

Notes:

1. We create a `TestView` to test `blog/serializers.py`
2. In `setUp`, we created a `BlogPage` and set it as root page of the site.

Next, let's run the Django test

```
$ docker-compose run --rm web python manage.py test --noinput -v 2

...
test_category_serializer (blog.tests.TestSerializer) ... ok
test_tag_serializer (blog.tests.TestSerializer) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.054s

OK
```

Note: In some cases, if you want to debug why the unittest fail, you can append option `--pdb` to your command like this `docker-compose run --rm web python manage.py test --noinput -v 2 --pdb`

Chapter 12

UnitTest (Part 2)

12.1 Objectives

By the end of this chapter, you should be able to:

1. Learn how to generate test data for StreamField.
2. How to use temp `MEDIA_ROOT` during test.
3. Generate code coverage reports with `Coverage.py`

12.2 Image Test Data

As I said in the previous chapter, `wagtail-factories` provides factory functions for Wagtail built-in models.

It has `ImageFactory` which can help us to generate Wagtail images quickly.

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> import factory
>>> from wagtail_factories.factories import ImageFactory
>>> img = ImageFactory(file=factory.django.ImageField(width=1000, height=1000))

# If you check `image` in Wagtail admin, you will see an image that has pure color.
# remember to delete it after check
>>> img.delete()
```

12.3 StreamField Test Data

People always complain it is not easy to create test data for the StreamField.

Here I'd like to show you a simple way to solve the problem.

Let's first get the Python representation of the sample StreamField data structure.

```
# please run code in new Django shell if you change something
$ docker-compose run --rm web python manage.py shell
```

```
>>> import pprint
>>> from blog.models import PostPage
>>> page = PostPage.objects.first().specific
>>> pprint.pprint(list(page.body.raw_data))
[{'id': '4f741399-9d5c-47ea-b4f7-e492f0cf6e54',
  'type': 'h1',
  'value': 'The Zen of Wagtail'},
 {'id': '36b0fd2f-df70-4201-8ab5-74e187d99c69',
  'type': 'paragraph',
  'value': '<p>Wagtail has been born out of many years of experience building '
           'websites, learning approaches that work and ones that don’t, and '
           'striking a balance between power and simplicity, structure and '
           'flexibility. We hope you’ll find that Wagtail is in that sweet '
           'spot.</p>'},
 {'id': '4a87b280-40b2-4621-a624-f29a918e848c',
  'type': 'image_carousel',
  'value': [3, 2]},
 {'id': 'd6fbd024-9c6c-4acf-8e35-dece3d3daf5d',
  'type': 'image_text',
  'value': {'image': 3,
            'reverse': False,
            'text': '<p>Wagtail is not an instant website in a box.</p><p>You '
                    'can’t make a beautiful website by plugging off-the-shelf '
                    'modules together - expect to write code.</p>'}},
 {'id': '0b32c34d-0155-4581-b17e-d62c7caf0b4d',
  'type': 'image_text',
  'value': {'image': 1,
            'reverse': True,
            'text': '<p>A CMS should get information out of an editor’s head '
                    'and into a database, as efficiently and directly as '
                    'possible.</p>'}}]
```

If you check `value` of `image_text`, you will see `'image': 3`, here `3` is the `pk` of the Wagtail image

We can copy the above Python code to our unittest and then modify the value of the image `pk` generated by the `ImageFactory`

After that, we use `json.dumps` to convert it from Python list to `JSON` format, set it to `body` when creating the page.

```
post_page = PostPageFactory.create(
    parent=self.blog_page, body=json.dumps(body_data)
)
```

Then the StreamField test data problem is resolved, and you will see full code in the next section.

12.4 Write Test

Now let’s start writing test.

Update *blog/tests.py*

```
import json
from django.test import TestCase
from wagtail.core.models import Site
import factory
from wagtail_factories.factories import ImageFactory

from blog.factories import (
    BlogCategoryFactory,
    PostPageBlogCategoryFactory,
```

```
    BlogPageFactory,
    PostPageTagFactory,
    PostPageFactory,
    TagFactory,
)

from blog.serializers import PostPageSerializer, CategorySerializer, TagSerializer


class TestSerializer(TestCase):
    def setUp(self):
        self.blog_page = BlogPageFactory.create()

        self.site = Site.objects.all().first()
        self.site.root_page = self.blog_page
        self.site.save()

    def test_post_page_body(self):
        img_1 = ImageFactory(file=factory.django.ImageField(width=1000, height=1000))
        img_2 = ImageFactory(file=factory.django.ImageField(width=1000, height=1000))
        img_3 = ImageFactory(file=factory.django.ImageField(width=1000, height=1000))

        body_data = [
            {
                'type': 'h1',
                'value': 'The Zen of Wagtail'
            },
            {
                'type': 'paragraph',
                'value': '<p>Wagtail has been born out of many years of experience building '
                         'websites, learning approaches that work and ones that don’t, and '
                         'striking a balance between power and simplicity, structure and '
                         'flexibility. We hope you’ll find that Wagtail is in that sweet '
                         'spot.</p>'
            },
            {
                'type': 'image_carousel',
                'value': [img_1.pk, img_2.pk]
            },
            {
                'type': 'image_text',
                'value': {
                    'image': img_3.pk,
                    'reverse': False,
                    'text': '<p>Wagtail is not an instant website in a box.</p><p>You '
                            'can’t make a beautiful website by plugging off-the-shelf '
                            'modules together - expect to write code.</p>'
                }
            },
            {
                'type': 'image_text',
                'value': {
                    'image': img_2.pk,
                    'reverse': True,
                    'text': '<p>A CMS should get information out of an editor’s head '
                            'and into a database, as efficiently and directly as '
                            'possible.</p>'
                }
            }
        ]

        post_page = PostPageFactory.create(
            parent=self.blog_page, body=json.dumps(body_data), header_image=img_3
```

```
        )

        data = PostPageSerializer(post_page).data

        # return the correct block value
        assert data["body"][1]["value"] == body_data[1]["value"]

        # check get_api_representation
        assert data["body"][3]["type"] == "image_text"
        assert data["body"][3]['value']['image']['width'] == 800
```

Note:

1. In `setUp` method, we changed root page of the default site to `self.blog_page`.
2. In unittest, we created 3 images using `ImageFactory`, they all have `width=1000`
3. `body_data` is copied from the above Django shell, we replaced the image `pk` with the image pk from the `ImageFactory`.
4. All block `id` in StreamField are removed to keep the code clean. (It can still work)
5. Based on my experience, this solution is much more flexible and easy to maintain compared with editing `fixture file`.
6. Some people might ask why I do not use `wagtail-factories` to do this, because the package does not work well when dealing with some complex and nested data structure.

Let's run test

```
$ docker-compose run --rm web python manage.py test --noinput -v 2

test_category_serializer (blog.tests.TestSerializer) ... ok
test_post_page_body (blog.tests.TestSerializer) ... ok
test_tag_serializer (blog.tests.TestSerializer) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.565s

OK
```

12.5 Rest Test

Let's keep adding unittests to the `TestSerializer`

```
class TestSerializer(TestCase):

    def test_post_page_category(self):
        post_page = PostPageFactory.create(parent=self.blog_page, )

        category_1 = BlogCategoryFactory.create()
        PostPageBlogCategoryFactory.create(
            page=post_page, blog_category=category_1,
        )

        data = PostPageSerializer(post_page).data

        assert data["categories"][0]["name"] == category_1.name
        assert data["categories"][0]["slug"] == category_1.slug

        category_2 = BlogCategoryFactory.create()
        PostPageBlogCategoryFactory.create(
```

```
            page=post_page, blog_category=category_2,
        )

        data = PostPageSerializer(post_page).data
        assert len(data["categories"]) == 2

    def test_post_page_tag(self):
        post_page = PostPageFactory.create(parent=self.blog_page,)

        tag_1 = TagFactory.create()
        PostPageTagFactory.create(
            content_object=post_page, tag=tag_1,
        )

        data = PostPageSerializer(post_page).data

        assert data["tags"][0]["name"] == tag_1.name
        assert data["tags"][0]["slug"] == tag_1.slug

        tag_2 = TagFactory.create()
        PostPageTagFactory.create(
            content_object=post_page, tag=tag_2,
        )
        data = PostPageSerializer(post_page).data
        assert len(data["tags"]) == 2
```

1. Here we add unittests to make sure `categories` and `tags` is working as expected on the REST API.

```
$ docker-compose run --rm web python manage.py test --noinput -v 2

test_category_serializer (blog.tests.TestSerializer) ... ok
test_post_page_body (blog.tests.TestSerializer) ... ok
test_post_page_category (blog.tests.TestSerializer) ... ok
test_post_page_tag (blog.tests.TestSerializer) ... ok
test_tag_serializer (blog.tests.TestSerializer) ... ok

----------------------------------------------------------------------
Ran 5 tests in 0.674s

OK
```

12.6 Temp MEDIA_ROOT

Now if you check the `media/original_images` directory, you will see a lot of `example_xxx` images, which are all generated during the above tests.

How to resolve this?

We should create tmp directory and use it as `MEDIA_ROOT` during the tests.

```
import json
import tempfile                                 # new
from django.test import TestCase
from django.test import override_settings       # new
from wagtail.core.models import Site

@override_settings(MEDIA_ROOT=tempfile.gettempdir())
class TestSerializer(TestCase):
    ...
```

Notes:

1. We use Python `tempfile.gettempdir()` to get the temp directory for media files.
2. `@override_settings(MEDIA_ROOT=tempfile.gettempdir())` to make the tests use the temp directory to store media files during the test.

Now if you run tests again, the test images would not be saved to `media/original_images`.

12.7 Test Coverage

> test coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with high test coverage, measured as a percentage, has had more of its source code executed during testing, which suggests it has a lower chance of containing undetected software bugs compared to a program with low test coverage

Add Coverage.py[31] to the *requirements.txt*

```
coverage==5.5                           # new
```

Rebuild the image, and run the tests with coverage:

```
$ docker-compose up -d --build

$ docker-compose run --rm web coverage run --source='.' manage.py test --noinput -v 2
$ docker-compose run --rm web coverage report

Name                                             Stmts   Miss  Cover
------------------------------------------------------------------
blog/__init__.py                                     0      0   100%
blog/admin.py                                        1      0   100%
blog/apps.py                                         4      0   100%
blog/blocks.py                                      23      0   100%
blog/factories.py                                   30      0   100%
blog/fields.py                                      13      4    69%
blog/management/__init__.py                          0      0   100%
blog/management/commands/__init__.py                 0      0   100%
blog/management/commands/load_initial_data.py       25     25     0%
blog/migrations/0001_initial.py                      8      0   100%
blog/migrations/0002_postpage_body.py                7      0   100%
blog/migrations/0003_alter_postpage_body.py          7      0   100%
blog/migrations/__init__.py                          0      0   100%
blog/models.py                                     126     58    54%
blog/serializers.py                                 25      0   100%
blog/tests.py                                       60      0   100%
blog/views.py                                        1      1     0%
manage.py                                           12      2    83%
nextjs/__init__.py                                   0      0   100%
nextjs/api.py                                       40      2    95%
nextjs_wagtail_app/__init__.py                       0      0   100%
nextjs_wagtail_app/asgi.py                           4      4     0%
nextjs_wagtail_app/settings.py                      25      0   100%
nextjs_wagtail_app/urls.py                          11      2    82%
nextjs_wagtail_app/wsgi.py                           4      4     0%
------------------------------------------------------------------
TOTAL                                              426    102    76%
```

You can also create an HTMl report to get more info (which line is not executed):

[31] https://coverage.readthedocs.io/

```
$ docker-compose run --rm web coverage html
$ open htmlcov/index.html

# check in your browser
```

12.8 Next Step:

Now you should have a great understanding about the unittest in Wagtail.

However, the code in `blog/tests.py` is all for `blog/serializers.py`, the code in `blog/models.py` is not fully tested yet.

To test models, we can create file structures like this:

```
├── blog
│   ├── tests
│   │   ├── __init__.py
│   │   ├── test_serializers.py
│   │   └── test_views.py
```

1. Move the `blog/tests.py` to `blog/tests/test_serializers.py`
2. Write test for `blog/models.py` in `blog/tests/test_views.py`, we can use Django test client to visit the page, and check if the JSON response is correct.
3. Do not forget to create `nextjs/tests.py` to test `nextjs/api.py`

Chapter 13

Setup Next.js project

13.1 Objectives

By the end of this chapter, you should be able to:

1. Create Next.js project using `create-next-app`

13.2 Frontend Workflow

In the previous chapters, we already built the API backend.

From this chapter, we will start building the frontend app (`nextjs` project)

The frontend app would be built with modern frontend tech such as `ES6`, `React`, and `SCSS`, it would fetch data from the REST API and display the content on the web page.

13.3 Create Project

First, please make sure you have `node` installed. It is recommended to use nvm[32] to install `node` on your local env.

```
$ node -v
v14.17.4

$ npm -v
6.14.14
```

```
# cd to the root directory
$ ls

manage.py
media
compose
docker-compose.yml
nextjs_wagtail_app
blog
requirements.txt
```

[32] https://github.com/nvm-sh/nvm

Next, We'll use the create-next-app[33] to generate a boilerplate that's all set up and ready to go. (It works like cookiecutter-django[34])

```
$ npx create-next-app frontend

# then wait for some mins

Success! Created frontend at ...
Inside that directory, you can run several commands:

  yarn dev
    Starts the development server.

  yarn build
    Builds the app for production.

  yarn start
    Runs the built app in production mode.

We suggest that you begin by typing:

  cd frontend
  yarn dev
```

13.4 Project structure

Let's check the `frontend` directory.

```
.
├── README.md
├── next.config.js
├── node_modules
├── package.json
├── pages
├── public
├── styles
└── yarn.lock
```

1. `node_modules` contains the dependency packages.

Let's take a look at the `package.json`

```
{
  "name": "frontend",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  },
  "dependencies": {
    "next": "11.0.1",
    "react": "17.0.2",
    "react-dom": "17.0.2"
  },
  "devDependencies": {
```

[33] https://nextjs.org/docs/api-reference/create-next-app
[34] https://github.com/pydanny/cookiecutter-django

```
    "eslint": "7.31.0",
    "eslint-config-next": "11.0.1"
  }
}
```

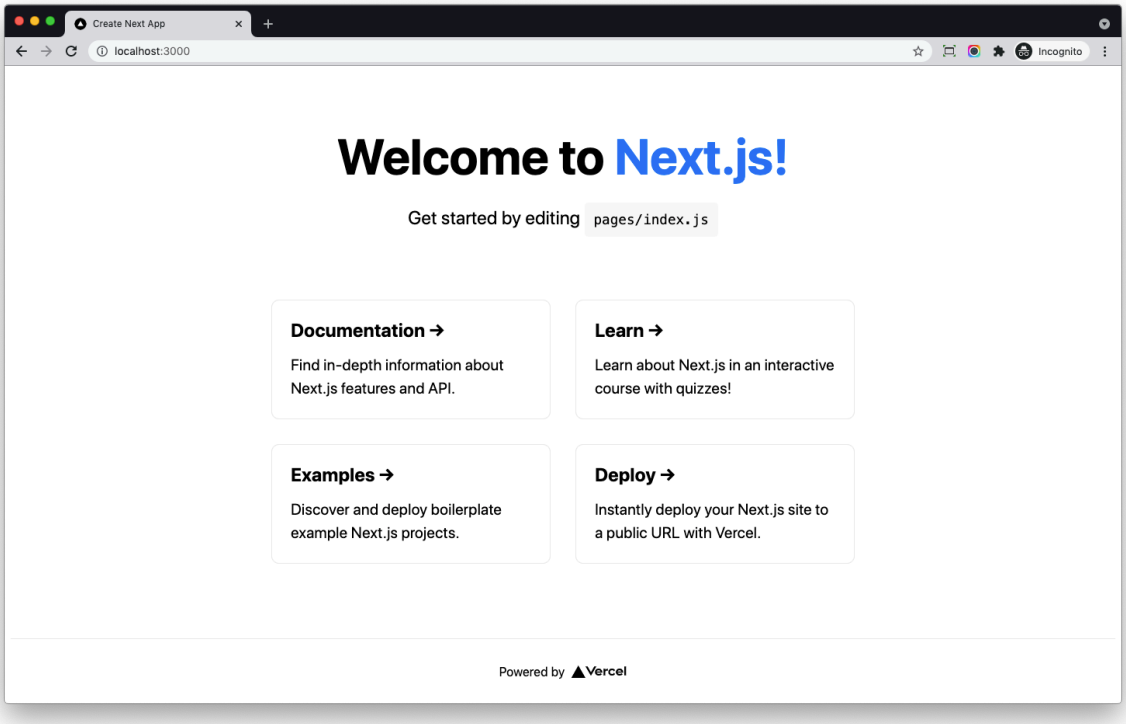
1. The `scripts` contains all available command we can run in this project.
2. The `dependencies` contains all dependency and it works like `requirements.txt` in python

13.5 Simple Test

Let's run simple test to check if the setup is working.

```
$ cd frontend
$ yarn dev
```

Now check http://localhost:3000/ on your browser, you will see `Welcome to Next.js`, which means the setup is working.



13.6 Config hosts

Next, let's use `www.local.test` for frontend project and `api.local.test` for backend API service.

Below command is for Mac, but you can ask Google for help if you use other OS.

```
$ sudo vi /etc/hosts

# add to the bottom
127.0.0.1 www.local.test
127.0.0.1 api.local.test
```

```
# clear DNS cache
$ sudo killall -HUP mDNSResponder
```

After you are done, please check on http://www.local.test:3000/

Chapter 14

Add SCSS support to Next.js project

14.1 Objectives

By the end of this chapter, you should be able to:

1. Make SCSS work with your Next.js project
2. Compile bootstrap theme in your Next.js project.

14.2 Bootstrap

This frontend app would build style based on popular open source framework Bootstrap[35]

We should use `SCSS` instead of `CSS` so we can do customization work.

We would also use React Bootstrap[36] to make the React component works with Bootstrap.

14.3 Install Dependency

By default, `create-next-app` does not support `SCSS`, so we should install some dependency here.

```
$ cd frontend
$ yarn add sass@1.32
# we specify version number here to avoid compatibility issue
$ yarn add bootstrap@5.1.0 react-bootstrap@2.0.0-beta.4
```

This is `dependencies` of *package.json*, as you can see, the above packages are already added by `yarn add` command.

```
"dependencies": {
  "bootstrap": "4.6.0",
    "next": "11.0.1",
    "react": "17.0.2",
    "react-bootstrap": "1.6.1",
    "react-dom": "17.0.2",
    "sass": "1.32"
},
```

[35] https://getbootstrap.com/
[36] https://react-bootstrap.github.io/

14.4 Import Bootstrap Theme

```
$ rm frontend/styles/Home.module.css
$ rm frontend/styles/globals.css
```

Create *frontend/styles/global.scss*

```
@import "~bootstrap/scss/bootstrap.scss";
```

Update *frontend/pages/_app.js*

```
import "../styles/global.scss";                    // new

function MyApp({ Component, pageProps }) {
  return <Component {...pageProps} />
}

export default MyApp
```

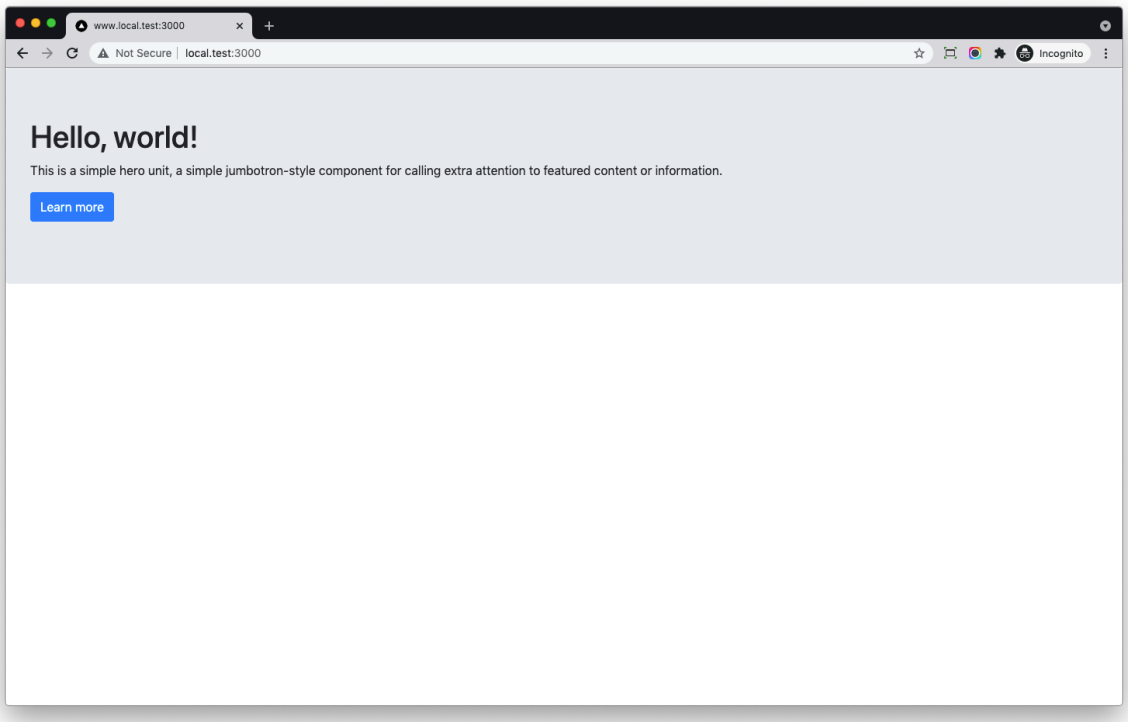
Notes:

1. This will make `../styles/global.scss` imported in all Next.js pages.

Update *frontend/pages/index.js*

```
import { Button } from "react-bootstrap";

export default function Home() {
  return (
    <div className="p-5 mb-4 bg-light">
      <div className="container-fluid py-5">
        <h1>Hello, world!</h1>
        <p>
          This is a simple hero unit, a simple jumbotron-style component for calling
          extra attention to featured content or information.
        </p>
        <Button variant="primary">Learn more</Button>
      </div>
    </div>
  );
}
```

Now check http://www.local.test:3000/, you can see something like this.


www.local.test:3000
Not Secure
local.test:3000
Incognito
Hello, world!
This is a simple hero unit, a simple jumbotron-style component for calling extra attention to featured content or information.
Learn more

Chapter 15

Building React Component (SideBar)

15.1 Objectives

By the end of this chapter, you should be able to:

1. Understand React Component `props` and `state`
2. Learn the basic syntax of React Function Component
3. Learn how to debug React component.

15.2 Basic Concepts

Before checking the content below, I wish you have a basic understanding of `props` and `state` of React Component.

15.2.1 props

`props` (short for properties) are a Component's **configuration,** its options if you may. They are received from above and **immutable** as far as the Component receiving them is concerned.

A Component **cannot change** its props, but it is responsible for putting together the props of its child Components.

15.2.2 state

The `state` starts with a default value when a Component mounts and then **suffers from mutations in time (mostly generated from user events).**

It's a **serializable** representation of one point in time—a snapshot.

A Component manages its own state internally, but—besides setting an initial state—has no business fiddling with the state of its children.

You could say the state is **private.**

We didn't say props are also serializable because it's pretty common to pass down callback functions through props.

15.2.3 React function components

1. In this course, we will React function components[37]
2. The `function component` just get `value` from the `props` and return the HTMl representation back.
3. Since `function component` is a JS function, so if you want to make it has `state` support like `class component`, you need to use React hook[38], I will talk about this in later chapter.

15.3 Pre-rendering

Next.js has two forms of pre-rendering: `Static Generation` and `Server-side Rendering`. The difference is in when it generates the HTML for a page.

- **Static Generation**: The HTML is generated at build time and will be reused on each request.
- **Server-side Rendering**: The HTML is generated on each request.

In this course, we will use **Static Generation (SSG)**, you can check Pre-rendering[39] to learn more.

15.4 First Component

Update *frontend/pages/index.js*

```
export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/postpage1/`)
  const data = await res.json()

  return {
    props: {...data}, // will be passed to the page component as props
  };
}

export default function CategoryWidget(props) {
  const categoriesList = props.categories_list;

  return (
    <div className="card mb-4">
      <h5 className="card-header">Categories</h5>
      <div className="card-body">
        <div className="row">
          <div className="col-lg-12">
            <ul className="list-unstyled mb-0">
              {categoriesList.map((category) => (
                <li key={category.slug}>
                  <a className="text-decoration-none" href={`${category.url}`}>{category.name}</a>
                </li>
              ))}
            </ul>
          </div>
        </div>
      </div>
    </div>
  );
}
```

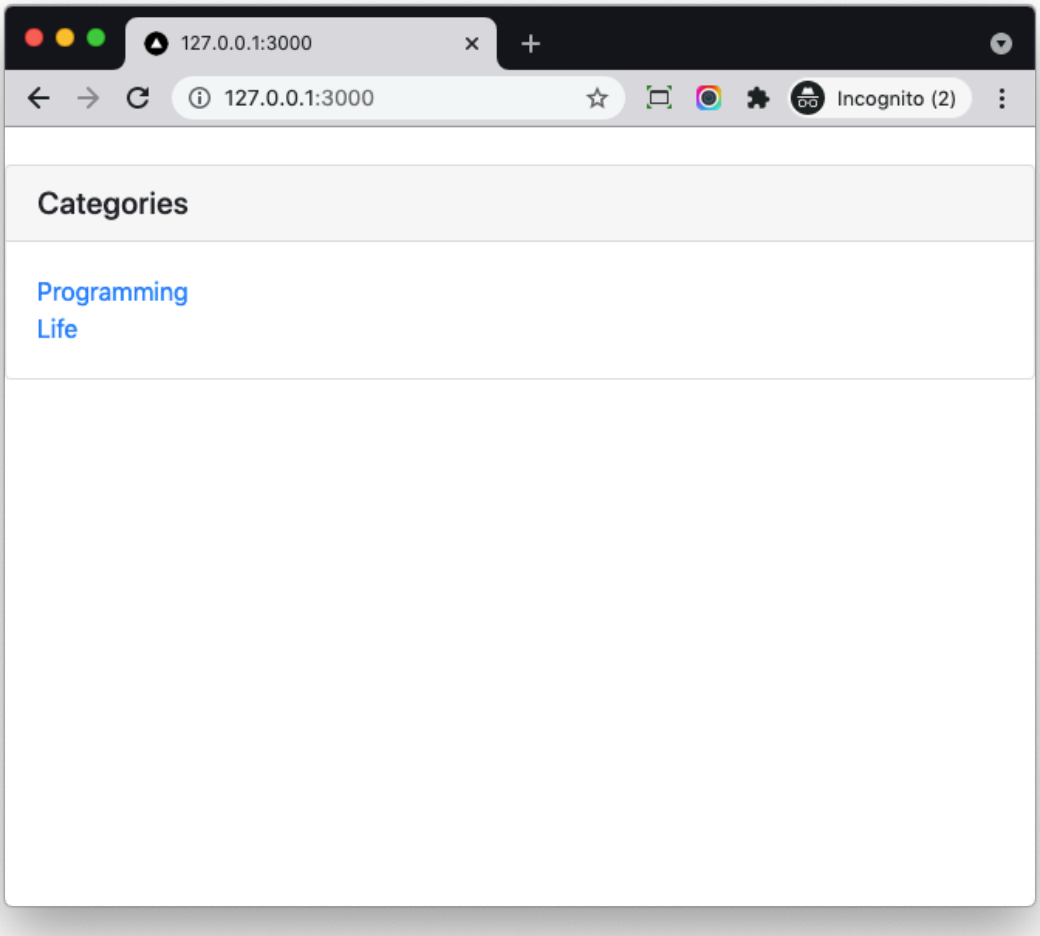
[37] https://reactjs.org/docs/components-and-props.html#function-and-class-components
[38] https://reactjs.org/docs/hooks-intro.html
[39] https://nextjs.org/docs/basic-features/pages#pre-rendering

```
$ docker-compose up -d
$ docker-compose logs -f

# run command in another terminal
$ cd frontend
$ yarn dev
```

Notes:

1. Here we export `getStaticProps` function, which `Next.js` will use to pre-render the page at build time
2. In the `getStaticProps`, we `fetch` data from backend API (a post page url), and return data in an object.
3. `{...data}` is Object spread operator, to learn more, please check https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
4. The `props` returned by `getStaticProps` will be passed to `CategoryWidget` (Next.js will find the default export of the *frontend/pages/index.js*), which will be used to render page UI.
5. `CategoryWidget` is a React functional component, it returns something like HTML, the syntax is called `JSX`. (It is more readable than pure JS code because you do not need to concatenate the HTML)
6. In `JSX`, we should use `className` if we want to specify class for HTML element, because `class` is a reserved keyword in ES6.
7. `categoriesList.map` is to render list in JSX, you can check basic-list-component[40] to learn more.

[40] https://reactjs.org/docs/lists-and-keys.html#basic-list-component



15.5 Camel case & Snake case

In Python community, people usually use Snake case[41]

However, in JS, people prefer to use Camel case[42].

Can we access the category list with `props.categoriesList;` instead of `props.categories_list;`

Let's solve this problem with `camelcase-keys`

```
$ cd frontend
$ yarn add camelcase-keys
```

Update *frontend/pages/index.js*

```
import camelcaseKeys from "camelcase-keys";                    // new

export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/postpage1/`)
  const data = await res.json()
```

[41] https://en.wikipedia.org/wiki/Snake_case
[42] https://en.wikipedia.org/wiki/Camel_case

```
  return {
    props: {...data}, // will be passed to the page component as props
  };
}
```

Notes:

1. We `import camelcaseKeys from "camelcase-keys";` at the top.

The `snake case` data would seem like this

```
{
   "page_type":"PostPage",
   "page_content":{
     ...
   },
   "categories_list":[
   ],
   "tags_list":[
   ],
}
```

The `camel case` format would seem like this

```
{
  pageType: 'PostPage',
  pageContent: {
  },
  categoriesList: [
  ],
  tagsList: [
  ],
}
```

Notes:

1. As you can see, `page_type` has been convert to `pageType`, `categories_list` has been convert to `categoriesList`.

15.6 Refactor

Next, let's create *frontend/components* directory

And create *frontend/components/CategoryWidget.js*

```
function CategoryWidget(props) {
  const { categoriesList } = props;

  return (
    <div className="card mb-4">
      <h5 className="card-header">Categories</h5>
      <div className="card-body">
        <div className="row">
          <div className="col-lg-12">
            <ul className="list-unstyled mb-0">
              {categoriesList.map((category) => (
                <li key={category.slug}>
                  <a href={`${category.url}`} className="text-decoration-none">{category.name}</a>
                </li>
              ))}
            </ul>
```

```
          </div>
        </div>
      </div>
    </div>
  );
}

export { CategoryWidget }
```

Update *frontend/pages/index.js*

```
import camelcaseKeys from "camelcase-keys";
import { CategoryWidget } from "../components/CategoryWidget";

export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/postpage1/`)
  const data = await res.json()

  return {
    props: {...camelcaseKeys(data, {deep: true})}, // will be passed to the page component as props
  };
}

export default CategoryWidget
```

Notes:

1. `camelcaseKeys(tmpData, {deep: true})`, the `deep: true` means `recurse nested objects and objects in arrays.`
2. We moved `CategoryWidget` to the `components/CategoryWidget` and let Next.js use it with `export default CategoryWidget`

15.7 TagWidget

Create *frontend/components/TagWidget.js*

```
import React from "react";
import { Badge } from "react-bootstrap";

function TagWidget(props) {
  const {tagsList} = props;

  return (
    <div className="card mb-4">
      <h5 className="card-header">Tags</h5>
      <div className="card-body">
        {tagsList.map((tag) => (
          <a key={tag.slug} href={tag.url} className="text-decoration-none">
            <Badge bg="secondary">{tag.name}</Badge>{" "}
          </a>
        ))}
      </div>
    </div>
  );
}

export { TagWidget };
```

The logic is very similar with the `CategoryWidget`

Let's keep creating *frontend/components/SideBar.js*

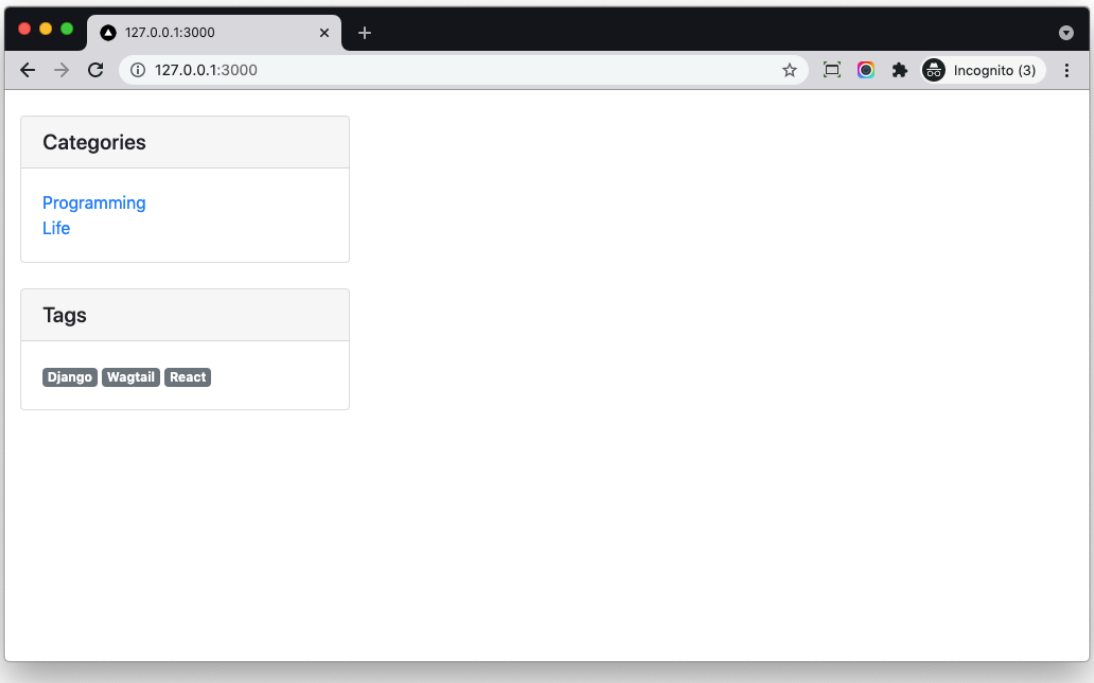
```
import { Col } from "react-bootstrap";
import { TagWidget } from "./TagWidget";
import { CategoryWidget } from "./CategoryWidget";

function SideBar(props) {
  return (
    <Col md={4}>
      <CategoryWidget {...props} />
      <TagWidget {...props} />
    </Col>
  );
}

export { SideBar };
```

Now the props is passed from `SideBar` to `TagWidget`

Update *frontend/pages/index.js*

```
import camelcaseKeys from "camelcase-keys";
import { SideBar } from "../components/SideBar";

export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/postpage1/`)
  const data = await res.json()

  return {
    props: {...camelcaseKeys(data, {deep: true})}, // will be passed to the page component as props
  };
}

export default SideBar
```

If we check http://www.local.test:3000/, we can see:

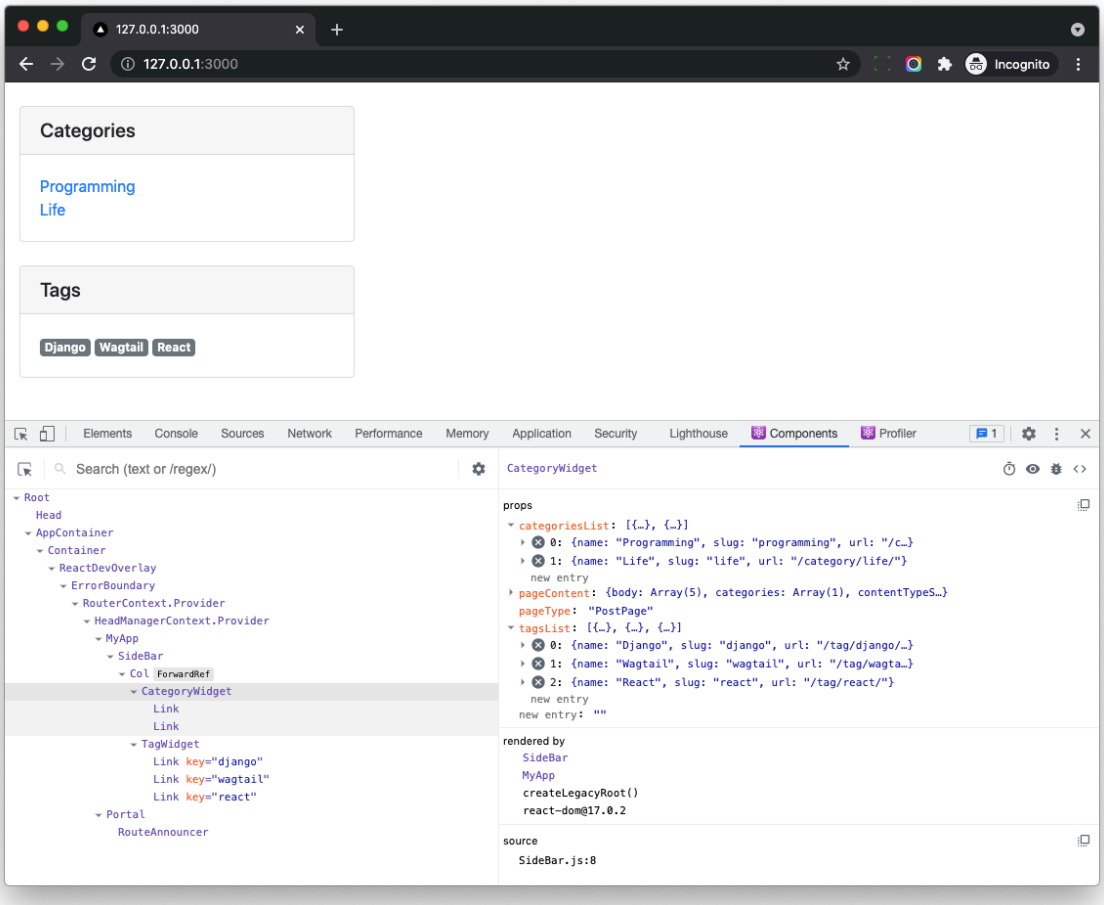


15.8 React Developer Tools

To help us better debug React components, here I recommend React Developer Tools[43]

Please install it in your browser.

[43] https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en



As you can see, we can use this tool to help us check the React component details, which is very convenient.

15.9 Component Hierarchy

Break The UI Into A Component Hierarchy has many benefits, and I recommend you to read Thinking in React[44].

[44] https://reactjs.org/docs/thinking-in-react.html

Chapter 16

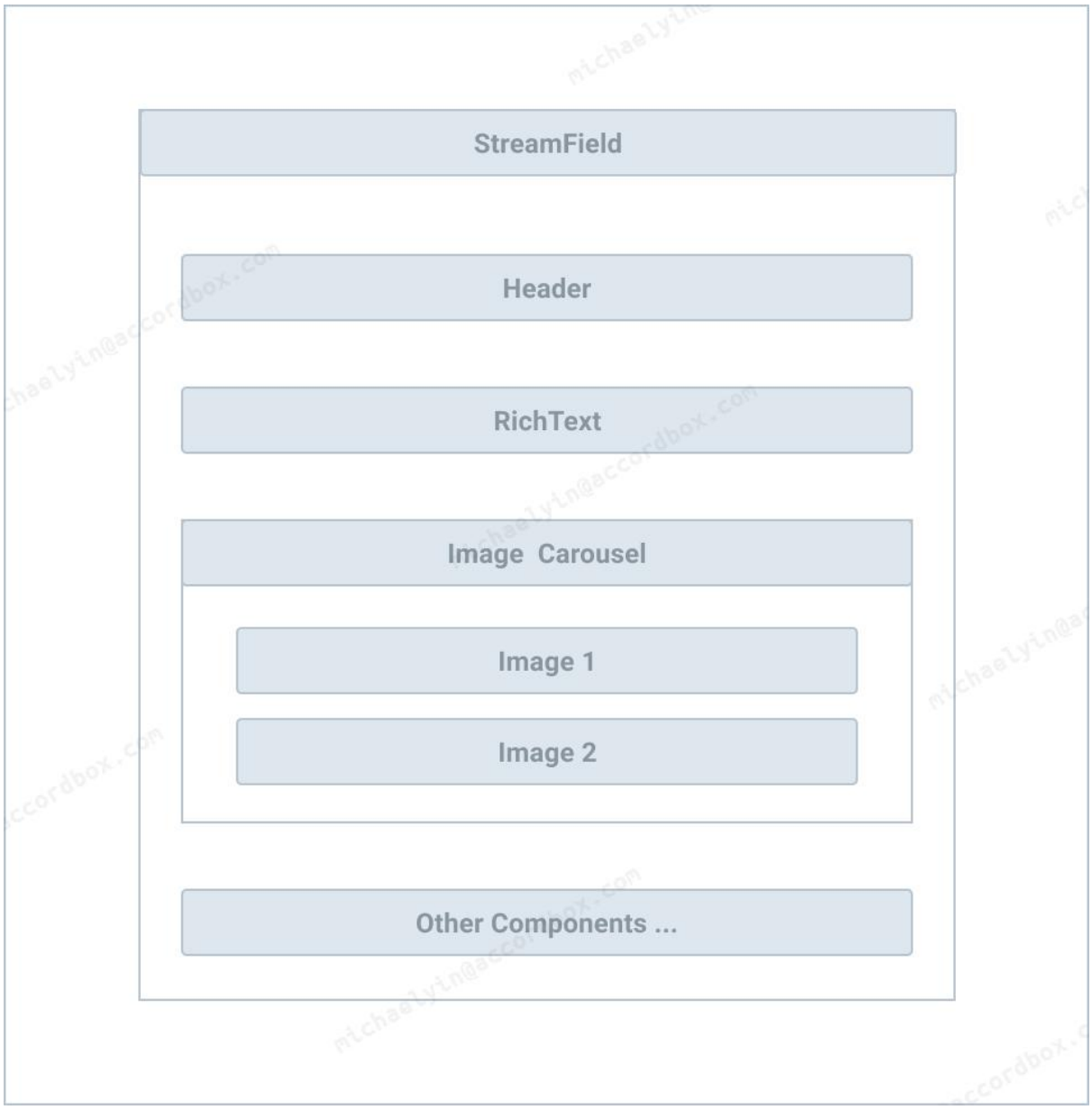
Building React Component (StreamField)

16.1 Objectives

By the end of this chapter, you should be able to:

1. Build StreamField components.

16.2 Design



16.3 BaseImage

The image object in backend API usually seem like this:

```
"image":{
   "url":"/media/images/image_3.width-800_mrgbWDQ.jpg",
   "width":800,
   "height":533,
   "alt":"image_3.jpeg"
}
```

Let's create `BaseImage` component for the image of Wagtail.

```
import React from "react";
```

```
function BaseImage({ img, className = "" }) {
  return (
    <React.Fragment>
      {/* eslint-disable-next-line */}
      <img
        className={`img-fluid ${className}`}
        alt={img.alt}
        height={img.height}
        width={img.width}
        src={`http://api.local.test:8000${img.url}`}
      />
    </React.Fragment>
  );
}

export { BaseImage };
```

Notes:

1. During development, the `url` returned by backend API is relative, so we use `http://api.local.test:8000${img.url}` to solve the problem.
2. The `http://api.local.test:8000` is hard-coded here, but we will make it configurable in later chapter.

The benefit of the `Component` is we can control specific behavior in better way.

For example, if we want to add `image lazy load` feature, we can use react-lazy-load-image-component[45]

```
<LazyLoadImage
  className={`img-fluid ${className}`}
  alt={img.alt}
  height={img.height}
  width={img.width}
  src={`${API_BASE}${img.url}`}
/>
```

16.4 StreamField Block Components

If you check *blog/blocks.py*, you will see we already have some blocks defined in the StreamField.

```
class BodyBlock(StreamBlock):
    h1 = CharBlock()
    h2 = CharBlock()
    paragraph = RichTextBlock()

    image_text = ImageText()
    image_carousel = ListBlock(CustomImageChooserBlock())
    thumbnail_gallery = ListBlock(CustomImageChooserBlock())
```

To make the code simple and easy to maintain, we can create `React Component` for respective blocks in StreamField.

16.4.1 ImageCarousel

Create *frontend/components/StreamField/ImageCarousel.js* (we will put all StreamField block components in the *frontend/components/StreamField*)

[45] https://github.com/Aljullu/react-lazy-load-image-component

```
import React from "react";
import { Carousel } from "react-bootstrap";
import { BaseImage } from "../BaseImage";

function ImageCarousel(props) {
  return (
    <div className="my-4">
      <Carousel>
        {props.value.map((item, index) => (
          <Carousel.Item key={`${index}.${item}`}>
            <BaseImage img={item} />
          </Carousel.Item>
        ))}
      </Carousel>
    </div>
  );
}

export { ImageCarousel };
```

1. The `Carousel` from `react-bootstrap` can help us make Bootstrap components work with React.

16.4.2 ImageText

Create *frontend/components/StreamField/ImageText.js*

```
import React from "react";
import { Container, Row, Col } from "react-bootstrap";
import { BaseImage } from "../BaseImage";

function ImageText(props) {
  const {value} = props;

  return (
    <Container className="py-4">
      <Row
        className={`align-items-center ${
          value.reverse ? "flex-row-reverse" : ""
        }`}
      >
        <Col xs={12} md={5}>
          <div dangerouslySetInnerHTML={{__html: value.text}}/>
        </Col>
        <Col xs={12} md={7}>
          <BaseImage img={value.image}/>
        </Col>
      </Row>
    </Container>
  );
}

export { ImageText };
```

Notes:

1. When we insert `RichText` string to React component, we need to assign it to `dangerouslySetInnerHTML`
2. You can find more details here dangerouslySetInnerHTML[46]

[46] https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml

16.4.3 ThumbnailGallery

Create *frontend/components/StreamField/ThumbnailGallery.js*

```
import React from "react";
import { Container } from "react-bootstrap";
import { BaseImage } from "../BaseImage";

function ThumbnailGallery(props) {
  const {value} = props;

  return (
    <Container>
      <div className="row text-center text-lg-left">
        {value.map((imageItem, index) => (
          <div
            className="col-lg-3 col-md-4 col-6"
            key={`${index}.${imageItem}`}
          >
            <a
              href={imageItem.url}
              className="d-block mb-4 h-100"
              target="_blank"
              rel="noopener noreferrer"
            >
              <BaseImage className="img-thumbnail" img={imageItem} />
            </a>
          </div>
        ))}
      </div>
    </Container>
  );
}

export { ThumbnailGallery };
```

We already created some StreamField block components, next, we will make them work together.

16.5 StreamField Component

Create *frontend/components/StreamField/StreamField.js*

```
import React from "react";
import { ThumbnailGallery } from "./ThumbnailGallery";
import { ImageText } from "./ImageText";
import { ImageCarousel } from "./ImageCarousel";

function StreamField(props) {
  const streamField = props.value;
  let html = [];

  for (let i = 0; i < streamField.length; i++) {
    const field = streamField[i];

    if (field.type === "h1") {
      html.push(
        <div key={`${i}.${field.type}`}>
          <h1>{field.value}</h1>
        </div>
      );
```

```
    } else if (field.type === "h2") {
      html.push(
        <div key={`${i}.${field.type}`}>
          {" "}
          <h2>{field.value}</h2>{" "}
        </div>
      );
    } else if (field.type === "paragraph") {
      html.push(
        <div key={`${i}.${field.type}`}>
          <div dangerouslySetInnerHTML={{ __html: field.value }} />
        </div>
      );
    } else if (field.type === "thumbnail_gallery") {
      html.push(
        <ThumbnailGallery value={field.value} key={`${i}.${field.type}`} />
      );
    } else if (field.type === "image_text") {
      html.push(<ImageText value={field.value} key={`${i}.${field.type}`} />);
    } else if (field.type === "image_carousel") {
      html.push(
        <ImageCarousel value={field.value} key={`${i}.${field.type}`} />
      );
    } else {
      // fallback empty div
      html.push(<div className={field.type} key={`${i}.${field.type}`} />);
    }
  }

  return html;
}

export { StreamField };
```

1. First, we import block Components we just created.
2. We build a `StreamField` Component, which iterate the `props.value` and decide which block component should be used according to the `block type`
3. We pass `field.value` to the child Component props, so they would use the `field.value` to render HTML.
4. The `key` is used to distinguish child in a list React keys[47]

16.6 PostDetail

Let's create `PostDetail` component, it would display post `title`, `header_image` and `body` (which is `StreamField`)

Create *frontend/components/PostDetail.js*

```
import React from "react";
import { StreamField } from "./StreamField/StreamField";
import { BaseImage } from "./BaseImage";

function PostDetail(props) {
  const { pageContent } = props;

  return (
```

[47] https://reactjs.org/docs/lists-and-keys.html#keys

```
    <div className="col-md-8">
      <BaseImage img={pageContent.headerImage} />
      <hr />
      <h1>{pageContent.title}</h1>
      <hr />
      <StreamField value={pageContent.body} />
    </div>
  );
}

export { PostDetail };
```

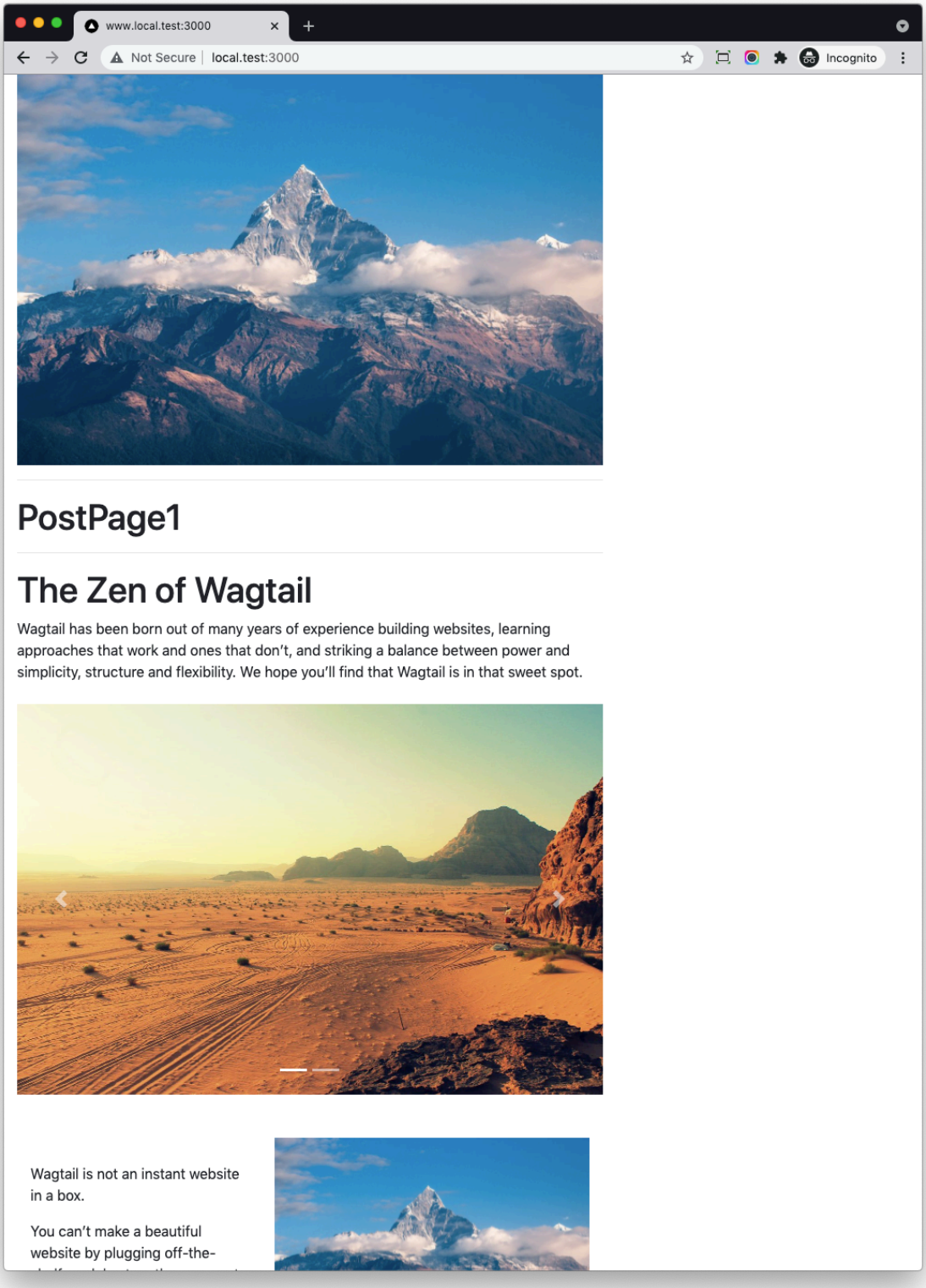
1. `<StreamField value={page_content.body}/>` would assign body to the `value` property.

Update *frontend/pages/index.js* to `export default PostDetail`

```
import camelcaseKeys from "camelcase-keys";
import { PostDetail } from "../components/PostDetail";        // new

export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/postpage1/`)
  const data = await res.json()

  return {
    props: {...camelcaseKeys(data, {deep: true})}, // will be passed to the page component as props
  };
}

export default PostDetail                    // new
```

Let's test

```
$ docker-compose up -d
$ docker-compose logs -f

# run command in another terminal
$ cd frontend
$ yarn dev
```


www.local.test:3000
Not Secure
local.test:3000
Incognito
PostPage1
The Zen of Wagtail
Wagtail has been born out of many years of experience building websites, learning approaches that work and ones that don't, and striking a balance between power and simplicity, structure and flexibility. We hope you'll find that Wagtail is in that sweet spot.
Wagtail is not an instant website in a box.
You can't make a beautiful website by plugging off-the-

Chapter 17

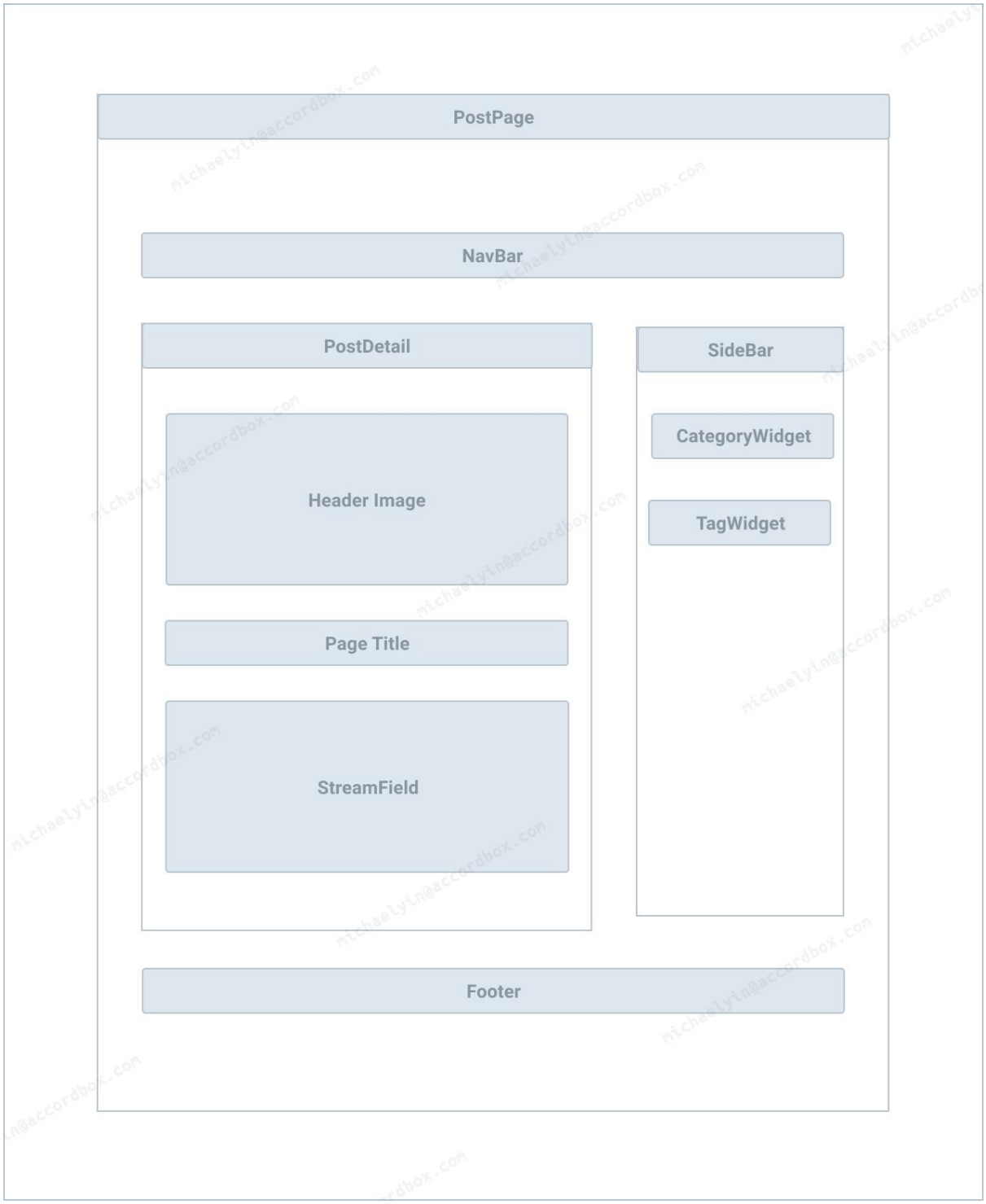
Building React Component (PostPage)

17.1 Objectives

By the end of this chapter, you should be able to:

1. Build `PostPage` component.
2. Understand Component Hierarchy better.

17.2 Design



17.3 PostPage

The `PostPage` would have classic two-column layout, top banner is the `navbar`, left part is the post content, and the right part is the sidebar.

Let's start from top to bottom and create *frontend/components/TopNav.js*

```
import React from "react";
import { Navbar, Nav, Container } from "react-bootstrap";
import Link from "next/link";

function TopNav() {
  return (
    <Navbar bg="dark" variant="dark" expand="lg" className="mb-2">
      <Container>
        <Link href="/">
          <a className="navbar-brand">Next.js Wagtail Demo</a>
        </Link>
        <Navbar.Toggle aria-controls="basic-navbar-nav" />
        <Navbar.Collapse id="basic-navbar-nav">
          <Nav className="mr-auto">
          </Nav>
        </Navbar.Collapse>
      </Container>
    </Navbar>
  );
}

export { TopNav };
```

1. `Navbar` and `Nav` from `react-bootstrap` make the code structure easy to understand.
2. Client-side transitions between routes can be enabled via the `Link` component exported by `next/link`, I will talk about this in later chapter.

Let's create *frontend/components/Footer.js*

```
import React from "react";

function Footer() {
  return (
    <footer className="py-5 bg-dark">
      <div className="container">
        <p className="m-0 text-center text-white">
          Built by <a href="https://www.accordbox.com/">MichaelYin</a>
        </p>
      </div>
    </footer>
  );
}

export { Footer };
```

Now let's create a `PostPage` component to use the above components.

Create *frontend/components/PostPage.js*

```
import React from "react";
import { Container, Row } from "react-bootstrap";
import { PostDetail } from "./PostDetail";
import { SideBar } from "./SideBar";
import { TopNav } from "./TopNav";
import { Footer } from "./Footer";

function PostPage(props) {
  return (
    <div>
      <TopNav />
      <Container>
        <Row>
```

```
          <PostDetail {...props} />
          <SideBar {...props} />
        </Row>
      </Container>
      <Footer />
    </div>
  );
}

export { PostPage };
```

Update *frontend/pages/index.js* to `export default PostPage`

```
import camelcaseKeys from "camelcase-keys";
import { PostPage } from "../components/PostPage";        // new

export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/postpage1/`)
  const data = await res.json()

  return {
    props: {...camelcaseKeys(data, {deep: true})}, // will be passed to the page component as props
  };
}

export default PostPage                     // new
```

Let's test

```
$ docker-compose up -d
$ docker-compose logs -f

# run command in another terminal
$ cd frontend
$ yarn dev
```

Next.js Wagtail Demo

Categories

Programming
Life

Tags

Django Wagtail React

PostPage1

The Zen of Wagtail

Wagtail has been born out of many years of experience building websites, learning approaches that work and ones that don't, and striking a balance between power and simplicity, structure and flexibility. We hope you'll find that Wagtail is in that sweet spot.

Wagtail is not an instant website in a box.

You can't make a beautiful website by plugging off-the-shelf modules together - expect to write code.

A CMS should get information out of an editor's head and into a database, as efficiently and directly as possible.

Built by MichaelYin

Chapter 18

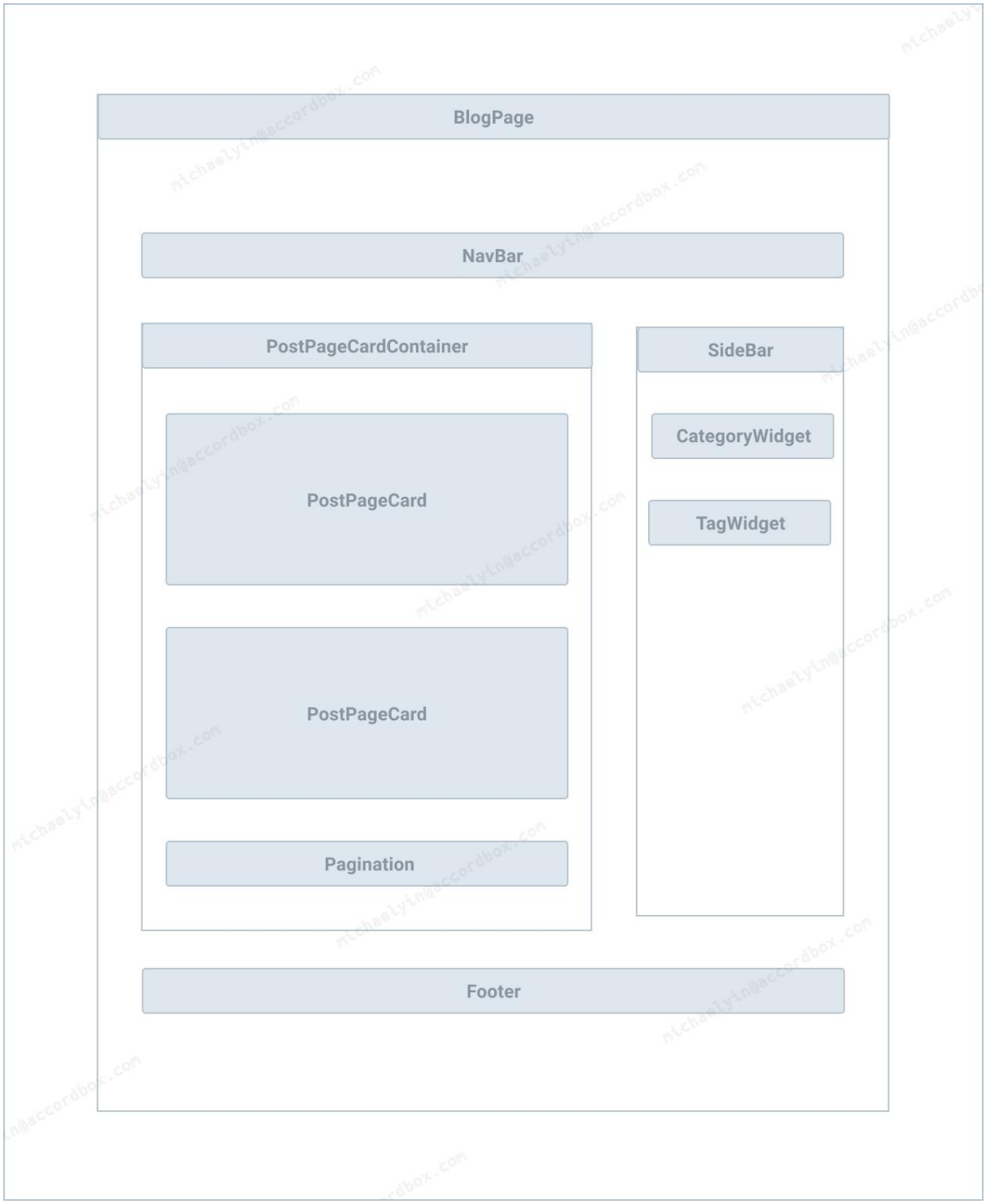
Building React Component (BlogPage)

18.1 Objectives

By the end of this chapter, you should be able to:

1. Build `BlogPage` and `PostPageCardContainer`component.

18.2 BlogPage



Create *frontend/components/BlogPage.js*

```
import React from "react";
import { Container, Row } from "react-bootstrap";
import { TopNav } from "./TopNav";
import { Footer } from "./Footer";
import { SideBar } from "./SideBar";

function BlogPage(props) {
```

```
  return (
    <div>
      <TopNav />
      <Container>
        <Row>
          <SideBar {...props} />
        </Row>
      </Container>
      <Footer />
    </div>
  );
}

export { BlogPage };
```

Notes:

1. Since `TopNav`, `Footer` and `SideBar` are already created in the previous chapter, here we use them to quickly build the page layout.

Let's create *frontend/pages/index.js*

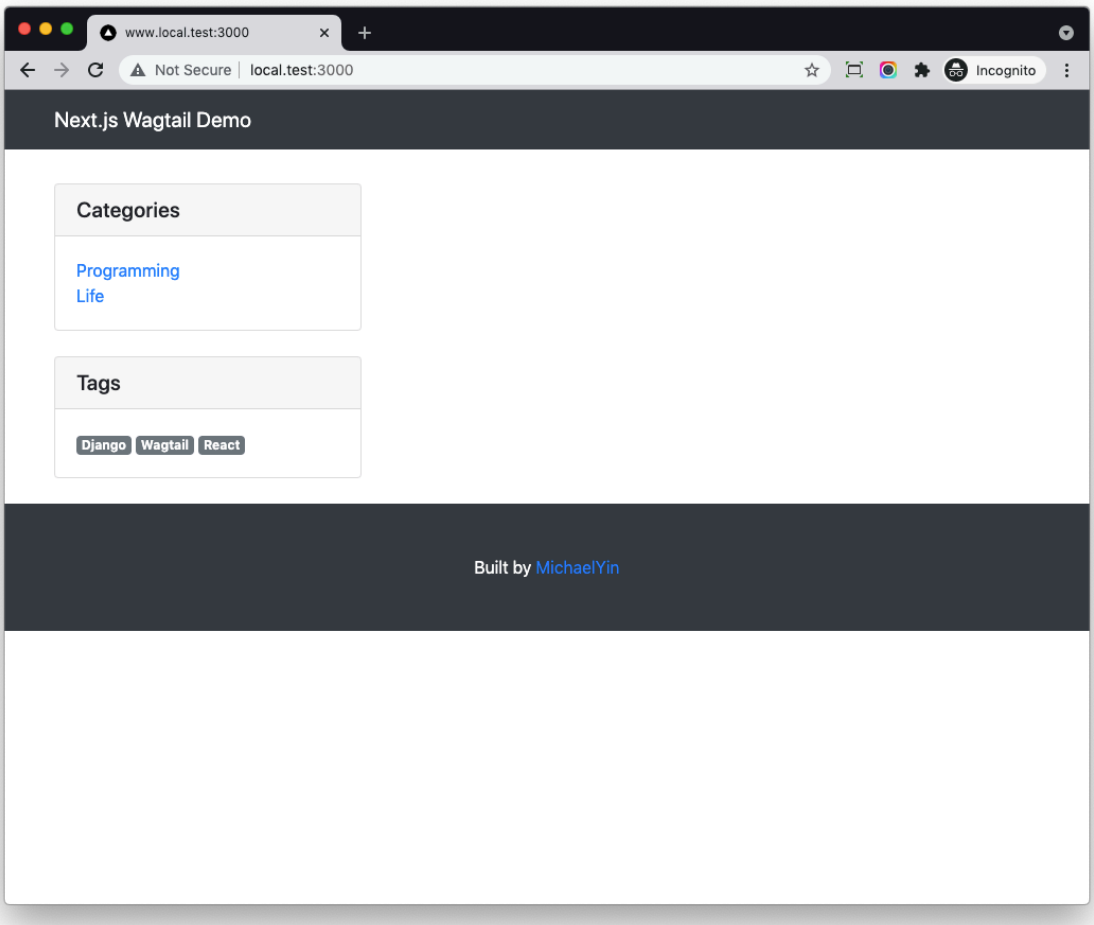
```
import camelcaseKeys from "camelcase-keys";
import { BlogPage } from "../components/BlogPage";                    // new

export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/`)              // new
  const data = await res.json()

  return {
    props: {...camelcaseKeys(data, {deep: true})}, // will be passed to the page component as props
  };
}

export default BlogPage                                               // new
```

Notes:

1. We fetch data from `http://api.local.test:8000/` because it will return JSON data of the blog page.
2. Here we `export default BlogPage`

```
$ docker-compose up -d
$ docker-compose logs -f

# run command in another terminal
$ cd frontend
$ yarn dev
```

check http://www.local.test:3000/



18.3 PostPageCard

Next, we will build `PostPageCard`, which display card info for the `PostPage`

Create *frontend/components/PostPageCard.js*

```
import React from "react";
import Link from "next/link";
import { BaseImage } from "./BaseImage";

function PostPageCard(props) {
  const {post} = props;
  const {pageContent} = post;
  const dateStr = new Date(pageContent.lastPublishedAt).toUTCString();

  return (
    <div className="card mb-4">
      <Link href={pageContent.url}>
        <a>
          <BaseImage img={pageContent.headerImage}/>
        </a>
      </Link>
      <div className="card-body">
        <h2 className="card-title">
          <a className="text-decoration-none" href={pageContent.url}>{pageContent.title}</a>
```

```
        </h2>
        <Link href={pageContent.url}>
          <a className="btn btn-primary">Read More →</a>
        </Link>
      </div>
      <div className="card-footer text-muted">Posted on {dateStr}</div>
    </div>
  );
}

export { PostPageCard };
```

Notes:

1. If you want to do more on datetime in JS, you can check day.js[48]

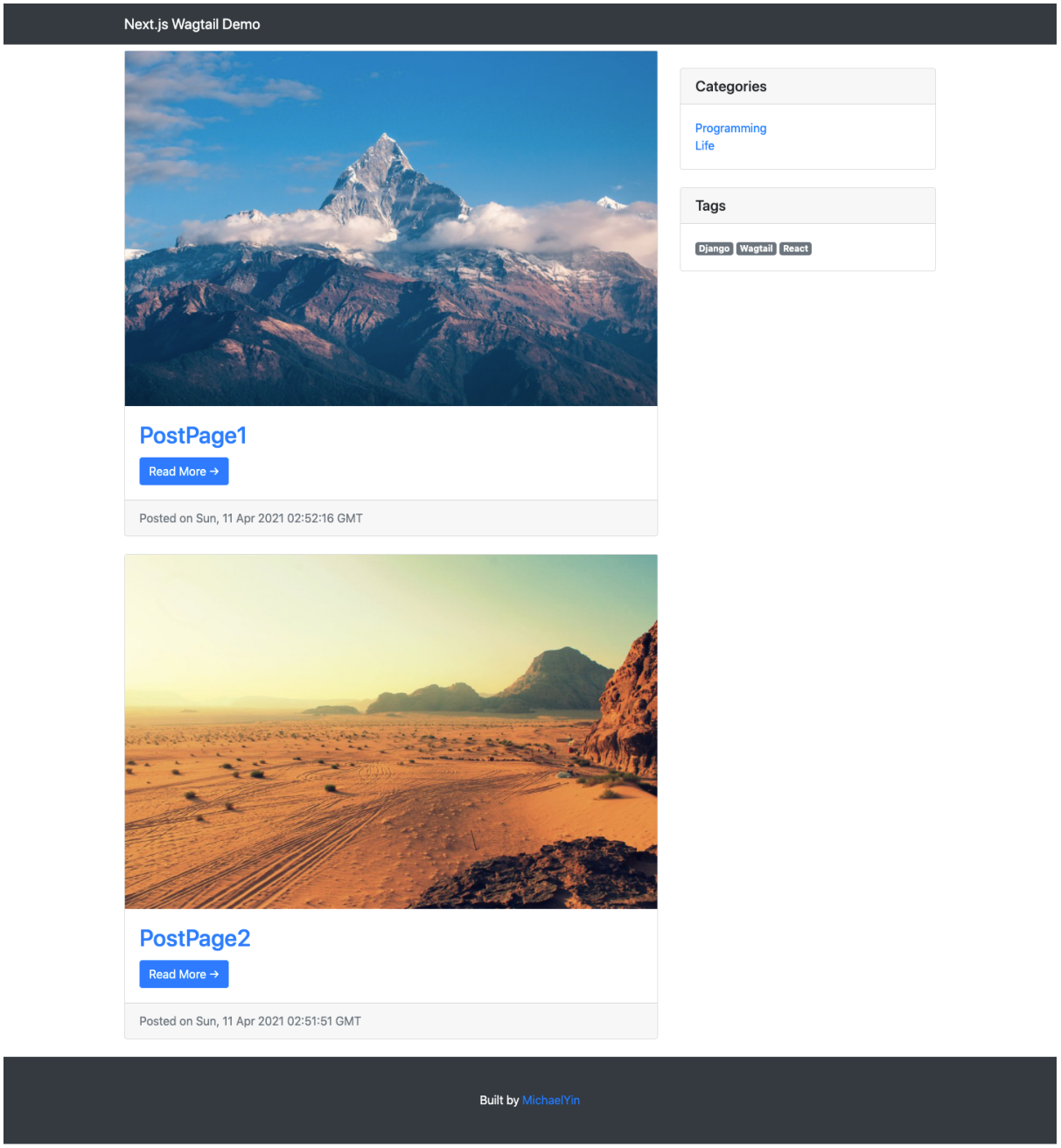
Create *frontend/components/PostPageCardContainer.js* to use the `PostPageCard`

```
import React from "react";
import { Col } from "react-bootstrap";
import { PostPageCard } from "./PostPageCard";

function PostPageCardContainer(props) {
  const {childrenPages} = props;

  return (
    <Col md={8}>
      {childrenPages.map((post) => (
        <PostPageCard post={post} key={post.pageContent.id}/>
      ))}
    </Col>
  );
}

export { PostPageCardContainer };
```

Update *frontend/components/BlogPage.js*

```
import React from "react";
import { Container, Row } from "react-bootstrap";
import { TopNav } from "./TopNav";
import { Footer } from "./Footer";
import { SideBar } from "./SideBar";
import { PostPageCardContainer } from "./PostPageCardContainer";

function BlogPage(props) {
  return (
    <div>
      <TopNav />
      <Container>
        <Row>
          <PostPageCardContainer {...props} />                // new
          <SideBar {...props} />
        </Row>
      </Container>
      <Footer />
    </div>
  );
}

export { BlogPage };
```

[48] https://day.js.org/


Next.js Wagtail Demo
Categories
Programming
Life
Tags
Django
Wagtail
React
PostPage1
Read More →
Posted on Sun, 11 Apr 2021 02:52:16 GMT
PostPage2
Read More →
Posted on Sun, 11 Apr 2021 02:51:51 GMT
Built by MichaelYin

Chapter 19

Building React Component (BlogPage Pagination)

19.1 Objectives

By the end of this chapter, you should be able to:

1. Add `Pagination` and `Filter Message` to the `BlogPage`

19.2 Pagination

In the JSON data returned by backend API, the `paginator` has `currentPage` and `numPages`, which we can use to generate the pagination links

Update *frontend/components/PostPageCardContainer.js*

```
import React from "react";
import { Col, Pagination, Alert } from "react-bootstrap";
import { PostPageCard } from "./PostPageCard";
import Link from "next/link";
import { useRouter } from "next/router";

function getPageItems(props, router) {
  const {paginator} = props;
  const {currentPage, numPages} = paginator;
  let items = [];

  const curPath = router.asPath;

  let prePageUrl, nextPageUrl;
  if (curPath.match(/\/page-[0-9]+/)) {
    prePageUrl = curPath.replace(/\/page-[0-9]+/, `/page-${currentPage - 1}`);
    nextPageUrl = curPath.replace(/\/page-[0-9]+/, `/page-${currentPage + 1}`);
  } else {
    prePageUrl = `${curPath}/page-${currentPage - 1}`.replace("//", "/");
    nextPageUrl = `${curPath}/page-${currentPage + 1}`.replace("//", "/");
  }

  items.push(
    <li
      key={prePageUrl}
      className={`page-item ${currentPage > 1 ? "" : "disabled"}`}
```

```
    >
      <Link href={prePageUrl}>
        <a className="page-link">Previous</a>
      </Link>
    </li>
  );

  items.push(
    <li
      key={nextPageUrl}
      className={`page-item ${currentPage >= numPages ? "disabled" : ""}`}
    >
      <Link href={nextPageUrl}>
        <a className="page-link">Next</a>
      </Link>
    </li>
  );

  return items;
}

function PostPageCardContainer(props) {
  const router = useRouter();
  const {childrenPages} = props;
  const pageItems = getPageItems(props, router);

  return (
    <Col md={8}>
      {childrenPages.map((post) => (
        <PostPageCard post={post} key={post.pageContent.id}/>
      ))}

      <Pagination>{pageItems}</Pagination>
    </Col>
  );
}

export { PostPageCardContainer };
```

Notes:

1. We add `getPageItems` to generate pagination items from the `paginator` and `router`.
2. If you want to access the `router` object inside any function component, you can use the `useRouter` hook

19.3 Filter Message

When we filter posts using `category widget` or `tag widget` in the sidebar, it is better to display some message.

The `filterMeta` can help us get this done.

Update *frontend/components/PostPageCardContainer.js*

```
// code omitted for brevity
function getFilterMsg(props) {                                    // new
  const { filterMeta } = props;

  let filterMsg = "";
  if (filterMeta.filterType) {
```

```
    filterMsg = (
      <Alert variant="primary">
        Results for{" "}
        <span>
        {filterMeta.filterType}: {filterMeta.filterTerm}
      </span>
      </Alert>
    );
  }
  return filterMsg;
}

function PostPageCardContainer(props) {
  const router = useRouter();
  const {childrenPages} = props;
  const pageItems = getPageItems(props, router);
  const filterMsg = getFilterMsg(props);                          // new

  return (
    <Col md={8}>
      {filterMsg}

      {childrenPages.map((post) => (
        <PostPageCard post={post} key={post.pageContent.id}/>
      ))}

      <Pagination>{pageItems}</Pagination>
    </Col>
  );
}

export { PostPageCardContainer };
```

Update *frontend/pages/index.js*

```
import camelcaseKeys from "camelcase-keys";
import { BlogPage } from "../components/BlogPage";

export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/page-2`)         // new
  const data = await res.json()

  return {
    props: {...camelcaseKeys(data, {deep: true})}, // will be passed to the page component as props
  };
}

export default BlogPage
```

Please check http://www.local.test:3000/ to see if the pagination links is generated in correct way.

You can also change the URL to fetch `http://api.local.test:8000/tag/django` to test if the `filter message` is working as expected.

Chapter 20

Next.js routing (Part 1)

20.1 Objectives

By the end of this chapter, you should be able to:

1. Learn how to do `dynamic import` in Next.js
2. Understand how Next.js routing works.

20.2 Dynamic import

In the previous chapters, we need to change the page component based on the fetch URL, which is inconvenient.

Let's solve this problem.

Create *frontend/components/LazyPages.js*

```
import dynamic from 'next/dynamic';

export default {
  PostPage: dynamic(() => import('./PostPage').then((mod) => mod.PostPage)),
  BlogPage: dynamic(() => import('./BlogPage').then((mod) => mod.BlogPage)),
};
```

Notes:

1. `LazyPages` is like a registry, all pages types are registered here.
2. The `next/dynamic` can help improve the load performance, I will talk about it in a bit.

Create *frontend/components/PageProxy.js*

```
import React from "react";
import LazyPages from './LazyPages';

function PageProxy(props) {
  const {pageType} = props;
  const PageComponent = LazyPages[pageType];

  if (PageComponent) {
    return (
      <React.Fragment>
        <PageComponent {...props} />
      </React.Fragment>
```

```
    );
  } else {
    return <div>Error when loading {pageType}</div>;
  }
}

export { PageProxy };
```

Notes:

1. We get `pageType` from the props, and then find the respective page component from `LazyPages`
2. If `PageComponent` can be found, use it to render the page UI.
3. For example, if the `pageType` is `BlogPage`, then `BlogPage` component will be imported. Since `PostPage` is not `imported` at build time, the code does not need to be downloaded, so the page will load faster.

Please check links below to know more.

1. Code splitting with dynamic imports in Next.js[49]
2. Dynamic Import[50]

Update *frontend/pages/index.js*

```
import camelcaseKeys from "camelcase-keys";
import { PageProxy } from "../components/PageProxy";                    // new

export async function getStaticProps(context) {
  // Call an external API endpoint to get posts
  const res = await fetch(`http://api.local.test:8000/`)
  const data = await res.json()

  return {
    props: {...camelcaseKeys(data, {deep: true})}, // will be passed to the page component as props
  };
}

export default PageProxy                                                // new
```

Now we can use `PageProxy` without caring about the fetch URL, the component will use respective page component from the `pageType`

Please check http://www.local.test:3000/ and it should still work without issue.

20.3 Install Dependency

In the previous chapter, we have learned to use `camelcase-keys` to convert object key to `Camel Case`

What if we want to convert object key to `Snake Case` so backend API can process in correct way?

```
$ cd frontend
$ yarn add snakecase-keys

# we need this package in a bit.
$ yarn add query-string
```

[49] https://web.dev/code-splitting-with-dynamic-imports-in-nextjs/
[50] https://nextjs.org/docs/advanced-features/dynamic-import

20.4 ENV

Create *frontend/.env.development* to add ENV to our Next.js project.

```
NEXT_PUBLIC_WAGTAIL_API_BASE=http://api.local.test:8000
```

Please check https://nextjs.org/docs/basic-features/environment-variables to learn more.

Update *frontend/components/BaseImage.js*

```
import React from "react";

const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;                // new

function BaseImage({ img, className = "" }) {
  return (
    <React.Fragment>
      {/* eslint-disable-next-line */}
      <img
        className={`img-fluid ${className}`}
        alt={img.alt}
        height={img.height}
        width={img.width}
        src={`${API_BASE}${img.url}`}                                      // new
      />
    </React.Fragment>
  );
}

export { BaseImage };
```

Please check http://www.local.test:3000/ to see if the image has been loaded successfully

20.5 wagtail.js

Create *frontend/utils/wagtail.js*

```
import querystring from "query-string";
import snakecaseKeys from "snakecase-keys";
import camelcaseKeys from "camelcase-keys";

const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;

export function cleanUrlPaths(urlArray) {
  let tmpUrls = urlArray.filter((x) => x);
  tmpUrls = tmpUrls.map((x) => x.split("/"));
  tmpUrls = tmpUrls.map((x) => x.filter((y) => y));
  tmpUrls = tmpUrls.filter((x) => x.length);
  return tmpUrls;
}

export async function getPage(path, params, options) {
  params = params || {};
  let relativePath = path
  if (relativePath.indexOf('/') !== 0) {
    relativePath = `/${relativePath}`
  }

  return await getRequest(
    `${API_BASE}${relativePath}`,
```

```
    params,
    options
  );
}

export async function getAllPages() {
  return await getPaginationContent(`${API_BASE}/api/v1/nextjs/page_relative_urls/`);
}

async function getPaginationContent(url) {
  let items = []
  while (true) {
    const data = await getRequest(
      url,
      {offset: items.length}
    )
    if (data.items.length > 0) {
      items = [...items, ...data.items];
    } else {
      break;
    }
  }

  return {items}
}

export async function getRequest(url, params, options) {
  params = params || {};
  params = snakecaseKeys(params, {deep: true});

  let headers = options?.headers || {};
  headers = {
    "Content-Type": "application/json",
    ...headers,
  };
  const queryString = querystring.stringify(params);
  const res = await fetch(`${url}?${queryString}`, {headers});

  const data = await res.json();
  return camelcaseKeys(data, {deep: true})
}
```

Notes:

1. `getRequest` is the core function here, it will send GET request to backend API to get data.
2. We can also set `params` to set querystring and `options` to set headers.
3. `getAllPages` will get all page urls and page titles
4. As we know, the Wagtail Rest API will return paginated data (WAGTAILAPI_LIMIT_MAX[51]), `getPaginationContent` can help us get all paginated data.
5. `getPage` will get data of the specific page URL.
6. `cleanUrlPaths` will convert the url from `String` to `Array`, and remove the empty path array.

Create *frontend/pages/[...path].js*

```
import { getPage, getAllPages, cleanUrlPaths } from "../utils/wagtail";
import { PageProxy } from "../components/PageProxy";
```

[51] https://docs.wagtail.io/en/stable/advanced_topics/api/v2/configuration.html?highlight=WAGTAILAPI_LIMIT_MAX#wagtailapi-limit-max

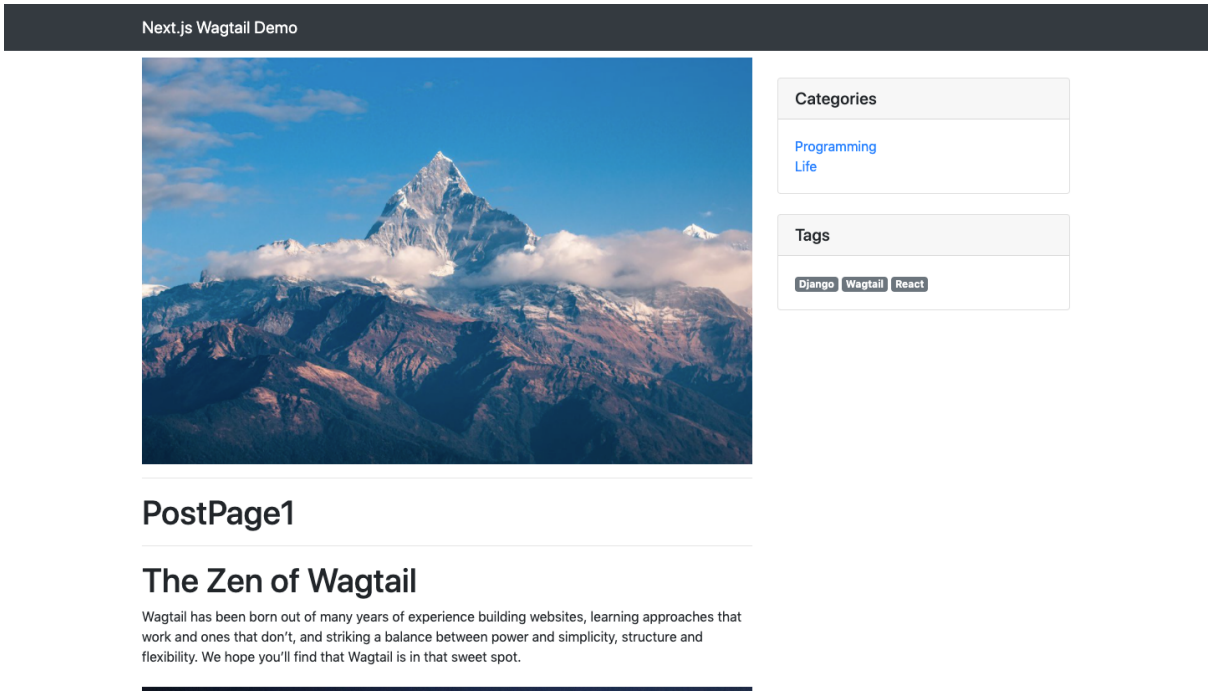
```
export default PageProxy;

async function getAllPaths() {
  const data = await getAllPages();

  let htmlUrls = data.items.map((x) => x.relativeUrl);
  htmlUrls = cleanUrlPaths(htmlUrls);

  return htmlUrls.map((x) => ({ params: { path: x } }));
}

export async function getStaticProps({ params }) {
  params = params || {};
  let path = params.path || [];
  path = path.join("/");

  const data = await getPage(path);
  return { props: data };
}

export async function getStaticPaths() {
  const postPaths = await getAllPaths();
  return {
    paths: postPaths,
    fallback: false,
  };
}
```

Notes:

1. `[...path].js` is a `Catch all` route, it can match `/postpage1`, `/blog/postpage1` and `/blog/sub/postpage1`. Please check https://nextjs.org/docs/routing/dynamic-routes#catch-all-routes to learn more.
2. `getStaticPaths` is used by Next.js in `Static Generation`, which will specify routes to pre-render pages.
3. `getStaticProps` is used by Next.js in `Static Generation`, which fetch data at built time.
4. Here, in `getStaticPaths` we fetch data from `/api/v1/nextjs/page_relative_urls/` to get all page title and urls, and then return the path back, so Next.js know which URL path it can handle.
5. In `getStaticProps`, we get `path` from the `params` object, and then use `getPage` to fetch the page JSON data from the backend API.

```
$ docker-compose up -d
$ docker-compose logs -f

# run command in another terminal
$ cd frontend
$ yarn dev
```

Try to visit http://www.local.test:3000/postpage1



The `postPaths` in `getStaticPaths` has value like this:

```
[{"params":{"path":["postpage1"]}},{"params":{"path":["postpage2"]}},{"params":{"path":["postpage3
↪"]}},{"params":{"path":["postpage4"]}}]
```

Try to visit:

- http://www.local.test:3000/postpage2[52]
- http://www.local.test:3000/postpage4

20.6 Home Page

Next.js will automatically route files named `index` to the root of the directory

```
pages/index.js → /
```

Let's update *frontend/pages/index.js*

```
export { default, getStaticProps } from "./[...path]";
```

Check http://www.local.test:3000/ and think about why this works.

[52] http://www.local.test:3000/postpage1

Chapter 21

Next.js routing (Part 2)

21.1 Objectives

By the end of this chapter, you should be able to:

1. Add pagination link to Next.js routes.
2. Learn how to add nested routes
3. Understand client-side transitions between routes

21.2 Pagination

Update *frontend/pages/[...path].js*

```
import { getPage, getAllPages, cleanUrlPaths } from "../utils/wagtail";
import { PageProxy } from "../components/PageProxy";

export default PageProxy;

async function getAllPaths() {
  const data = await getAllPages();

  let htmlUrls = data.items.map((x) => x.relativeUrl);
  htmlUrls = cleanUrlPaths(htmlUrls);

  return htmlUrls.map((x) => ({ params: { path: x } }));
}

async function getPaginationPaths() {                              // new
  const data = await getPage("/");
  let pageCount = data.paginator.numPages;
  const pageIndexArray = Array.from({length: pageCount}, (_, i) => i + 1);
  let pageUrls = pageIndexArray.map((x) => `/page-${x}`);
  pageUrls = cleanUrlPaths(pageUrls);
  return pageUrls.map((x) => ({params: {path: x}}));
}

export async function getStaticProps({ params }) {
  params = params || {};
  let path = params.path || [];
  path = path.join("/");

  const data = await getPage(path);
```

```
  return { props: data };
}

export async function getStaticPaths() {
  const postPaths = await getAllPaths();
  const paginationPaths = await getPaginationPaths();              // new
  return {
    paths: [...postPaths, ...paginationPaths],                     // new
    fallback: false,
  };
}
```

```
$ docker-compose up -d
$ docker-compose logs -f

# run command in another terminal
$ cd frontend
$ yarn dev
```

Notes:

1. We add `getPaginationPaths` to generate pagination links (`/page-1`, `/page-2`) from `paginator`.
2. The `paginationPaths` would seem like `[{"params":{"path":["page-1"]}}, {"params":{"path":["page-2"]}}]`
3. Now if user visit http://www.local.test:3000/, `frontend/pages/index.js` will be used.
4. If user visit http://www.local.test:3000/page-1, `frontend/pages/[...path].js` will be used.

21.3 Category

Update *frontend/utils/wagtail.js* to add two functions.

```
export async function getAllCategories() {
  return await getPaginationContent(`${API_BASE}/api/v1/nextjs/category/`);
}

export async function getAllTags() {
  return await getPaginationContent(`${API_BASE}/api/v1/nextjs/tag/`);
}
```

Here we add `getAllCategories` and `getAllTags`, which will be used in a bit.

Create *frontend/pages/category[...path].js*

This route will handle URL `/category/xxx`

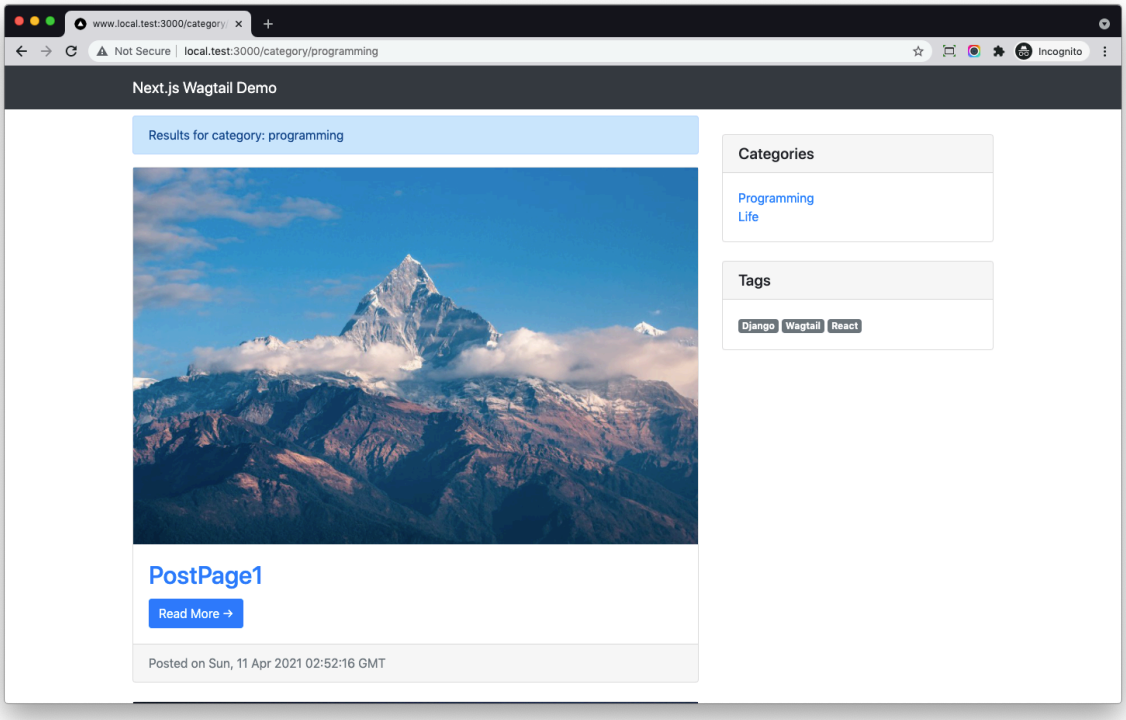
```
import { getPage, getAllCategories, cleanUrlPaths } from "../../utils/wagtail";

export { default } from "../[...path]";

export async function getStaticProps({ params }) {
  let path = params.path || [];
  path = path.join("/");
  // full path
  path = `category/${path}`;

  const data = await getPage(path);
  return { props: data };
}
```

```
export async function getStaticPaths() {
  const categoryList = await getAllCategories();
  let totalUrls = [];

  for (const category of categoryList.items) {
    const data = await getPage(`category/${category.slug}`);

    let pageCount = data.paginator.numPages;
    const pageIndexArray = Array.from({ length: pageCount }, (_, i) => i + 1);

    let paginationPaths = pageIndexArray.map((x) => `${category.slug}/page-${x}`);
    // do not forget to add the default category page
    totalUrls.push(...paginationPaths, `${category.slug}`);
  }

  totalUrls = cleanUrlPaths(totalUrls);
  const totalPaths = totalUrls.map((x) => ({ params: { path: x } }));
  return {
    paths: totalPaths,
    fallback: false,
  };
}
```

Notes:

1. The logic here is very simple, in `getStaticPaths`, we call `getAllCategories` to get all category info.
2. And then generate pagination links, combine urls with `totalUrls.push(...paginationPaths, `${category.slug}`)`
3. The `totalPaths` would seem like `[{"params":{"path":["programming","page-1"]}}, {"params":{"path":["programming"]}},{"params":{"path":["life","page-1"]}}, {"params":{"path":["life"]}}]`

Please test on http://www.local.test:3000/category/programming



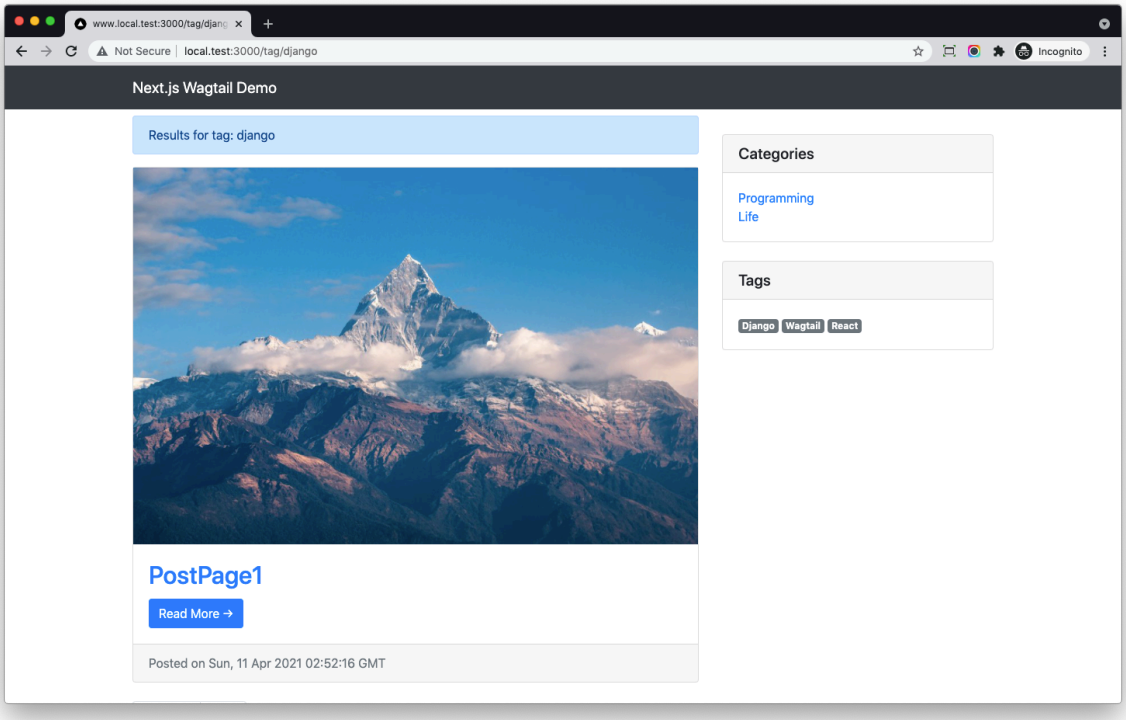
21.4 Tag

Create *frontend/pages/tag/[…path].js*

This route will handle URL `/tag/xxx`

```
import { getPage, getAllTags, cleanUrlPaths } from "../../utils/wagtail";

export { default } from "../[...path]";

export async function getStaticProps({ params }) {
  let path = params.path || [];
  path = path.join("/");
  // full path
  path = `tag/${path}`;

  const data = await getPage(path);
  return { props: data };
}

export async function getStaticPaths() {
  const tagList = await getAllTags();
  let totalUrls = [];

  for (const tag of tagList.items) {
    const data = await getPage(`tag/${tag.slug}`);

    let pageCount = data.paginator.numPages;
    const pageIndexArray = Array.from({ length: pageCount }, (_, i) => i + 1);

    let paginationPaths = pageIndexArray.map((x) => `${tag.slug}/page-${x}`);
    // do not forget to add the default tag page
    totalUrls.push(...paginationPaths, `${tag.slug}`);
  }

  totalUrls = cleanUrlPaths(totalUrls);
  const totalPaths = totalUrls.map((x) => ({ params: { path: x } }));
  return {
    paths: totalPaths,
    fallback: false,
  };
}
```

Please test on http://www.local.test:3000/tag/django



The route structure seem like this:

```
./pages
├── [...path].js
├── _app.js
├── api
│   └── hello.js
├── category
│   └── [...path].js
├── index.js
└── tag
    └── [...path].js
```

21.5 next/link

> Client-side transitions between routes can be enabled via the Link component exported by next/link

Please open devtool, `network` tab on http://www.local.test:3000/

1. Click post page link, check network requests
2. Click category or tag link in the sidebar, check network requests
3. You will see, if you click post page link, no http GET request was sent to `http://www.local.test:3000/postpage1`, because Next.js did client-side navigation.

Update *frontend/components/CategoryWidget.js*

```
import Link from "next/link";

function CategoryWidget(props) {
  const { categoriesList } = props;
```

```
  return (
    <div className="card mb-4">
      <h5 className="card-header">Categories</h5>
      <div className="card-body">
        <div className="row">
          <div className="col-lg-12">
            <ul className="list-unstyled mb-0">
              {categoriesList.map((category) => (
                <li key={category.slug}>
                  <Link href={`${category.url}`}>               // new
                    <a className="text-decoration-none">{category.name}</a>
                  </Link>
                </li>
              ))}
            </ul>
          </div>
        </div>
      </div>
    </div>
  );
}

export { CategoryWidget }
```

Update *frontend/components/TagWidget.js*

```
import React from "react";
import { Badge } from "react-bootstrap";
import Link from "next/link";

function TagWidget(props) {
  const {tagsList} = props;

  return (
    <div className="card mb-4">
      <h5 className="card-header">Tags</h5>
      <div className="card-body">
        {tagsList.map((tag) => (
          <Link href={`${tag.url}`} key={tag.slug}>
            <a>
              <Badge bg="secondary">{tag.name}</Badge>{" "}
            </a>
          </Link>
        ))}
      </div>
    </div>
  );
}

export { TagWidget };
```

Notes:

1. We use `Link` from `next/link` to wrap the `a` element.
2. Now you can test again to understand `next/link` better, and you can check https://nextjs.org/docs/api-reference/next/link to learn more.

Chapter 22

Search Page

22.1 Objectives

By the end of this chapter, you should be able to:

1. Learn how to add search page to Next.js
2. Understand what is CORS and how to do it with `django-cors-headers`
3. What is `useEffect` and how to use it to fetch search results on client side.

22.2 Problem

As we know, in this project, we will use Next.js to do `Static Generation` (SSG).

When building, Next.js will fetch data from backend API and generate static HTML files.

So what is the solution if we want to add `Search` function? Because it seems not a good idea to generate one page for each `search keyword`

22.3 Solution

1. First, display the search page without data. Parts of the page can be pre-rendered using Static Generation.
2. Then, **on the client side**, send search keyword to backend API and display search results.

22.4 CORS

> Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served

For example, if the Django app is serving on `http://api.local.test:8000/`, the Ajax requests from `http://www.local.test:3000` would get `forbidden` error because of Same-origin policy[53]

We can change this behavior by using django-cors-headers[54]

[53] https://en.wikipedia.org/wiki/Same-origin_policy
[54] https://github.com/adamchainz/django-cors-headers

Add it to *requirements.txt*

```
django-cors-headers==3.7.0
```

Update *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    # code omitted for brevity
    "corsheaders",                                  # new
]


MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'corsheaders.middleware.CorsMiddleware',        # new
    # code omitted for brevity
]

CORS_ORIGIN_ALLOW_ALL = True                        # new
```

Notes:

1. We update `INSTALLED_APPS` and `MIDDLEWARE` to make it work in our project.
2. `CORS_ORIGIN_ALLOW_ALL = True` means all origins will be allowed. (We can config to only allow specific origins on production site)

```
$ docker-compose up -d
$ docker-compose logs -f
```

```
$ curl -H "Origin: http://www.local.test:3000/" --verbose \
  http://api.local.test:8000/

# will see response header like this

Access-Control-Allow-Origin: *
```

22.5 SearchPage

Create *frontend/pages/search.js*

```
import { SearchPage } from "../components/SearchPage";

export default SearchPage;
```

Create *frontend/components/SearchPage.js*

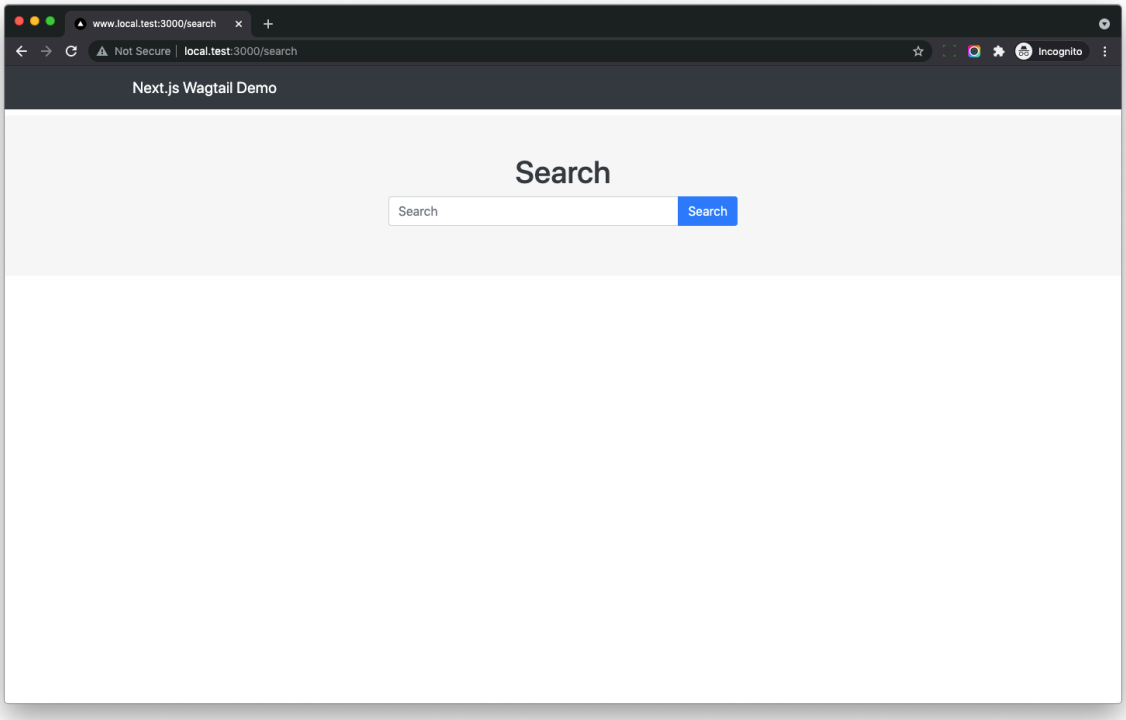
```
import React from "react";
import {
  Container,
  Row,
  InputGroup,
  FormControl,
  Button,
  Form,
} from "react-bootstrap";
import { TopNav } from "./TopNav";
import { Footer } from "./Footer";
```

```
function SearchPage() {
  return (
    <div>
      <TopNav />
      <Container fluid className="min-vh-100">
        <Row className="row bg-light text-dark py-5">
          <div className="col-xl-4 col-md-6 mx-auto">
            <div className="mx-auto">
              <h1 className="text-center">Search</h1>
              <Form className="mb-3">
                <InputGroup>
                  <FormControl
                    placeholder="Search"
                    type="search"
                  />
                <Button variant="primary" type="submit">
                  Search
                </Button>
                </InputGroup>
              </Form>

            </div>
          </div>
        </Row>
      </Container>
      <Footer />
    </div>
  );
}

export { SearchPage };
```

Notes:

1. We add a SearchPage component

```
$ cd frontend
$ yarn dev
```



22.6 useEffect

Next, we will add some code to run on `browser side`

After user visit the page, code will get `search keywords` from the querystring, and then send request to `http://api.local.test:8000//api/v1/nextjs/pages/` to get the search results.

> What does useEffect do? By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates.
>
> useEffect is run after render, so it is not executed by Next.js. We can put the `data fetching` code there.

Update *frontend/components/SearchPage.js*

```
import React, { useEffect, useState } from "react";
import { useRouter } from "next/router";
import {
  Container,
  Row,
  InputGroup,
  FormControl,
  Button,
  Form,
  ListGroup,
} from "react-bootstrap";
import { TopNav } from "./TopNav";
import { Footer } from "./Footer";
import Link from "next/link";
import { getRequest } from "../utils/wagtail";

const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;
```

```
function SearchPage() {
  const router = useRouter();
  const [state, setState] = useState({
    searchQuery: "",
    inputSearch: "",
    loading: true,
    hasMore: false,
    loadResults: [],
    btnSearch: false,
  });
  const { searchQuery, inputSearch, loading, hasMore, loadResults, btnSearch } =
    state;

  let searchQueryString = ''
  if (typeof window !== 'undefined') {
    // only run on browser
    searchQueryString = router.query.query ?? "";
  }

  useEffect(() => {
    // When page load with querystring, we set searchQuery
    if (searchQueryString && !searchQuery && !btnSearch) {
      setState((state) => ({
        ...state,
        searchQuery: searchQueryString,
        inputSearch: searchQueryString,
      }));
    }
  }, [searchQueryString, searchQuery, btnSearch]);

  useEffect(() => {
    // if searchQuery is available, we can query search results
    if (searchQuery && loading) {
      getRequest(`${API_BASE}/api/v1/nextjs/pages/`, {
        search: searchQuery,
        offset: loadResults.length,
        limit: 2,
      }).then((res) => {
        const data = res;
        const newResults = [...loadResults, ...data.items];
        const hasMore = data.meta.totalCount > newResults.length;

        setState((state) => ({
          ...state,
          loading: false,
          loadResults: newResults,
          hasMore: hasMore,
        }));
      });
    }
  }, [searchQuery, loading]);

  return (
    <div>
      <TopNav />
      <Container fluid className="min-vh-100">
        <Row className="row bg-light text-dark py-5">
          <div className="col-xl-4 col-md-6 mx-auto">
            <div className="mx-auto">
              <h1 className="text-center">Search</h1>
              <Form className="mb-3">
                <InputGroup>
                  <FormControl
```

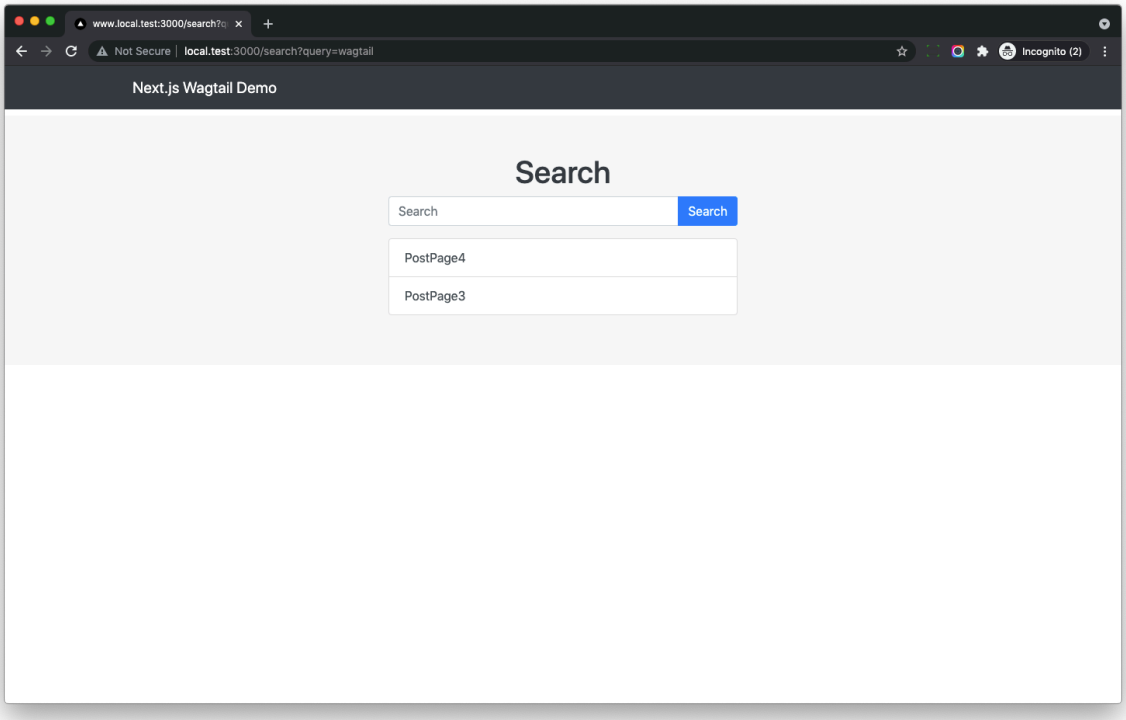
```
                placeholder="Search"
                type="search"
              />
            <Button variant="primary" type="submit">
              Search
            </Button>
            </InputGroup>
          </Form>

          {loadResults.length === 0 && !loading && (
            <div>No Results Found</div>
          )}

          {loadResults.length > 0 && (
            <ListGroup className="mb-3">
              {loadResults.map((searchResult) => (
                <Link
                  href={searchResult.meta.htmlUrl}
                  key={searchResult.id}
                >
                  <ListGroup.Item action>
                    {searchResult.title}
                  </ListGroup.Item>
                </Link>
              ))}
            </ListGroup>
          )}

        </div>
      </div>
    </Row>
  </Container>
  <Footer />
 </div>
 );
}

export { SearchPage };
```

Notes:

1. We use `useState` to add an object to the state to the component.
2. We use `typeof window !== 'undefined'` to run code on the browser side to get the querystring from `router`
3. Here we have **TWO** useEffect, which do different work.
4. If `searchQueryString` is available while `searchQuery` is empty, we set the value to `searchQuery` of state.
5. If `searchQuery` is available while `loading` is `true`, it will send request to the `/api/v1/nextjs/pages/` to get search results. Set state after receiving the data.
6. Here we pass array to the `useEffect` for better performance, you can check Optimizing Performance by Skipping Effects[55] to learn more.
7. To update state, we pass a function to the `setState`, please check Functional updates[56]
8. Please note to debug code in `useEffect`, we should do it in the web browser devtool.

Now if we visit http://www.local.test:3000/search?query=wagtail

[55] https://reactjs.org/docs/hooks-effect.html#tip-optimizing-performance-by-skipping-effects
[56] https://reactjs.org/docs/hooks-reference.html#functional-updates



1. Next.js will render search page which has empty `searchQueryString`
2. When browser load the page, `searchQueryString` is set from the querystring.
3. The code in `useEffect` will run to send request to backend API to get the search results.

22.7 Event Handler

Let's add some button and click handler

```
import React, { useEffect, useState } from "react";
import { useRouter } from "next/router";
import {
  Container,
  Row,
  InputGroup,
  FormControl,
  Button,
  Form,
  ListGroup,
} from "react-bootstrap";
import { TopNav } from "./TopNav";
import { Footer } from "./Footer";
import Link from "next/link";
import { getRequest } from "../utils/wagtail";

const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;

function SearchPage() {
  const router = useRouter();
  const [state, setState] = useState({
    searchQuery: "",
    inputSearch: "",
    loading: true,
```

```
  hasMore: false,
  loadResults: [],
  btnSearch: false,
});
const { searchQuery, inputSearch, loading, hasMore, loadResults, btnSearch } =
  state;

let searchQueryString = ''
if (typeof window !== 'undefined') {
  // only run on browser
  searchQueryString = router.query.query ?? "";
}

useEffect(() => {
  // When page load with querystring, we set searchQuery
  if (searchQueryString && !searchQuery && !btnSearch) {
    setState((state) => ({
      ...state,
      searchQuery: searchQueryString,
      inputSearch: searchQueryString,
    }));
  }
}, [searchQueryString, searchQuery, btnSearch]);

useEffect(() => {
  // if searchQuery is available, we can query search results
  if (searchQuery && loading) {
    getRequest(`${API_BASE}/api/v1/nextjs/pages/`, {
      search: searchQuery,
      type: 'blog.PostPage',
      offset: loadResults.length,
      limit: 2,
    }).then((res) => {
      const data = res;
      const newResults = [...loadResults, ...data.items];
      const hasMore = data.meta.totalCount > newResults.length;

      setState((state) => ({
        ...state,
        loading: false,
        loadResults: newResults,
        hasMore: hasMore,
      }));
    });
  }
}, [searchQuery, loading]);

const handleLoadMoreClick = (e) => {
  e.preventDefault();

  setState((state) => ({
    ...state,
    loading: true,
  }));
};

const handleSubmit = (e) => {
  e.preventDefault();

  setState((state) => ({
    ...state,
    btnSearch: true,
    searchQuery: inputSearch,
```

```
    loading: true,
    hasMore: false,
    loadResults: [],
  }));
};

return (
  <div>
    <TopNav />
    <Container fluid className="min-vh-100">
      <Row className="row bg-light text-dark py-5">
        <div className="col-xl-4 col-md-6 mx-auto">
          <div className="mx-auto">
            <h1 className="text-center">Search</h1>
            <Form onSubmit={handleSubmit} className="mb-3">
              <InputGroup>
                <FormControl
                  placeholder="Search"
                  type="search"
                  defaultValue={inputSearch}
                  onChange={(e) =>
                    setState((state) => ({
                      ...state,
                      inputSearch: e.target.value,
                    }))
                  }
                />
              <Button variant="primary" type="submit">
                Search
              </Button>
              </InputGroup>
            </Form>

            {loadResults.length === 0 && !loading && (
              <div>No Results Found</div>
            )}

            {loadResults.length > 0 && (
              <ListGroup className="mb-3">
                {loadResults.map((searchResult) => (
                  <Link
                    href={searchResult.meta.htmlUrl}
                    key={searchResult.id}
                  >
                    <ListGroup.Item action>
                      {searchResult.title}
                    </ListGroup.Item>
                  </Link>
                ))}
              </ListGroup>
            )}

            {hasMore && (
              <Button variant="primary" onClick={handleLoadMoreClick}>
                Load More
              </Button>
            )}

          </div>
        </div>
      </Row>
    </Container>
    <Footer />
```

```
    </div>
  );
}

export { SearchPage };
```

Notes:

1. We add `handleLoadMoreClick` to load more items if user click the `Load More` button.
2. We add `handleSubmit` so user can keep searching other keywords.
3. In the `FormControl`, we add event handler to change the `inputSearch` of state, here I have a question, does it cause component re-render?
4. If you are confused with the above question, please check Optimizing Performance by Skipping Effects[57] again!

22.8 Url

If we check the response of the backend API

```
{
    "meta": {
        "total_count": 4
    },
    "items": [
        {
            "id": 7,
            "meta": {
                "type": "blog.PostPage",
                "detail_url": "http://localhost/api/v1/nextjs/pages/7/",
                "html_url": "http://localhost/postpage4/",
                "slug": "postpage4",
                "first_published_at": "2021-08-11T02:50:53.016000Z"
            },
            "title": "PostPage4"
        },
        {
            "id": 6,
            "meta": {
                "type": "blog.PostPage",
                "detail_url": "http://localhost/api/v1/nextjs/pages/6/",
                "html_url": "http://localhost/postpage3/",
                "slug": "postpage3",
                "first_published_at": "2021-08-11T02:50:39.671000Z"
            },
            "title": "PostPage3"
        }
    ]
}
```

As you can see, the `html_url` is `"http://localhost/postpage4/"`, the domain and port is generated from Wagtail site config.

So let's config it in Wagtail admin, after we fix it, the response would have correct URL:

```
{
    "meta": {
        "total_count": 4
```

[57] https://reactjs.org/docs/hooks-effect.html#tip-optimizing-performance-by-skipping-effects

```
    },
    "items": [
        {
            "id": 7,
            "meta": {
                "type": "blog.PostPage",
                "detail_url": "http://api.local.test:8000/api/v1/nextjs/pages/7/",
                "html_url": "http://api.local.test:8000/postpage4/",
                "slug": "postpage4",
                "first_published_at": "2021-08-11T02:50:53.016000Z"
            },
            "title": "PostPage4"
        },
        {
            "id": 6,
            "meta": {
                "type": "blog.PostPage",
                "detail_url": "http://api.local.test:8000/api/v1/nextjs/pages/6/",
                "html_url": "http://api.local.test:8000/postpage3/",
                "slug": "postpage3",
                "first_published_at": "2021-08-11T02:50:39.671000Z"
            },
            "title": "PostPage3"
        }
    ]
}
```


Next, update *frontend/.env.development*

```
NEXT_PUBLIC_WAGTAIL_API_BASE=http://api.local.test:8000
NEXT_PUBLIC_NEXT_BASE=http://www.local.test:3000
```

Update *frontend/utils/wagtail.js*

```
const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;
const NEXT_BASE = process.env.NEXT_PUBLIC_NEXT_BASE;


export function convertWagtailUrlToNext(url){
  return url.replace(API_BASE, NEXT_BASE)
}
```

Update *frontend/components/SearchPage.js*

```
{loadResults.length > 0 && (
  <ListGroup className="mb-3">
    {loadResults.map((searchResult) => (
      <Link
        href={convertWagtailUrlToNext(searchResult.meta.htmlUrl)}           // new
        key={searchResult.id}
      >
        <ListGroup.Item action>
          {searchResult.title}
        </ListGroup.Item>
      </Link>
    ))}
  </ListGroup>
)}
```

Now, if we click the search results, we will get redirected to the target page smoothly.

22.9 SearchForm

Create *frontend/components/SearchForm.js*

```
import React, { useState } from "react";
import { useRouter } from "next/router";
import { Button, Form, FormControl, InputGroup } from "react-bootstrap";

function SearchForm() {
  const router = useRouter();

  const [searchQuery, setSearchQuery] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();

    router.push({
      pathname: "/search",
      query: { query: searchQuery },
    });
  };

  return (
    <div className="card mb-4">
      <h5 className="card-header">Search</h5>
      <div className="card-body">
        <Form onSubmit={handleSubmit}>
          <InputGroup>
            <FormControl
              placeholder="Search"
              type="search"
              onChange={(e) => setSearchQuery(e.target.value)}
            />
          <Button variant="dark" type="submit">
            Go
          </Button>
          </InputGroup>
        </Form>
      </div>
    </div>
  );
}

export { SearchForm };
```

Notes:

1. In `handleSubmit`, we use `router.push` jump to search page on the client side.

Next, update *frontend/components/SideBar.js*

```
import { Col } from "react-bootstrap";
import { TagWidget } from "./TagWidget";
import { CategoryWidget } from "./CategoryWidget";
import { SearchForm } from "./SearchForm";

function SideBar(props) {
  return (
    <Col md={4}>
      <SearchForm {...props} />
      <CategoryWidget {...props} />
      <TagWidget {...props} />
    </Col>
```

```
  );
}

export { SideBar };
```

Now, we can search on the sidebar.

Chapter 23

SEO

23.1 Objectives

By the end of this chapter, you should be able to:

1. Add SEO related fields to the serializers
2. Use `next/head` to append elements to the head of the page

23.2 Seo data

Update *blog/models.py* to add relevant `property`

```
class BasePage(Page):

    @property
    def seo_json_title(self):
        return self.seo_title or self.title

    @property
    def seo_json_description(self):
        return self.search_description
```

Update *blog/serializers.py* to add the above fields to `BasePageSerializer`

```
class BasePageSerializer(serializers.ModelSerializer):
    serializer_field_mapping = (
        serializers.ModelSerializer.serializer_field_mapping.copy()
    )
    serializer_field_mapping.update(
        {fields.StreamField: wagtail_serializers.StreamField}
    )

    class Meta:
        model = BasePage
        fields = (
            "id",
            "slug",
            "title",
            "url",
            "last_published_at",
            "seo_json_title",                       # new
```

```
            "seo_json_description",                # new
        )
```

23.3 next/head

Create *frontend/components/HeadMeta.js*

```
import React from "react";
import Head from "next/head";

function HeadMeta(props) {
  const { seoJsonTitle, seoJsonDescription } = props.pageContent;
  return (
    <Head>
      <title>{seoJsonTitle}</title>
      <link rel="icon" href="/favicon.ico" />
      {!!seoJsonDescription && (
        <meta name="description" content={seoJsonDescription} />
      )}
    </Head>
  );
}

export { HeadMeta };
```

Notes:

1. This component is to generate meta info to the head of the page
2. You can check https://nextjs.org/docs/api-reference/next/head to learn more.

Update *frontend/components/PageProxy.js*

```
import React from "react";
import LazyPages from './LazyPages';
import { HeadMeta } from "./HeadMeta";

function PageProxy(props) {
  const {pageType} = props;
  const PageComponent = LazyPages[pageType];

  if (PageComponent) {
    return (
      <React.Fragment>
        <HeadMeta {...props} />                    // new
        <PageComponent {...props} />
      </React.Fragment>
    );
  } else {
    return <div>Error when loading {pageType}</div>;
  }
}

export { PageProxy };
```

Now if we check the HTML source code, we can see `title` and `description`.

Notes:

1. We can use `cached_property` to replace `property` for better performance. You can check Django doc[58] to learn more.
2. To add more meta fields to your page models, you can check https://github.com/coderedcorp/wagtail-seo/blob/main/wagtailseo/models.py or https://github.com/neon-jungle/wagtail-metadata/blob/master/wagtailmetadata/models.py

[58] https://docs.djangoproject.com/en/3.2/ref/utils/#module-django.utils.functional

Chapter 24

Add Preview Support

24.1 Objective

1. Make Wagtail preview work with the Next.js.

24.2 Wagtail WorkFlow

Before we start, let's think about how to solve this problem.

1. When editor click link `preview draft`, Wagtail admin create a `token` and save the `token` and `draft page content` to db.
2. The Wagtail admin redirect editor to the Next.js app, the url contains the `token` in querystring.
3. The Next.js app use the `token` to send request to get the `draft page content`, and display it.

So this is the basic workflow of `wagtail-headless-preview`.

24.3 Next.js Preview Mode

As we know, we use `Static Generation` (SSG) in this project, if some page is not published, the static HTML would not even exist, so how do we solve this problem?

Fortunately, Next.js has a feature called `Preview Mode` can help us in this case, I will talk about it in a bit.

24.4 Wagtail headless preview

Update *requirements.txt*

```
# other packages
wagtail-headless-preview==0.1.4
```

Update *.env/.dev-sample*

```
DEBUG=1
SECRET_KEY='randome_key'
DJANGO_ALLOWED_HOSTS=*

SQL_ENGINE=django.db.backends.postgresql
```

```
SQL_DATABASE=wagtail_dev
SQL_USER=wagtail_dev
SQL_PASSWORD=wagtail_dev
SQL_HOST=db
SQL_PORT=5432

NEXT_PUBLIC_NEXT_BASE=http://www.local.test:3000                # new
```

Update *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    ...
    "wagtail_headless_preview",
]


NEXT_PUBLIC_NEXT_BASE = os.environ.get("NEXT_PUBLIC_NEXT_BASE")
HEADLESS_PREVIEW_CLIENT_URLS = {
    "default": f'{NEXT_PUBLIC_NEXT_BASE}/api/preview/',
}
```

Notes:

1. We add `wagtail_headless_preview` to the `INSTALLED_APPS`
2. And we also set `HEADLESS_PREVIEW_CLIENT_URLS` so `wagtail_headless_preview` will use it to generate redirect URL after creating token.

Next, let's update *blog/models.py*

```
import urllib.parse
from wagtail_headless_preview.models import HeadlessPreviewMixin


class BasePage(HeadlessPreviewMixin, Page):

    class Meta:
        abstract = True
```

24.5 Rest API

Update *blog/api.py*

```
from django.urls import path
from django.contrib.contenttypes.models import ContentType
from rest_framework import serializers
from wagtail.api.v2.router import WagtailAPIRouter
from wagtail.api.v2.views import BaseAPIViewSet, PagesAPIViewSet
from wagtail.core.models import Page
from blog.serializers import CategorySerializer, TagSerializer
from blog.models import BlogCategory, Tag
from wagtail_headless_preview.models import PagePreview


class PagePreviewAPIViewSet(PagesAPIViewSet):
    known_query_parameters = PagesAPIViewSet.known_query_parameters.union(
        ["content_type", "token"]
    )

    def listing_view(self, request):
```

```
        page = self.get_object()
        return page.serve(request)

    def detail_view(self, request, pk):
        page = self.get_object()
        return page.serve(request)

    def get_object(self):
        app_label, model = self.request.GET["content_type"].split(".")
        content_type = ContentType.objects.get(app_label=app_label, model=model)

        page_preview = PagePreview.objects.get(
            content_type=content_type, token=self.request.GET["token"]
        )
        page = page_preview.as_page()
        if not page.pk:
            # fake primary key to stop API URL routing from complaining
            page.pk = 0

        return page


api_router.register_endpoint("page_preview", PagePreviewAPIViewSet)
```

Notes:

1. The key point is `PagePreviewAPIViewSet.get_object`, which would get `content_type` and `token` from the querystring, and find the `PagePreview` instance, which contains `json` representation of the draft content.

After all those are done, let's migrate db

```
$ docker-compose up --build -d
$ docker-compose logs -f
```

24.6 Next.js Preview

Update *frontend/utils/wagtail.js*

```
export async function getPagePreview(contentType, token, params, options) {
  params = params || {};
  params = {
    contentType,
    token,
    ...params,
  };

  return await getRequest(
    `${API_BASE}/api/v1/nextjs/page_preview/`,
    params,
    options
  );
}
```

We will use this function in a bit.

Create *frontend/pages/api/preview.js*

```
export default async function preview(req, res) {
  const { content_type: contentType, token } = req.query;
```

```
  if (!contentType || !token) {
    return res
      .status(401)
      .json({ message: "Missing contentType and/or token" });
  }

  res.setPreviewData({ contentType, token });
  res.redirect("/_preview");
  res.end();
}
```

Notes:

1. We put `preview.js` in `pages/api` to create API route.
2. If `contentType` and `token` both available, call `res.setPreviewData` to write `contentType` and `token` to cookie.
3. `res.setPreviewData` sets some cookies on the browser which turns on the preview mode. Any requests to Next.js containing these cookies will be considered as the preview mode, and **the behavior for statically generated pages will change**
4. And then we redirect to `_preview` page.
5. You can check Preview Mode[59] to learn more.

Create *frontend/pages/_preview.js*

```
import { getPagePreview } from "../utils/wagtail";
export { default } from "./[...path]";

export async function getStaticProps({ params, preview, previewData }) {
  if (!preview) {
    return { props: {} };
  }

  const { contentType, token } = previewData;

  const pagePreviewData = await getPagePreview(
    contentType,
    token,
    {}
  );

  return {
    props: pagePreviewData,
  };
}
```

Notes:

1. If you request a page which has `getStaticProps` with the preview mode cookies set (via `res.setPreviewData`), then `getStaticProps` will be called at request time (instead of at build time).
2. In `getStaticProps`, we send request using `getPagePreview` to get the page draft data, and then return it to render the page UI.

[59] https://nextjs.org/docs/advanced-features/preview-mode

24.7 Steps

1. When editor click link `preview draft`, Wagtail admin create a `token` and save the `token` and `draft page content` to db.
2. In `http://api.local.test:8000/cms-admin/pages/4/edit/preview/`, the `wagtail_headless_preview` use `iframe` to visit `http://www.local.test:3000/api/preview/?content_type=blog.postpage&token=id%3D4%3A1m8xLI%3AmVhj9gO8YwEjYXEb81vR7OjCXjnrCf26k5YacDqxwTk`, the url contains the `token` and `content_type`
3. The Next.js API router turn on `Preview Mode`, sets `token` and `content_type` to Cookie.
4. The `_preview.js` uses `getPagePreview` to request draft content of the page, and then use `ProxyPage` to render the page UI.

24.8 Live Redirect

Now the `page preview` is working with Wagtail and Next.js

If we try to check Live version of the page, we will see JSON representation. Is it possible to let us see the live page on Next.js?

Update *blog/models.py*

```
import urllib.parse
from django.http.response import JsonResponse, HttpResponseRedirect
from django.conf import settings


class BasePage(HeadlessPreviewMixin, Page):

    def serve(self, request, *args, **kwargs):
        if request.content_type == 'application/json':
            context = self.get_context(request, *args, **kwargs)
            return JsonResponse(context['page_component'])
        else:
            full_path = request.get_full_path()
            return HttpResponseRedirect(urllib.parse.urljoin(settings.NEXT_PUBLIC_NEXT_BASE, full_
↪path))
```

Notes:

1. As we know, in the `getRequest` function (`frontend/utils/wagtail.js`), we set `"Content-Type": "application/json"`
2. In backend API, If the http request come from `getRequest`, we return the JSON representation.
3. If the http request come from normal browser, we redirect to Next.js page.

Now if we click the `Live` button in Wagtail admin, we will be redirected to the frontend app.

You can also use command below to test

```
# return 302 redirect response
$ curl --verbose http://api.local.test:8000/

# return JSON
$ curl --header "Content-Type: application/json" --verbose http://api.local.test:8000/
```

Chapter 25

Import Comment Model

25.1 Objective

By the end of this chapter, you should be able to:

1. Import `django-contrib-comments` to add comment feature to the PostPage
2. Understand what is `Generic relation` and the benefit

25.2 Setup

Add `django-contrib-comments` to the *requirements.txt*

```
django-contrib-comments==2.1.0
```

Update *wagtail_bootstrap_blog/settings/base.py*

```
INSTALLED_APPS = [
    ...
    'django_comments',
    'django.contrib.sites',
]

SITE_ID = 1
```

Notes:

1. Add `django_comments` to the `INSTALLED_APPS`
2. Add `django.contrib.sites` to the `INSTALLED_APPS`, which is required by `django_comments`
3. Add `SITE_ID = 1` at the bottom, which is required by `django.contrib.sites`

Let's rebuild our docker image

```
$ docker-compose up -d --build
$ docker-compose logs -f
```

25.3 Generic relation

Let's take a look at core model of the `django_comments`, you can also check on Github[60]

[60] https://github.com/django/django-contrib-comments/blob/2.0.0/django_comments/abstracts.py#L15

```
class BaseCommentAbstractModel(models.Model):
    # Content-object field
    content_type = models.ForeignKey(ContentType,
                                     verbose_name=_('content type'),
                                     related_name="content_type_set_for_%(class)s",
                                     on_delete=models.CASCADE)
    object_pk = models.TextField(_('object ID'))
    content_object = GenericForeignKey(ct_field="content_type", fk_field="object_pk")
```

Notes:

1. The Django `ForeignKey` can only build relationship between two models.
2. With `GenericForeignKey`, we can use it to build relationship with other models. (not limited to one model)
3. Django `contenttype` is very powerful and you can check Django doc: Generic relations[61] to learn more.

25.4 custom_comments app

Here, let's create a new Django app for comment customization

```
$ docker-compose run --rm web bash

# create a Django app
(container)$ ./manage.py startapp custom_comments
(container)$ exit
```

Update *wagtail_bootstrap_blog/settings/base.py*

```
INSTALLED_APPS = [
    ...
    'django_comments',
    'django.contrib.sites',
    'custom_comments',
]
```

Notes:

1. We added `custom_comments` to the `INSTALLED_APPS`

25.5 serializers.py

Create *custom_comments/serializers.py*

```
from django.apps import apps
from django_comments.models import Comment
from django.contrib.contenttypes.models import ContentType
from django.utils import timezone
from django.conf import settings
from rest_framework import serializers


class CommentSerializer(serializers.ModelSerializer):
    content_type = serializers.CharField()
```

[61] https://docs.djangoproject.com/en/3.1/ref/contrib/contenttypes/#generic-relations

```
    class Meta:
        model = Comment
        fields = (
            "pk",
            "content_type",
            "object_pk",

            "user_name",
            "user_email",
            "comment",
            "submit_date",
        )

    def create(self, validated_data):
        # django_comments/views/comments.py
        # django_comments/forms.py
        app_label, model = validated_data.get("content_type").split(".")
        content_type = ContentType.objects.get(app_label=app_label, model=model)
        object_pk = validated_data.get("object_pk")

        try:
            model = apps.get_model(app_label, model)
            model.objects.get(pk=object_pk)
        except Exception:
            raise Exception('can not find comment target object')

        comment_data = dict(
            content_type=content_type,
            object_pk=object_pk,
            user_name=validated_data["user_name"],
            user_email=validated_data["user_email"],
            comment=validated_data["comment"],
            submit_date=timezone.now(),
            site_id=getattr(settings, 'SITE_ID'),
            is_public=True,
            is_removed=False,
        )

        return super().create(comment_data)
```

Notes:

1. Here we add a `CommentSerializer` for `Comment` model.
2. In the `create` method, we can control how the comment instance is created, you can check https://www.django-rest-framework.org/api-guide/serializers/#saving-instances to learn more.

25.6 ViewSet

Update *custom_comments/views.py*

```
from django_comments.models import Comment
from rest_framework import viewsets
from .serializers import CommentSerializer


class CommentViewSet(viewsets.ModelViewSet):
    serializer_class = CommentSerializer
    queryset = Comment.objects.all()
    http_method_names = ["get", "post"]
```

Update *custom_comments/api.py*

```
from rest_framework.routers import SimpleRouter
from .views import CommentViewSet

# Create a router and register our viewsets with it.
comment_router = SimpleRouter()
comment_router.register(r'comments', CommentViewSet)
```

Create *custom_comments/urls.py*

```
from django.urls import path, include
from .api import comment_router

urlpatterns = [
    path('api/v1/', include(comment_router.urls)),
]
```

Update *nextjs_wagtail_app/urls.py*

```
urlpatterns = [
    path('admin/', admin.site.urls),

    path('cms-admin/', include(wagtailadmin_urls)),
    path('documents/', include(wagtaildocs_urls)),

    path('api/v1/nextjs/', api_router.urls),
    path('', include('custom_comments.urls')),                # new

    # For anything not caught by a more specific rule above, hand over to
    # Wagtail's serving mechanism
    path('', include(wagtail_urls)),
]
```

25.7 Manual test

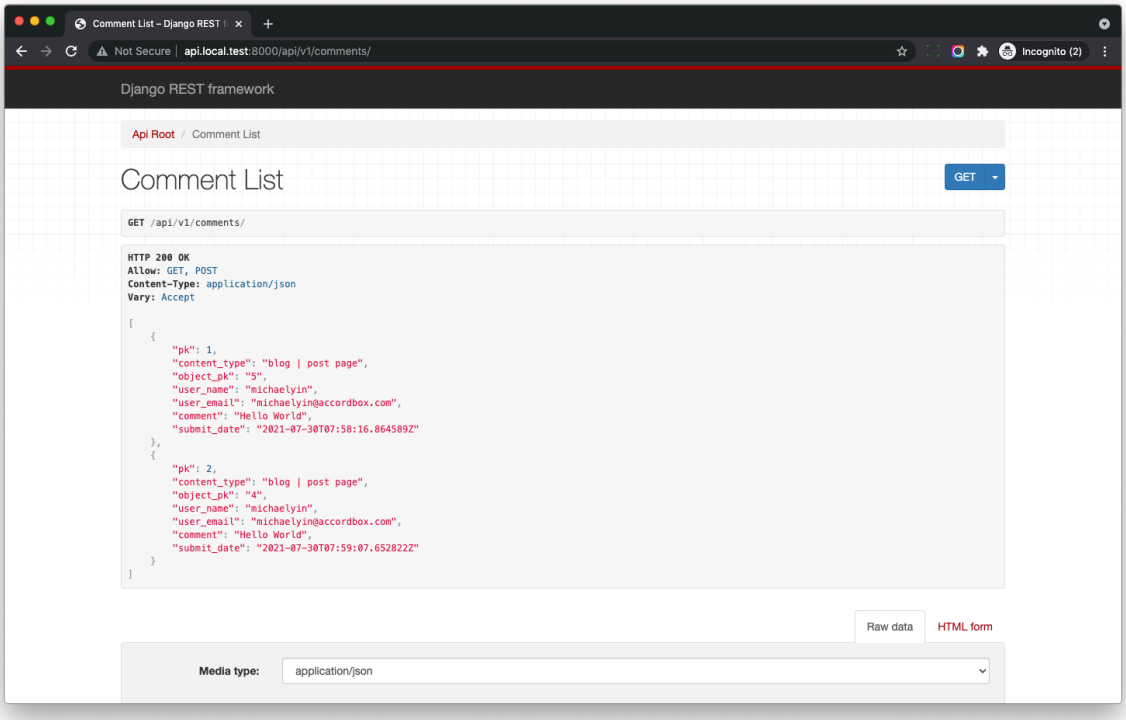
```
$ docker-compose up -d
$ docker-compose logs -f
```

Visit http://api.local.test:8000/api/v1/comments/

Here we can use the HTML form to submit comment data

Let's use `Raw Data / application/json`

```
{
  "content_type": "blog.postpage",
  "object_pk": "4",
  "user_name": "michaelyin",
  "user_email": "michaelyin@accordbox.com",
  "comment": "Hello World"
}
```



Next, let's use Django shell to inspect the data.

```
$ docker-compose run --rm web python manage.py shell
```

```
>>> from django_comments.models import Comment
>>> Comment.objects.all()
<QuerySet [<Comment: michaelyin: Hello World...>, <Comment: michaelyin: Hello World...>]>

>>> Comment.objects.first().content_object
<PostPage: PostPage2>

>>> exit()
```

Notes:

1. Here we can get the page instance with `content_object` attribute of comment instance.
2. The `Generic relations` enable us to make comment model work with **ANY** Django models, which is very flexible and powerful.

Chapter 26

Manage comment in Wagtail

26.1 Objective

By the end of this chapter, you should be able to:

1. Use `ModelAdmin` to manage comment in Wagtail admin.
2. Learn how to customize `ModelAdmin`

26.2 ModelAdmin

Update *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    ...
    'wagtail.contrib.modeladmin',            # new
]
```

We add `wagtail.contrib.modeladmin` to the `INSTALLED_APPS`

Create *custom_comments/wagtail_hooks.py*

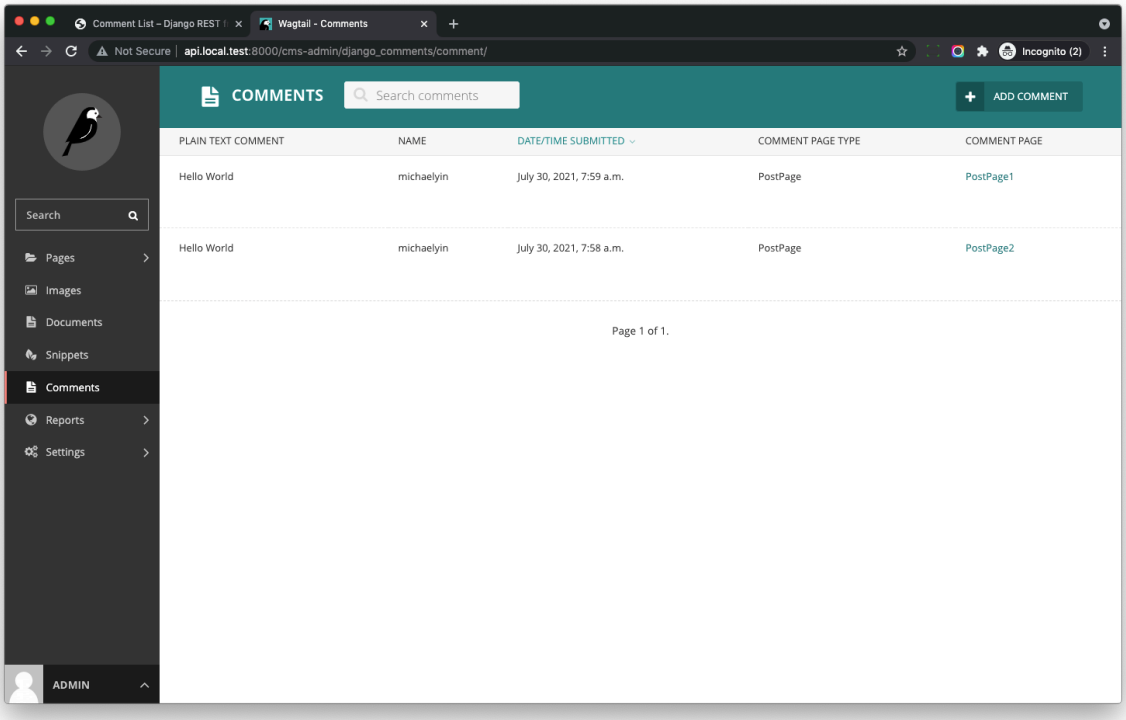
```
from django_comments.models import Comment
from django.contrib.contenttypes.models import ContentType
from django.utils.html import format_html, strip_tags
from wagtail.contrib.modeladmin.options import ModelAdmin, modeladmin_register


class CommentAdmin(ModelAdmin):
    model = Comment
    menu_label = "Comments"
    menu_icon = "doc-full-inverse"
    add_to_settings_menu = False

    list_display = (
        "plain_text_comment",
        "name",
        "submit_date",
        "comment_page_type",
        "comment_page",
    )
    ordering = ("-submit_date",)
    search_fields = ("comment",)
    list_per_page = 10
```

```
    form_fields_exclude = [
        "submit_date",
        "ip_address",
        "is_removed",
        "is_public",
        "user_url",
        "user",
        "content_type",
        "object_pk",
        "content_object",
        "site",
    ]

    def comment_page_type(self, obj):
        ct = ContentType.objects.get(pk=obj.content_object.content_type_id)
        return ct.model_class().__name__

    def comment_page(self, obj):
        return format_html(
            f"<a href='{obj.content_object.specific.url}'>{obj.content_object.specific.title}</a>"
        )

    def plain_text_comment(self, obj):
        return strip_tags(obj.comment)


modeladmin_register(CommentAdmin)
```

Notes:

1. When loading, Wagtail will search for any Django app with the file `wagtail_hooks.py` and execute the code.
2. We create `CommentAdmin` that inherit from `ModelAdmin`, which tell Wagtail admin how to manage the comment.
3. The syntax is very similar with Django ModelAdmin[62]
4. `comment_page_type`, `comment_page` and `plain_text_comment` are methods that help to generate extra fields in the listing view.

[62] https://docs.djangoproject.com/en/3.1/ref/contrib/admin/#modeladmin-objects



26.3 Custom View

In some cases, we do not want to delete the comment record but want to set some flag to hide it.

`django_comments.models.Comment` also has bool field `is_removed` for us to use.

Create *custom_comments/views.py*

```
from django.shortcuts import redirect
from wagtail.admin import messages
from wagtail.contrib.modeladmin.views import DeleteView


class CommentDeleteView(DeleteView):

    page_title = "Delete comment"

    def post(self, request, *args, **kwargs):
        try:
            self.instance.is_removed = True
            self.instance.save()
            msg = f"{self.verbose_name} '{self.instance}' deleted."
            messages.success(request, msg)
            return redirect(self.index_url)
        except Exception as e:
            raise e
```

Notes:

1. We created a custom `DeleteView`, in the `post` method, we set the `is_removed` field to `True` (which means the comment is "deleted")

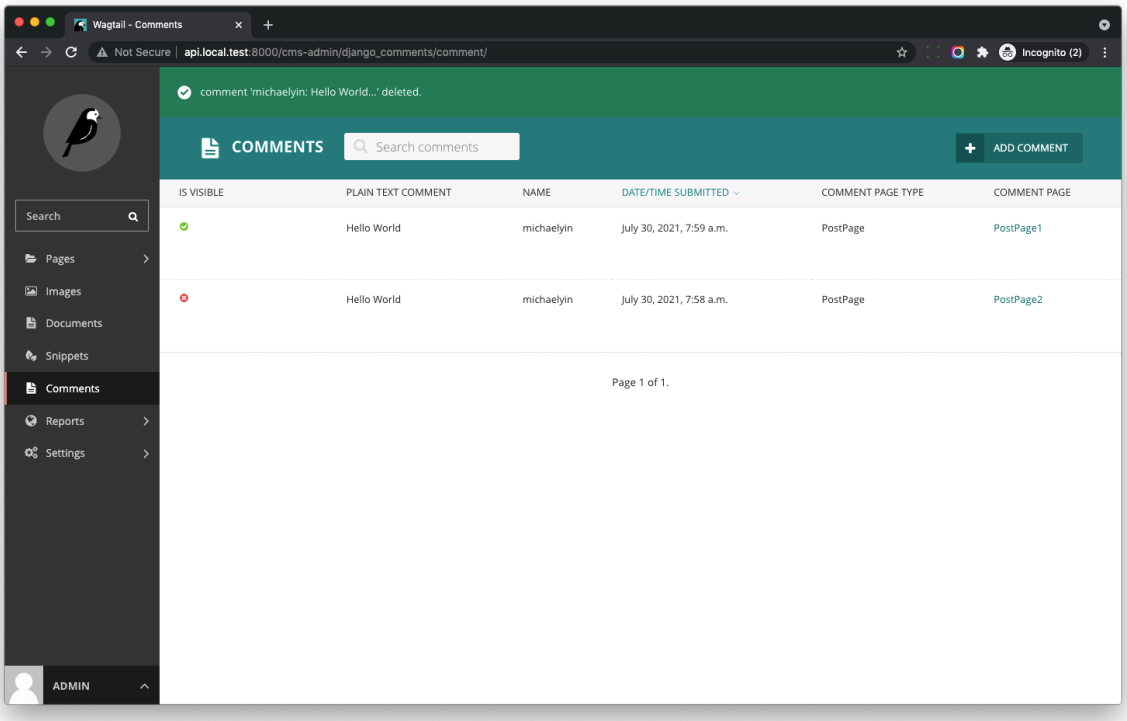
Update *custom_comments/wagtail_hooks.py*

```
from django_comments.models import Comment
from django.contrib.contenttypes.models import ContentType
from django.utils.html import format_html, strip_tags
from wagtail.contrib.modeladmin.options import ModelAdmin, modeladmin_register

from .views import CommentDeleteView


class CommentAdmin(ModelAdmin):
    model = Comment
    menu_label = "Comments"
    menu_icon = "doc-full-inverse"
    add_to_settings_menu = False
    delete_view_class = CommentDeleteView                # new

    list_display = (
        "is_visible",
        "plain_text_comment",
        "name",
        "submit_date",
        "comment_page_type",
        "comment_page",
    )
    ordering = ("-submit_date",)
    search_fields = ("comment",)
    list_per_page = 10
    form_fields_exclude = [
        "submit_date",
        "ip_address",
        "is_removed",
        "is_public",
        "user_url",
        "user",
        "content_type",
        "object_pk",
        "content_object",
        "site",
    ]

    def comment_page_type(self, obj):
        ct = ContentType.objects.get(pk=obj.content_object.content_type_id)
        return ct.model_class().__name__

    def comment_page(self, obj):
        return format_html(
            f"<a href='{obj.content_object.specific.url}'>{obj.content_object.specific.title}</a>"
        )

    def plain_text_comment(self, obj):
        return strip_tags(obj.comment)

    def is_visible(self, obj):
        from django.contrib.admin.templatetags.admin_list import _boolean_icon
        return _boolean_icon(not obj.is_removed)


modeladmin_register(CommentAdmin)
```

Notes:

1. We tell `modeladmin` to use custom `delete_view` with `delete_view_class = CommentDeleteView`
2. In Wagtail admin, if user delete the comment, the `is_removed` field is set to `False`.

3. We added a new `is_visible` method, to tell us if the comment is visible to users. It is more straightforward compared with `is_removed` field



26.4 Reference:

1. `django-contrib-comments` supports `moderation`, and you can check Generic comment moderation[63] to learn more.
2. If you want to know more about custom view, please check Wagtail doc: Customising CreateView, EditView and DeleteView[64]

[63] https://django-contrib-comments.readthedocs.io/en/latest/moderation.html
[64] https://docs.wagtail.io/en/stable/reference/contrib/modeladmin/create_edit_delete_views.html

Chapter 27

CommentForm

27.1 Objective

By the end of this chapter, you should be able to:

1. Build `CommentForm` component to let user submit comment.

27.2 content_type_str

Update *blog/serializers.py*

```
class PostPageSerializer(serializers.ModelSerializer):
    tags = TagField()
    categories = CategoryField()
    body = StreamField()
    header_image = ImageRenditionField("max-1000x800")
    content_type_str = serializers.SerializerMethodField()        # new

    class Meta:
        model = PostPage
        fields = BasePageSerializer.Meta.fields + (
            "tags",
            "categories",
            "body",
            "header_image",
            "content_type_str",                                   # new
        )

    def get_content_type_str(self, obj):                          # new
        return obj._meta.label_lower
```

Notes:

1. `SerializerMethodField` will create a read-only field by calling the relevant method.
2. Here we add `content_type_str`, which will be used by the `CommentForm` component.

```
$ docker-compose up -d
$ docker-compose logs -f
```

```
# check content_type_str
$ curl --header "Content-Type: application/json" http://api.local.test:8000/postpage1/
```

```
"content_type_str": "blog.postpage"
```

27.3 postRequest

Update *frontend/utils/wagtail.js*

```
export async function postRequest(url, params, options) {
  params = params || {};
  params = snakecaseKeys(params, {deep: true});

  let headers = options?.headers || {};
  headers = {
    "Content-Type": "application/json",
    ...headers,
  };
  const res = await fetch(
    url, {
      method: "POST",
      headers,
      body: JSON.stringify(params),
    }
  );

  const data = await res.json();
  return camelcaseKeys(data, {deep: true})
}
```

Notes:

1. Here we add `postRequest` to send POST request to the backend API.

27.4 CommentForm

Create *frontend/components/CommentForm.js*

```
import React, { useState, useEffect } from "react";
import { Form, Button } from "react-bootstrap";
import { postRequest } from "../utils/wagtail";

const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;

function CommentForm(props) {
  const {pageContent} = props;
  const {id: objectPk, contentTypeStr: contentType} = pageContent;

  const commentInput = React.useRef();
  const [displayCommentForm, setDisplayCommentForm] = useState(false);
  const [isPosting, setIsPosting] = useState(false);

  const handleFormSubmit = (e) => {
    e.preventDefault();
    setIsPosting(true);
    let formData = new FormData(e.target),
      formDataObj = Object.fromEntries(formData.entries());
    formDataObj.contentType = contentType;
    formDataObj.objectPk = objectPk;
```

```
    postRequest(`${API_BASE}/api/v1/comments/`, formDataObj)
      .then((data) => {
        e.target.reset();
        setIsPosting(false);
      })
  };

  if (displayCommentForm) {
    return (
      <div className="mb-4">
        <Form onSubmit={handleFormSubmit}>
          <Form.Group className="mb-3" controlId="formBasicUsername">
            <Form.Label>Username</Form.Label>
            <Form.Control type="text" name="user_name"/>
          </Form.Group>
          <Form.Group className="mb-3" controlId="formBasicEmail">
            <Form.Label>Email address</Form.Label>
            <Form.Control type="email" name="user_email"/>
          </Form.Group>
          <Form.Group className="mb-3" controlId="formBasicComment">
            <Form.Label>Comment</Form.Label>
            <Form.Control
              as="textarea"
              rows={6}
              ref={commentInput}
              name="comment"
            />
          </Form.Group>
          <Button
            variant="primary"
            type="submit"
            className={`btn btn-primary ${isPosting ? "disabled" : ""}`}
          >
            {isPosting ? "Submitting" : "Submit"}
          </Button>
        </Form>
      </div>
    );
  } else {
    return (
      <div className="mb-4">
        <button
          className="btn btn-primary btn-block"
          onClick={() => setDisplayCommentForm(true)}
        >
          Write Comment
        </button>
      </div>
    );
  }
}

export { CommentForm };
```

Notes:

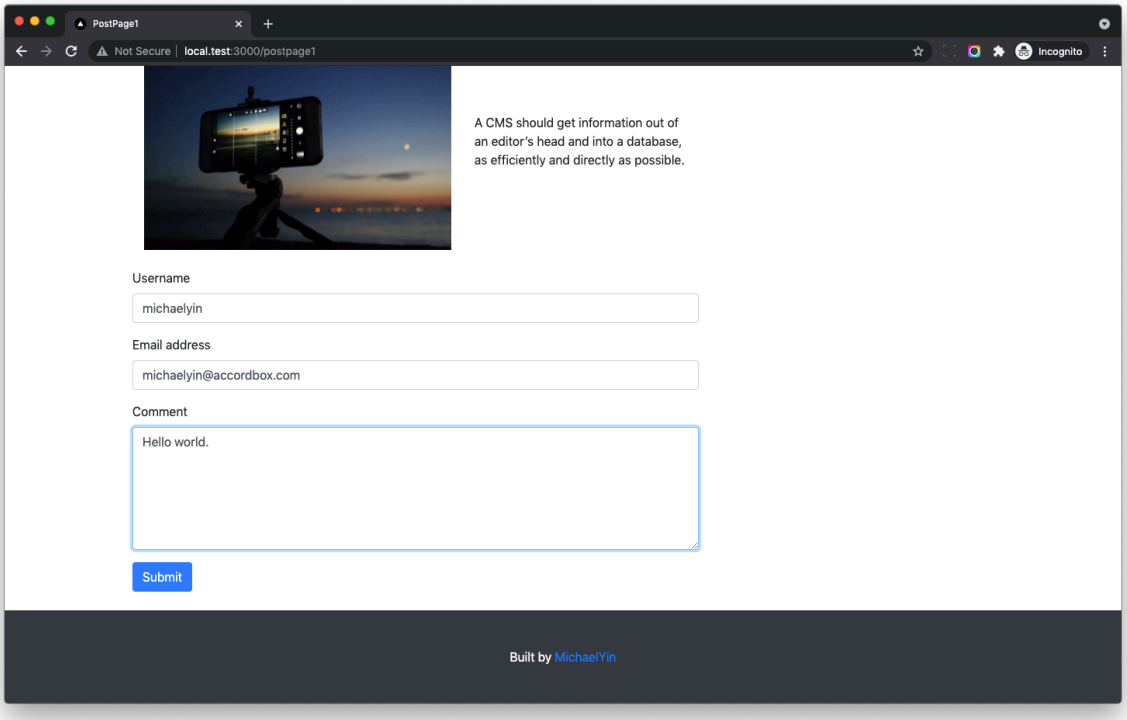
1. We use `displayCommentForm` state to control if display the form or not.
2. In `handleFormSubmit`, we use `postRequest` to send POST request.
3. Please do not forget to add `contentType` and `objectPk` to the `formDataObj`.

Update *frontend/components/PostDetail.js*

```
import React from "react";
import { StreamField } from "./StreamField/StreamField";
import { BaseImage } from "./BaseImage";
import { CommentForm } from "./CommentForm";

function PostDetail(props) {
  const { pageContent } = props;

  return (
    <div className="col-md-8">
      <BaseImage img={pageContent.headerImage} />
      <hr />
      <h1>{pageContent.title}</h1>
      <hr />
      <StreamField value={pageContent.body} />
      <CommentForm {...props}/>                                    // new
    </div>
  );
}

export { PostDetail };
```

```
$ docker-compose up -d
$ docker-compose logs -f

# run command in another terminal
$ cd frontend
$ yarn dev
```

Let's go to http://www.local.test:3000/postpage1 to submit comment.



If the code works without problem, we should be able to see the comments in Wagtail admin. (http://api.local.test:8000/cms-admin/django_comments/comment/)

Chapter 28

Add Mention support to comment form with Tribute.js

28.1 Objective

By the end of this chapter, you should be able to:

1. Use `Tribute.js` to add mention support to HTML textarea element.
2. Learn how to access the DOM in React with `ref`

Let's make the comment form supports `@ mention`

28.2 API

Update *custom_comments/views.py*

```
from django_comments.models import Comment
from django.contrib.contenttypes.models import ContentType
from rest_framework.response import Response


class CommentMentionViewSet(viewsets.ViewSet):

    def list(self, request):
        content_type = request.GET.get("content_type")
        object_pk = request.GET.get("object_pk")

        app_label, model = content_type.split(".")
        content_type = ContentType.objects.get(app_label=app_label, model=model)

        comment_model = Comment
        data = comment_model.objects.filter(
            content_type=content_type, object_pk=object_pk,
        ).filter(
            is_removed=False
        ).values(
            "user_name"
        ).order_by('user_name').distinct()

        resp = {"result": list(data)}
        return Response(resp)
```

Update *custom_comments/api.py*

```
from rest_framework.routers import SimpleRouter
from .views import CommentViewSet, CommentMentionViewSet

# Create a router and register our viewsets with it.
comment_router = SimpleRouter()
comment_router.register(r'comments', CommentViewSet)
comment_router.register(r'comment-mentions', CommentMentionViewSet, basename='comment-mention')
```

```
$ docker-compose exec web python manage.py shell
```

```
>>> import requests
>>> resp = requests.get('http://127.0.0.1:8000/api/v1/comment-mentions/?content_type=blog.postpage&
↪object_pk=4')

>>> resp.json()
{'result': [{'user_name': 'michaelyin'}]}
>>> exit()
```

28.3 Install Packages

`Tributejs` is a cross-browser @mention engine written in ES6, let's use it in our Next.js project.

```
$ cd frontend
$ yarn add tributejs
```

28.4 Mention Style

`Tribute` also has SCSS file for the mention style, we can find it here https://github.com/zurb/tribute/blob/5.1.3/src/tribute.scss

Create *frontend/styles/_comment.scss*

```
.tribute-container {
  position: absolute;
  top: 0;
  left: 0;
  height: auto;
  max-height: 300px;
  max-width: 500px;
  overflow: auto;
  display: block;
  z-index: 999999;

  ul {
    margin: 0;
    margin-top: 2px;
    padding: 0;
    list-style: none;
    background: #efefef;
  }

  li {
    padding: 5px 5px;
    cursor: pointer;
```

```
    &.highlight {
      background: #ddd;
    }

    span {
      font-weight: bold;
    }

    &.no-match {
      cursor: default;
    }
  }

  .menu-highlighted {
    font-weight: bold;
  }
}
```

Update *frontend/styles/global.scss*

```
@import "~bootstrap/scss/bootstrap.scss";
@import "comment";
```

28.5 Javascript

Update *frontend/components/CommentForm.js*

```
import React, { useState, useEffect } from "react";
import { Form, Button } from "react-bootstrap";
import { postRequest, getRequest } from "../utils/wagtail";

const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;

async function getMentionTribute(Tribute, queryData) {
  const {url, ...params} = queryData;
  let tribute;

  try {
    const data = await getRequest(url, params);

    let values = [];
    for (const index in data.result) {
      const userValue = data.result[index];
      values.push({
        key: userValue.userName,
        value: userValue.userName,
      });
    }

    tribute = new Tribute({
      values: values,
    });
  } catch (error) {
    console.error(error);
  }

  return tribute;
}

function CommentForm(props) {
```

```
  const {pageContent} = props;
  const {id: objectPk, contentTypeStr: contentType} = pageContent;

  const commentInput = React.useRef();
  const [displayCommentForm, setDisplayCommentForm] = useState(false);
  const [isPosting, setIsPosting] = useState(false);
  const [commentFormSetup, setCommentFormSetup] = useState(false);

  useEffect(() => {
    if (displayCommentForm && !commentFormSetup) {
      const queryData = {
        url: `${API_BASE}/api/v1/comment-mentions/`,
        contentType,
        objectPk,
      };

      import("tributejs").then(function (Tribute) {
        getMentionTribute(Tribute.default, queryData).then(function (tribute) {
          if (tribute) {
            tribute.attach(commentInput.current);
          }
        });
        setCommentFormSetup(true);
      });
    }
  });

  // code omitted for brevity
}
```

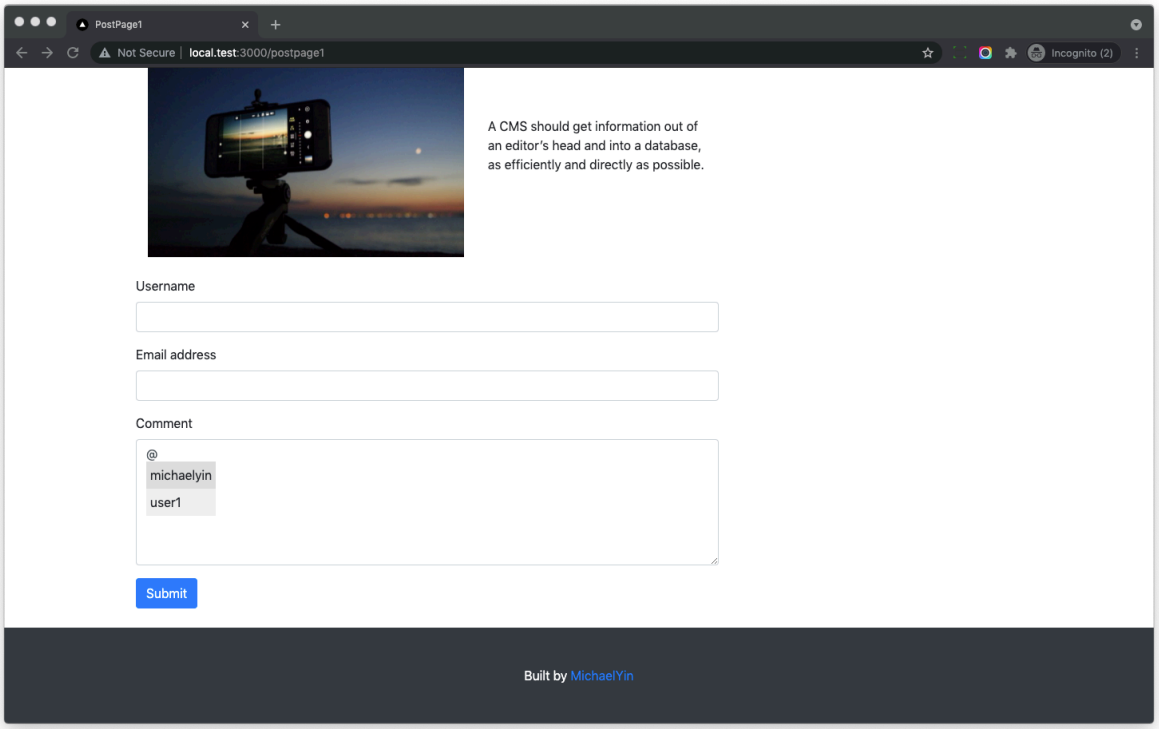
Notes:

1. We add `getMentionTribute` to fetch comment mentions from backend API, and return `tribute` object.
2. The `tribute.attach` need to attach the raw DOM element. Here we use `React.useRef()`, please check useRef doc[65] and Refs and the DOM[66]
3. `Tribute` can only work on the browser side (not Node env), so her we `Dynamic Import` in `useEffect`.

Now visit http://www.local.test:3000/postpage1

If we press `@` in the comment editor, we will see a popup window to let us do mention operation.

[65] https://reactjs.org/docs/hooks-reference.html#useref
[66] https://reactjs.org/docs/refs-and-the-dom.html



Notes:

1. Here we can see user mentions, if we keep typing `michael`, then only `michaelyin` would show up
2. We can use UP and DOWN arrow key to select
3. If we select the mention, for example, `michaelyin` here, then `@michaelyin` would be inserted to the textarea as plain text.

28.6 Extract Mention

As we know, HTML `textarea` is s multi-line plain-text editor. So we need to extract the mention from the plain text to know who is mentioned.

The `django-contrib-comments` has signal `comment_was_posted` and we can write signal receiver for it.

If you are new to `Django Signals`, please check Django Signals[67]

Update *custom_comments/serializers.py*

```
from django_comments.signals import comment_was_posted

class CommentSerializer(serializers.ModelSerializer):

    # code omitted for brevity

    def create(self, validated_data):
        # django_comments/views/comments.py
        # django_comments/forms.py
        app_label, model = validated_data.get("content_type").split(".")
        content_type = ContentType.objects.get(app_label=app_label, model=model)
        object_pk = validated_data.get("object_pk")

        try:
```

[67] https://docs.djangoproject.com/en/3.2/topics/signals/

```
            model = apps.get_model(app_label, model)
            model.objects.get(pk=object_pk)
        except Exception:
            raise Exception('can not find comment target object')

        comment_data = dict(
            content_type=content_type,
            object_pk=object_pk,
            user_name=validated_data["user_name"],
            user_email=validated_data["user_email"],
            comment=validated_data["comment"],
            submit_date=timezone.now(),
            site_id=getattr(settings, 'SITE_ID'),
            is_public=True,
            is_removed=False,
        )

        instance = super().create(comment_data)

        comment_was_posted.send(sender=instance.__class__,                          # new
                                comment=instance,
                                request=self.context['request'])

        return instance
```

Notes:

1. After comment instance is created, use `comment_was_posted.send` to fire signal.

Create *custom_comments/receivers.py*

```
import re

from django.dispatch import receiver
from django_comments.signals import comment_was_posted


@receiver(comment_was_posted)
def process_comment(sender, **kwargs):
    comment_instance = kwargs["comment"]
    comment_msg = comment_instance.comment

    mention_user_names = re.findall(r"@([^:\s]+)", comment_msg)
    print(f"user {mention_user_names} were just mentioned")
```

Notes:

1. Here we use Python `regex` module to help us extract the mention.
2. To help people understand, the logic is very simple, in some cases, you might need to validate the username in your project.

Next, let's do some config to register the signal receiver in the initialization process.

Update *custom_comments/apps.py*

```
from django.apps import AppConfig


class CustomCommentsConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'custom_comments'

    def ready(self):
        import custom_comments.receivers
```

Notes:

1. We import the above `recievers` in the `ready` method.
2. I recommend you to check Django doc: How applications are loaded[68] to better understand it.

Update `INSTALLED_APPS` in *wagtail_bootstrap_blog/settings/base.py*

```
INSTALLED_APPS = [
    'custom_comments.apps.CustomCommentsConfig',
]
```

Notes:

1. We change `custom_comments` to `custom_comments.apps.CustomCommentsConfig`.
2. `CustomCommentsConfig.ready` will not run until we do this.

```
$ docker-compose stop
$ docker-compose up -d --build
$ docker-compose logs -f
```

If you submit new comment which contains user mention, you will see something like this in the terminal output

```
web_1       | user ['michaelyin'] were just mentioned
```

28.7 Notes:

You can use Django `send_email` to send email notification in the signal receiver Django doc: Email sending[69]

For better performance, I would recommend move comment processing logic to Asynchronous task queue.

You can check The Definitive Guide to Celery and Django[70].

This course is also written by me (but published on testdriven.io) to help user get started with Celery and Django in quick way.

[68] https://docs.djangoproject.com/en/3.2/ref/applications/#initialization-process
[69] https://docs.djangoproject.com/en/3.1/topics/email/
[70] https://testdriven.io/courses/django-celery/

Chapter 29

Add Emoji support to Django form with Tribute.js

29.1 Objective

By the end of this chapter, you should be able to:

1. Learn how to customize Tribute.js

29.2 Workflow

1. Github has `emoji` api (https://api.github.com/emojis), which return emoji shortname and emoji image
2. We will create a new Tribute instance which has trigger key `:` (the same as Github and Slack)
3. As for the `values`, the key is the emoji short name, and the value is the emoji image.
4. User will see emoji image on the popup window, and text like `:smile:` will be inserted after selection.
5. The emoji shortname will be saved to db, and we can convert the emoji shortname to emoji unicode to display on the browser.

29.3 Implementation

Update *frontend/styles/_comment.scss*

```
/* stylelint-disable */
// for emoji image
.tribute-container {
  li {
    img {
      max-height: 16px;
    }
  }
}
/* stylelint-enable */
```

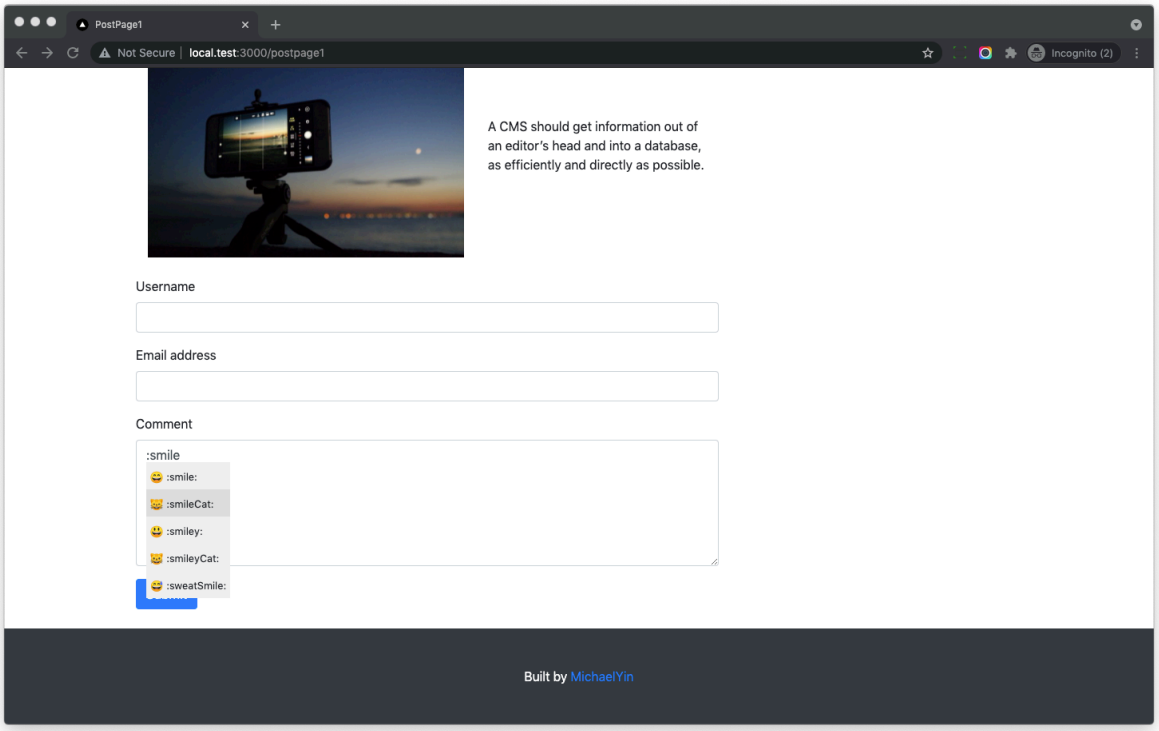
Update *frontend/components/CommentForm.js*

```
async function getEmojiTribute(Tribute) {
  const url = "https://api.github.com/emojis";
  let tribute;

  try {
    const data = await getRequest(url);

    let values = [];
    for (const key in data) {
      const value = data[key];
      values.push({
        key: key,
        value: value,
      });
    }
    tribute = new Tribute({
      trigger: ":",
      values: values,
      menuItemTemplate: function (item) {
        return `<img src="${item.original.value}"/> <small>:${item.original.key}:</small>`;
      },
      selectTemplate: function (item) {
        return `:${item.original.key}:`;
      },
      menuItemLimit: 5,
    });
  } catch (error) {
    console.error(error);
  }

  return tribute;
}

function CommentForm(props) {

  // code omitted for brevity

  useEffect(() => {
    if (displayCommentForm && !commentFormSetup) {
      const queryData = {
        url: `${API_BASE}/api/v1/comment-mentions/`,
        contentType,
        objectPk,
      };

      import("tributejs").then(function (Tribute) {
        getMentionTribute(Tribute.default, queryData).then(function (tribute) {
          if (tribute) {
            tribute.attach(commentInput.current);
          }
        });
        getEmojiTribute(Tribute.default).then(function (tributeEmoji) {
          if (tributeEmoji) {
            tributeEmoji.attach(commentInput.current);
          }
        });
        setCommentFormSetup(true);
      });
    }
  });

  // code omitted for brevity
}
```

Notes:

1. We added a new `getEmojiTribute` function, which would send Ajax request to the Github emoji API to get the emoji data.
2. When we create `Tribute` instance, we did some customization with the option object.
3. The `trigger` is `:` and the `menuItemTemplate` help us to display the emoji image from Github API.

   WARNING: Github API has rate limit, you can send request to your own Django view to solve this problem.



In the next chapter, I will talk about how to convert the emoji shortname to emoji unicode to display on the browser.

Chapter 30

Lazy Load CommentList

30.1 Objective

By the end of this chapter, you should be able to:

1. Understand what is SWR `stale-while-revalidate`
2. Learn to create custom React Hooks to extract component logic into reusable functions.
3. How to build CommentList which supports lazy load.

30.2 SWR

The name "SWR" is derived from `stale-while-revalidate`, SWR is a strategy to first return the data from cache (stale), then send the fetch request (revalidate), and finally come with the up-to-date data.

Let's install it

```
$ cd frontend
$ yarn add swr
```

30.3 Backend API

Update *custom_comments/views.py*

```
class CommentViewSet(viewsets.ModelViewSet):
    serializer_class = CommentSerializer
    queryset = Comment.objects.all()
    http_method_names = ["get", "post"]

    def get_queryset(self):                                    # new
        queryset = self.queryset.filter(is_removed=False)
        content_type = self.request.query_params.get('content_type')
        object_pk = self.request.query_params.get('object_pk')
        if content_type:
            app_label, model = content_type.split(".")
            content_type_instance = ContentType.objects.get(app_label=app_label, model=model)
            queryset = queryset.filter(content_type=content_type_instance)
        if object_pk:
            queryset = queryset.filter(object_pk=object_pk)
```

```
        return queryset
```

Here we add `get_queryset` to the `CommentViewSet` so it can filter comments based on `content_type` and `object_pk`

Update *nextjs_wagtail_app/settings.py*

```
REST_FRAMEWORK = {
    "DEFAULT_PAGINATION_CLASS": "rest_framework.pagination.LimitOffsetPagination",
    "PAGE_SIZE": 2,
}
```

Now if we visit http://api.local.test:8000/api/v1/comments/, the JSON response will also contain the pagination info.

```
{
  "count": 5,
  "next": "http://api.local.test:8000/api/v1/comments/?limit=2&offset=2",
  "previous": null,
  "results": [
    {
      "pk": 2,
      "content_type": "blog | post page",
      "object_pk": "4",
      "user_name": "michaelyin",
      "user_email": "michaelyin@accordbox.com",
      "comment": "Hello World",
      "submit_date": "2021-07-30T07:59:07.652822Z"
    },
    {
      "pk": 3,
      "content_type": "blog | post page",
      "object_pk": "4",
      "user_name": "michaelyin",
      "user_email": "michaelyin@accordbox.com",
      "comment": "Test and I love it",
      "submit_date": "2021-07-31T01:11:07.175843Z"
    }
  ]
}
```

30.4 useOnScreen

Create *frontend/hooks/useOnScreen.js*

```
import { useState, useEffect } from "react";

const useOnScreen = (ref) => {
  const [isIntersecting, setIntersecting] = useState(false);

  useEffect(() => {
    const observer = new IntersectionObserver(([entry]) =>
      setIntersecting(entry.isIntersecting)
    );

    observer.observe(ref.current);
    // Remove the observer as soon as the component is unmounted
    return () => {
      observer.disconnect();
```

```
    };
  }, []);

  return isIntersecting;
};

export default useOnScreen;
```

1. Building custom Hooks lets us extract component logic into reusable functions.
2. This `useOnScreen` can help us detect visibility of the `ref` element
3. If you want to know more about `IntersectionObserver`, check https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API
4. If you want to know more about `Custom React Hook`, check https://reactjs.org/docs/hooks-custom.html

30.5 CommentList

Create *frontend/components/CommentList.js*

```
import React, { useState, useEffect, useRef } from "react";
import { Media } from "react-bootstrap";
import useSWR, { mutate } from "swr";
import { CommentForm } from "./CommentForm";
import { getRequest } from "../utils/wagtail";
import useOnScreen from "../hooks/useOnScreen";

const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;

// https://swr.vercel.app/docs/arguments
const fetcher = (...args) => {
  const [url, objectPk, contentType] = args;
  return getRequest(url, {
    objectPk: objectPk,
    contentType: contentType,
  });
};

function CommentList(props) {
  const {pageContent} = props;
  const {id: objectPk, contentTypeStr: contentType} = pageContent;

  const ref = React.useRef();
  const isVisible = useOnScreen(ref);

  const [commentsCount, setCommentsCount] = useState(0);
  const [loadComments, setLoadComments] = useState([]);

  const COMMENTS_API_URL = `${API_BASE}/api/v1/comments/`;
  const key = [COMMENTS_API_URL, objectPk, contentType];
  const { data } = useSWR(key, fetcher);

  const refreshForNewComment = () => {
    mutate(key);
  };

  useEffect(() => {
    if (data && commentsCount !== data.count) {
      setCommentsCount(data.count);
```

```
    }
  }, [data, commentsCount]);

  useEffect(() => {
    if (isVisible && loadComments.length !== commentsCount) {
      getRequest(COMMENTS_API_URL, {
        limit: 10,
        offset: loadComments.length,
        objectPk: objectPk,
        contentType: contentType,
      }).then((res) => {
        // combine
        const newComments = [...loadComments, ...res.results];
        setLoadComments(newComments);
      });
    }
  }, [isVisible, commentsCount, loadComments, objectPk, contentType]);

  return (
    <React.Fragment>
      {loadComments.map((commentObj) =>
        <div className="d-flex my-4" key={commentObj.pk}>
          {/* eslint-disable-next-line */}
          <img
            width={64}
            height={64}
            src="http://via.placeholder.com/64"
            alt="Generic placeholder"
          />
          <div className="ms-3">
            <div className="comment-date">
              <strong className="text-primary">{commentObj.userName}</strong>{" "}
              <small>{new Date(commentObj.submitDate).toString()}</small>
            </div>
            <div dangerouslySetInnerHTML={{__html: commentObj.comment}}/>
          </div>
        </div>
      )}
      <div ref={ref} />
      <CommentForm refreshForNewComment={refreshForNewComment} {...props} />
    </React.Fragment>
  );
}

export { CommentList };
```

Notes:

1. We passed a `key` array and `fetcher` function to the `useSWR`.
2. The `fetcher` function will receive the key array and send request to get the comment data.
3. In `refreshForNewComment` function, we call `mutate(key)` to tell all SWRs with this key to revalidate (it will send request to revalidate again)
4. With `data` returned by `useSWR`, if the `data.count` does not equal `commentsCount`, call `setCommentsCount(data.count)` to set the state.
5. In the other `useEffect`, if `isVisible`, try to load comments and set to `loadComments`.
6. We set `<CommentForm refreshForNewComment={refreshForNewComment} {...props} />`, so if new comment is posted, the form will call `refreshForNewComment` to revalidate.

Update *frontend/components/CommentForm.js*

```
function CommentForm(props) {
  const { pageContent, refreshForNewComment} = props;
  const { id: objectPk, contentTypeStr: contentType} = pageContent;

  const handleFormSubmit = (e) => {
    e.preventDefault();
    setIsPosting(true);
    let formData = new FormData(e.target),
      formDataObj = Object.fromEntries(formData.entries());
    formDataObj.contentType = contentType;
    formDataObj.objectPk = objectPk;

    postRequest(`${API_BASE}/api/v1/comments/`, formDataObj)
      .then((data) => {
        e.target.reset();
        refreshForNewComment();                                          // new
        setIsPosting(false);
      })
  };

}
```

Update *frontend/components/PostDetail.js*

```
import React from "react";
import { StreamField } from "./StreamField/StreamField";
import { BaseImage } from "./BaseImage";
import { CommentList } from "./CommentList";

function PostDetail(props) {
  const { pageContent } = props;

  return (
    <div className="col-md-8">
      <BaseImage img={pageContent.headerImage} />
      <hr />
      <h1>{pageContent.title}</h1>
      <hr />
      <StreamField value={pageContent.body} />
      <CommentList {...props}/>
    </div>
  );
}

export { PostDetail };
```

Now, please open devtools and check network requests on the http://www.local.test:3000/postpage1

1. If we load the page, `SWR` will send request to check if the response data has changed.
2. If the response data has changed, the first `useEffect` will call `setCommentsCount(data.count)`
3. Please check another tab of the browser, and then click back. `SWR` will send a new request to check if the data has changed. This feature called `Revalidate on Focus`, and you can check https://swr.vercel.app/docs/revalidation
4. The real comments will not be loaded until we scroll to the bottom of the page.

30.6 Render Emoji

Next, let's work to convert emoji shortname to unicode.

We will use https://github.com/carpedm20/emoji/ to help us.

Add `emoji` to the *requirements.txt*

```
emoji==0.6.0
```

```
$ docker-compose up -d --build
$ docker-compose logs -f
```

Update *custom_comments/serializers.py*

```
class CommentSerializer(serializers.ModelSerializer):
    content_type = serializers.CharField()
    pretty_comment = serializers.SerializerMethodField()               # new

    class Meta:
        model = Comment
        fields = (
            "pk",
            "content_type",
            "object_pk",

            "user_name",
            "user_email",
            "comment",
            "pretty_comment",
            "submit_date",
        )

    def get_pretty_comment(self, obj):                                 # new
        import emoji
        from django.template.defaultfilters import linebreaks
        return linebreaks(emoji.emojize(obj.comment, use_aliases=True))
```

Notes:

1. We add `pretty_comment` field, which uses the `emoji` package to convert emoji shortname to unicode.
2. The `django.template.defaultfilters.linebreaks` would convert linebreaks to html `<br>` element.

Update *frontend/components/CommentList.js*

```
return (
  <React.Fragment>
    {loadComments.map((commentObj) =>
      <div className="d-flex my-4" key={commentObj.pk}>
        {/* eslint-disable-next-line */}
        <img
          width={64}
          height={64}
          src="http://via.placeholder.com/64"
          alt="Generic placeholder"
        />
        <div className="ms-3">
          <div className="comment-date">
            <strong className="text-primary">{commentObj.userName}</strong>{" "}
            <small>{new Date(commentObj.submitDate).toString()}</small>
          </div>
          <div dangerouslySetInnerHTML={{__html: commentObj.prettyComment}}/>         // new
        </div>
      </div>
    )}
```

```
    <div ref={ref} />
    <CommentForm refreshForNewComment={refreshForNewComment} {...props} />
  </React.Fragment>
);
```

We display `commentObj.prettyComment` instead of the `commentObj.Comment`

Chapter 31

Next.js UnitTest Guide (Part 1)

31.1 Objective

By the end of this chapter, you should be able to:

1. Understand what is `jest` and how to set it up with Next.js
2. Write unittest for React component.
3. Learn how to do `snapshot` test

31.2 Setup Jest

`Jest` is a delightful JavaScript Testing Framework with a focus on simplicity. It works with projects using: Babel, TypeScript, Node, React, Angular, Vue and more!

`Jest` provides basic features for us to test normal JS code, if we want to test UI components in clean way, we still need testing-library[71]

> The @testing-library family of packages helps you test UI components in a user-centric way.

1. `@testing-library/jest-dom` is a companion library for Testing Library that provides custom DOM element matchers for Jest
2. `@testing-library/react` builds on top of DOM Testing Library by adding APIs for working with React components.

```
$ cd frontend
$ yarn add jest babel-jest @testing-library/jest-dom @testing-library/react jest-next-dynamic babel-
↪plugin-dynamic-import-node
```

Next, update *frontend/package.json*

```
{
  "scripts": {
    "dev": "next dev",
      "build": "next build",
      "start": "next start",
      "lint": "next lint",
      "test": "jest"                // new
  },
}
```

[71] https://testing-library.com/docs/

Notes:

1. We add a new command `"test": "jest"` to the `scripts`

Create *frontend/jest.config.js*

```
// https://stackoverflow.com/questions/54627028/jest-unexpected-token-when-importing-cs

module.exports = {
  testPathIgnorePatterns: ["/node_modules/", "/.next/"],
  globalSetup: "<rootDir>/setupTestEnv.js",
  setupFilesAfterEnv: ["<rootDir>/setupTests.js"],
  clearMocks: true,
  testEnvironment: 'jsdom',
  moduleNameMapper: {
    "\\.(css|less|scss|sass)$": "identity-obj-proxy"
  },
  modulePathIgnorePatterns: [
    '.next/',
  ],
};
```

Notes:

1. `frontend/jest.config.js` is the config file of `jest`
2. We will create `setupTestEnv.js` and `setupTests.js` in a bit.

Create *frontend/setupTestEnv.js*

```
import { loadEnvConfig } from '@next/env'

export default async () => {
  const projectDir = process.cwd()
  loadEnvConfig(projectDir)
}
```

Notes:

1. The above code will make Next.js load `.env.test` during the test.

Create *frontend/setupTests.js*

```
import '@testing-library/jest-dom';
// https://github.com/vercel/next.js/discussions/13678
import next from 'next';

jest.isolateModules(() => {
  const preloadAll = require('jest-next-dynamic');

  beforeEach(async () => {
    await preloadAll();
  });
});
```

Notes:

1. The `import next from 'next';` will help resolve `ReferenceError: fetch is not defined` issue.
2. The `jest-next-dynamic` package can help us resolve Next.js dynamic import during Jest tests.

Create *frontend/.env.test*

```
# https://stackoverflow.com/questions/63934104/environment-variables-undefined-in-nextjs-when-
↪running-jest
```

```
NEXT_PUBLIC_WAGTAIL_API_BASE=http://api.local.test:8000
NEXT_PUBLIC_NEXT_BASE=http://www.local.test:3000
```

Create *frontend/.babelrc*

```
{
  "presets": [
    "next/babel"
  ],
  "env": {
    "test": {
      "plugins": [
        "babel-plugin-dynamic-import-node"
      ]
    }
  }
}
```

31.3 First Test

Create *frontend/components/TagWidget.test.js*

The filename `TagWidget.test.js` tell us it is the test file for `TagWidget.js`, you can also check Jest testMatch[72] to learn more.

```
test('Simple Test', () => {
  // write test code here
  expect(1 + 1).toBe(2);
})
```

Notes:

1. `Simple Test` is the test name.
2. The `1+1` is the expression to test, in most cases, it can be DOM elements, or JS function call.

Let's run test

```
$ yarn test

 PASS  components/TagWidget.test.js
  ✓ Simple Test (4 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.211 s
Ran all test suites.
✨  Done in 2.92s.
```

31.4 TagWidget Test

Now, we will write real test for our `TagWidget` component.

[72] https://jestjs.io/docs/configuration#testmatch-arraystring

31.4.1 AAA pattern

When writing unittest, you should follow `AAA` pattern.

1. `Arrange`: You whould make the test env ready, such as mock operation, or create some `test data`.
2. `Act`: Execute the code
3. `Assert`: Check returned value and other objects to make sure everything works as expected.

31.4.2 Test philosophy

Some people who are new to `testing-library` feel confused because they can not get the `props` and `state` of component. Let's check the words from testing-library doc[73]

> Testing Library encourages you to avoid testing implementation details like the internals of a component you're testing (though it's still possible). The Guiding Principles of this library emphasize a focus on tests that closely resemble how your web pages are interacted by the users.

So `testing-library` is more focused on the DOM and user interactions, instead of the component internal details.

If you want to test by checking component `props` and `state`, you can take a look at another framework Enzyme[74], but we will not use it in this course.

31.4.3 Test Data

Create *frontend/components/mockData.js* , and copy JSON response from the `http://api.local.test:8000/`

And then we convert the JSON string to JS object on https://www.convertsimple.com/convert-json-to-javascript/

```
const blogPageData = {
  page_type: "BlogPage",
  page_content: {
    // code omitted for brevity
  },
  categories_list: [
    // code omitted for brevity
  ],
  tags_list: [
    // code omitted for brevity
  ],
  children_pages: [
    // code omitted for brevity
  ],
  paginator: {
    // code omitted for brevity
  },
  filter_meta: {
    // code omitted for brevity
  }
}

export { blogPageData };
```

[73] https://testing-library.com/docs/
[74] https://enzymejs.github.io/enzyme/

31.4.4 Test Code

Update *frontend/components/TagWidget.test.js*

```
import { render, screen } from "@testing-library/react";
import { TagWidget } from "./TagWidget.js";
import { blogPageData } from "./mockData";
import camelcaseKeys from "camelcase-keys";

test("TagWidget Test", () => {
  const data = camelcaseKeys(blogPageData, {deep: true})
  render(<TagWidget {...data} />);

  const { tagsList } = data;

  const el = screen.getByText(tagsList[0].name);
  expect(el.tagName).toEqual("SPAN");
  expect(el).toHaveClass("badge bg-secondary");

  tagsList.map((tag) =>
    expect(screen.getByText(tag.name)).toBeInTheDocument()
  );

});
```

Notes:

1. We render `TagWidget` with test data first.
2. Use `screen.getByText` to find the element, and check the tag and class.

```
$ yarn test

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.262 s
Ran all test suites.
```

31.5 Snapshot Test

> Snapshot tests are a very useful tool whenever you want to make sure your UI does not change unexpectedly.

The logic of `Snapshot tests` is it would compare `snapshot` or the component and make sure the UI would be consistent during the test.

Update *frontend/components/TagWidget.test.js*

```
import { render, screen } from "@testing-library/react";
import { TagWidget } from "./TagWidget.js";
import { blogPageData } from "./mockData";
import camelcaseKeys from "camelcase-keys";

test("TagWidget Test", () => {
  const data = camelcaseKeys(blogPageData, {deep: true})
  const { asFragment } = render(<TagWidget {...data} />);

  const { tagsList } = data;

  const el = screen.getByText(tagsList[0].name);
```

```
  expect(el.tagName).toEqual("SPAN");
  expect(el).toHaveClass("badge bg-secondary");

  tagsList.map((tag) =>
    expect(screen.getByText(tag.name)).toBeInTheDocument()
  );

  expect(asFragment()).toMatchSnapshot();
});
```

Notes:

1. `expect(asFragment()).toMatchSnapshot();` is the code line we add here.

```
$ yarn test

 PASS  components/TagWidget.test.js
  ✓ TagWidget Test (51 ms)

 › 1 snapshot written.
Snapshot Summary
 › 1 snapshot written from 1 test suite.

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 written, 1 total
Time:        1.608 s, estimated 2 s
Ran all test suites.
```

Notes:

1. As you see, `1 snapshot written` is in the above terminal output.
2. And we see a file *frontend/components/**snapshots**/TagWidget.test.js.snap* is created.
3. If you check the file, you will see it contains HTML of the component.

```
$ yarn test

 PASS  src/components/TagWidget.test.js
  ✓ TagWidget Test (98 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   1 passed, 1 total
Time:        2.904 s, estimated 3 s
Ran all test suites.
```

1. Jest would check the snapshot, this can help us detect unexpected UI change of the component.
2. You can check Snapshot Testing[75] to learn more.

[75] https://jestjs.io/docs/snapshot-testing

Chapter 32

Next.js UnitTest Guide (Part 2)

32.1 Objective

By the end of this chapter, you should be able to:

1. Use `Jest` to do mock

32.2 Background

Sometimes, mounting a whole component (such as `BlogPage`) with all its child components might become slow or cumbersome.

There are some ways to solve this:

1. **Shallow Rendering**, this is supported in Enzyme[76]
2. **Mock component**, this is the method we will use in this chapter.

32.3 BlogPage

Create *frontend/components/BlogPage.test.js*

```
import { BlogPage } from "./BlogPage";
import { screen, render } from "@testing-library/react";
import { blogPageData } from "./mockData";
import { PostPageCardContainer } from "./PostPageCardContainer";
import { SideBar } from "./SideBar";
import camelcaseKeys from "camelcase-keys";


jest.mock("./PostPageCardContainer", () => ({
  PostPageCardContainer:
    jest.fn(({ children }) => <div data-testid="PostPageCardContainer">{children}</div>)
}));

jest.mock("./SideBar", () => ({
  SideBar:
    jest.fn(({ children }) => <div data-testid="SideBar">{children}</div>)
}));
```

[76] https://enzymejs.github.io/enzyme/docs/api/shallow.html

```
test('BlogPage Test', () => {
  const data = camelcaseKeys(blogPageData, {deep: true})

  const {asFragment} = render(
    <BlogPage {...data}/>
  );

  expect(screen.queryByTestId("PostPageCardContainer")).toBeInTheDocument();
  expect(PostPageCardContainer).toHaveBeenCalledWith(
    data,
    expect.anything(),
  );

  expect(screen.queryByTestId("SideBar")).toBeInTheDocument();
  expect(SideBar).toHaveBeenCalledWith(
    data,
    expect.anything(),
  );

  expect(asFragment()).toMatchSnapshot();
});
```

Notes:

1. We mock `PostPageCardContainer` and `SideBar` at the top.
2. The `data-testid` can help us check if the component has been rendered (`screen.queryByTestId("PostPageCardContainer")`)
3. We use `toHaveBeenCalledWith` to check if the React `props` has been passed to child components in correct way.
4. The `expect.anything()` in toHaveBeenCalledWith can tell Jest only check specific parameters.

```
$ yarn test
```

You can see code below in *frontend/components/**snapshots**/BlogPage.test.js.snap*

```
<div
  class="row"
>
  <div
    data-testid="PostPageCardContainer"
  />
  <div
    data-testid="SideBar"
  />
</div>
```

32.4 PageProxy

Create *frontend/components/PageProxy.test.js*

```
import { PageProxy } from "./PageProxy";
import { screen, render } from "@testing-library/react";
import { blogPageData } from "./mockData";
import { PostPageCardContainer } from "./PostPageCardContainer";
import { SideBar } from "./SideBar";
import camelcaseKeys from "camelcase-keys";
```

```
jest.mock("./PostPageCardContainer", () => ({
  PostPageCardContainer:
    jest.fn(({children}) => <div data-testid="PostPageCardContainer">{children}</div>)
}));

jest.mock("./SideBar", () => ({
  SideBar:
    jest.fn(({children}) => <div data-testid="SideBar">{children}</div>)
}));

test('BlogPage', () => {
  const data = camelcaseKeys(blogPageData, {deep: true})

  const {asFragment} = render(
    <PageProxy {...data}/>
  );

  expect(screen.queryByTestId("PostPageCardContainer")).toBeInTheDocument();
  expect(PostPageCardContainer).toHaveBeenCalledWith(
    data,
    expect.anything(),
  );

  expect(screen.queryByTestId("SideBar")).toBeInTheDocument();
  expect(SideBar).toHaveBeenCalledWith(
    data,
    expect.anything(),
  );

  expect(asFragment()).toMatchSnapshot();
});
```

Notes:

1. The test logic is very similar with `BlogPage`.
2. Here we render `PageProxy` instead of `PostPage`.
3. If you see some error caused by `'next/dynamic'`, please check jest-next-dynamic[77]

[77] https://github.com/FormidableLabs/jest-next-dynamic

Chapter 33

Next.js UnitTest Guide (Part 3)

33.1 Objective

By the end of this chapter, you should be able to:

1. Use `msw` to mock API response.
2. Learn how to use `mockFn.mockImplementation`
3. Generate code coverage report

33.2 Mock API response

During the test, sometimes, we need to mock the network response.

There are multiple ways to do this:

1. Use `jest.mock`
2. Use 3-party packages such as `axios-mock-adapter` or `fetch-mock`

Here I'd like to recommend another tool msw (Mock Service Worker)[78]

This package works on network level, and it can work for both `Node.js` or `browser`

```
$ cd frontend
$ yarn add msw
```

33.3 SearchPage

Let's create *frontend/components/SearchPage.test.js*

```
import { rest } from 'msw'
import {render, fireEvent, waitFor, screen} from '@testing-library/react'
import { setupServer } from 'msw/node'
import { useRouter } from 'next/router'
import { SearchPage } from "./SearchPage";

const API_BASE = process.env.NEXT_PUBLIC_WAGTAIL_API_BASE;

jest.mock('next/router', () => ({
```

[78] https://github.com/mswjs/msw

```
  useRouter: jest.fn()
}))
```

Notes:

1. We mock `next/router` because it is used by `SearchPage`

```
const page1Response = {
  ...
}

const page2Response = {
  ...
}

const server = setupServer(
  rest.get(`${API_BASE}/api/v1/nextjs/pages/`, (req, res, ctx) => {
    const offset = req.url.searchParams.get('offset')
    if (offset === '0') {
      return res(
        ctx.json(page1Response),
      )
    } else if (offset === '2'){
      return res(
        ctx.json(page2Response),
      )
    }
  }),
)

// Enable API mocking before tests.
beforeAll(() => server.listen())

// Reset any runtime request handlers we may add during the tests.
afterEach(() => server.resetHandlers())

// Disable API mocking after the tests are done.
afterAll(() => server.close())
```

Notes:

1. In `setupServer`, we use `rest.get` to define how to handle GET request to `${API_BASE}/api/v1/nextjs/pages/`
2. And we return different response based on the `offset` querystring.

```
test('Search on page load', async () => {
  useRouter.mockImplementation(() => ({
    query: {
      query: 'wagtail'
    }
  }))

  render(<SearchPage/>);

  await waitFor(() => expect(
    screen.getByText('Load More', { selector: 'button' })).toBeInTheDocument()
  );

  await waitFor(() => expect(
    screen.getByText('PostPage4', { selector: 'button' })).toBeInTheDocument()
  );
  await waitFor(() => expect(
```

```
    screen.getByText('PostPage3', { selector: 'button' })).toBeInTheDocument()
  );

  fireEvent.click(screen.getByText('Load More'))

  // Load 2 page
  await waitFor(() => expect(
    screen.getByText('PostPage2', { selector: 'button' })).toBeInTheDocument()
  );

  await waitFor(() => expect(
    screen.getByText('PostPage1', { selector: 'button' })).toBeInTheDocument()
  );

})
```

Notes:

1. In the test, we use `useRouter.mockImplementation` to make `route.query.query` available.
2. We will test if the search page can search on page load, and `load more` button works without problem.

```
$ yarn test

 PASS  components/TagWidget.test.js
 PASS  components/PageProxy.test.js
 PASS  components/BlogPage.test.js
 PASS  components/SearchPage.test.js

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   3 passed, 3 total
Time:        3.244 s
Ran all test suites.
```

33.4 Test Coverage

Test coverage report can print stats about the test, which can give us confidence.

```
$ yarn test -- --coverage

-----------------------|---------|----------|---------|---------|-------------------
File                   | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------------|---------|----------|---------|---------|-------------------
All files              |   27.44 |    28.57 |   30.99 |   26.73 |
 components            |   35.07 |    40.43 |   43.48 |   33.07 |
  BaseImage.js         |      50 |        0 |       0 |      50 | 6
  BlogPage.js          |     100 |      100 |     100 |     100 |
  CommentForm.js       |    1.82 |        0 |       0 |    1.82 | 8-151
  CommentList.js       |       8 |        0 |       0 |       8 | 12-61
  Footer.js            |     100 |      100 |     100 |     100 |
  HeadMeta.js          |     100 |       50 |     100 |     100 | 10
  LazyPages.js         |     100 |      100 |     100 |     100 |
  PageProxy.js         |      80 |       50 |     100 |      80 | 17
  PostDetail.js        |       0 |      100 |       0 |       0 | 7-9
  PostPage.js          |       0 |      100 |       0 |       0 | 9
  SearchPage.js        |   83.87 |       85 |   69.23 |   88.46 | 82-84,109
  TagWidget.js         |     100 |      100 |     100 |     100 |
  TopNav.js            |     100 |      100 |     100 |     100 |
```

```
  mockData.js           |     100 |      100 |     100 |     100 |
 components/StreamField |       0 |        0 |       0 |       0 |
  ImageCarousel.js      |       0 |      100 |       0 |       0 | 6-10
  ImageText.js          |       0 |        0 |       0 |       0 | 6-8
  StreamField.js        |       0 |        0 |       0 |       0 | 7-48
  ThumbnailGallery.js   |       0 |      100 |       0 |       0 | 6-12
 hooks                  |   11.11 |      100 |       0 |   11.11 |
  useOnScreen.js        |   11.11 |      100 |       0 |   11.11 | 4-18
 utils                  |   23.91 |    18.75 |   13.33 |   26.83 |
  wagtail.js            |   23.91 |    18.75 |   13.33 |   26.83 | 10-72,92-109
------------------------|---------|----------|---------|---------|-------------------

Test Suites: 4 passed, 4 total
Tests:       4 passed, 4 total
Snapshots:   3 passed, 3 total
Time:        3.532 s
```

33.5 Next Step:

1. Now we already learned ho to test the `individual page components`
2. For the page routes (`getStaticPaths`, `getStaticProps`), we can test them as normal JS functions.
3. If you want to do `integration tests`, you can check cypress[79]

[79] https://www.cypress.io/

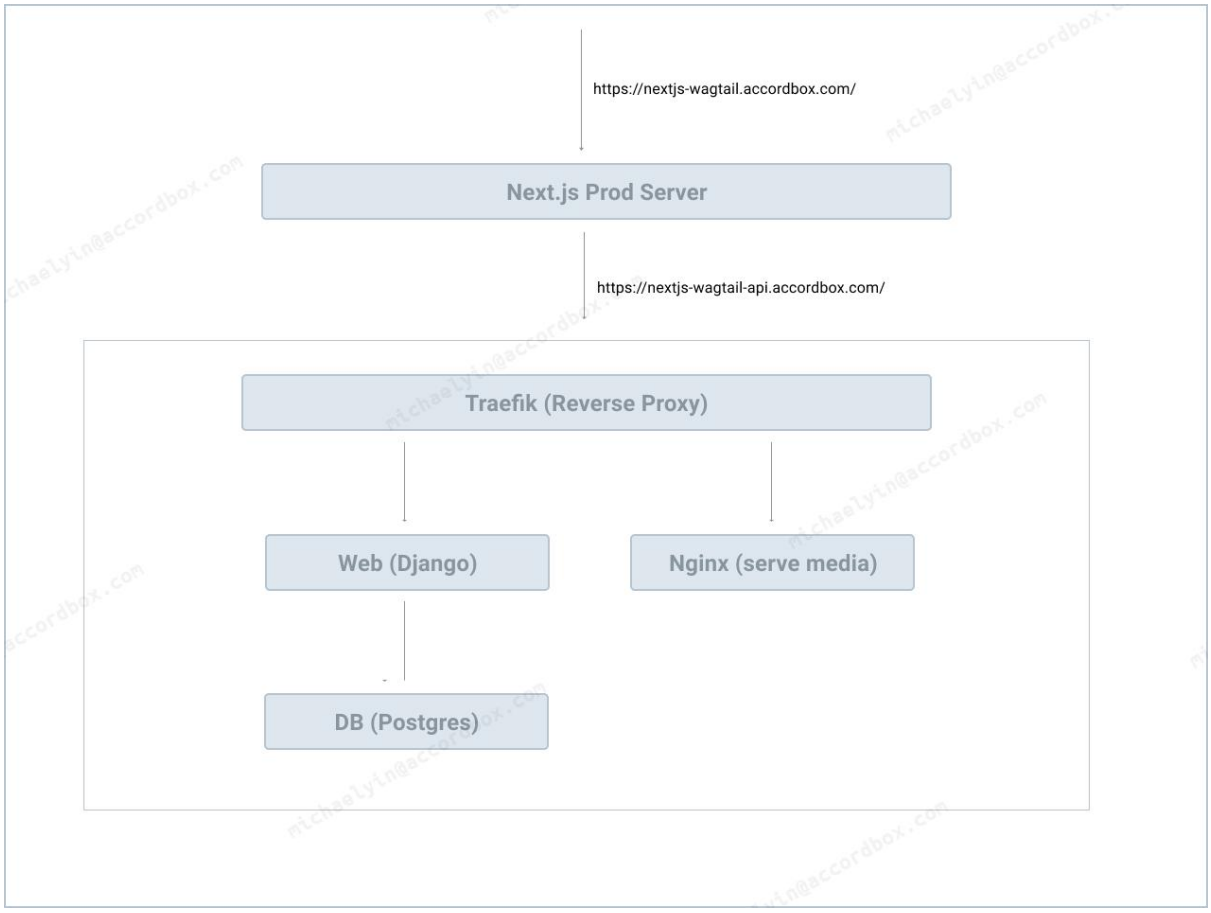
Chapter 34

Deploy REST API

34.1 Objective

In this chapter, we will learn how to deploy the Wagtail app to DigitalOcean[80] with Docker Compose. The backend API will serve on `https://nextjs-wagtail-api.accordbox.com`

34.2 Workflow



[80] https://www.digitalocean.com/

34.3 Compose File

Let's create *docker-compose.prod.yml*, the `prod` means this compose file is for our `production` app.

```
version: '3.7'

services:

  traefik:
    build:
      context: .
      dockerfile: ./compose/production/traefik/Dockerfile
    volumes:
      - production_traefik:/etc/traefik/acme:z
    ports:
      - "0.0.0.0:80:80"
      - "0.0.0.0:443:443"

  nginx:
    build:
      context: .
      dockerfile: ./compose/production/nginx/Dockerfile
    volumes:
      - mediafiles:/app/media
    depends_on:
      - web

  web:
    build:
      context: .
      dockerfile: ./compose/production/django/Dockerfile
    command: /start
    volumes:
      - mediafiles:/app/media
    env_file:
      - ./.env/.prod-sample
    depends_on:
      - db

  db:
    image: postgres:12.0-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment:
      - POSTGRES_DB=react_wagtail_dev
      - POSTGRES_USER=react_wagtail
      - POSTGRES_PASSWORD=react_wagtail

volumes:
  postgres_data:
  mediafiles:
  production_traefik:
```

Notes:

1. Here we create 4 services, `traefik` is `reverse proxy` for `web` service and `nginx` service.
2. Here we use `nginx` to serve the media files, if you use object storage such as AWS S3, then you do not need this service.
3. For `web` service, we created `staticfiles` and `mediafiles` docker volume to store the assets.
4. All env variables are stored in `.env/.prod-sample`

Create `production` directory under the `compose`, and then create sub directory `web` and `nginx`.

So you would have file structure like this.

```
compose
├── local
│   └── django
│       ├── Dockerfile
│       ├── entrypoint
│       └── start
└── production
    ├── django
    │   ├── Dockerfile
    │   ├── entrypoint
    │   └── start
    ├── nginx
    │   ├── Dockerfile
    │   └── nginx.conf
    └── traefik
        ├── Dockerfile
        └── traefik.yml
```

34.4 Traefik Service

Create *compose/production/traefik/Dockerfile*

```
FROM traefik:v2.2.11
RUN mkdir -p /etc/traefik/acme \
  && touch /etc/traefik/acme/acme.json \
  && chmod 600 /etc/traefik/acme/acme.json
COPY ./compose/production/traefik/traefik.yml /etc/traefik
```

Create *compose/production/traefik/traefik.yml*

```
log:
  level: INFO

entryPoints:
  web:
    # http
    address: ":80"

http:
  routers:
    web-media-router:
      rule: "Host(`api.local.test`) && PathPrefix(`/media/`)"
      entryPoints:
        - web
      middlewares:
        - csrf
      service: django-media

    web-secure-router:
      rule: "Host(`api.local.test`)"
      entryPoints:
        - web
      middlewares:
        - csrf
      service: django

  middlewares:
```

```
    csrf:
      # https://docs.traefik.io/master/middlewares/headers/#hostsproxyheaders
      # https://docs.djangoproject.com/en/dev/ref/csrf/#ajax
      headers:
        hostsProxyHeaders: ["X-CSRFToken"]

  services:
    django:
      loadBalancer:
        servers:
          - url: http://web:8000
    django-media:
      loadBalancer:
        servers:
          - url: http://nginx:80

providers:
  # https://docs.traefik.io/master/providers/file/
  file:
    filename: /etc/traefik/traefik.yml
    watch: true
```

Notes:

1. In the `http.router`, if the request URL match prefix `/media/`, then it will be sent to Nginx. Other requests will be sent to `web` service.
2. Please note the `api.local.test` is hard-coded in the file.

34.5 Nginx Service

Create *compose/production/nginx/Dockerfile*:

```
FROM nginx:1.19.2-alpine

RUN rm /etc/nginx/conf.d/default.conf
COPY ./compose/production/nginx/nginx.conf /etc/nginx/conf.d
```

Create *compose/production/nginx/nginx.conf*:

```
server {
    listen 80;
    client_max_body_size 20M;

    location /media/ {
        alias /app/media/;
    }
}
```

Notes:

1. `client_max_body_size 20M;` is to solve "`Request Entity Too Large`" error when we upload image in Wagtail admin.
2. `/app/media/` point to the docker volume `mediafiles`, where Nginx would find files.

34.6 Web Service

Before we start, let's take a look at the file structures

```
├── compose
│   └── production
│       ├── django
│       │   ├── Dockerfile
│       │   ├── entrypoint
│       │   └── start
```

34.6.1 DockerFile

Create *compose/production/django/Dockerfile*

```
FROM python:3.8-slim-buster

ENV PYTHONUNBUFFERED 1

RUN apt-get update \
  # dependencies for building Python packages
  && apt-get install -y build-essential netcat \
  # psycopg2 dependencies
  && apt-get install -y libpq-dev \
  # Translations dependencies
  && apt-get install -y gettext \
  # cleaning up unused files
  && apt-get purge -y --auto-remove -o APT::AutoRemove::RecommendsImportant=false \
  && rm -rf /var/lib/apt/lists/*

RUN addgroup --system django \
    && adduser --system --ingroup django django

# Requirements are installed here to ensure they will be cached.
COPY ./requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

COPY ./compose/production/django/entrypoint /entrypoint
RUN sed -i 's/\r$//g' /entrypoint
RUN chmod +x /entrypoint
RUN chown django /entrypoint

COPY ./compose/production/django/start /start
RUN sed -i 's/\r$//g' /start
RUN chmod +x /start
RUN chown django /start

WORKDIR /app

# avoid 'permission denied' error
RUN mkdir /app/media

# copy project code
COPY . .

RUN chown -R django:django /app

USER django

ENTRYPOINT ["/entrypoint"]
```

Notes:

1. We added a `django` user and used it to run the entrypoint command for security.

2. We use `RUN mkdir /app/media` combined with `RUN chown -R django:django /app` to solve the `permission denied` problem.

34.6.2 Entrypoint

Create *compose/production/django/entrypoint*

```
#!/bin/bash

set -o errexit
set -o pipefail
set -o nounset

postgres_ready() {
python << END
import sys

import psycopg2

try:
    psycopg2.connect(
        dbname="${SQL_DATABASE}",
        user="${SQL_USER}",
        password="${SQL_PASSWORD}",
        host="${SQL_HOST}",
        port="${SQL_PORT}",
    )
except psycopg2.OperationalError:
    sys.exit(-1)
sys.exit(0)

END
}
until postgres_ready; do
  >&2 echo 'Waiting for PostgreSQL to become available...'
  sleep 1
done
>&2 echo 'PostgreSQL is available'

exec "$@"
```

Notes:

1. We defined a `postgres_ready` function which is called in loop.
2. The `exec "$@"` is used to make the entrypoint a pass through to ensure that Docker runs the command the user passes in (`command: /start`, in our case). For more, check this Stack Overflow answer[81].

34.6.3 Start script

Update *compose/production/django/start*

```
#!/bin/bash

set -o errexit
set -o pipefail
set -o nounset
```

[81] https://stackoverflow.com/a/39082923/2371995

```
python /app/manage.py collectstatic --noinput
python /app/manage.py migrate

/usr/local/bin/gunicorn nextjs_wagtail_app.wsgi:application --bind 0.0.0.0:8000 --chdir=/app
```

Notes:

1. We should `collecstatic` to collect static assets for `production` app Serving static files in production[82]
2. We use `gunicorn` to run Django app

   The Gunicorn "Green Unicorn" (pronounced jee-unicorn or gun-i-corn)[2] is a Python Web Server Gateway Interface (WSGI) HTTP server.

34.7 Environment Variables

Create *.env/.prod-sample*, which contains `env variables` for our `production` app

```
DEBUG=0
SECRET_KEY=dbaa1_i7%*3r9-=z-+_mz4r-!qeed@(-a_r(g@k8jo8y3r27%m
DJANGO_ALLOWED_HOSTS=*

SQL_ENGINE=django.db.backends.postgresql
SQL_DATABASE=react_wagtail_dev
SQL_USER=react_wagtail
SQL_PASSWORD=react_wagtail
SQL_HOST=db
SQL_PORT=5432

NEXT_PUBLIC_NEXT_BASE=https://nextjs-wagtail.accordbox.com
```

Here `NEXT_PUBLIC_NEXT_BASE` is the base URL of the Next.js project, please update it in your project.

Please make sure .env is not excluded in the .gitignore, so it can be added to Git repo

34.7.1 Config

Update *nextjs_wagtail_app/settings.py*:

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware',          # new
    ...
]

STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'

STATIC_ROOT = os.path.join(BASE_DIR, 'static')
STATIC_URL = '/static/'

MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

1. Add `whitenoise` to the `MIDDLEWARE`
2. set `STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'`

[82] https://docs.djangoproject.com/en/3.1/howto/static-files/deployment/

3. `STATIC_URL` and `MEDIA_URL` should match the above Nginx `location` config.

Add `gunicorn` and `whitenoise` to *requirements.txt*

```
gunicorn==20.1.0
whitenoise==5.3.0
```

Notes:

1. `whitenoise` can help serve static assets in Django app.

34.8 Test Build

Let's test the config on local env. (This can help us find problem)

```
# cleanup
$ docker-compose stop
$ docker-compose down

$ docker-compose ps
Name   Command   State   Ports
------------------------------
```

Command above can help us remove the `dev` docker containers, while keeping the `docker volumes`.

```
$ docker-compose -f docker-compose.prod.yml -p nextjs-wagtail-prod build
$ docker-compose -f docker-compose.prod.yml -p nextjs-wagtail-prod up -d
$ docker-compose -f docker-compose.prod.yml -p nextjs-wagtail-prod logs -f

web_1      | [2021-08-01 03:04:46 +0000] [15] [INFO] Starting gunicorn 20.1.0
web_1      | [2021-08-01 03:04:46 +0000] [15] [INFO] Listening at: http://0.0.0.0:8000 (15)
web_1      | [2021-08-01 03:04:46 +0000] [15] [INFO] Using worker: sync
web_1      | [2021-08-01 03:04:46 +0000] [17] [INFO] Booting worker with pid: 17
```

Notes:

1. We specify `docker compose file` by using `-f docker-compose.prod.yml` (please note default `docker-compose.yml` is for `dev` app)
2. `-p nextjs-wagtail-prod` specify the value prepended along with the service name. So the test would not ruin our local development env. You can check Docker doc[83] for more details.

Next, let's test on local env to check if the config is correct

```
# create superuser
$ docker-compose -f docker-compose.prod.yml -p nextjs-wagtail-prod exec web python manage.py
↪createsuperuser
```

Notes:

1. Visit https://api.local.test/cms-admin in browser, try to load images to see if everything is working.

```
# cleanup
$ docker-compose -f docker-compose.prod.yml -p nextjs-wagtail-prod stop
# delete containers and volumes
$ docker-compose -f docker-compose.prod.yml -p nextjs-wagtail-prod down -v
$ docker-compose -f docker-compose.prod.yml -p nextjs-wagtail-prod ps

Name   Command   State   Ports
------------------------------
```

[83] https://docs.docker.com/compose/reference/envvars/#compose_project_name

34.9 Config Traefik SSL

Let's update *compose/production/traefik/traefik.yml*

```
log:
  level: INFO

entryPoints:
  web:
    # http
    address: ":80"
    http:
      # https://docs.traefik.io/routing/entrypoints/#entrypoint
      redirections:
        entryPoint:
          to: web-secure

  web-secure:
    # https
    address: ":443"



certificatesResolvers:
  letsencrypt:
    # https://docs.traefik.io/master/https/acme/#lets-encrypt
    acme:
      email: "michaelyin@accordbox.com"
      storage: /etc/traefik/acme/acme.json
      # https://docs.traefik.io/master/https/acme/#httpchallenge
      httpChallenge:
        entryPoint: web


http:
  routers:
    web-media-router:
      rule: "Host(`nextjs-wagtail-api.accordbox.com`) && PathPrefix(`/media/`)"
      entryPoints:
        - web-secure
      middlewares:
        - csrf
      service: django-media
      tls:
        # https://docs.traefik.io/master/routing/routers/#certresolver
        certResolver: letsencrypt

    web-secure-router:
      rule: "Host(`nextjs-wagtail-api.accordbox.com`)"
      entryPoints:
        - web-secure
      middlewares:
        - csrf
      service: django
      tls:
        # https://docs.traefik.io/master/routing/routers/#certresolver
        certResolver: letsencrypt

  middlewares:
    csrf:
      # https://docs.traefik.io/master/middlewares/headers/#hostsproxyheaders
      # https://docs.djangoproject.com/en/dev/ref/csrf/#ajax
```

```
      headers:
        hostsProxyHeaders: ["X-CSRFToken"]

  services:
    django:
      loadBalancer:
        servers:
          - url: http://web:8000
    django-media:
      loadBalancer:
        servers:
          - url: http://nginx:80

providers:
  # https://docs.traefik.io/master/providers/file/
  file:
    filename: /etc/traefik/traefik.yml
    watch: true
```

1. Change host to `nextjs-wagtail-api.accordbox.com`, which is the URL of our live backend API.
2. Config `entryPoints` to redirect traffic from 80 to 443
3. Add `certificatesResolvers` section, add `tls` to the `routers`, change the `entryPoints` to `web-secure`so our Wagtail app can work with `HTTPS` automatically.
4. Please update email in the `certificatesResolvers` to your personal email.

34.10 Deploy to DigitalOcean

In this section:

1. `(local)$` means that the command should be run on your local environment
2. `(server)$` means that the command should be run on the remote server.

34.10.1 Server Setup

First, sign up for a DigitalOcean account[84] (if you don't already have one), and then generate[85] an API token so you can access the DigitalOcean API.

Add the token to your environment:

```
(local)$ export DIGITAL_OCEAN_ACCESS_TOKEN=[your_digital_ocean_token]
```

Next, create a Droplet with Docker pre-installed[86], you can copy shell code from that page and update the command:

```
# create Droplet
curl -X POST -H 'Content-Type: application/json' \
     -H 'Authorization: Bearer '$DIGITAL_OCEAN_ACCESS_TOKEN'' -d \
    '{"name":"nextjs-wagtail","region":"sfo2","size":"s-2vcpu-4gb","image":"docker-20-04"}' \
    "https://api.digitalocean.com/v2/droplets"

# check status
curl \
```

[84] https://m.do.co/c/b585bd8722ec
[85] https://www.digitalocean.com/docs/apis-clis/api/
[86] https://marketplace.digitalocean.com/apps/docker

```
    -H 'Content-Type: application/json' \
    -H 'Authorization: Bearer '$DIGITAL_OCEAN_ACCESS_TOKEN'' \
    "https://api.digitalocean.com/v2/droplets?nextjs-wagtail"
```

Notes:

1. We use `nextjs-wagtail` as the `Droplet` name, you can modify it.
2. When the `Droplet` is available, you should receive an email which contains the login credentials.

34.11 Config DNS

After the `Droplet` is created, let's create DNS records to point domain to the IP address of the server.

Here I config `nextjs-wagtail.accordbox.com` (you need change it) to point to the IP of the server, after waiting for some minutes, let's run test.

```
$ dig nextjs-wagtail-api.accordbox.com
```

34.11.1 Config SSH

Notes: This section assumes you have no experience with SSH, and it would help you get it work quickly.

```
(local)$ ssh root@<YOUR_INSTANCE_IP>

# type the root password in the email and set the new password
```

After you set the new password, generate an SSH key:

```
(server)$ ssh-keygen -t rsa

# press ENTER multiple times
```

This will generate a public and private key – `.ssh.id_rsa.pub` and `.ssh/id_rsa`, respectively.

Copy the above private key to your `system clipboard` and then set it as an environment variable on your local machine:

```
(server)$ cat ~/.ssh/id_rsa.pub > ~/.ssh/authorized_keys
(server)$ cat ~/.ssh/id_rsa

(local)$ export PRIVATE_KEY='-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAqy8065H+/bn6e0NbPdoKgl7BI8bCLwJ2W1goI6UfVKN/w40P
yVEu0QDJgvZuzLqBEvZkeookpvYotQ4TddfY2ksVf3svDXsd6NZClJ/e8LawwVoP
VXL9Pdbo8X7PtCmvdD/lvuhcg8iFhwJR8YqxeZhRvds5PzwIhYx9/n7f3y6goR0s
8J71z47xZs6phQD96o3dG692E8gUBbt525p08+ysOQBLbv8DTdv0xoCOkV83I2z1
...
-----END RSA PRIVATE KEY-----'
```

Add the key to your ssh-agent[87]:

```
(local)$ ssh-add - <<< "${PRIVATE_KEY}"

Identity added
```

To test, run:

[87] https://en.wikipedia.org/wiki/Ssh-agent

```
(local)$ ssh -o StrictHostKeyChecking=no root@$<YOUR_INSTANCE_IP> whoami

root
```

Notes:

1. You can save the SSH private key to `$HOME/.ssh/id_rsa` locally, so SSH still works after you restart your local machine. Github Doc[88]
2. To keep your server secure, if you can log in using the SSH private key, you should disable SSH password login.

34.11.2 Upload source code and Build Image

Before we start, please make sure to use `git commit` to commit your code.

Next, let's write a bash script to upload source code to DigitalOcean.

Create *compose/auto_deploy_do.sh*:

```
#! /bin/bash

# This shell script quickly deploys your project to your
# DigitalOcean Droplet

if [ -z "$DIGITAL_OCEAN_IP_ADDRESS" ]
then
    echo "DIGITAL_OCEAN_IP_ADDRESS not defined"
    exit 0
fi

# generate TAR file from git
git archive --format tar --output ./project.tar master

echo 'Uploading project...'
rsync ./project.tar root@$DIGITAL_OCEAN_IP_ADDRESS:/tmp/project.tar
echo 'Uploaded complete.'

echo 'Building image...'
ssh -o StrictHostKeyChecking=no root@$DIGITAL_OCEAN_IP_ADDRESS << 'ENDSSH'
    mkdir -p /app
    rm -rf /app/* && tar -xf /tmp/project.tar -C /app
    docker-compose -f /app/docker-compose.prod.yml build
ENDSSH
echo 'Build complete.'
```

Notes:

1. First, we export the latest version from the `master` branch to *project.tar*. (Please update if you use new `main` branch)
2. Then, we upload *project.tar* to the server, clean up the "/app" directory, and decompress the source code to "/app".
3. Finally, we re-build the Docker image.

```
(local)$ export DIGITAL_OCEAN_IP_ADDRESS=<YOUR_INSTANCE_IP>

(local)$ bash compose/auto_deploy_do.sh

Uploading project...
```

[88] https://docs.github.com/en/free-pro-team@latest/github/authenticating-to-github/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent

```
Uploaded complete.
Building image...

...

Build complete.
```

Now let's test on the server:

```
(local)$ ssh root@<YOUR_INSTANCE_IP>

(server)$ cd /app
(server)$ docker-compose -f docker-compose.prod.yml up
```

Now you can visit https://nextjs-wagtail-api.accordbox.com/cms-admin to see if the Wagtail admin is up and running. Press `Ctrl+c` to stop the running containers.

```
# check
(server)$ docker-compose -f docker-compose.prod.yml ps
```

Next, rather than manually running the containers, let's let `Supervisor` handle this for us.

34.11.3 Process Manager

Start by installing Supervisor:

```
(server)$ apt-get update
(server)$ apt-get install -y supervisor
```

Next, add the following config to */etc/supervisor/conf.d/nextjs-wagtail.conf*:

```
[program:nextjs-wagtail]
directory=/app
command=docker-compose -f docker-compose.prod.yml up
autostart=true
autorestart=true
```

Restart:

```
(server)$ supervisorctl

supervisor> reload
Really restart the remote supervisord process y/N? y
Restarted supervisord

supervisor> status
nextjs-wagtail                   STARTING
```

Now the containers should automatically run on boot. We can also restart the containers via the `supervisorctl restart nextjs-wagtail` command. Let's run the command after the image is built.

```
#! /bin/bash

# This shell script quickly deploys your project to your
# DigitalOcean Droplet

if [ -z "$DIGITAL_OCEAN_IP_ADDRESS" ]
then
    echo "DIGITAL_OCEAN_IP_ADDRESS not defined"
    exit 0
```

```
fi

# generate TAR file from git
git archive --format tar --output ./project.tar master

echo 'Uploading project...'
rsync ./project.tar root@$DIGITAL_OCEAN_IP_ADDRESS:/tmp/project.tar
echo 'Uploaded complete.'

echo 'Building image...'
ssh -o StrictHostKeyChecking=no root@$DIGITAL_OCEAN_IP_ADDRESS << 'ENDSSH'
    mkdir -p /app
    rm -rf /app/* && tar -xf /tmp/project.tar -C /app
    docker-compose -f /app/docker-compose.prod.yml build
    supervisorctl restart nextjs-wagtail
ENDSSH
echo 'Build complete.'
```

Now, after the new images are built, `supervisorctl` is used to restart the containers:

```
(local)$ bash compose/auto_deploy_do.sh

Uploading project...
Uploaded complete.
Building image...

...

nextjs-wagtail: stopped
nextjs-wagtail: started
Build complete.
```

34.12 Config site

Now, let's create admin login credential

```
(local)$ ssh root@<YOUR_INSTANCE_IP>

(server)$ cd /app
(server)$ docker-compose -f docker-compose.prod.yml exec web python manage.py createsuperuser
```

After you are done, you can use the login credential to login Wagtail admin https://nextjs-wagtail-api.accordbox.com/cms-admin/ and setup your site, upload images and create pages. (`BlogPage`, `PostPage`)

Please remember to config `Wagtail site` to make the generated url from REST API has the correct hostname and port.

Chapter 35

Deploy Next.js to Netlify

35.1 Objective

1. We will learn how to run Next.js in production mode.
2. We will deploy Next.js to the Netlify platform.

The Next.js project will serve on `https://nextjs-wagtail.accordbox.com`

35.2 Yarn Build

```
$ docker-compose up -d
$ docker-compose logs -f
```

Create *frontend/.env.production*

```
NEXT_PUBLIC_WAGTAIL_API_BASE=http://api.local.test:8000
NEXT_PUBLIC_NEXT_BASE=http://www.local.test:3000
```

```
$ cd frontend
$ yarn build

Page                                Size     First Load JS
┌ ○ /                               899 B          70.1 kB
├   /_app                           0 B            65.3 kB
├ ○ /_preview                       905 B          70.1 kB
├ ○ /[...path]                      884 B            70 kB
├   ├ /postpage1
├   ├ /postpage2
├   ├ /postpage3
├   └ [+3 more paths]
├ ○ /404                            3.18 kB        68.5 kB
├ λ /api/hello                      0 B            65.3 kB
├ λ /api/preview                    0 B            65.3 kB
├ ○ /category/[...path]             910 B          70.1 kB
├   ├ /category/programming/page-1
├   ├ /category/programming
├   ├ /category/life/page-1
├   └ /category/life
├ ○ /search                         19.4 kB        84.7 kB
└ ○ /tag/[...path]                  908 B          70.1 kB
    ├ /tag/django/page-1
```

```
    ├ /tag/django
    ├ /tag/wagtail/page-1
    └ [+3 more paths]
+ First Load JS shared by all       65.3 kB
  ├ chunks/framework.1eefeb.js      42.6 kB
  ├ chunks/main.655ad0.js           20.3 kB
  ├ chunks/pages/_app.ae7c21.js     576 B
  ├ chunks/webpack.fdfc2c.js        1.85 kB
  └ css/d3c9c7e1900a22a104af.css    24.8 kB

λ  (Server)  server-side renders at runtime (uses getInitialProps or getServerSideProps)
□  (Static)  automatically rendered as static HTML (uses no initial props)
●  (SSG)     automatically generated as static HTML + JSON (uses getStaticProps)
   (ISR)     incremental static regeneration (uses revalidate in getStaticProps)
```

If we check the build files

```
$ tree .next/server/pages

.next/server/pages
├── 404.html
├── 500.html
├── [...path].js
├── _app.js
├── _document.js
├── _error.js
├── _preview.html
├── _preview.js
├── _preview.json
├── api
│   ├── hello.js
│   └── preview.js
├── category
│   ├── [...path].js
│   ├── life
│   │   ├── page-1.html
│   │   └── page-1.json
│   ├── life.html
│   ├── life.json
│   ├── programming
│   │   ├── page-1.html
│   │   └── page-1.json
│   ├── programming.html
│   └── programming.json
├── index.html
├── index.js
├── index.json
├── page-1.html
├── page-1.json
├── page-2.html
├── page-2.json
├── postpage1.html
├── postpage1.json
├── postpage2.html
├── postpage2.json
├── postpage3.html
├── postpage3.json
├── postpage4.html
├── postpage4.json
├── search.html
└── tag
    ├── [...path].js
    ├── django
```

```
    │   ├── page-1.html
    │   └── page-1.json
    ├── django.html
    ├── django.json
    ├── react
    │   ├── page-1.html
    │   └── page-1.json
    ├── react.html
    ├── react.json
    ├── wagtail
    │   ├── page-1.html
    │   └── page-1.json
    ├── wagtail.html
    └── wagtail.json
```

Notes:

1. `yarn build` will generate all pages to static HTML files.

Let's run Node.js server

```
$ yarn start

yarn run v1.22.10
$ next start
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
```

Now visit http://www.local.test:3000/ to test.

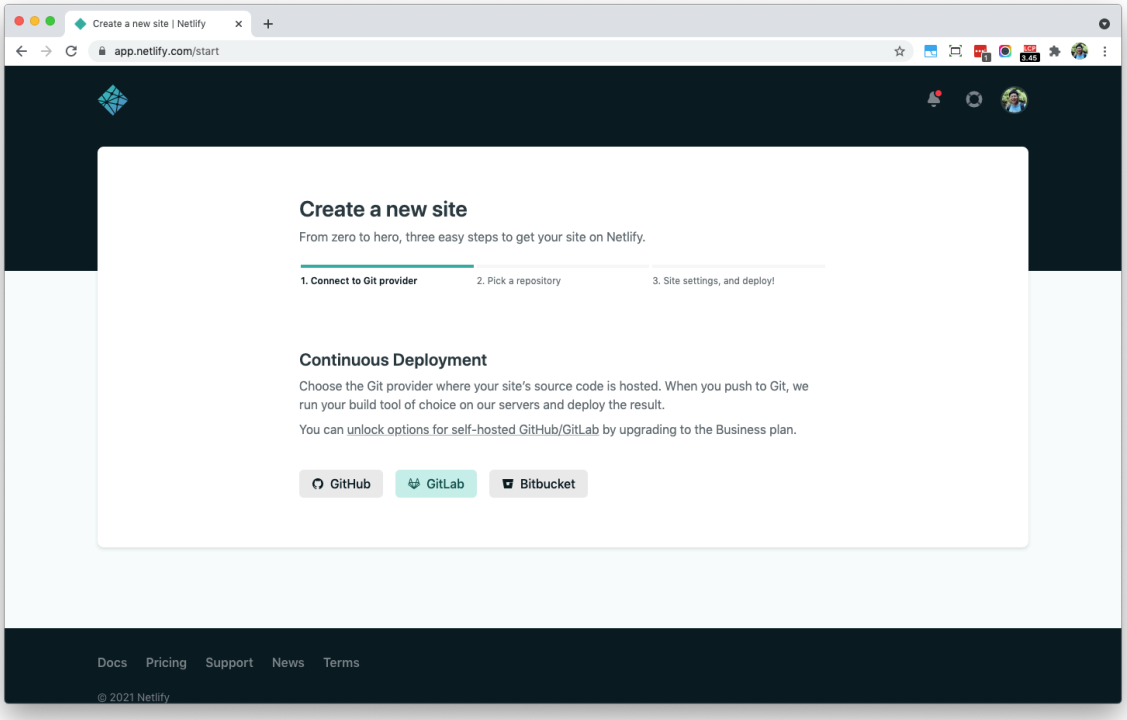
If everything works fine, press `Ctrl+C` to terminate the Node.js server.

35.3 Create Netlify Site

First, please signup Netlify[89]

And then, when we see the dashboard, click `New site from Git` button

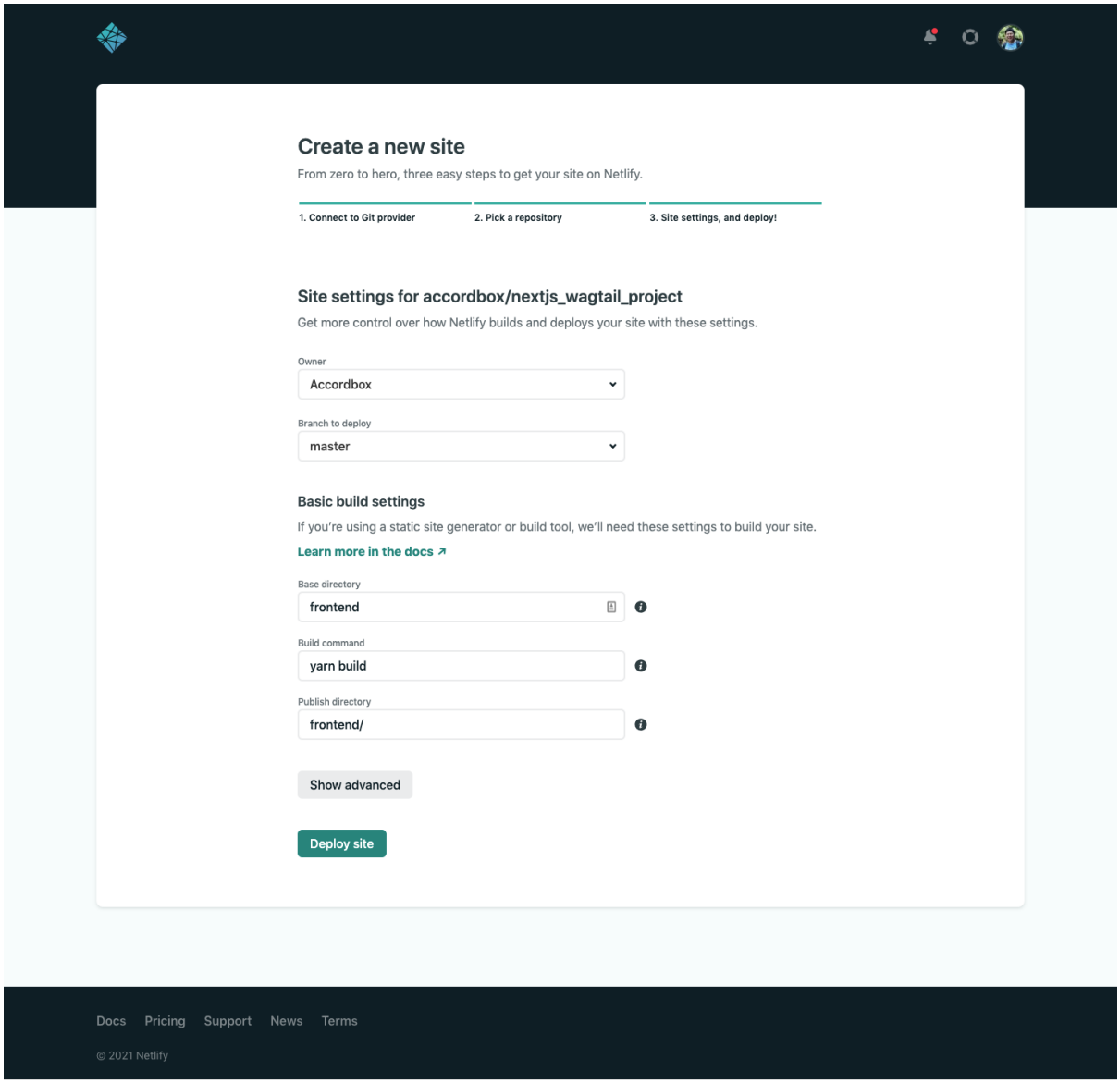
[89] https://app.netlify.com/signup



Notes:

1. Here we can click the button to connect Github, Gitlab or Bitbucket account with Netlify account.

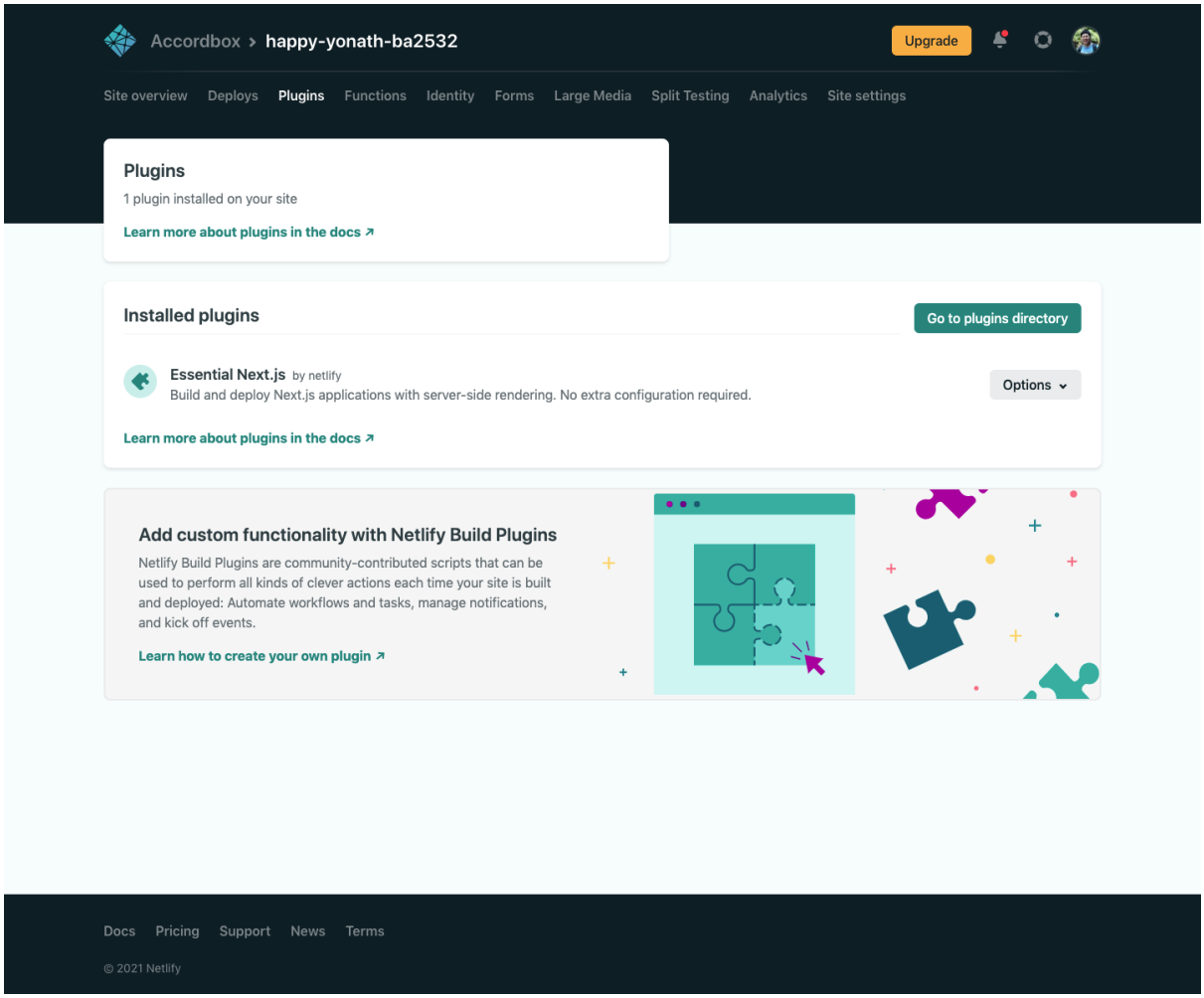
Next, config the Netlify site



1. We config Netlify to deploy the `master` branch of our Git repo. If you are using `main` branch, please change the value.
2. Since the Next.js project is placed in `frontend` directory in this project, so we set `base directory` to `frontend`

35.4 Activate NextJS plugin

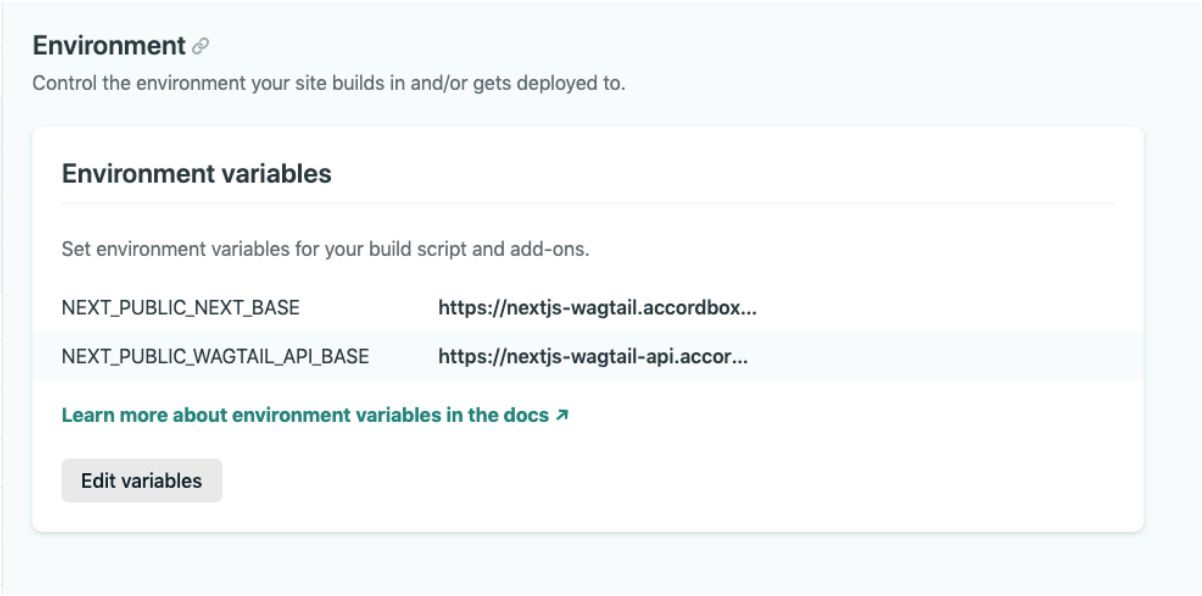
Go to `plugins` tab, enable `Essential Next.js` Plugin



35.5 ENV variable

Go to `Site Settings / Build & Deploy / Environment` to set production env for the Next.js project.

```
NEXT_PUBLIC_NEXT_BASE=https://nextjs-wagtail.accordbox.com
NEXT_PUBLIC_WAGTAIL_API_BASE=https://nextjs-wagtail-api.accordbox.com
```

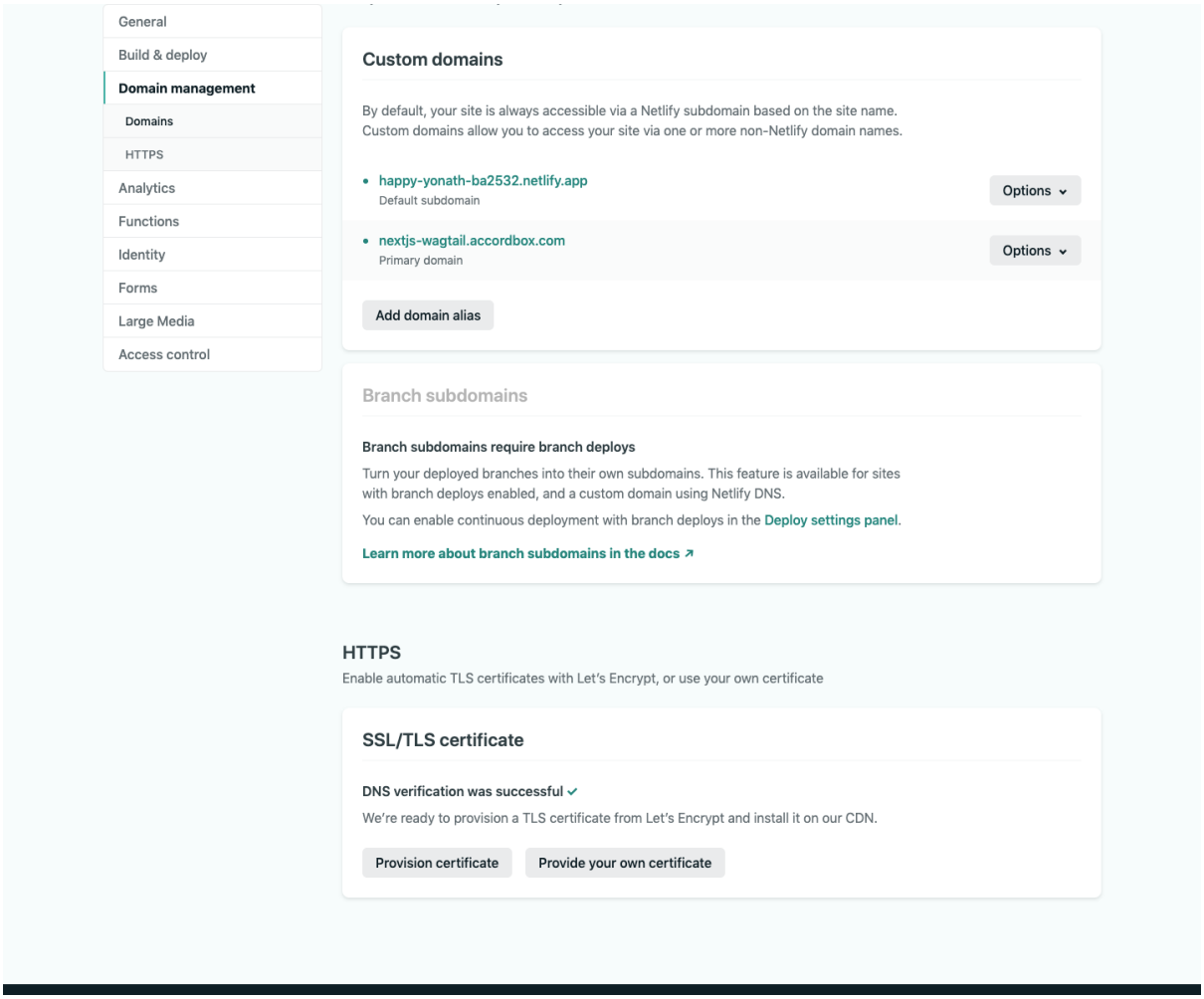


35.6 Custom Domain

In the `Site Settings / Domain Management`, click `Add custom domain` button to add `nextjs-wagtail.accordbox.com` with our Netlify site.

Please follow the instruction to add `CNAME` record in our domain manager.

Do not forget to setup `SSL/TLS certificate` to enable the HTTPS on the `Domain Management` page.



35.7 Deploy

Now go to `deploys` tab, click `Trigger Deploy / Deploy Site` to check if everything works.

35.8 Manual Test

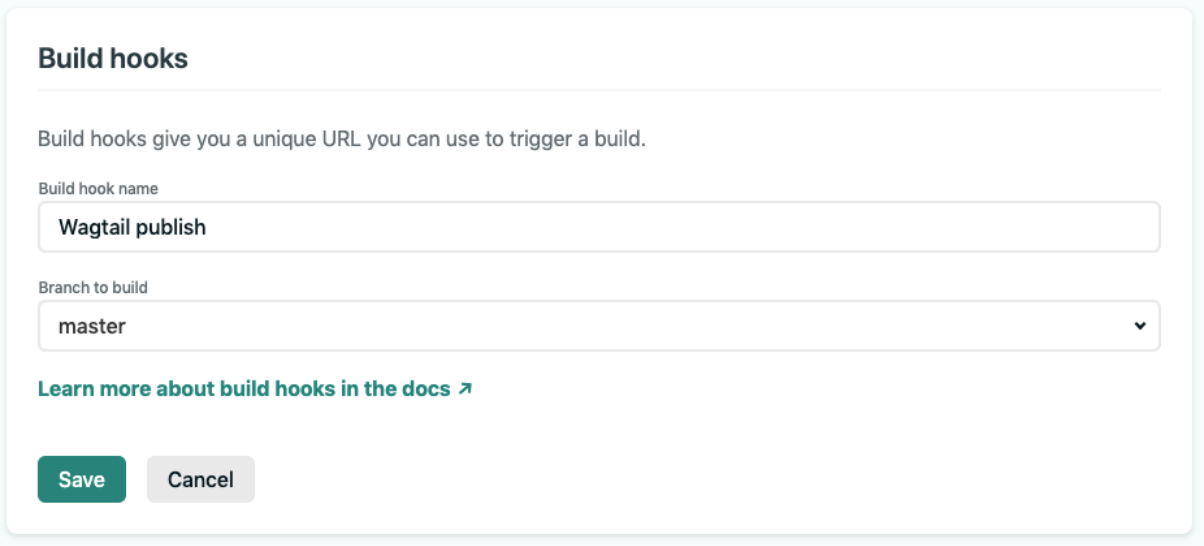
Now, let's go to https://nextjs-wagtail.accordbox.com/ to check relevant functions

35.9 Build on page publish

Next, we will add some code, to make `netlify` auto rebuild if we publish or un-publish pages in Wagtail admin.

35.9.1 Build Hook

On Netlify, go to `Site settings > Build & deploy > Continuous deployment > Build hooks` to add new build hook



Now the build hook will look something like `https://api.netlify.com/build_hooks/xxxxxxxxxxxxx`

Update *.env/.prod-sample*

```
NETLIFY_BUILD_HOOK=https://api.netlify.com/build_hooks/xxxxxxxxxxxxx
```

Update *nextjs_wagtail_app/settings.py*

```
NETLIFY_BUILD_HOOK = os.environ.get("NETLIFY_BUILD_HOOK", "")
```

35.9.2 Signal handler

Create *blog/receivers.py*

```
import requests
from django.apps import apps
from django.conf import settings

from wagtail.core.signals import page_published, page_unpublished


def nextjs_rebuild():
    if not settings.DEBUG and settings.NETLIFY_BUILD_HOOK:
        resp = requests.post(settings.NETLIFY_BUILD_HOOK)
        resp.raise_for_status()


def page_published_signal_handler(instance, **kwargs):
    nextjs_rebuild()


def page_unpublished_signal_handler(instance, **kwargs):
    nextjs_rebuild()


def register_signal_handlers():
    # Get list of models that are page types
    Page = apps.get_model('wagtailcore', 'Page')
    indexed_models = [model for model in apps.get_models() if issubclass(model, Page)]

    # Loop through list and register signal handlers for each one
    for model in indexed_models:
```

```
        page_published.connect(page_published_signal_handler, sender=model)
        page_unpublished.connect(page_unpublished_signal_handler, sender=model)
```

Notes:

1. We use `page_published.connect` and `page_unpublished.connect` to attach signal handler.
2. If any Wagtail page is publish or un-publish, `nextjs_rebuild` will be called.

Update *blog/apps.py*

```
from django.apps import AppConfig


class BlogConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'blog'

    def ready(self):
        from blog.receivers import register_signal_handlers
        register_signal_handlers()
```

Update *nextjs_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    ...

    'django_comments',
    'django.contrib.sites',
    'custom_comments.apps.CustomCommentsConfig',

    'blog.apps.BlogConfig',            # new
    'nextjs',
]
```

Notes:

1. Commit the above code to git repo.
2. Run `bash compose/auto_deploy_do.sh` to deploy to the backend server.
3. If we publish pages on Wagtail admin, Netlify will rebuild the site automatically.

35.10 Reference

https://www.netlify.com/blog/2020/11/30/how-to-deploy-next.js-sites-to-netlify/

Chapter 36

Next Steps

Congratulations on making it through. I hope you have learned a lot in this book.

Below are some things I wish you can check for next steps:

1. **Wagtail-Pipit**: Wagtail-Pipit[90] is a boilerplate which built with Next.js and Wagtail.
2. **UnlyEd/next-right-now**: https://github.com/UnlyEd/next-right-now is a boilerplate with Next.js 11, Vercel and TypeScript.
3. **django-storages**: Check django-storages[91].
4. **django-debug-toolbar**: Try to use django-debug-toolbar[92] to help you debug.
5. **django-silk**: django-silk[93] can help you find performance issue of your project.
6. **Storybook**: Try to use storybook[94] to help build React component in isolated env.
7. **React Context**: Check React Context[95]
8. **Type Checking**: Check Typechecking With PropTypes[96] and Static Type Checking[97]

If you want your project has good code style:

1. prettier[98] for Javascript.
2. black[99], isort[100] and flake8[101] for Python.
3. pre-commit[102] to check code style while `git commit`

36.1 Thank You

Thank you spending time reading my book.

— Michael Yin

[90] https://github.com/Frojd/Wagtail-Pipit
[91] https://github.com/jschneier/django-storages
[92] https://github.com/jazzband/django-debug-toolbar
[93] https://github.com/jazzband/django-silk
[94] https://storybook.js.org/
[95] https://reactjs.org/docs/context.html
[96] https://reactjs.org/docs/typechecking-with-proptypes.html
[97] https://reactjs.org/docs/static-type-checking.html
[98] https://prettier.io/
[99] https://black.readthedocs.io/
[100] https://github.com/PyCQA/isort
[101] https://flake8.pycqa.org/
[102] https://pre-commit.com/

Chapter 37

Backend FAQ

37.1 Troubleshoot

If you run into problems, you can view the logs at:

```
$ docker-compose logs -f
```

Sometimes, you may want to remove the docker-compose app to start over again.

```
# stop and remove containers, networks, images
$ docker-compose down
```

If you want to also remove the data in `docker volume` (db data in this project)

```
# stop and remove containers, networks, images, volume
$ docker-compose down -v
```

37.2 Useful Commands

To enter the shell of a container that's up and running, run the following command:

```
$ docker-compose exec <service-name> bash

# for example:
# docker-compose exec web bash
```

If you want to run a command against a new container that's not currently running, run:

```
$ docker-compose run --rm web bash
```

> The `--rm` option tells docker to delete the container after you exit the shell.

To stop the docker compose application

```
$ docker-compose run --rm web bash
```