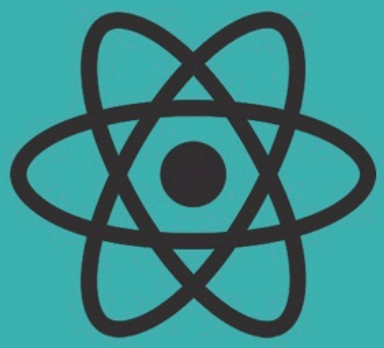


Build SPA *with*

React

Wagtail

MichaelYin@Accordbox

Contents

Chapter 1

Introduction

1.1 Objectives

This course will teach you how to build a `SPA` (single-page application) using `React` and `Wagtail CMS`. By the end of this course, you will be able to:

1. Understand `Docker` and use `Docker Compose` to do development
2. Build a REST API for Wagtail CMS
3. Use the Django shell to test code and check data.
4. Test the REST API and generate test coverage report
5. Use the `factory` package to help create test data
6. Build a React app from `create-react-app`
7. Understand React Components and the component lifecycle
8. Understand React router
9. Use `Storybook` to develop React Components
10. Test React components and the frontend app
11. Make `React app` work with Wagtail preview
12. Deploy the production app to DigitalOcean

1.2 Who is this course for

1. People who need a CMS + SPA solution.
2. Backend developers who want to learn modern frontend technologies.
3. Frontend developers who wish to learn Wagtail and Django.

1.3 What is included

1. A PDF ebook which contains about 30 chapters.
2. 10+ screenshots and 7 diagrams, all created by me.

3. The source code is available on Github/wagtail-react-blog[1]

1.4 How to use the source code

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[2]

```
$ git clone https://github.com/AccordBox/wagtail-react-blog react_wagtail
$ cd react_wagtail
$ 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://127.0.0.1:3000
- http://127.0.0.1:6006
- http://127.0.0.1:8000/cms-admin

1.5 Demo

The demo is also online if you want to check.

- React app Demo[3]
- Storybook Demo[4]
- Wagtail Demo[5]

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

- 2020-12-05: First release
- 2020-11-28: Review done

[1] https://github.com/AccordBox/wagtail-react-blog
[2] https://docs.docker.com/get-docker/
[3] http://react-wagtail.accordbox.com
[4] http://react-wagtail-storybook.accordbox.com
[5] http://react-wagtail-api.accordbox.com/cms-admin

- 2020-11-02: Draft finished
- 2020-07-25: Start writing

Chapter 2

Setup project

2.1 Objectives

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

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

2.2 Create Django Project

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

You can also use other tool such as Poetry[6] or Pipenv[7]

Create *requirements.txt*

```
django==3.1
```

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

You will see structure like this

```
(env)$ app tree -L 1
.
├── env
├── manage.py
└── react_wagtail_app

2 directories, 1 file
```

2.3 Import Wagtail

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

[6] https://python-poetry.org/
[7] https://pipenv.pypa.io/

```
wagtail==2.10.2
```

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

Add the apps to `INSTALLED_APPS` in the *react_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",
    "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 *react_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",
]
```

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

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

WAGTAIL_SITE_NAME = 'My Project'
```

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

```
from django.contrib import admin
from django.urls import path, include, re_path
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
    re_path(r'', 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. Remember to put `re_path(r'', 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
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
October 10, 2020 - 02:25:27
Django version 3.1, using settings 'react_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[8]
3. You will see `db.sqlite3` is created at the project directory, here we did not specify Django to use other db so default sqlite is used by default.

2.5 Reference

Integrating Wagtail into a Django project[9]

[8] https://github.com/wagtail/wagtail/blob/v2.10.2/wagtail/core/migrations/0002_initial_data.py#L30
[9] 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[10] if you haven't already done so.

```
$ docker --version
Docker version 18.09.2, build 6247962

$ docker-compose --version
docker-compose version 1.23.2, build 1110ad01
```

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
```

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

```
├── react_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[11], which is a great project for people who want to learn Django.

3.4 Compose file

Note: we can ignore the `frontend` and `storybook` here.

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
    image: react_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:
```

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

```
      - postgres_data:/var/lib/postgresql/data/
    environment:
      - POSTGRES_DB=react_wagtail_dev
      - POSTGRES_USER=react_wagtail
      - POSTGRES_PASSWORD=react_wagtail

volumes:
  postgres_data:
```

Notes:

1. Here we defined two services, one is `web` (django devserver), 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=react_wagtail_dev
SQL_USER=react_wagtail
SQL_PASSWORD=react_wagtail
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` *react_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 \
  # 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 volumn `.:/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[12])

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 "$@"
```

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[13].

3.7 Start script

Now, let's add start script.

compose/local/django/start

```
#!/bin/bash
```

[12] https://github.com/docker-library/postgres/issues/146
[13] https://stackoverflow.com/a/39082923/2371995

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

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

3.8 Start application

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

```
django==3.1
wagtail==2.10.2
psycopg2-binary
```

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 3.1, using settings 'react_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.

```
$  docker-compose ps
  Name                 Command              State           Ports
-------------------------------------------------------------------------
app_db_1    docker-entrypoint.sh postgres   Up      5432/tcp
app_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
```

```
├── react_wagtail_app
└── requirements.txt
```

Add `blog` to the `INSTALLED_APPS` of *react_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[14] 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[15] 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"),
```

[14] https://docs.djangoproject.com/en/3.1/topics/db/models/#proxy-models
[15] 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[16] 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 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 (
```

[16] 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

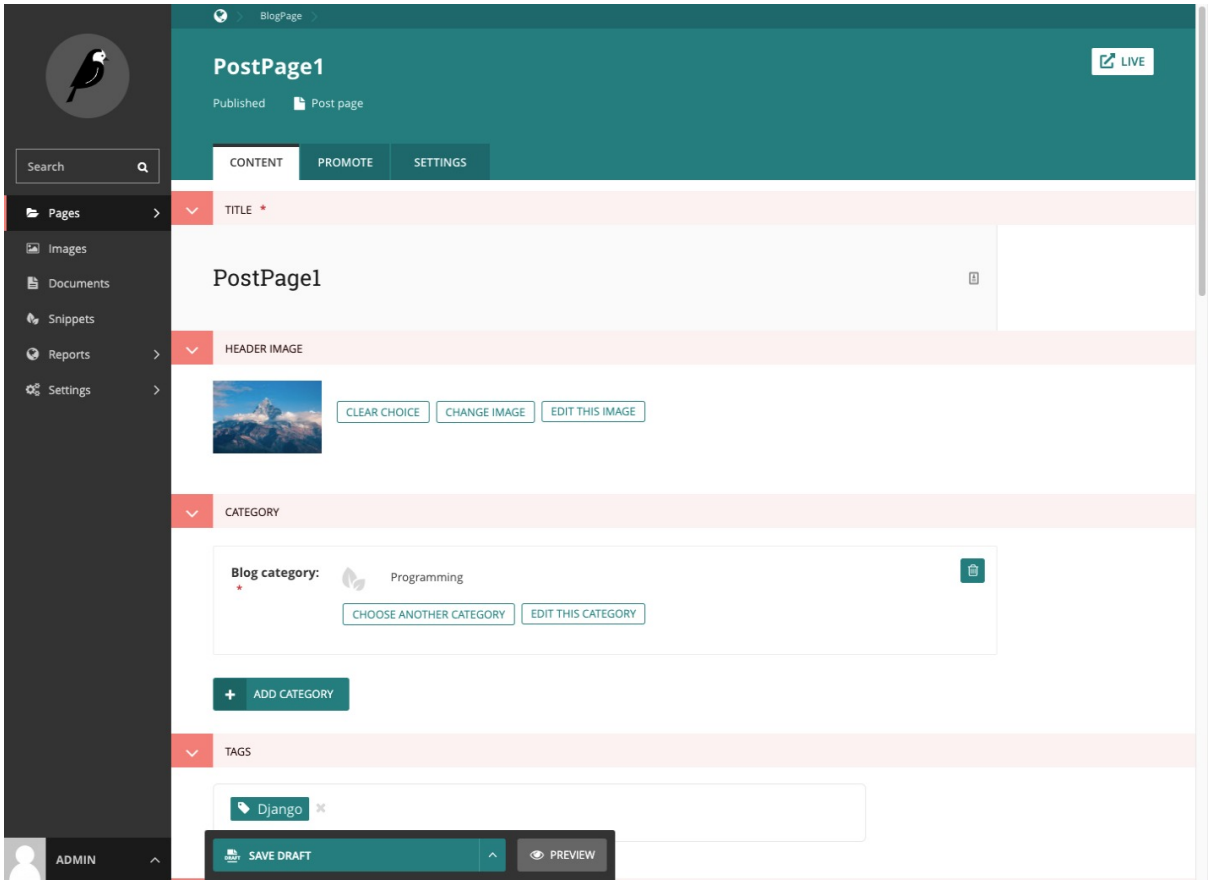
```
# create superuser and password
$ docker-compose run --rm web python manage.py createsuperuser

$ 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 cretaed.
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 visists http://127.0.0.1:8000/ you will see `TemplateDoesNotExist` exception. This is correct and we will fix it later, do not worry.

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

Many pepole have not much exprieence on Django when they reach Wagtail. So 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 whch support `preview`, 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)

Category: category 1 (pk=1)

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

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 plan?

1. You need to create something like `PostPageCategory (blog_category=2, page=1)` in `memory` and not write it to `PostPageCategory` table. (Because if you do, it will affect the `live` page)
2. You need to write code to convert page page data, and the above `PostPageCategory` 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.

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

That is why django-modelcluster[17] is created and `ParentalKey` is introduced.

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[18]
2. And define the `PostPageCategory.page` as `ParentalKey` field.
3. So the page (ClusterableModel) can hold the `PostPageCategory` in memory even the data is not created in db.
4. We can then serialize the page to JSON format and save to `revision` table.
5. Now editor can preview the page before publishing it.

4.12.1 Tip

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` relashiship with `Page` in `other` class, for example, `PostPageBlogCategory.page`, use `ParentalKey`.

[17] https://github.com/wagtail/django-modelcluster
[18] 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 block can also contains 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[19] 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.

[19] https://docs.djangoproject.com/en/3.1/ref/models/fields/#field-types

blog/blocks.py

```
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*

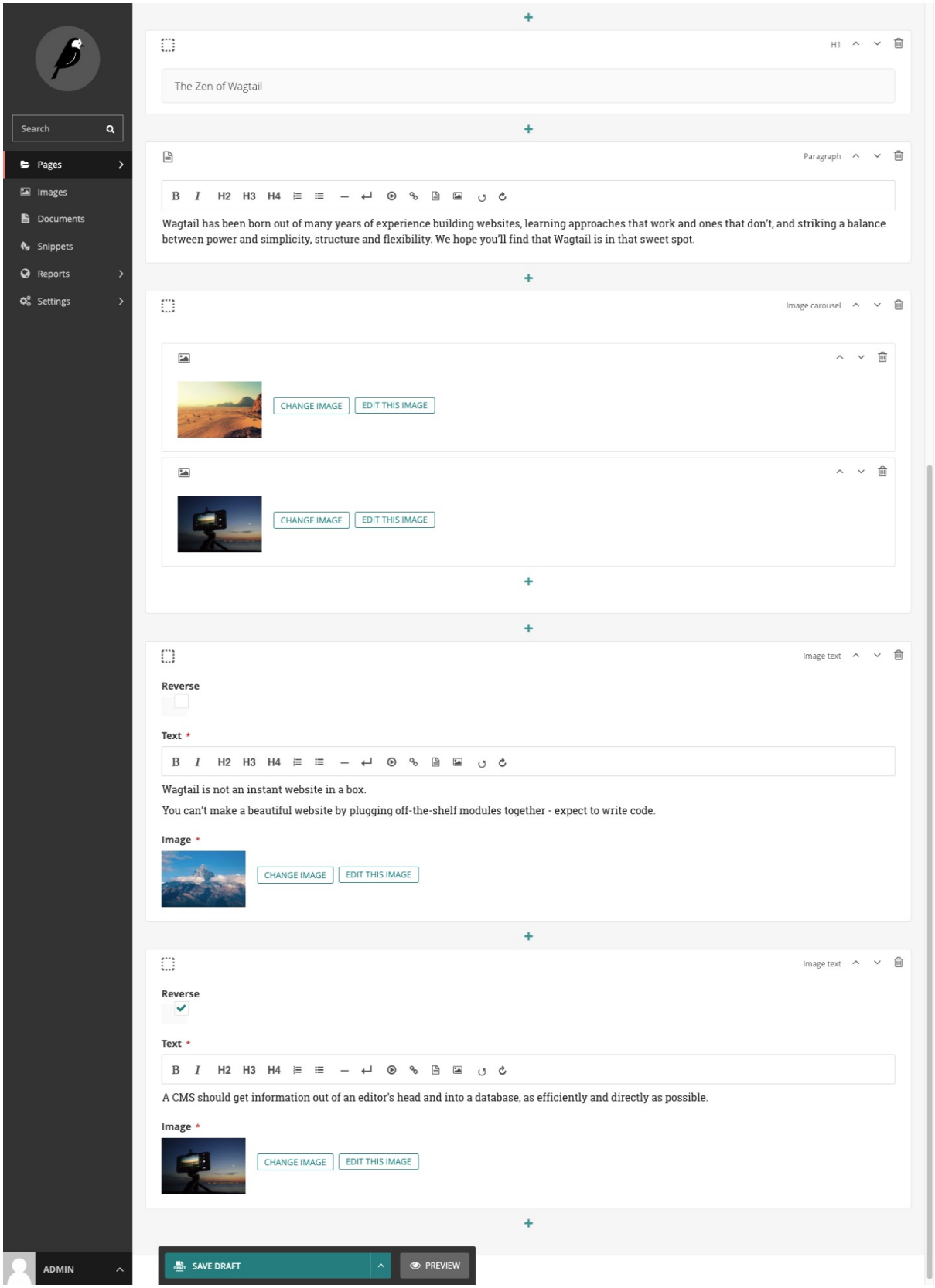
```
from wagtail.core.fields import StreamField
from .blocks import BodyBlock


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)

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

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

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.stream_data
[{'type': 'h1', 'value': 'The Zen of Wagtail', 'id': '0dd3e943-4cbc-4c13-94ce-423c91ab9800'}, {'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': 'c111ca2d-a55f-4956-81ce-ec5cde9a4bb0'}, {'type': 'image_carousel', 'value': [3, 2],
↪'id': 'd3cdcc2b-02c9-4a8d-96e2-fa4102c9bb82'}, {'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':
↪'05b07aa7-be47-477b-aba1-a77d61243e6b'}, {'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': 'a45251b7-c9b3-4a60-9fe7-ac6fc225ad6f'}]

# let's make the data look more clear
>>> import pprint
>>> pprint.pprint(page.body.stream_data)
[{'id': '0dd3e943-4cbc-4c13-94ce-423c91ab9800',
  'type': 'h1',
  'value': 'The Zen of Wagtail'},
 {'id': 'c111ca2d-a55f-4956-81ce-ec5cde9a4bb0',
  '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': 'd3cdcc2b-02c9-4a8d-96e2-fa4102c9bb82',
  'type': 'image_carousel',
  'value': [3, 2]},
 {'id': '05b07aa7-be47-477b-aba1-a77d61243e6b',
  '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': 'a45251b7-c9b3-4a60-9fe7-ac6fc225ad6f',
  '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.

I also recommend you to run some code on your local env and check the data. This can help better understand the data structures of SteramField.

Chapter 6

Build REST API with Django REST Framework

6.1 Objectives

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

1. Understand what is REST API
2. Use `Django REST Framework` (DRF) to add REST API to Django project.
3. Make Wagtail page models support REST API by setting `api_fields`

6.2 Django REST Framework

Django REST Framework[20] (`DRF`) is a powerful and flexible tool for building Web APIs for Django project.

There are some basic concepts in `DRF`

1. routers[21], similar with Django urls.
2. viewsets[22], similar with Django view, which handle request and return response.
3. serializer[23], convert Django model instances to JSON, XML or vice versa.

6.3 Install Django REST Framework

requirements.txt

```
django==3.1
wagtail==2.10.2
psycopg2-binary
djangorestframework
```

Note: `djangorestframework` is also dependency package of Wagtail CMS, but is is good manner to add it here.

Update `INSTALLED_APPS` of *eact_wagtail_app/settings.py*

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

```
INSTALLED_APPS = [
    # code omitted for brevity

    "rest_framework",

    "blog",
]
```

6.4 Serializer

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

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 control the serialization behavior

6.5 Serializer Field

If we use the above `PostPageSerializer` get JSON string of the `PostPage`, we would get `id`, `slug` and `title`.

```
>>> 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",
        )
```

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>}
```

So now we see a problem, we need to tell `serializers` `how to process the data with other Django model`.

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
from .models import BlogCategory, PostPage, Tag


class PostPageSerializer(serializers.ModelSerializer):
    api_tags = TagField(source="tags")

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

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

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', 'api_tags': [{'name': 'Django', 'slug': 'django
↪', 'id': 1}]}
```

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

6.6 Viewsets & Router

Next, let's add `viewsets` to the `blog/views.py`

```
from rest_framework import viewsets

from .models import BlogCategory, PostPage, Tag
from .serializers import CategorySerializer, PostPageSerializer, TagSerializer


class PostPageSet(viewsets.ModelViewSet):
    serializer_class = PostPageSerializer
    queryset = PostPage.objects.all()
    http_method_names = ["get"]
```

```
class CategorySet(viewsets.ModelViewSet):
    queryset = BlogCategory.objects.all()
    serializer_class = CategorySerializer
    http_method_names = ["get"]


class TagSet(viewsets.ModelViewSet):
    queryset = Tag.objects.all()
    serializer_class = TagSerializer
    http_method_names = ["get"]
```

Create *blog/api.py*

```
from rest_framework import routers

from blog.views import CategorySet, PostPageSet, TagSet

# Below is custom router which has some advanced feature not implemented by Wagtail
blog_router = routers.DefaultRouter()
blog_router.register(r"posts", PostPageSet)
blog_router.register(r"categories", CategorySet)
blog_router.register(r"tags", TagSet)
```

Update *react_wagtail_app/urls.py*

```
from blog.api import blog_router


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

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

    path('api/blog/', include(blog_router.urls)),

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

Notes

1. `blog_router` is working on prefix `api/blog`
2. `blog_router` registered three viewsets (`posts`, `categories`, `tags`)

The three API URL would look like this, you can check the URL in your browser.

1. http://127.0.0.1:8000/api/blog/posts/
2. http://127.0.0.1:8000/api/blog/categories/
3. http://127.0.0.1:8000/api/blog/tags/

Or you can test in Django shell

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

```
>>> import requests
>>> requests.get('http://web:8000/api/blog/posts/').json()
[{'id': 4, 'slug': 'postpage1', 'title': 'PostPage1', 'api_tags': [{'name': 'Django', 'slug':
↪'django', 'id': 1}]}]
```

```
>>> requests.get('http://web:8000/api/blog/categories/').json()
[{'id': 1, 'slug': 'programming', 'name': 'Programming'}]
```

Note: because the django devserver is running on `web` container, so here the hostname is `web:8000` instead of `127.0.0.1:8000`

6.7 Filter

Next, I would add the filter function to the `PostPageSet`, so we can filter blog posts by `category` and `tag`

```
class PostPageSet(viewsets.ModelViewSet):
    serializer_class = PostPageSerializer
    queryset = PostPage.objects.all()
    http_method_names = ["get"]

    def get_queryset(self):
        queryset = PostPage.objects.all()
        category = self.request.query_params.get("category", None)
        tag = self.request.query_params.get("tag", None)
        if category is not None and category != "*":
            queryset = queryset.filter(categories__blog_category__slug=category)
        if tag is not None and tag != "*":
            queryset = queryset.filter(tags__slug=tag)
        return queryset
```

You can test in your browser

1. http://127.0.0.1:8000/api/blog/posts/?tag=djangoa
2. http://127.0.0.1:8000/api/blog/posts/?tag=*
3. http://127.0.0.1:8000/api/blog/posts/?tag=django

If you do not understand the __ in the ORM query, check Django Doc[24]

6.8 Pagination

To make our REST API support pagination by default, let's update *react_wagtail_app/settings.py*

```
REST_FRAMEWORK = {
    "DEFAULT_PAGINATION_CLASS": "rest_framework.pagination.LimitOffsetPagination",
    "PAGE_SIZE": 20,
}
```

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

```
>>> import requests, pprint
>>> requests.get('http://web:8000/api/blog/categories/').json()
>>> pprint.pprint(requests.get('http://web:8000/api/blog/categories/').json())
{'count': 1,
 'next': None,
 'previous': None,
 'results': [{'id': 1, 'name': 'Programming', 'slug': 'programming'}]}
```

Notes:

[24] https://docs.djangoproject.com/en/3.1/topics/db/queries/#lookups-that-span-relationships

1. All data has been moved to `results`
2. And `count`, `next` and `previous` contains the pagination info.

6.9 Conclusion

Now we have three api endpoints

1. http://127.0.0.1:8000/api/blog/posts/ which is to list blog posts, support filter function
2. http://127.0.0.1:8000/api/blog/categories/ which is to list blog categories.
3. http://127.0.0.1:8000/api/blog/tags/ which is to list blog categories.

Chapter 7

Build REST API for Wagtail Page

7.1 Objectives

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

1. Understand how to config REST API for Wagtail page models
2. Learn how to improve REST API representation for StreamField

7.2 Config

As I said, Wagtail's has already supported REST API based on Django REST Framework[25] (`DRF`)

Add `wagtail.api.v2` to `INSTALLED_APPS` of the *react_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    # code omitted for brevity

    "wagtail.api.v2",
    "rest_framework",

    "blog",
]
```

Update *blog/api.py* to add router `cms_api_router`

```
from rest_framework import routers
from wagtail.api.v2.router import WagtailAPIRouter
from wagtail.api.v2.views import PagesAPIViewSet
from wagtail.documents.api.v2.views import DocumentsAPIViewSet
from wagtail.images.api.v2.views import ImagesAPIViewSet

from .views import CategorySet, PostPageSet, TagSet

cms_api_router = WagtailAPIRouter("wagtailapi")

# Add the three endpoints using the "register_endpoint" method.
# The first parameter is the name of the endpoint (eg. pages, images). This
# is used in the URL of the endpoint
# The second parameter is the endpoint class that handles the requests
cms_api_router.register_endpoint("pages", PagesAPIViewSet)
cms_api_router.register_endpoint("images", ImagesAPIViewSet)
```

[25] https://www.django-rest-framework.org/

```
cms_api_router.register_endpoint("documents", DocumentsAPIViewSet)

# Below is custom router which has some advanced feature not implemented by Wagtail
blog_router = routers.DefaultRouter()
blog_router.register(r"posts", PostPageSet)
blog_router.register(r"categories", CategorySet)
blog_router.register(r"tags", TagSet)
```

1. We create `cms_api_router` from `WagtailAPIRouter`.
2. We registered three endpoints using `register_endpoint` method
3. We can query data for Wagtail models `Page`, `Image` and `Document`

In the previous chapter, we have built REST API on url prefix `api/blog/`, here we config to make the Wagtail REST API work on url prefix `api/cms/`

Update *react_wagtail_app/urls.py*

```
from blog.api import blog_router, cms_api_router


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

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

    path('api/blog/', include(blog_router.urls)),
    path('api/cms/', cms_api_router.urls),

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

1. We have added `path('api/cms/', cms_api_router.urls)`, to the `urlpatterns`
2. **Please not use `include` to wrap the `cms_api_router.urls` or you might get error.**

7.3 Simple Test

Now, let's test in Django shell

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

```
>>> import requests, pprint
# you can also check the URL on your browser
>>> pprint.pprint(requests.get('http://web:8000/api/cms/pages/').json())

{'items': [{'id': 3,
            'meta': {'detail_url': 'http://localhost/api/cms/pages/3/',
                     'first_published_at': '2020-10-12T03:11:37.418006Z',
                     'html_url': 'http://localhost/',
                     'slug': 'blogpage',
                     'type': 'blog.BlogPage'},
            'title': 'BlogPage'},
           {'id': 4,
            'meta': {'detail_url': 'http://localhost/api/cms/pages/4/',
                     'first_published_at': '2020-10-12T03:40:14.162646Z',
```

```
                    'html_url': 'http://localhost/postpage1/',
                    'slug': 'postpage1',
                    'type': 'blog.PostPage'},
            'title': 'PostPage1'}],
 'meta': {'total_count': 2}}
```

Note: the `detail_url` has hostname `localhost` because it is generated from Wagtail `Site`, you can change the port number to `8000` in Wagtail admin so the url would be correct. (we can ignore this problem if we do not follow the url)

1. By default, `/api/cms/pages/` would return all `wagtail.core.models.Page`, that is why you can see both `blog.BlogPage` and `blog.PostPage` here.
2. Because core logic of `/api/cms/pages/` is from Wagtail, it is not easy to customize (for example, filter posts by `tag`)
3. To solve the above problem, we need some other routers and viewsets. (That is why we built `blog_router`)

7.4 API fields

Let's keep checking the `PostPage` content in the Django shell we just opened.

```
>>> pprint.pprint(requests.get('http://web:8000/api/cms/pages/4/').json())

{'id': 4,
 'meta': {'detail_url': 'http://localhost/api/cms/pages/4/',
          'first_published_at': '2020-10-12T03:40:14.162646Z',
          'html_url': 'http://localhost/postpage1/',
          'parent': {'id': 3,
                     'meta': {'detail_url': 'http://localhost/api/cms/pages/3/',
                              'html_url': 'http://localhost/',
                              'type': 'blog.BlogPage'},
                     'title': 'BlogPage'},
          'search_description': '',
          'seo_title': '',
          'show_in_menus': False,
          'slug': 'postpage1',
          'type': 'blog.PostPage'},
 'title': 'PostPage1'}
```

Even the page model has `header_image`, `tags`, `categories` and `body`, we can not see them from REST API.

We can solve this problem by setting `api_fields`

Update *blog/models.py*

```
from wagtail.api import APIField
from .fields import TagField


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)
```

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

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

    api_fields = (
        "header_image",
        "body",
        APIField("owner"),
        APIField("api_tags", serializer=TagField(source="tags")),
        APIField(
            "pub_date",
            serializer=DateTimeField(format="%d %B %Y", source="last_published_at"),
        ),
    )
```

Notes:

1. We add some custom fields to the `api_fields`
2. We use `wagtail.api.APIField` and custom `TagField` together to return tag info of the `PostPage`
3. Code in `api_fields` is very similar with code in `blog/serializers.py`, you can check and compoare.

Let's check it again.

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

```
>>> import requests, pprint
# you can also check the URL on your browser
>>> pprint.pprint(requests.get('http://web:8000/api/cms/pages/4/').json())
{'api_tags': [{'id': 1, 'name': 'Django', 'slug': 'django'}],
 'body': [{'id': '0dd3e943-4cbc-4c13-94ce-423c91ab9800',
           'type': 'h1',
           'value': 'The Zen of Wagtail'},
          {'id': 'c111ca2d-a55f-4956-81ce-ec5cde9a4bb0',
           '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': 'd3cdcc2b-02c9-4a8d-96e2-fa4102c9bb82',
           'type': 'image_carousel',
           'value': [3, 2]},
          {'id': '05b07aa7-be47-477b-aba1-a77d61243e6b',
           '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': 'a45251b7-c9b3-4a60-9fe7-ac6fc225ad6f',
           '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>'}}],
 'header_image': {'id': 1,
                  'meta': {'detail_url': 'http://localhost/api/cms/images/1/',
                           'download_url': '/media/original_images/photo-1506765515384-028b60a970df.
↪jpeg',
                           'type': 'wagtailimages.Image'},
                  'title': 'photo-1506765515384-028b60a970df.jpeg'},
 'id': 4,
 'meta': {'detail_url': 'http://localhost/api/cms/pages/4/',
          'first_published_at': '2020-10-12T03:40:14.162646Z',
          'html_url': 'http://localhost/postpage1/',
          'parent': {'id': 3,
                     'meta': {'detail_url': 'http://localhost/api/cms/pages/3/',
                              'html_url': 'http://localhost/',
                              'type': 'blog.BlogPage'},
                     'title': 'BlogPage'},
          'search_description': '',
          'seo_title': '',
          'show_in_menus': False,
          'slug': 'postpage1',
          'type': 'blog.PostPage'},
 'owner': {'id': 1, 'meta': {'type': 'auth.User'}},
 'pub_date': '12 October 2020',
 'title': 'PostPage1'}
```

1. Now `body`, `api_tags` and `header_image` can be seen in the REST API response.

7.5 Control Image Size

If you check `header_image` in the above JSON response, you will find the image `download_url` contains the original image url.

What if you want to conrol the image size?

The answer is `serializer filed`!

The good news is Wagtail has already built this.

Update *blog/models.py*

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


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)

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

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

```
        InlinePanel("categories", label="category"),
        FieldPanel("tags"),
        StreamFieldPanel("body"),
    ]

    api_fields = (
        APIField(
            "header_image_url",
            serializer=ImageRenditionField("max-1000x800", source="header_image"),
        ),
        "body",
        APIField("owner"),
        APIField("api_tags", serializer=TagField(source="tags")),
        APIField(
            "pub_date",
            serializer=DateTimeField(format="%d %B %Y", source="last_published_at"),
        ),
    )
```

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
```

```
>>> import requests, pprint
# you can also check the URL on your browser
>>> pprint.pprint(requests.get('http://web:8000/api/cms/pages/4/').json())

'header_image_url': {'alt': 'photo-1506765515384-028b60a970df.jpeg',
                     'height': 667,
                     'url': '/media/images/photo-1506765515384-028b60a970df.max-1000x800.jpg',
                     'width': 1000},
```

7.6 StreamField Image

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.

In Wagtail StreamField block, there is a method `get_api_representation`, we can use it to control block behavior when generating api response.

Update *blog/blocks.py*

```
class CustomImageChooserBlock(ImageChooserBlock):
    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")
```

1. `CustomImageChooserBlock` now accepet 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
```

```
>>> import requests, pprint
# you can also check the URL on your browser
>>> pprint.pprint(requests.get('http://web:8000/api/cms/pages/4/').json())

'body': [{'id': '0dd3e943-4cbc-4c13-94ce-423c91ab9800',
          'type': 'h1',
          'value': 'The Zen of Wagtail'},
         {'id': 'c111ca2d-a55f-4956-81ce-ec5cde9a4bb0',
          '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': 'd3cdcc2b-02c9-4a8d-96e2-fa4102c9bb82',
          'type': 'image_carousel',
          'value': [{'alt': 'photo-1531256379416-9f000e90aacc.jpeg',
                     'height': 1175,
                     'url': '/media/images/photo-1531256379416-9f000e90aacc.original.jpg',
                     'width': 1567},
                    {'alt': 'photo-1515524738708-327f6b0037a7.jpeg',
                     'height': 1300,
                     'url': '/media/images/photo-1515524738708-327f6b0037a7.original.jpg',
                     'width': 1950}]},
         {'id': '05b07aa7-be47-477b-aba1-a77d61243e6b',
          'type': 'image_text',
          'value': {'image': {'alt': 'photo-1531256379416-9f000e90aacc.jpeg',
                              'height': 599,
                              'url': '/media/images/photo-1531256379416-9f000e90aacc.width-800.jpg',
                              'width': 800},
                    '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': 'a45251b7-c9b3-4a60-9fe7-ac6fc225ad6f',
          'type': 'image_text',
          'value': {'image': {'alt': 'photo-1506765515384-028b60a970df.jpeg',
                              'height': 534,
                              'url': '/media/images/photo-1506765515384-028b60a970df.width-800.jpg',
                              'width': 800},
                    '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. Now all the image block in `body` has `url`
2. The image in `image_text` has `800` width
3. The image in `image_carousel` has `original` width

7.7 Category

Now let's add `category` to the REST API, it is easy since we already make `tag` work.

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/models.py*

```
class PostPage(Page):

    api_fields = (
        # other fields

        APIField("api_categories", serializer=CategoryField(source="categories")),
    )
```

Chapter 8

UnitTest REST API (Part 1)

8.1 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
4. Understand how to test REST API `/api/blog/`

8.2 Workflow

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

1. `Arrange`: You whould 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 exepcted.

8.3 Fixture

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

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

1. Django doc[26]
2. create test fixtures for Wagtail[27]

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.

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

2. The test rely on the fixture file and the logic of the test seems not that straightforward. (You might need to check the fixutre file to figure out the logic)
3. The fixture file is slow to load.

8.4 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==2.12.0
wagtail-factories==2.0.0
```

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

```
$ docker-compose build
```

Next, create *blog/factories.py*

```
from factory import DjangoModelFactory, 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))
```

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 the about the implementation details.

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

8.5 Wagtail Factories

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

```
from django.utils.text import slugify
from factory import (
    DjangoModelFactory,
    LazyAttribute,
    Sequence,
)
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).

8.6 Test RestAPI

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,
)


class TestView(TestCase):
    """
    Test blog.views
    """
    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_view(self):
        # arrange
        category_1 = BlogCategoryFactory.create()

        # act
        response = self.client.get("/api/blog/categories/")
        response_data = response.json()

        # assert
        assert response_data["results"][0]["name"] == category_1.name
        assert response_data["results"][0]["slug"] == category_1.slug

        # arrange
        BlogCategoryFactory.create()

        # act
        response = self.client.get("/api/blog/categories/")
        response_data = response.json()

        # assert
        assert len(response_data["results"]) == 2
```

1. We create a `TestView` to test `blog.views`
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_view (blog.tests.TestView) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.028s

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`

8.7 Rest Test

Let's add more tests to the `TestView`

```
class TestView(TestCase):
    """
    Test blog.views
    """

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

        response = self.client.get("/api/blog/tags/")
        response_data = response.json()

        assert response_data["results"][0]["name"] == tag_1.name
        assert response_data["results"][0]["slug"] == tag_1.slug

        TagFactory.create()
        response = self.client.get("/api/blog/tags/")
        response_data = response.json()
        assert len(response_data["results"]) == 2

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

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

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

        response = self.client.get(
            f"/api/blog/posts/?category={category_1.slug}&tag=*"
        )
        response_data = response.json()
        assert response_data["results"][0]["id"] == post_page.pk

        response = self.client.get(
            f"/api/blog/posts/?category=*&tag={tag_1.slug}"
        )
        response_data = response.json()
        assert response_data["results"][0]["id"] == post_page.pk

        response = self.client.get("/api/blog/posts/")
        response_data = response.json()
        assert response_data["results"][0]["id"] == post_page.pk

        # empty list
        tag_2 = TagFactory.create()
        response = self.client.get(
            f"/api/blog/posts/?category=*&tag={tag_2.slug}"
        )
        response_data = response.json()
        assert response_data["count"] == 0

        category_2 = BlogCategoryFactory.create()
        response = self.client.get(
```

```
            f"/api/blog/posts/?category={category_2.slug}&tag=*"
        )
        response_data = response.json()
        assert response_data["count"] == 0
```

1. In `test_post_page_view`, we test the `filter` functions

Let's run test again.

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

test_category_view (blog.tests.TestView) ... ok
test_post_page_view (blog.tests.TestView) ... ok
test_tag_view (blog.tests.TestView) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.099s
```

Chapter 9

UnitTest REST API (Part 2)

9.1 Objectives

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

1. Learn how to generate test data for StreamField.
2. Test Wagtail Rest API `/api/cms/pages/`
3. Generate code coverage reports with `Coverage.py`

9.2 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()
```

9.3 StreamField Test Data

People alwasy 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(page.body.stream_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.

9.4 Write Test

Now let's start writing test.

Update *blog/tests.py*

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

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

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


class TestPostPageAPI(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(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
        )
```

```
        response = self.client.get(f"/api/cms/pages/{post_page.pk}/")
        response_data = response.json()

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

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

Note:

1. In `setUp` method, we changed root page of the default site to `self.blog_page`. Because Wagtai would filter pages based on the current site[28]
2. In unittest, we created 3 images using `ImageFactory`, they all have `width=1000`
3. `body_data` copied from the above Django shell, we replced the image `pk` with the image from the `ImageFactory`.
4. All `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_post_page (blog.tests.TestPostPageAPI) ... ok
test_category_view (blog.tests.TestView) ... ok
test_post_page_view (blog.tests.TestView) ... ok
test_tag_view (blog.tests.TestView) ... ok

----------------------------------------------------------------------
Ran 4 tests in 0.628s

OK
```

9.5 Rest Test

Let's keep adding unittests to the `TestPostPageAPI`

```
class TestPostPageAPI(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,
        )

        response = self.client.get(f"/api/cms/pages/{post_page.pk}/")
        response_data = response.json()
```

[28] https://github.com/wagtail/wagtail/blob/c9e740324c1a2197454274f5d18514b9a0752374/wagtail/api/v2/endpoints.py#L427-L432

```
        assert response_data["api_categories"][0]["name"] == category_1.name
        assert response_data["api_categories"][0]["slug"] == category_1.slug

        category_2 = BlogCategoryFactory.create()
        PostPageBlogCategoryFactory.create(
            page=post_page, blog_category=category_2,
        )

        response = self.client.get(f"/api/cms/pages/{post_page.pk}/")
        response_data = response.json()
        assert len(response_data["api_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,
        )

        response = self.client.get(f"/api/cms/pages/{post_page.pk}/")
        response_data = response.json()

        assert response_data["api_tags"][0]["name"] == tag_1.name
        assert response_data["api_tags"][0]["slug"] == tag_1.slug

        tag_2 = TagFactory.create()
        PostPageTagFactory.create(
            content_object=post_page, tag=tag_2,
        )
        response = self.client.get(f"/api/cms/pages/{post_page.pk}/")
        response_data = response.json()
        assert len(response_data["api_tags"]) == 2
```

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

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

test_post_page (blog.tests.TestPostPageAPI) ... ok
test_post_page_category (blog.tests.TestPostPageAPI) ... ok
test_post_page_tag (blog.tests.TestPostPageAPI) ... ok
test_category_view (blog.tests.TestView) ... ok
test_post_page_view (blog.tests.TestView) ... ok
test_tag_view (blog.tests.TestView) ... ok

----------------------------------------------------------------------
Ran 6 tests in 0.817s

OK
```

9.6 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[29] to the *requirements.txt*

```
django==3.1
wagtail==2.10.2
psycopg2-binary
djangorestframework

factory-boy==2.12.0
wagtail-factories==2.0.0
coverage
```

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/api.py                                      14      0   100%
blog/apps.py                                      3      3     0%
blog/blocks.py                                   23      0   100%
blog/factories.py                                29      0   100%
blog/fields.py                                   13      4    69%
blog/migrations/0001_initial.py                   8      0   100%
blog/migrations/0002_postpage_body.py             7      0   100%
blog/migrations/0003_auto_20201014_0318.py        7      0   100%
blog/migrations/__init__.py                       0      0   100%
blog/models.py                                   47      1    98%
blog/serializers.py                              16      0   100%
blog/tests.py                                   100      0   100%
blog/views.py                                    24      0   100%
manage.py                                        12      2    83%
react_wagtail_app/__init__.py                     0      0   100%
react_wagtail_app/asgi.py                         4      4     0%
react_wagtail_app/settings.py                    23      0   100%
react_wagtail_app/urls.py                        11      2    82%
react_wagtail_app/wsgi.py                         4      4     0%
-----------------------------------------------------------------
TOTAL                                           346     20    94%
```

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

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

# check in your browser
```

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

Chapter 10

Setup frontend project

10.1 Objectives

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

1. Understand the frontend workflow.
2. Create frontend project using `create-react-app`

10.2 Frontend Workflow

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

From this chapter, we will start building a frontend app (`SPA`).

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.

10.3 Project Setup

First, please make sure you have `node` installed. It is recommended to use nvm[30] to install `node` on your local env.

```
$ node -v
v12.18.4

$ npm -v
6.14.6
```

Next, We'll use the Create React App[31] to generate a boilerplate that's all set up and ready to go. (It works like cookiecutter-django[32])

```
# cd to the root directory
$ ls

manage.py
media
compose
```

[30] https://github.com/nvm-sh/nvm
[31] https://create-react-app.dev/https://create-react-app.dev/
[32] https://github.com/pydanny/cookiecutter-django

```
docker-compose.yml
react_wagtail_app
blog
requirements.txt

# https://create-react-app.dev/docs/getting-started
$ npx create-react-app frontend

# then wait for some mins
```

Notes:

1. We create `frontend` app using `npx create-react-app frontend`
2. After the command finish, you will see some output like this from terminal

```
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd frontend
  yarn start

Happy hacking!
```

If you see some command `npm start`, `npm build` when you run command on your local env, that is also ok and I will explain in a bit.

10.4 Project structure

Let's check the `frontend` directory.

```
.
├── frontend
|   ├── README.md
|   ├── node_modules
|   ├── package.json
|   ├── public
|   ├── src
|   └── yarn.lock
```

1. `node_modules` contains the dependency packages.
2. `src` contains all `source code`.

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

```
{
  "name": "frontend",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "react": "^16.14.0",
    "react-dom": "^16.14.0",
    "react-scripts": "3.4.3"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": "react-app"
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

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

10.5 Simple Test

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

```
$ cd frontend
$ yarn start
```

Now check http://localhost:3000/ on your browser, you will see a React icon and `Edit src/App.js and save to reload.`, which means the setup is working without problem.

`yarn start` run a devserver on `3000` port and it also monitor the source code in `src` directory, if you make changes to the code, it would rebuild automatically. (This behavior is similar with Django's `devserver`)

If everything works without problem, please press `CTRL-C` to quit.

Chapter 11

Dockerizing React project

11.1 Objectives

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

1. Learn how to develop the `frontend` app in docker container.
2. Understand `docker volume` better.

11.2 Compose file

Note: We can ignore the `storybook` here

Before we start, please delete the `frontend/node_modules` on the local env.

```
$ rm -r frontend/node_modules/
```

The *docker-compose.yml* already has two services `web` and `db`, next, let's add a new service `frontend`.

Update *docker-compose.yml*

```yaml
services:
  web:
    # code omitted for brevity

  db:
    # code omitted for brevity
```

```yaml
  frontend:
    build:
      context: .
      dockerfile: ./compose/local/node/Dockerfile
    image: react_wagtail_app_frontend
    command: yarn start
    volumes:
      - .:/app
      # http://jdlm.info/articles/2016/03/06/lessons-building-node-app-docker.html
      - /app/frontend/node_modules
    ports:
      - 3000:3000
    depends_on:
      - web
    stdin_open: true
```

Notes

1. We use a custom `./compose/local/node/Dockerfile` to build the docker image, we will create the file in a bit.
2. The command to run in docker conainer is `yarn start`, which would launch a webpack-dev-server[33]
3. `stdin_open: true` is also required to make the app run in docker.

Here I want to talk about the `node_modules`

1. As you know, we already deleted `frontend/node_modules` on docker host.
2. Now we would install frontned packages in `docker build` stage.
3. To make docker container can use the `node_modules` created in `docker build` stage, we need to mount it to the container. That is why we need to add `/app/frontend/node_modules` to the volumes. You can check this link to learn more Lessons from Building a Node App in Docker[34]

11.3 Dockerfile

Next, let's create *compose/local/node/Dockerfile*

```dockerfile
FROM node:12-stretch-slim

WORKDIR /app/frontend

COPY ./frontend/package.json /app/frontend
COPY ./frontend/yarn.lock /app/frontend

ENV PATH ./node_modules/.bin/:$PATH

RUN yarn install
```

11.4 Simple Test

[33] https://webpack.js.org/guides/development/#using-webpack-dev-server
[34] http://jdlm.info/articles/2016/03/06/lessons-building-node-app-docker.html

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

You will see some output like this in the terminal

```
frontend_1  | You can now view frontend in the browser.
frontend_1  |
frontend_1  |   Local:            http://localhost:3000
frontend_1  |   On Your Network:  http://172.23.0.4:3000
frontend_1  |
frontend_1  | Note that the development build is not optimized.
frontend_1  | To create a production build, use yarn build.
frontend_1  |
```

Now, check http://127.0.0.1:3000/ to see if it work, and **do not close this page.**

Next, edit *frontend/src/App.js*, change the `Learn React` to `Learn React Test`, and then save in the editor.

You would see new output in termila

```
frontend_1  | Compiling...
frontend_1  | Compiled successfully!
```

If you check the http://127.0.0.1:3000/ again, you will see the link text has changed.

You do not have to manually refresh the page to see the change and this awesome features is called HMR (hot module replacement), which is brought by `create-react-app`.

Chapter 12

Build React Component with StoryBook

12.1 Objectives

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

1. Understand Storybook and the benefits.
2. Install Storybook and dockerize it

12.2 Background

I have seen many online courses or blogs teaching developers to write React Component pull data from REST API directly when developing.

However, this brings some problems:

1. Your React Component depends on the REST API, which dependes on the data in db sometime. (You have to write React Component after REST API is ready to serve)
2. Frontend developers can not test, manipulate specific React component in an easy way.

   Storybook is an open source tool for developing UI components in isolation for React, Vue, Angular, and more. It makes building stunning UIs organized and efficient.

Storybook can help solve the above problems in a great way, and that is why I want to show you how to do it in my course.

12.3 Install Storybook

Let's setup `storybook` in our `frontend` project.

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

# enter frontend container
$ docker-compose exec frontend bash

(container)$ pwd
/app/frontend

# https://create-react-app.dev/docs/developing-components-in-isolation/#getting-started-with-
↪storybook
(container)$ npx -p @storybook/cli sb init
```

The above command would update `package.json`, install dependency packages and create some files.

After the command finish, you should see file structures like this.

```
frontend
├── .gitignore
├── .storybook
│   ├── main.js
│   └── preview.js
├── public
    # code omitted for brevity
├── src
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   ├── logo.svg
│   ├── serviceWorker.js
│   ├── setupTests.js
│   └── stories
│       ├── Button.js
│       ├── Button.stories.js
│       ├── Header.js
│       ├── Header.stories.js
│       ├── Introduction.stories.mdx
│       ├── Page.js
│       ├── Page.stories.js
```

```
│       ├── assets
│       │   ├── code-brackets.svg
│       │   ├── colors.svg
│       │   ├── comments.svg
│       │   ├── direction.svg
│       │   ├── flow.svg
│       │   ├── plugin.svg
│       │   ├── repo.svg
│       │   └── stackalt.svg
│       ├── button.css
│       ├── header.css
│       └── page.css
└── yarn.lock
```

Notes:

1. `frontend/.storybook` contains some config file for the storybook
2. `frontend/src/stories` contains some sample React Compooent and the sample Storybook code.
3. New command `storybook` and `build-storybook` has been added to the `scripts` of the *frontend/package.json*
4. Now check `storybook` command in *frontend/package.json*, we see `start-storybook -p 6006 -s public`. `6006` is the port number of the storybook devserver.

Let's add new service `storybook` which work on `6006` to the docker compose file.

Update the *frontend/package.json*

```yaml
services:
  web:
    # code omitted for brevity

  db:
    # code omitted for brevity

  frontend:
    # code omitted for brevity

  storybook:
    build:
      context: .
      dockerfile: ./compose/local/node/Dockerfile
    image: react_wagtail_app_storybook
    command: yarn storybook
    volumes:
      - .:/app
      # http://jdlm.info/articles/2016/03/06/lessons-building-node-app-docker.html
      - /app/frontend/node_modules
    ports:
      - 6006:6006
    depends_on:
      - web
    stdin_open: true
```

Notes:

1. `frontend` and `storybook` services have many things in common so here I will only tell the difference.
2. The image is `react_wagtail_app_storybook`
3. The command is `yarn storybook`
4. Port is `6006`

Let's rerun the whole application

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


storybook_1  | |   Storybook 6.0.26 started                      |
storybook_1  | |   18 s for manager and 19 s for preview         |
storybook_1  | |                                                 |
storybook_1  | |    Local:            http://localhost:6006/     |
storybook_1  | |    On your network:  http://172.18.0.5:6006/    |
```

1. Frontend app is working on http://localhost:3000/
2. StoryBook is working on http://localhost:6006/

If you see `Module not found: Error: Can't resolve` error on your local env, check details here *Frontend FAQ*

12.4 Explore StoryBook

Now visit http://localhost:6006/ in browser, and check `button/primary`

You will see a single `Button`, check the top `Docs` tab, you will see doc for the `Button` component.

If you change the `size` of the button in the `size` row, the top button wuold change in realtime.

As you can see, this is very powerful way to let you check UI of the React components in an isolated environment. And you can even use storybook to build component library in your project.

12.5 Storybook config

Check *frontend/.storybook/main.js*

You will see something like this

```
module.exports = {
  "stories": [
    "../src/**/*.stories.mdx",
    "../src/**/*.stories.@(js|jsx|ts|tsx)"
  ],
  "addons": [
    "@storybook/addon-links",
    "@storybook/addon-essentials",
    "@storybook/preset-create-react-app"
  ]
}
```

1. `stories` is used to define rules to find the stories.
2. `addons` is to defined some addon.

12.6 Cleanup

In the next chapters, we will start writing our own storybooks, let's delete the all files in *frontend/src/stories*

```
$ rm app/frontend/src/stories/*
```

Chapter 13

Add SCSS support to React project

13.1 Objectives

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

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

13.2 Bootstrap

This frontend app would build style based on popular open source framwork 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.

13.3 Install Dependency

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

```
$ docker-compose up -d

# run command in a new terminal
$ docker-compose exec frontend bash

(container)$ yarn add node-sass
# we better specify version number here
(container)$ yarn add bootstrap@4.5.3

(container)$ yarn add react-bootstrap
(container)$ exit

# sync dependency in storybook
$ docker-compose exec storybook yarn install
```

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

[35] https://getbootstrap.com/docs/4.0/getting-started/theming/
[36] https://react-bootstrap.github.io/

```
"dependencies": {
  "@testing-library/jest-dom": "^4.2.4",
  "@testing-library/react": "^9.3.2",
  "@testing-library/user-event": "^7.1.2",
  "bootstrap": "4.5.3",
  "node-sass": "^4.14.1",
  "react": "^16.14.0",
  "react-bootstrap": "^1.3.0",
  "react-dom": "^16.14.0",
  "react-scripts": "3.4.3"
},
```

13.4 Simple Test

Change suffix of *frontend/src/index.css* to *frontend/src/index.scss*

Update `import ./index.css';` to `import ./index.scss';` in *frontend/src/index.js* (SCSS syntax is CSS compatible so this can be done without problem)

You will see logs like this

```
frontend_1   | Compiled successfully!
```

Now check http://127.0.0.1:3000/ and it can still work without problem.

13.5 Test with Bootstrap

Let's keep testing.

Please delete all code from *frontend/src/index.scss*, and then add below code

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

Now visit http://127.0.0.1:3000/ and try to check `elements` in browser devtool.

You can find `style` element in `head` which has this

```
/*!
 * Bootstrap v4.5.3 (https://getbootstrap.com/)
 * Copyright 2011-2020 The Bootstrap Authors
 * Copyright 2011-2020 Twitter, Inc.
 * Licensed under MIT (https://github.com/twbs/bootstrap/blob/main/LICENSE)
```

Which means bootstrap `SCSS` files have been compiled to the css and imported to the frontend project sucessufly.

Chapter 14

Building React Component (Part 1)

14.1 Objectives

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

1. The basic syntax of React Component
2. Learn React Component Life Lifecycle

14.2 Basic Concepts

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

14.2.1 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.

14.2.2 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.

14.3 Simple React Component

Create *frontend/src/components/TagWidget.js*

```
import React from "react";
import '../index.scss';

class TagWidget extends React.Component {
  render() {
    return (
      <div className="card my-4">
        <h5 className="card-header">Tags</h5>
        <div className="card-body">
          Loading...
        </div>
      </div>
    );
  }
}

export { TagWidget };
```

Notes:

1. We import `index.scss` which contains `@import "~bootstrap/scss/bootstrap";`, which means Bootstrap styles are dependency of this component. We can also do similar things on the `font`, `image` files.
2. React has **more than one** kinds of components, here we start with `React.Component` subclasses because it is easy to learn and it is popular.
3. In `render` method, we return a something like HTML, the syntax is called `JSX`. (It is more readable than pure JS code becaues you do not need to concatenate the HTML)
4. In `JSX`, we should use `className` if we want to specify class for HTML element, because `class` is a reserved keyword in JS.

Next, we will create a story for the above component

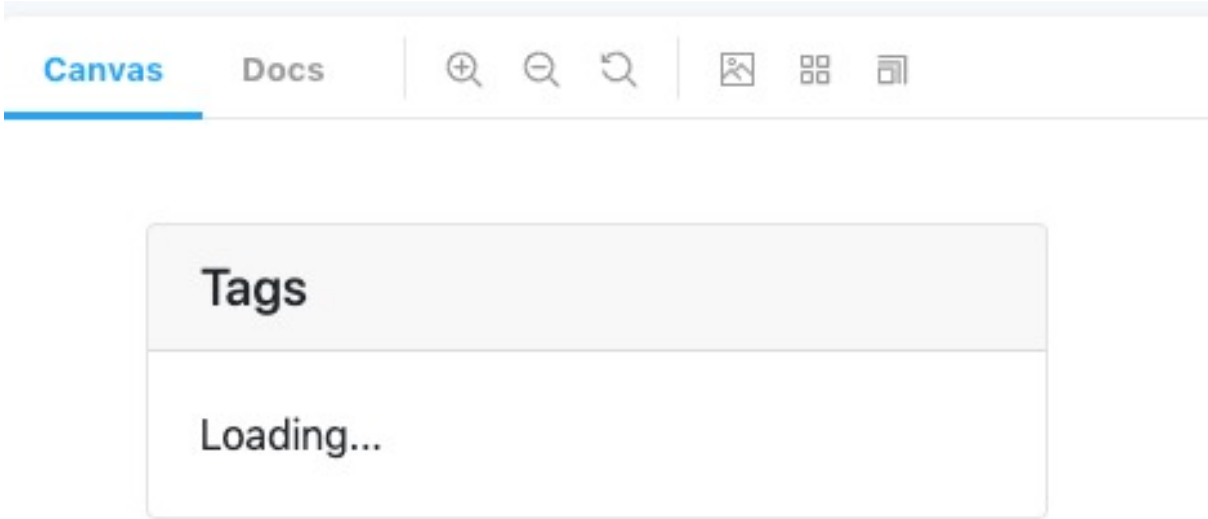
Create *frontend/src/stories/TagWidget.stories.js*

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

export default {
  title: "TagWidget",
  component: TagWidget,
};

export const Example = () => {
  return (
    <Container>
      <Row>
        <Col md={4}>
          <TagWidget />
        </Col>
      </Row>
    </Container>
  );
};
```

1. The filename `TagWidget.stories.js` tell us this is story for the `TagWidget.js`

2. We import `Container, Row, Col` from `react-bootstrap` to make the `JSX` strcuture more clear than `<div className='container'>`
3. The `Example` is a JS function, we return JSX in the function and the JSX would render in the storybook

Now check http://127.0.0.1:6006/, and click `TagWidget/Example` in the sidebar, you will see component already dispaly on the page.



Notes:

1. As you can see, we use Storybook to quickly display the React component.
2. The Component now display `loading`, which means it is loading data, I will talk about how to use Ajax to pull data to the React component in the next sections.

14.4 React Component Lifecycle

React component has `Lifeccyle`, which we can override the methods to run custom code.

Let's check below methods (we can learn step by step)

```
constructor()                -> Initirate instantiate, Like __init__() in Python
render()                     -> Return HTML representation of the Component to be mounted
componentDidMount()          -> Invoked immediately after a component is mounted (inserted into the
↪tree)
```

From React doc[37]

> You should populate data with AJAX calls in the componentDidMount lifecycle method. This is so you can use setState to update your component when the data is retrieved.

Let's update *frontend/src/components/TagWidget.js*

[37] https://reactjs.org/docs/faq-ajax.html#where-in-the-component-lifecycle-should-i-make-an-ajax-call

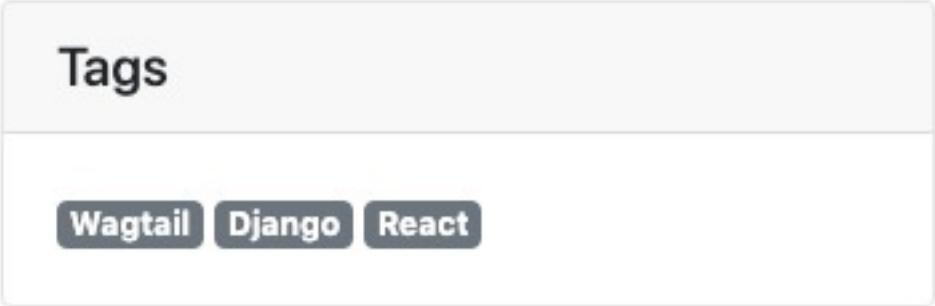
```
import React from "react";
import '../index.scss';

class TagWidget extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      tags: [],
      loading: true,
    };
  }

  componentDidMount() {
    const tags = [
      {
        slug: "wagtail",
        name: "Wagtail",
      },
      {
        slug: "django",
        name: "Django",
      },
      {
        slug: "react",
        name: "React",
      },
    ];
    this.setState({
      tags,
      loading: false
    });
  }

  render() {
    let content;
    if (this.state.loading) {
      content = 'Loading...';
    } else {
      content = this.state.tags.map((tag) => (
        <a href={`/tag/${tag.slug}`} key={tag.slug}>
          <span className="badge badge-secondary">{tag.name}</span>{" "}
        </a>
      ))
    }

    return (
      <div className="card my-4">
        <h5 className="card-header">Tags</h5>
        <div className="card-body">
          {content}
        </div>
      </div>
    );
  }
}

export { TagWidget };
```

Notes:

1. In `constructor` method, we set init value to the Component `state`
2. In `render` method, we check `state.loading` to know if the tag data is ready.

3. In `componentDidMount` method, we set the `tags` value to the `state.tags` and change the `stateloading` to `false`, which means the loading process is finished and data is ready.
4. When `state` has been updated, the `render` method would run again to return the tag HTML.
5. We use `this.state.tags.map` to do `for-loop` operation, the `key` is used to distinguish child in a list React keys[38]

If you check in the Storybook, you would seem something like this.



14.5 Ajax and Mock

Now, we will update the `componentDidMount` to pull data using `Ajax`, and then set the data of the response to the `state.tags` to make the component work as expected.

Because the REST API is ready, so some people might thinkging about sending requests to the `127.0.0.1:8000` directly.

However, this is not the best practise.

It is better use some way to mock the API so the storybook can run without the real API server.

Let's first install axios[39], which is a popular AJAX client, and axios-mock-adapter[40], which allow to mock request.

```
# add the dependency
$ docker-compose exec frontend bash
(container)$ yarn add axios
(container)$ yarn add axios-mock-adapter
(container)$ exit

# sync dependency in storybook
$ docker-compose exec storybook yarn install
```

Update `componentDidMount` of *frontend/src/components/TagWidget.js*

```
import axios from "axios";

class TagWidget extends React.Component {

  componentDidMount() {
    axios.get("/api/blog/tags/").then((res) => {
      const tags = res.data.results;
      this.setState({
```

[38] https://reactjs.org/docs/lists-and-keys.html#keys
[39] https://www.npmjs.com/package/axios
[40] https://github.com/ctimmerm/axios-mock-adapter

```
        tags,
        loading: false
      });
    });
  }

}
```

Notes:

1. Import dependency `import axios from "axios";`
2. Use `axios` to send Ajax request, and set the data to the component `state`

Create *frontend/src/stories/mockUtils.js*

```
import MockAdapter from "axios-mock-adapter";
import axios from "axios";

const mockTag = (mockAxios) => {
  const API_REQUEST = "/api/blog/tags/";
  mockAxios.onGet(API_REQUEST).reply(200, {
    results: [
      {
        slug: "wagtail",
        name: "Wagtail",
      },
      {
        slug: "django",
        name: "Django",
      },
      {
        slug: "react",
        name: "React",
      },
    ],
  });
};

export { mockTag };
```

Notes:

1. We create `mockUtils.js` and put all mock code in this file.
2. As you can see, the `axios-mock-adapter` is easy to use and we can define Axios response in JS code.

Update *frontend/src/stories/TagWidget.stories.js*

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

import axios from "axios";
import MockAdapter from "axios-mock-adapter"
import { mockTag } from "./mockUtils";

export default {
  title: "TagWidget",
  component: TagWidget,
};

export const Example = () => {
  const mock = new MockAdapter(axios);
```

```
  mockTag(mock);

  return (
    <Container>
      <Row>
        <Col md={4}>
          <TagWidget />
        </Col>
      </Row>
    </Container>
  );
};
```

Notes:

1. In the story, we create a `MockAdapter` instance, and defined the Ajax response in `mockTag` method.
2. And then return the `JSX` which contains the `TagWidget`

Now please check the storybook to make sure it works on your local env.

14.6 Global SCSS for Storybook

Now we have `import '../index.scss';` in our `TagWidget` to make the style work, However, this is not a good way.

Please remove `index.scss` from *frontend/src/components/TagWidget.js*, and then update *frontend/.storybook/preview.js*

```
// Make scss available on all stories
import '../src/index.scss'
```

1. `preview.js` is a global config file for storybook.
2. So all stories would have `index.scss` now.

14.7 Reference

React Lifecycle Methods Diagram[41], which provides an awesome online diagram for refrenece.

[41] https://github.com/wojtekmaj/react-lifecycle-methods-diagram

Chapter 15

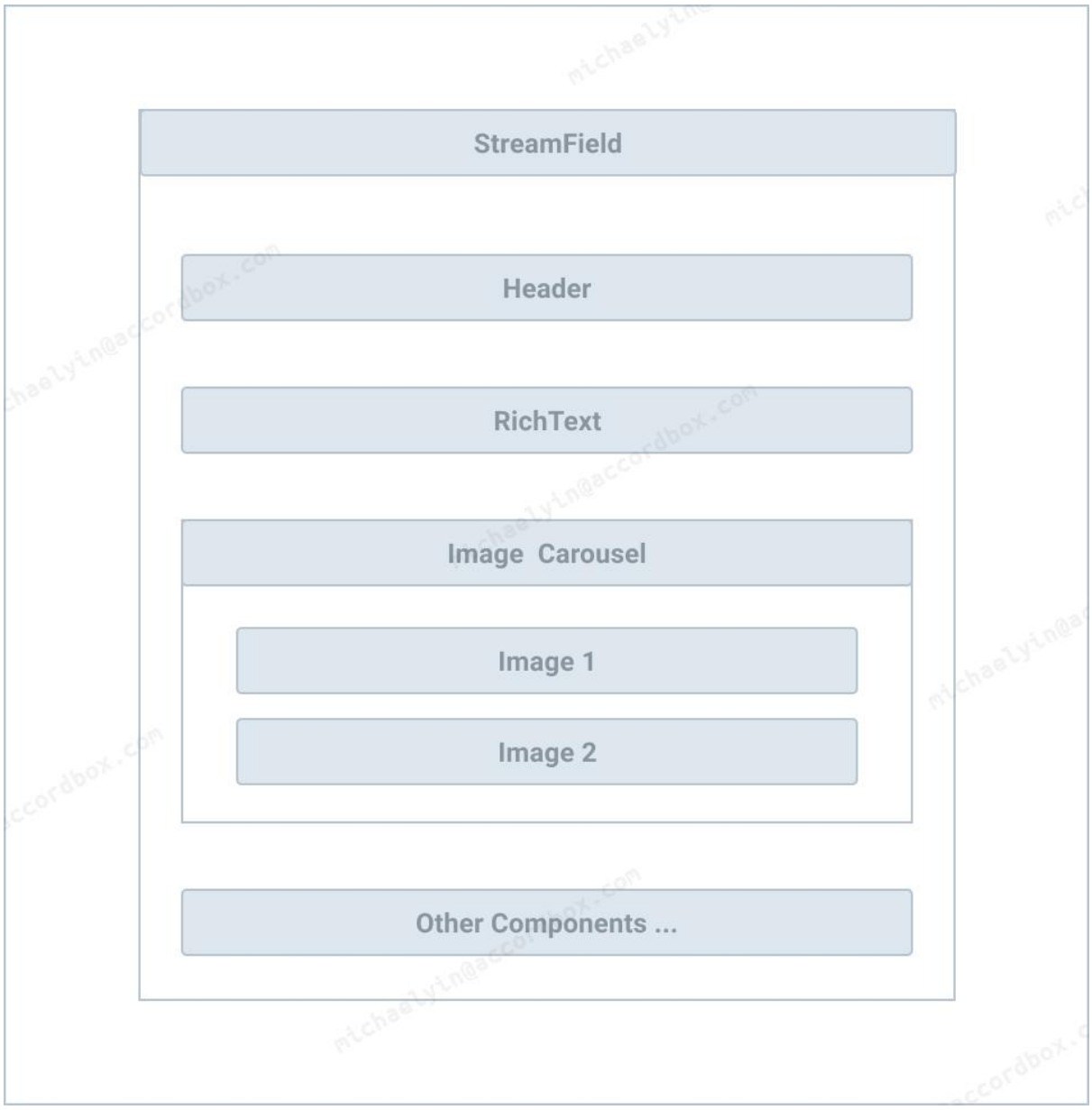
Building React Component (Part 2)

15.1 Objectives

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

1. Understand React component `props` and `state` better.
2. Build StreamField components and relevant stories.
3. Generate mock data for StreamField components.

15.2 Design



15.3 Install DOMPurify

Before we start, please first install `DOMPurify`, and I will explain in the below section.

```
$ docker-compose up -d

(container)$ yarn add dompurify
(container)$ exit

# sync dependency in storybook
$ docker-compose exec storybook yarn install
```

15.4 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.

15.4.1 ImageCarousel

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

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

function ImageCarousel(props) {
  return (
    <div className="my-4">
      <Carousel>
        {props.value.map((item, index) => (
          <Carousel.Item key={`${index}.${item}`}>
            <img className="d-block w-100" src={item.url} alt="" />
          </Carousel.Item>
        ))}
      </Carousel>
    </div>
  );
}

export { ImageCarousel };
```

1. Here we used React function components[42]
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 check React hook[43].
4. The `Carousel` from `react-bootstrap` can help us make Bootstrap components work with React. (You can check Bootstrap doc[44] to compare the difference)

15.4.2 ImageText

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

[42] https://reactjs.org/docs/components-and-props.html#function-and-class-components
[43] https://reactjs.org/docs/hooks-intro.html
[44] https://getbootstrap.com/docs/4.0/components/carousel/

```
import React from "react";
import { Container, Row, Col } from "react-bootstrap";
import { sanitize } from 'dompurify';

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

export { ImageText };
```

Notes:

1. When we insert `RichText` string to React component, we need to assign it to `dangerouslySetInnerHTML`
2. To remove risks from the HTML, we use DOMPurify[45] which is a sanitizer for HTML to help us.
3. You can find more details here dangerouslySetInnerHTML[46]

15.4.3 ThumbnailGallery

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

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

function ThumbnailGallery(props) {
  return (
    <Container>
      <div className="row text-center text-lg-left">
        {props.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"
            >
              <img
                className="img-fluid img-thumbnail"
```

[45] https://github.com/cure53/DOMPurify
[46] https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml

```
                src={imageItem.url}
                alt=""
              />
            </a>
          </div>
        ))}
      </div>
    </Container>
  );
}

export { ThumbnailGallery };
```

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

15.5 StreamField Component

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

```
import React from "react";
import { sanitize } from 'dompurify';
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: `${sanitize(field.value)}` }} />
        </div>
      );
    } else if (field.type === "thumbnail_gallery") {
      html.push(<ThumbnailGallery value={field.value} key={`${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]

15.6 Mock Data

Next, let's get some mock data ready to use in the Storybook.

First, please download some images from unsplash[48] and put them at *frontend/src/stories/assets*

You should have files like this below.

```
└── stories
    ├── assets
    │   ├── image_1.jpeg
    │   ├── image_2.jpeg
    │   ├── image_3.jpeg
```

Then import the images in *frontend/src/stories/mockUtils.js*

```
// top
import cardImage from "./assets/image_1.jpeg";
import cardImage2 from "./assets/image_2.jpeg";
import cardImage3 from "./assets/image_3.jpeg";
```

Then we can add StreamField mock data to the *frontend/src/stories/mockUtils.js*

```
const richtext1 = `
<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>
`;

const mockStreamFieldData = [
  {
    type: "h2",
    value: "The Zen of Wagtail",
  },
  {
    type: "paragraph",
    value: richtext1,
  },
  {
    type: "thumbnail_gallery",
    value: [
      {
        url: cardImage,
      },
```

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

```
      {
        url: cardImage2,
      },
      {
        url: cardImage3,
      },
      {
        url: cardImage2,
      },
      {
        url: cardImage,
      },
    ],
  },
  {
    type: "image_carousel",
    value: [
      {
        url: cardImage2,
      },
      {
        url: cardImage3,
      },
      {
        url: cardImage2,
      },
    ],
  },
  {
    type: "h2",
    value: "ImageText Example",
  },
  {
    type: "image_text",
    value: {
      image: {
        url: cardImage,
      },
      text: `<div class="rich-text"><p><b>Wagtail</b> CMS's multi-site feature is awesome! Client
↪can edit content of
          different sites in an efficient way.</p></div>`,
      reverse: true,
    },
  },
  {
    type: "image_text",
    value: {
      image: {
        url: cardImage,
      },
      text: `<div class="rich-text"><p><b>Wagtail</b> CMS's multi-site feature is awesome! Client
↪can edit content of
          different sites in an efficient way.</p></div>`,
      reverse: false,
    },
  },
];
```

Notes:

1. `richtext1` contains some RAW HTML, in most cases, this type of data is created by `RichTextBlock` or `RawHTMLBlock`

2. `mockStreamFieldData` is the StreamField mock data, which is a list contains objects which have `type` and `value`.
3. Please remember to export `mockStreamFieldData` using `export { mockStreamFieldData, };` so we can import in the story.

15.7 StoryBook

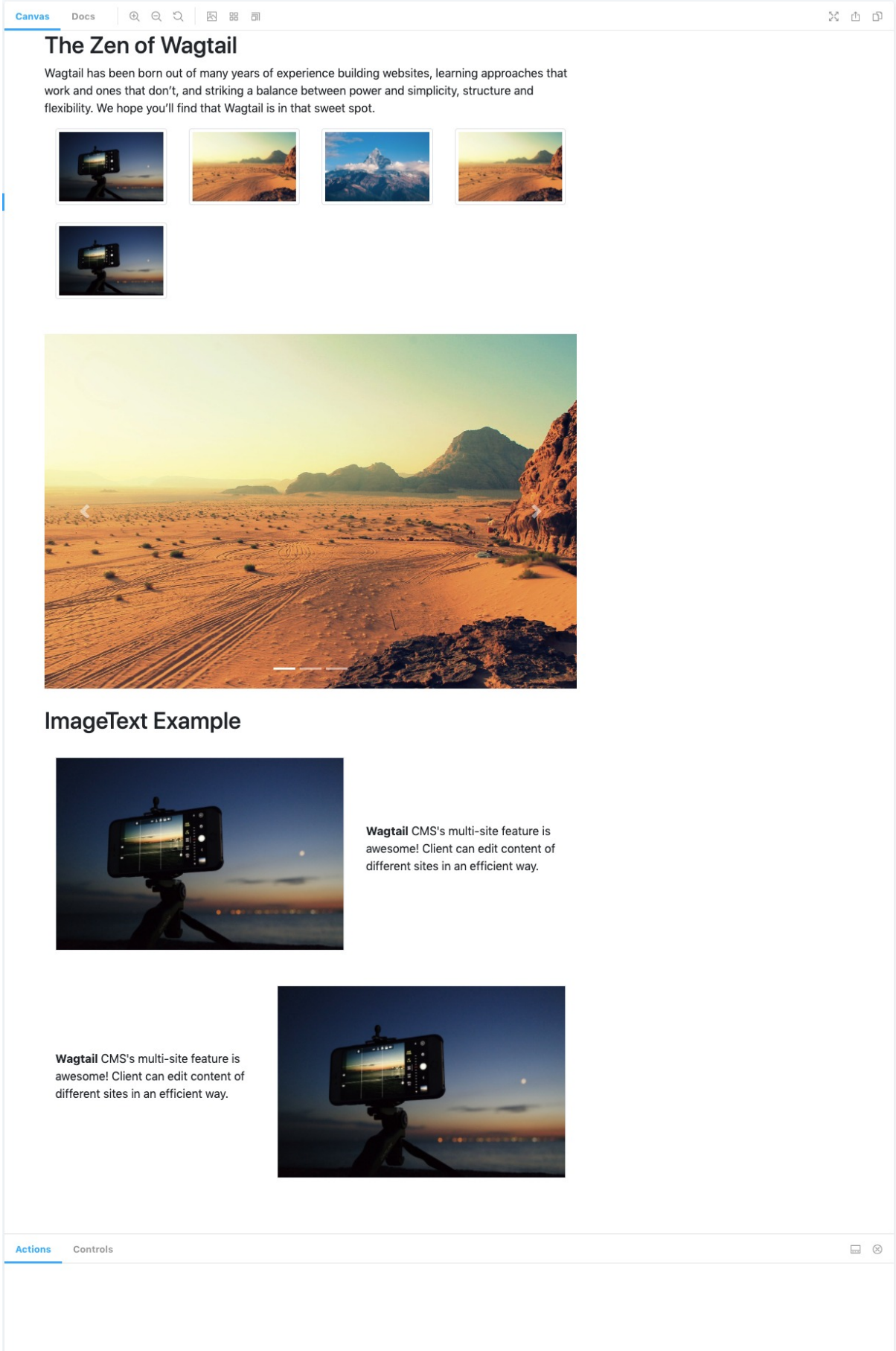
Create *frontend/src/stories/StreamField.stories.js*

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

import { mockStreamFieldData } from "./mockUtils";

export default {
  title: "StreamField",
  component: StreamField,
};

export const Example = () => {

  return (
    <Container>
      <Row>
        <Col md={8}>
          <StreamField value={mockStreamFieldData}/>
        </Col>
      </Row>
    </Container>
  );
};
```

Notes:

1. We pass `mockStreamFieldData` to the `props.value` of `StreamField`

Now you can check the `StreamField` in your storybook.


Canvas
Docs
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.
ImageText Example
Wagtail CMS's multi-site feature is awesome! Client can edit content of different sites in an efficient way.
Wagtail CMS's multi-site feature is awesome! Client can edit content of different sites in an efficient way.
Actions
Controls

15.8 Notes

1. In some projects, you can write story for each StreamField block component.
2. `Storybook` give us very flexible way for us to quickly check UI compooents without the backend API. (with the help of Mock data)

Chapter 16

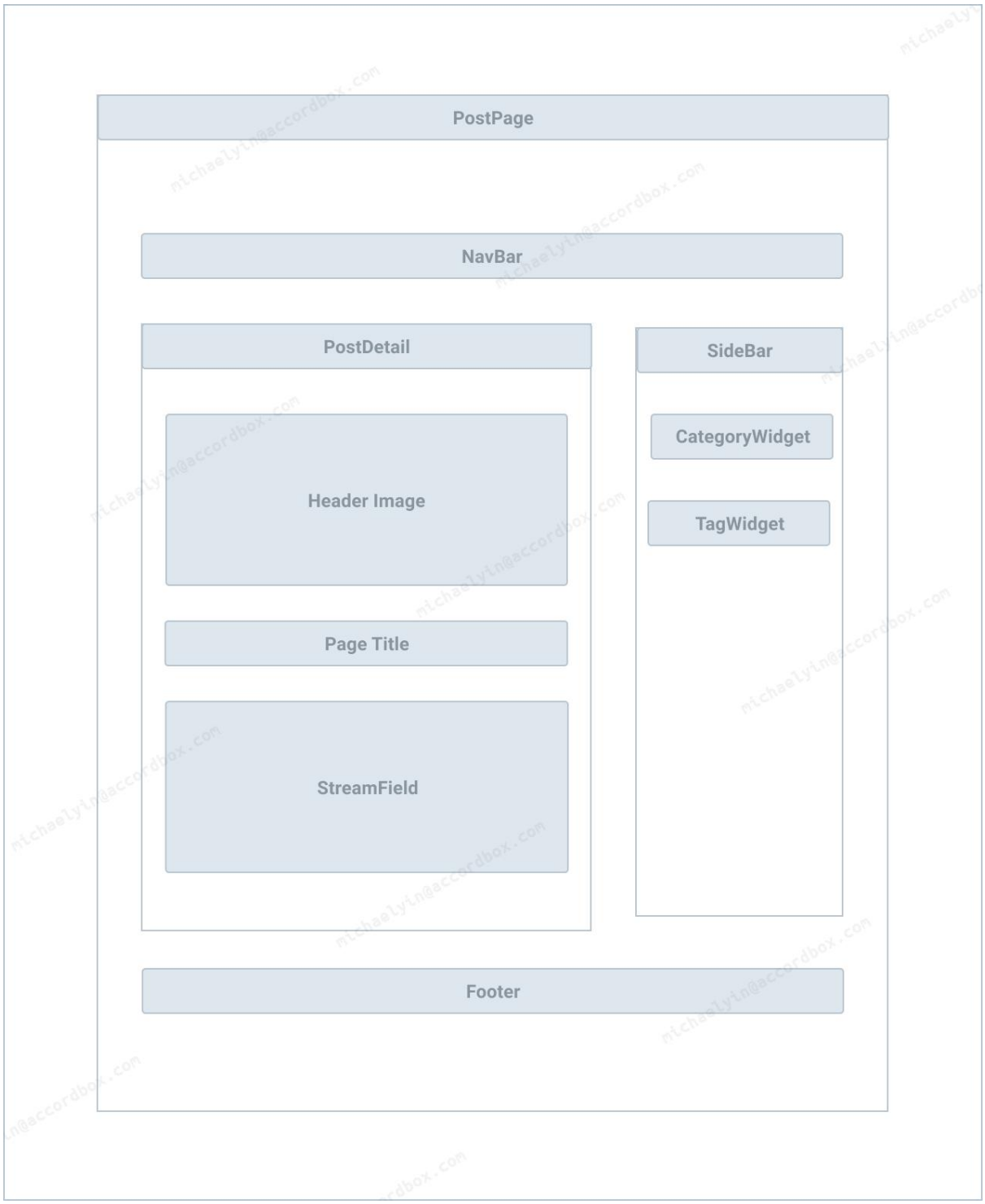
Building React Component (Part 3)

16.1 Objectives

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

1. Build `PostPage` component
2. Build story for `PostPage` and check the page in Storybook.

16.2 Design



16.3 PostDetail

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

Create *frontend/src/components/PostDetail.js*

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

class PostDetail extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      post: [],
      loading: true,
    };
  }

  componentDidMount() {
    axios.get(`/api/cms/pages/1/`).then((res) => {
      const post = res.data;
      this.setState({
        post,
        loading: false
      });
    })
  }

  render() {
    if (!this.state.loading) {
      const post = this.state.post;

      return (
        <div className="col-md-8">
          <img
            src={post.header_image_url.url}
            className="img-fluid rounded"
            alt=""
          />
          <hr />
          <h1>{post.title}</h1>
          <hr />
          <StreamField value={post.body} />
        </div>
      );
    } else {
      return <div className="col-md-8">Loading...</div>;
    }
  }
}

export { PostDetail };
```

1. We build a `class component` because we need to send `Ajax` request in `componentDidMount` method.
2. `/api/cms/pages/1/` is the REST API which can let us get the blog post detail from the `PostPage.api_fields`

16.3.1 Mock Data

Update *frontend/src/stories/mockUtils.js* to make `axios` can receive the `mockStreamFieldData`

```
// const mockStreamFieldData
// code omitted for brevity

const mockPost = (mockAxios) => {
```

```
  mockAxios.onGet(`/api/cms/pages/1/`).reply(200, {
    id: 1,
    title: "Love React 1",
    excerpt: "category: programming",
    header_image_url: {
      url: cardImage,
    },
    // py datetime.strftime('%s000')
    pub_date: 1597720114000,
    body: mockStreamFieldData,
  });

};

const mockTag = () => {
  // code omitted for brevity
};

export { mockStreamFieldData, mockTag, mockPost };
```

16.3.2 PostDetail Story

Let's create story for this component.

Create *frontend/src/stories/PostDetail.stories.js*

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

import axios from "axios";
import MockAdapter from "axios-mock-adapter"
import { mockPost } from "./mockUtils";

export default {
  title: "PostDetail",
  component: PostDetail,
};

export const Example = () => {
  const mock = new MockAdapter(axios);
  mockPost(mock);

  return (
    <Container>
      <Row>
        <PostDetail />
      </Row>
    </Container>
  );
};
```

Now `frontend/src/` would have files like this

```
├── App.css
├── App.js
├── App.test.js
├── components
|   ├── PostDetail.js
|   ├── StreamField
```

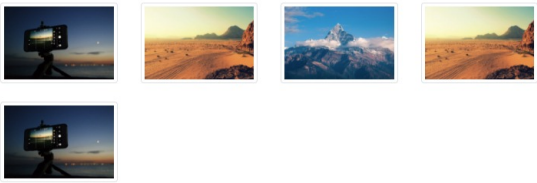
```
│   │   ├── ImageCarousel.js
│   │   ├── ImageText.js
│   │   ├── StreamField.js
│   │   └── ThumbnailGallery.js
│   └── TagWidget.js
├── index.js
├── index.scss
├── logo.svg
├── serviceWorker.js
├── setupTests.js
└── stories
    ├── PostDetail.stories.js
    ├── StreamField.stories.js
    ├── TagWidget.stories.js
    ├── assets
    │   ├── image_1.jpeg
    │   ├── image_2.jpeg
    │   └── image_3.jpeg
    └── mockUtils.js
```

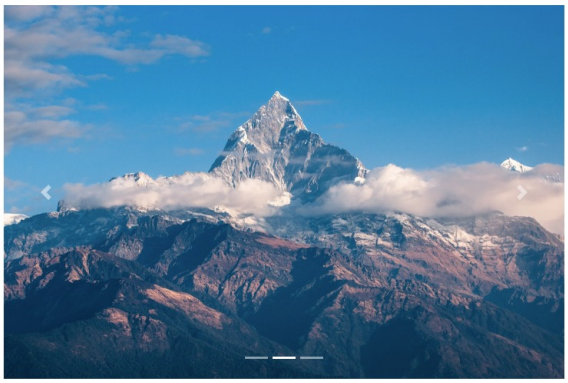
Let's check the component in storybook.

Love React 1

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.





ImageText Example

Wagtail CMS's multi-site feature is awesome! Client can edit content of different sites in an efficient way.

Wagtail CMS's multi-site feature is awesome! Client can edit content of different sites in an efficient way.

16.4 Other Components

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/src/components/TopNav.js*

```
import React from "react";
import { Navbar, Nav, Container } from "react-bootstrap";

class TopNav extends React.Component {
  render() {
    return (
      <Navbar bg="dark" variant="dark" expand="lg" className="mb-2">
        <Container>
          <Navbar.Brand href="#">React Wagtail Demo</Navbar.Brand>
          <Navbar.Toggle aria-controls="basic-navbar-nav" />
          <Navbar.Collapse id="basic-navbar-nav">
            <Nav className="mr-auto">
              <Nav.Link href="#">Link</Nav.Link>
              <Nav.Link href="#">Link</Nav.Link>
            </Nav>
          </Navbar.Collapse>
        </Container>
      </Navbar>
    );
  }
}

export { TopNav };
```

1. `Navbar` and `Nav` from `react-bootstrap` make the code structure easy to understand.
2. When using classic `Bootstrap`, you might need to import `jQuery` to make some components such as `dropdown` in menu to work. This is not required when using `react-bootstrap`.

Let's create *frontend/src/components/SideBar.js*, which is a container for sidebar widgets

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

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

export { SideBar };
```

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

```
import React from "react";

class Footer extends React.Component {
  render() {
    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 };
```

16.5 PostPage

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

Create *frontend/src/components/PostPage.js*

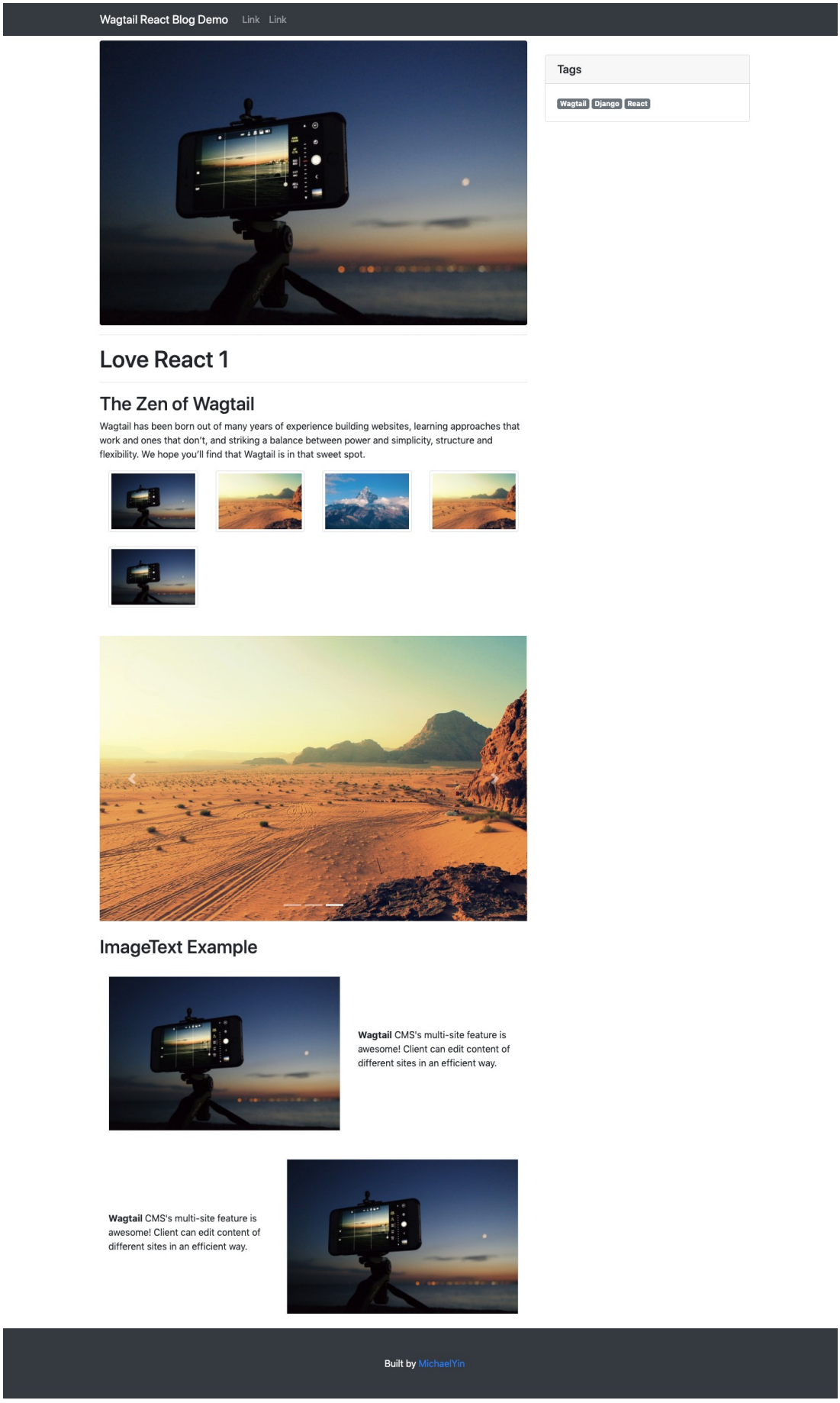
```
import React from "react";
import { Container, Row } from "react-bootstrap";
import { TopNav } from "./TopNav";
import { Footer } from "./Footer";
import { SideBar } from "./SideBar";
import { PostDetail } from "./PostDetail";

class PostPage extends React.Component {
  render() {
    return (
      <div>
        <TopNav/>
        <Container>
          <Row>
            <PostDetail/>
            <SideBar/>
          </Row>
        </Container>
        <Footer/>
      </div>
    );
  }
}

export { PostPage };
```

Let's create story for `PostPage`, *frontend/src/stories/PostPage.stories.js*

```
import React from "react";

import { PostPage } from "../components/PostPage";

import axios from "axios";
import MockAdapter from "axios-mock-adapter"
import { mockPost, mockTag } from "./mockUtils";

export default {
  title: "PostPage",
  component: PostPage,
};

export const Example = () => {
  const mock = new MockAdapter(axios);
  mockPost(mock);
  mockTag(mock);
```

```
  return (
    <PostPage />
  );
};
```

1. Because we need to display `PostPage` data and tag data in the story, so we should call methods get mock data ready.


Wagtail React Blog Demo
Link
Link
Tags
Wagtail
Django
React
Love React 1
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.
ImageText Example
Wagtail CMS's multi-site feature is awesome! Client can edit content of different sites in an efficient way.
Wagtail CMS's multi-site feature is awesome! Client can edit content of different sites in an efficient way.
Built by MichaelYin

As you can see, we build a page component step by step and now we can also check the final page in the storybook.

You might notice we send Ajax to `/api/cms/pages/1/` which is a staic url, we will change it in later.

Chapter 17

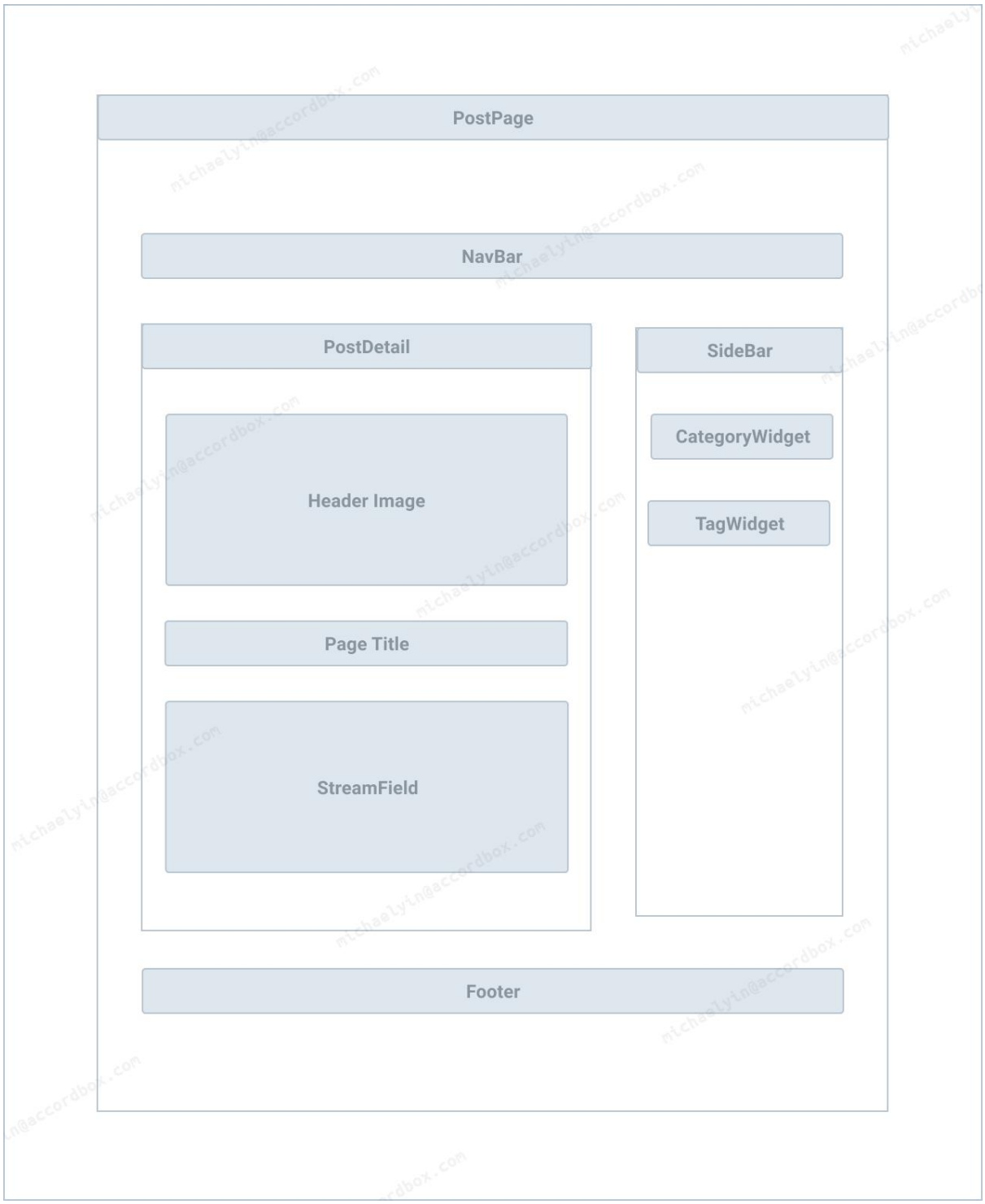
React Router (Part 1)

17.1 Objectives

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

1. Understand how React Router works
2. Make React Router work in Storybook

17.2 Design



17.3 Install

`React Reouter` provide some `navigational components` for us to use in our React app. Which make navigation between different components possible.

```
$ docker-compose up -d

$ docker-compose exec frontend bash
(container)$ yarn add react-router-dom
(container)$ exit

# sync dependency in storybook
$ docker-compose exec storybook yarn install
```

Notes:

1. `react-router-dom` contains componnetes which can be used in `web` projects.
2. Some core components (such as `Route`) are in `react-router` package, which is the dependency of `react-douter-dom`, so here we do not need to install it again.

17.4 PostDetail

Current version of `PostDetail` component will send Ajax to static url `/api/cms/pages/1/`, let's change it to send Ajax request based on the route params.

First, let's update *frontend/src/stories/PostDetail.stories.js*

```
import React from "react";
import { Container, Row } from "react-bootstrap";
import { MemoryRouter } from "react-router-dom";
import { Route } from "react-router";
import { PostDetail } from "../components/PostDetail";

import { mockPost } from "./mockUtils";

export default {
  title: "PostDetail",
  component: PostDetail,
};

export const Example = () => {
  mockPost();

  return (
    <Container>
      <Row>
        <MemoryRouter initialEntries={["/post/1/"]}>
          <Route path="/post/:id" component={PostDetail} />
        </MemoryRouter>
      </Row>
    </Container>
  );
};
```

1. We use `MemoryRouter` and set `initialEntries` to `/post/1/`, `MemoryRouter` is very helpful if we want to test navigation in the storybook or the test.
2. We declared a `Route`, if the `path` is matched with the `initialEntries`, then `PostDetail` would be used to render.

Let's update *frontend/src/components/PostDetail.js*

```
class PostDetail extends React.Component {
  // code omitted for brevity
```

```
  componentDidMount() {
    const pk = this.props.match.params.id;

    axios.get(`/api/cms/pages/${pk}/`).then((res) => {
      const post = res.data;
      this.setState({
        post,
        loading: false
      });
    })
  }

}
```

1. When `MemoryRouter` use `PostDetail` to render, it would pass match[49] object to `PostDetail`, which contains info about the router match information.
2. So we can get the `PostPage` primary key from `this.props.match.params.id`, and use it to send Ajax request.
3. Please check `PostDetail/Example` in storybook, and then try to change `initialEntries` to see if it still works.

17.5 PostPage

Let's make the React router work with `PostPage`

Update *frontend/src/stories/PostPage.stories.js*

```
import React from "react";
import { MemoryRouter } from "react-router-dom";
import { Route, Switch } from "react-router";
import { PostPage } from "../components/PostPage";

import { mockPost, mockTag } from "./mockUtils";

export default {
  title: "PostPage",
  component: PostPage,
};

export const Example1 = () => {
  mockPost();
  mockTag();

  return (
    <MemoryRouter initialEntries={['/post/1/']}>
      <Switch>
        <Route path="/post/:id" component={PostPage}/>
      </Switch>
    </MemoryRouter>
  );
};

export const Example2 = () => {
  mockPost();
  mockTag();

  return (
```

[49] https://reactrouter.com/web/api/match

```
    <MemoryRouter initialEntries={['/post/2/']}>
      <Switch>
        <Route path="/post/:id" component={PostPage}/>
      </Switch>
    </MemoryRouter>
  );
};
```

Notes:

1. Here we defined two stories for the Component, they have diffrent `initialEntries`
2. In `Example1`, `PostPage` will send Ajax requests to `/api/cms/pages/1/`
3. In `Example2`, `PostPage` will send Ajax requests to `/api/cms/pages/2/`

Let's update *frontend/src/components/PostPage.js*

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

class PostPage extends React.Component {
  render() {
    return (
      <div>
        <TopNav/>
        <Container>
          <Row>
            <PostDetail {...this.props} />
            <SideBar/>
          </Row>
        </Container>
        <Footer/>
      </div>
    );
  }
}

export { PostPage };
```

1. Now we already know component passed to `Route` would have `match` object.
2. So `PostPage.props` would have `match` object, however, you should know parent component `props` does not passed to child component by default.
3. We can use `{...this.props}` to pass all parent component `props` to child `component`. So `PostDetail` can access the `match`.

Let's update *frontend/src/stories/mockUtils.js* to add mock data for the `/api/cms/pages/2/`

```
  mockAxios.onGet(`/api/cms/pages/1/`).reply(200, {
    // code omitted for brevity
  });

  mockAxios.onGet(`/api/cms/pages/2/`).reply(200, {
    id: 2,
    title: "Love React 2",
    excerpt: "tag: react",
    header_image_url: {
      url: cardImage2,
    },
```

```
    // py datetime.strftime('%s000')
    pub_date: 1597720114002,
    body: mockStreamFieldData,
  });
```

Notes:

1. They have different titles and header_image
2. Now you can check `PostPage` in the storybook, you will see two stories which are different.

Chapter 18

React Router (Part 2)

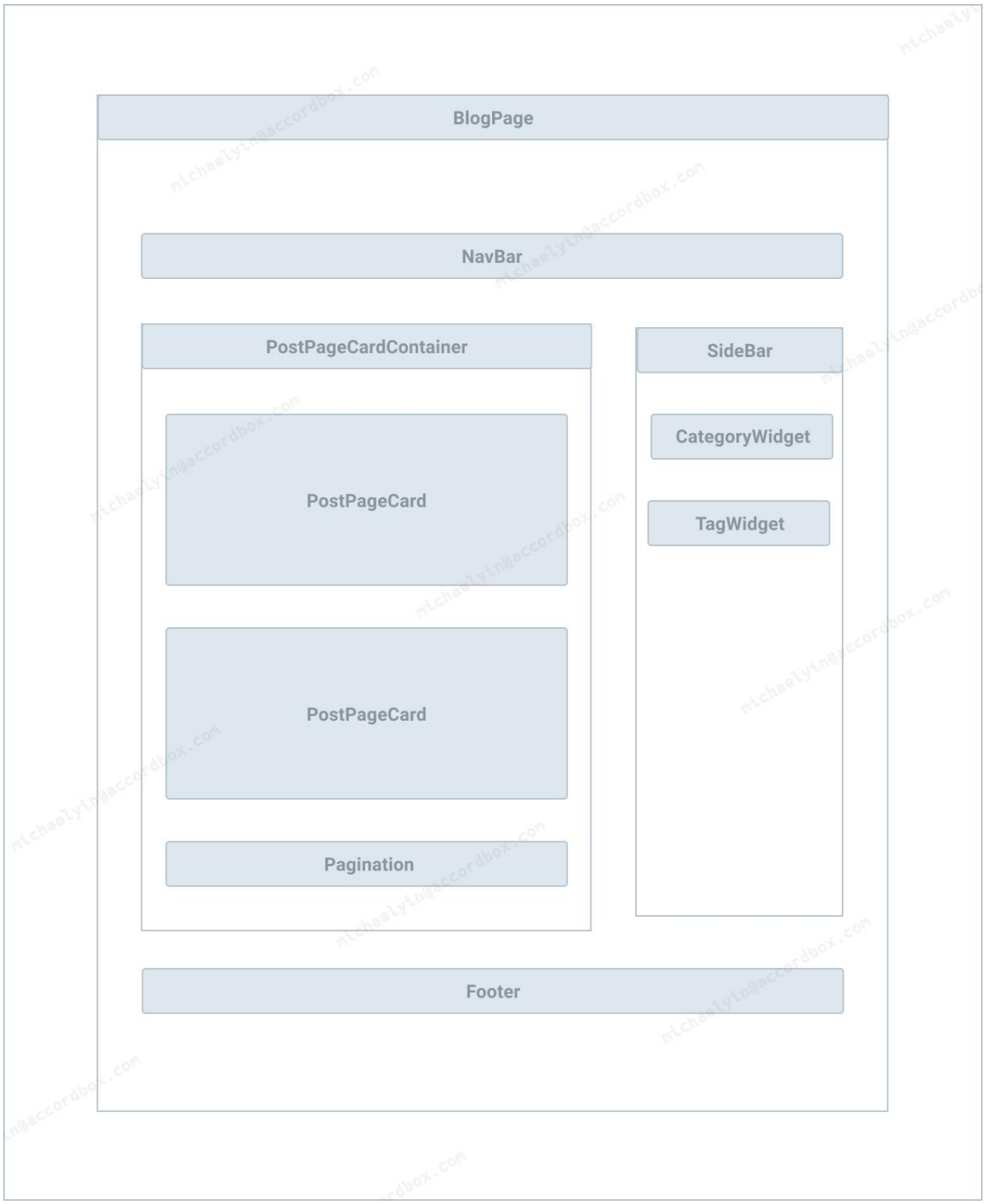
18.1 Objective

In the last chapter, we already make `PostPage` work with `React router`, in this chapter, we will start building the `BlogPage`

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

1. Build `PostPageCardContainer` component
2. Use `componentDidUpdate` method to make pagination work and understand `React Lifecycle` better.

18.2 Design



18.3 React Router Link

`react-router-dom` has a `Link` component to let us provide navigation in our application.

For example

```
<Link to="/about">About</Link>
```

Will generate HTML like `<a href="/about">About</a>`, but it can work with other `react router` components.

18.4 PostPageCard

Let's build `PostPageCard` component, it would contain basic info for `PostPage` and have links point to the `PostPage`

Create *frontend/src/components/PostPageCard.js*

```
import React from "react";
import { Link } from "react-router-dom";
import axios from "axios";

class PostPageCard extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: null,
      loading: true,
    };
  }

  componentDidMount() {
    axios.get(`/api/cms/pages/${this.props.postPk}/`).then((res) => {
      this.setState({
        data: res.data,
        loading: false
      });
    });
  }

  renderPost(data) {
    const dateStr = new Date(data.pub_date).toLocaleString();

    return (
      <div className="card mb-4">
        <Link to={`/post/${data.id}`}>
          <img
            src={data.header_image_url.url}
            className="card-img-top"
            alt=""
          />
        </Link>
        <div className="card-body">
          <h2 className="card-title">
            <Link to={`/post/${data.id}`}>{data.title}</Link>
          </h2>
          <p className="card-text">{data.excerpt}</p>
          <Link to={`/post/${data.id}`} className="btn btn-primary">
            Read More →
          </Link>
        </div>
        <div className="card-footer text-muted">Posted on {dateStr}</div>
      </div>
    );
  }
```

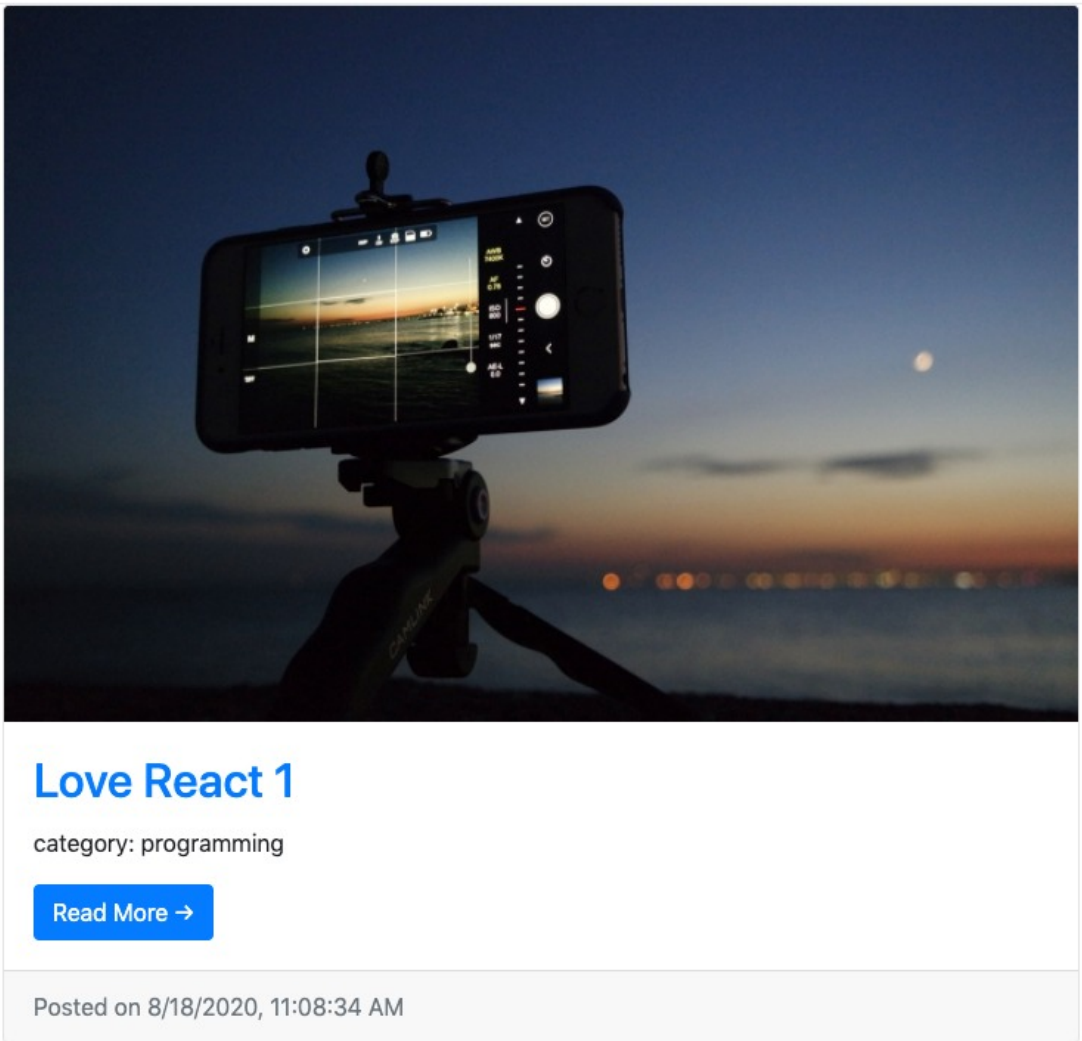
```
  render() {
    if (this.state.loading) {
      return 'Loading...';
    } else {
      return this.renderPost(this.state.data);
    }
  }
}

export { PostPageCard };
```

1. `PostPageCard` will query page detail info by sending Ajax request in `componentDidMount`
2. `props.postPk` tell us which page we need to query
3. We use `Link` from `react-router-dom` to represent the link instead of `a` tag.

Let's create story for the component, create *frontend/src/stories/PostPageCard.stories.js*

```
import React from "react";
import { Container, Row, Col } from "react-bootstrap";
import { MemoryRouter } from "react-router-dom";
import { PostPageCard } from "../components/PostPageCard";

import { mockPost } from "./mockUtils";

export default {
  title: "PostPageCard",
  component: PostPageCard,
};

export const Example = () => {
  mockPost();

  return (
    <Container>
      <Row>
        <Col md={8}>
          <MemoryRouter>
            <PostPageCard postPk={1} />
          </MemoryRouter>
        </Col>
      </Row>
    </Container>
  );
};
```

1. In the story, we set the props in this way `postPk={1}`



18.5 PostPageCardContainer

`BlogPage` will act like index page. It would have below functions

1. Provides `pagination` so user can check all blog posts
2. Provides `filter` function so uer can filter by using something like `tag`

Here let's create a `container` component which support the above feature.

This is an important component in this course

Let's create *frontend/src/components/PostPageCardContainer.js*

```
import React from "react";
import axios from "axios";
import { Col } from "react-bootstrap";
import { Link } from "react-router-dom";
import { generatePath } from "react-router";
import _ from 'lodash';

import { PostPageCard } from "./PostPageCard";
```

```
class PostPageCardContainer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      pageCount: 0,
      pageStep: 2,
    };
    this.getPosts = this.getPosts.bind(this);
  }

  componentDidMount() {
    this.getPosts();
  }

  getCurPage() {
    // return the page number from the url
    const page = this.props.match.params.page;
    return page === undefined ? 1 : parseInt(page);
  }

  getPrePageUrl() {
    const target = _.clone(this.props.match.params);
    target.page = this.getCurPage() - 1;
    return generatePath(this.props.match.path, target);
  }

  getNextPageUrl() {
    const target = _.clone(this.props.match.params);
    target.page = this.getCurPage() + 1;
    return generatePath(this.props.match.path, target);
  }

  getPosts() {
    let category =
      this.props.match.params.category === undefined
        ? "*"
        : this.props.match.params.category;
    let tag =
      this.props.match.params.tag === undefined
        ? "*"
        : this.props.match.params.tag;

    let offset = (this.getCurPage() - 1) * this.state.pageStep;
    const url = `/api/blog/posts/?limit=${this.state.pageStep}&offset=${offset}&category=${category}
↪&tag=${tag}`;
    axios.get(
      url
    ).then((res) => {
      const posts = res.data.results;
      this.setState({
        posts,
        pageCount: Math.ceil(parseInt(res.data.count) / this.state.pageStep),
      });
    });
  }

  render() {
    return (
      <Col md={8}>
        {this.state.posts.map((post) => (
          <PostPageCard postPk={post.id} key={post.id} />
        ))}
```

```
        <nav aria-label="Page navigation example">
          <ul className="pagination">
            <li
              className={
                this.getCurPage() <= 1 ? "page-item disabled" : "page-item"
              }
            >
              <Link
                to={this.getPrePageUrl()}
                className="page-link"
              >
                Previous
              </Link>
            </li>
            <li
              className={
                this.getCurPage() >= this.state.pageCount
                  ? "page-item disabled"
                  : "page-item"
              }
            >
              <Link
                to={this.getNextPageUrl()}
                className="page-link"
              >
                Next
              </Link>
            </li>
          </ul>
        </nav>
      </Col>
    );

  }
}

export { PostPageCardContainer };
```

Notes

1. In react component, if we want to access Component props (`this.props`) and state in `non-default` methods, then we should `bind it to component instance`. That is why you see `this.getPosts = this.getPosts.bind(this);` in `constructor` and more details can be found How do I bind a function to a component instance[50]
2. `getCurPage`, `getPrePageUrl` and `getNextPageUrl` can help us get the pre and next page index.
3. `generatePath` can help us generate the link url from the react router `match path` and `match.params`. (It works like Django `django.urls.reverse`)
4. In `getPosts`, we use `limit` and `offset` to do the pagination, and use `tag` and `category` to do filter function
5. When Ajax request get the response, it would write data to the `component state`, and component would run `render` method, `PostPageCard` would be used to display detail info of the post page.

18.6 PostPageCardContainer Story

Before writing story, let's make the mock data ready.

[50] https://reactjs.org/docs/faq-functions.html#bind-in-constructor-es2015

Update *frontend/src/stories/mockUtils.js*

```
const mockPost = () => {

  mockAxios
    .onGet(`/api/blog/posts/?limit=2&offset=0&category=*&tag=*`).reply(200, {
    results: [{ id: 1 }, { id: 2 }],
    count: 4,
  });

  mockAxios
    .onGet(`/api/blog/posts/?limit=2&offset=2&category=*&tag=*`).reply(200, {
    results: [{ id: 3 }, { id: 4 }],
    count: 4,
  });

  mockAxios
    .onGet(`/api/blog/posts/?limit=2&offset=0&category=*&tag=react`)
    .reply(200, {
      results: [{ id: 2 }, { id: 4 }],
      count: 2,
    });

  mockAxios
    .onGet(`/api/blog/posts/?limit=2&offset=0&category=*&tag=wagtail`)
    .reply(200, {
      results: [],
      count: 0,
    });

  mockAxios
    .onGet(`/api/blog/posts/?limit=2&offset=0&category=*&tag=django`)
    .reply(200, {
      results: [],
      count: 0,
    });

  mockAxios.onGet(`/api/cms/pages/1/`).reply(200, {
    // code omitted for brevity
  });

  mockAxios.onGet(`/api/cms/pages/2/`).reply(200, {
    // code omitted for brevity
  });

  mockAxios.onGet(`/api/cms/pages/3/`).reply(200, {
    id: 3,
    title: "Love React 3",
    excerpt: "category: programming",
    header_image_url: {
      url: cardImage,
    },
    // py datetime.strftime('%s000')
    pub_date: 1597720114002,
    body: mockStreamFieldData,
  });

  mockAxios.onGet(`/api/cms/pages/4/`).reply(200, {
    id: 4,
    title: "Love React 4",
    excerpt: "tag: react",
    header_image_url: {
      url: cardImage,
```

```
    },
    // py datetime.strftime('%s000')
    pub_date: 1597720114002,
    body: mockStreamFieldData,
  });

};
```

Notes:

1. Now we have 4 posts in the mock data.
2. Post 2 and 4 have tag `react`

Cretae *frontend/src/stories/PostPageCardContainer.stories.js*

```
import React from "react";

import { Route, Switch } from "react-router";
import { MemoryRouter } from "react-router-dom";
import { Container, Row } from "react-bootstrap";

import { PostPageCardContainer } from "../components/PostPageCardContainer";

import { mockPost } from "./mockUtils";

export default {
  title: "PostPageCardContainer",
  component: PostPageCardContainer,
};

export const Pagination = () => {
  mockPost();

  return (
    <Container>
      <Row>
        <MemoryRouter initialEntries={["/"]}>
          <Switch>
            <Route path="/tag/:tag/:page([\d]+)?" component={PostPageCardContainer}/>
            <Route path="/:page([\d]+)?" component={PostPageCardContainer}/>
          </Switch>
        </MemoryRouter>
      </Row>
    </Container>
  );
};

export const TagFilter = () => {
  mockPost();

  return (
    <Container>
      <Row>
        <MemoryRouter initialEntries={["/tag/react"]}>
          <Switch>
            <Route path="/tag/:tag/:page([\d]+)?" component={PostPageCardContainer}/>
            <Route path="/:page([\d]+)?" component={PostPageCardContainer}/>
          </Switch>
        </MemoryRouter>
      </Row>
    </Container>
  );
};
```

Notes:

1. Here we created two stories, the difference is the `MemoryRouter` has different initialEntries
2. The `Pagination` story would not filter blog posts
3. `TagFilter` would filter posts which have `tag=react`
4. The `React router` also support custom regex (`([\d]+)?` here), you can check more details on custom-matching-parameters[51])

18.7 Manual Test

Now you can check the `PostPageCardContainer` stories.

1. You will see `Love React 1` and `Love React 2` in the `Pagination`.
2. You wil see `Love React 2` and `Love React 4` in the `Tag Filter`

But you also find some problem.

There are 4 posts in the mock data, but the pagination button in the `Pagination` story seems not working!

Let's solve it in the next section.

18.8 Component Update

Please check `componentDidUpdate` in React Lifecycle Methods Diagram[52]

> componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.
>
> Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props

After we click the pagination link, the `props` of `PostPageCardContainer` have changed, however, we did not send Ajax request based on the new `props`

So let's fix it by adding a method to the component.

```
class PostPageCardContainer extends React.Component {

  componentDidUpdate(prevProps) {
    if (prevProps.location !== this.props.location) {
      this.getPosts();
    }
  }

}
```

Notes:

1. `location` is from `react-router`, here you can see it something like URL.
2. If the previous URL is different with the current URL, which means user do pagination or filter operation, then we call `getPosts` to update `state`
3. And then `render` method would be called to reflect the change.

[51] https://github.com/pillarjs/path-to-regexp#custom-matching-parameters
[52] https://github.com/wojtekmaj/react-lifecycle-methods-diagram

4. In most cases, logic in `componentDidMount` and `componentDidUpdate` have something in common, so you can move them to a seperate method (`getPosts` here) to make the code clean.

Now you can test in the storybook and the pagination should work as expected!

Chapter 19

Build App Component

19.1 Objective

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

1. Build `BlogPage` and `App` component
2. Check `BlogPage` in the storybook

19.2 BlogPage

In the previous chapter, we already built `PostPageCardContainer`, which is the core component of the `BlogPage`

Now, let's create `frontend/src/components/BlogPage.js`

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

class BlogPage extends React.Component {
  render() {
    return (
      <div>
        <TopNav />
        <Container>
          <Row>
            <PostPageCardContainer {...this.props} />
            <SideBar />
          </Row>
        </Container>
        <Footer />
      </div>
    );
  }
}

export { BlogPage };
```

Notes:

1. The structure is very similar with `PostPage` component.
2. We pass the router property to `PostPageCardContainer` using `{...this.props}`

19.3 App

Now `BlogPage` and `PostPage` are both finished, let's create the `App` component to make both component work together.

Update *frontend/src/App.js*

```
import React from "react";
import { Route, Switch } from "react-router";
import { Container, Row } from "react-bootstrap";
import { BlogPage } from "./components/BlogPage";
import { PostPage } from "./components/PostPage";

function App() {
  return (
    <Switch>
      <Route path="/post/:id([\d]+)" component={PostPage}/>
      <Route path="/tag/:tag/:page([\d]+)?" component={BlogPage}/>
      <Route path="/:page([\d]+)?" component={BlogPage}/>
      <Route
        path="*"
        component={() => (
          <Container>
            <Row>
              <h1>404</h1>
            </Row>
          </Container>
        )}
      />
    </Switch>
  );
}

export default App;
```

Notes:

1. `Switch` can make sure only one `route` would render. Switch doc[53]
2. In the path, we use `regex` expression to write flexible route rules. For example, `/` and `/1/` will both match `/:page([\d]+)?`.
3. The last route is a fallback route and show `404` message.

Considering the `App.css` is not needed anymore, let's delete it.

```
$ rm frontend/src/App.css
```

Next, let's write story for the `App.js`

Create *frontend/src/stories/App.stories.js*

```
import React from "react";
import { MemoryRouter } from "react-router-dom";

import App from "../App";
```

[53] https://reactrouter.com/web/api/Switch

```
import axios from "axios";
import MockAdapter from "axios-mock-adapter"
import { mockPost, mockTag  } from "./mockUtils";

export default {
  title: "App",
  component: App,
  decorators: [],
};

export const Example = () => {
  const mock = new MockAdapter(axios);
  mockPost(mock);
  mockTag(mock);

  return (
    <MemoryRouter initialEntries={["/"]}>
      <App/>
    </MemoryRouter>
  );
};
```

Notes:

1. Considering the `App` already contains `route` config, we only need to use `MemoryRouter` to wrap it.

19.4 TagWidget

To make the `App` works in storybook, we still need to update some components.

Edit *frontend/src/components/TagWidget.js* to replace the `a` with `Link` of `react-router-dom`

```
class TagWidget extends React.Component {
  // code omitted for brevity

  render() {
    let content;
    if (this.state.loading) {
      content = 'Loading...';
    } else {
      content = this.state.tags.map((tag) => (
        <Link to={`/tag/${tag.slug}`} key={tag.slug}>
          <span className="badge badge-secondary">{tag.name}</span>{" "}
        </Link>
      ))
    }

    return (
      <div className="card my-4">
        <h5 className="card-header">Tags</h5>
        <div className="card-body">
          {content}
        </div>
      </div>
    );
  }

}
```

Please note that the `Link` need be located in `router`, or you will get `You should not use <Link> outside a <Router>` error.

Update *frontend/src/stories/TagWidget.stories.js*

```
export const Example = () => {
  const mock = new MockAdapter(axios);
  mockTag(mock);

  return (
    <MemoryRouter>
      <Container>
        <Row>
          <Col md={4}>
            <TagWidget/>
          </Col>
        </Row>
      </Container>
    </MemoryRouter>
  );
};
```

Notes:

1. Here we put `TagWidget` in `MemoryRouter`
2. Please check `TagWidget` story in storybook

19.5 TopNav

Update *frontend/src/components/TopNav.js*

```
import React from "react";
import { Navbar, Nav, Container } from "react-bootstrap";
import { Link } from "react-router-dom";

class TopNav extends React.Component {
  render() {
    return (
      <Navbar bg="dark" variant="dark" expand="lg" className="mb-2">
        <Container>
          <Link to="/" className="navbar-brand">React Wagtail Demo</Link>
          <Navbar.Toggle aria-controls="basic-navbar-nav" />
          <Navbar.Collapse id="basic-navbar-nav">
            <Nav className="mr-auto">
              <Nav.Link href="#">Link</Nav.Link>
              <Nav.Link href="#">Link</Nav.Link>
            </Nav>
          </Navbar.Collapse>
        </Container>
      </Navbar>
    );
  }
}

export { TopNav };
```

Here we updated `React Wagtail Demo` link.

19.6 Category

The category component would work the similar way as `Tag` component.

Let's add it to our project.

Create *frontend/src/components/CategoryWidget.js*

```
import React from "react";
import axios from "axios";
import { Link } from "react-router-dom";

class CategoryWidget extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      categories: [],
      loading: true,
    };
  }

  componentDidMount() {
    axios.get("/api/blog/categories/").then((res) => {
      const categories = res.data.results;
      this.setState({
        categories,
        loading: false
      });
    });
  }

  render() {
    let content;
    if (this.state.loading) {
      content = 'Loading...';
    } else {
      content = <div className="row">
        <div className="col-lg-12">
          <ul className="list-unstyled mb-0">
            {this.state.categories.map((category) => (
              <li key={category.slug}>
                <Link to={`/category/${category.slug}`}>
                  {category.name}
                </Link>
              </li>
            ))}
          </ul>
        </div>
      </div>
    }

    return (
      <div className="card my-4">
        <h5 className="card-header">Categories</h5>
        <div className="card-body">
          {content}
        </div>
      </div>
    );
  }
}

export { CategoryWidget };
```

Update *frontend/src/stories/mockUtils.js* to get mock data ready.

```
const mockPost = (mockAxios) => {
  // code omitted for brevity

  mockAxios
    .onGet(`/api/blog/posts/?limit=2&offset=0&category=programming&tag=*`)
    .reply(200, {
      results: [{ id: 1 }, { id: 3 }],
      count: 2,
    });

  mockAxios
    .onGet(`/api/blog/posts/?limit=2&offset=0&category=life&tag=*`)
    .reply(200, {
      results: [],
      count: 0,
    });
}

const mockCategory = (mockAxios) => {
  const API_REQUEST = "/api/blog/categories/";

  mockAxios.onGet(API_REQUEST).reply(200, {
    results: [
      {
        slug: "programming",
        name: "Programming",
      },
      {
        slug: "life",
        name: "Life",
      },
    ],
  });
};

export { mockStreamFieldData, mockCategory, mockTag, mockPost};
```

Update *frontend/src/components/SideBar.js*

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

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

export { SideBar };
```

Update *frontend/src/stories/App.stories.js*

```
import React from "react";
import { MemoryRouter } from "react-router-dom";

import App from "../App";

import axios from "axios";
```

```
import MockAdapter from "axios-mock-adapter"
import { mockPost, mockTag, mockCategory } from "./mockUtils";

export default {
  title: "App",
  component: App,
  decorators: [],
};

export const Example = () => {
  const mock = new MockAdapter(axios);
  mockCategory(mock);
  mockPost(mock);
  mockTag(mock);

  return (
    <MemoryRouter initialEntries={["/"]}>
      <App/>
    </MemoryRouter>
  );
};
```

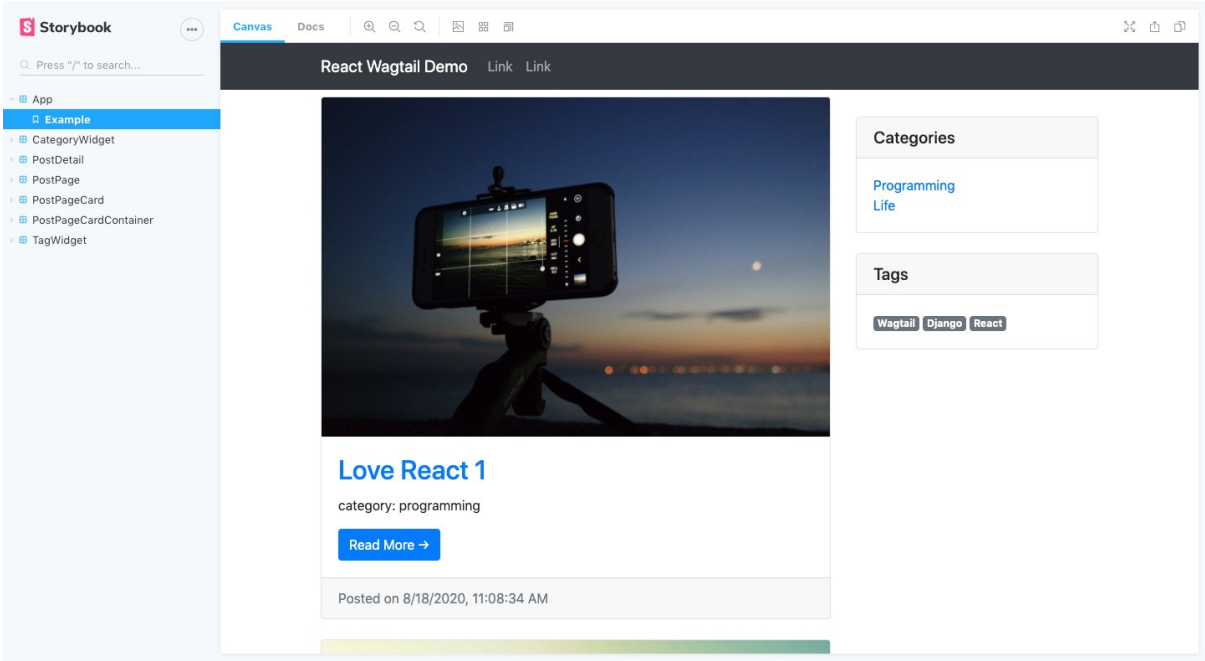
Notes:

1. We should also do the same thing on *frontend/src/stories/PostPage.stories.js*

19.7 Manual Test

Now you can check the `App` in the storybook

1. You can click the site name in the top navbar
2. You can click the `pagination` button.
3. You can click the post title to check post detail
4. You can click the `tag` link to filter the posts. (please note only `react` tag would return data)



19.8 Storybook

As you can see, we build the whole App in the storybook step by step, without sending requests to our backend.

1. Storybook provides us isolated env for our components.
2. `axios-mock-adapter` let us create mock data in very elegant way.

Chapter 20

Unittest React Component (Part 1)

20.1 Objectvie

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

1. Understand what is `jest` and the workflow
2. Use `jest` to do mock
3. Use `testing-library` to test UI interaction and asynchronous code (Ajax)
4. Learn how to do `snapshot` test

20.2 Jest

Let's first take a look at the *frontend/src/App.test.js*

You will see something like this

```
import React from 'react';
import { render } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  const { getByText } = render(<App />);
  const linkElement = getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Notes:

1. `App.test.js` means it is test file for `App.js`
2. We are using jest[54] to run test in CRA[55], `jest` is a simple `JavaScript Testing Framework` and `CRA` already config it for us, so here we can just use it directly.
3. `renders learn react link` is the name of the test.
4. The anonymous function contains the logic of the test, you can ignore the first lines here.
5. `expect(linkElement).toBeInTheDocument();` is an `assert` statement.

Let's try to run test

[54] https://jestjs.io/en/
[55] https://github.com/facebook/create-react-app

```
$ docker-compose exec frontend bash
$ yarn test

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        5.859s
Ran all test suites.
```

`yarn test` is the command to run test, the test fail becaues we already modified `App.js`, that is not a problem.

Now, please delete the *frontend/src/App.test.js* and we will add it back soon.

20.3 Testing Library

`jest` provides basic features for us to test normal JS code, if we want to test UI componetns in clean way, we still need testing-library[56]

> The @testing-library family of packages helps you test UI components in a user-centric way.

1. `@testing-library/dom` is a very light-weight solution for testing DOM nodes
2. `@testing-library/react` builds on top of DOM Testing Library by adding APIs for working with React components.

20.4 Test philosophy

Some people who are new to `testing-library` feel confused becasue they can not get the `props` and `state` of component. Let's check the words from testing-library doc[57]

> 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 intenal details.

If you want to test by checking component `props` and `state`, you can take a look at another framework Enzyme[58], but we will not use it in this course.

`CRA` already include `@testing-library/react`, so next we will use it to test our application.

20.5 Test TagWidget

The first component we built is `TagWidget`, so let's write our first test for it.

Create *frontend/src/components/TagWidget.test.js* (We can keep the test file beside the component file)

```
import React from "react";
import { render, screen, wait} from "@testing-library/react";
import { MemoryRouter } from "react-router-dom";
```

[56] https://testing-library.com/docs/
[57] https://testing-library.com/docs/
[58] https://enzymejs.github.io/enzyme/

```
import { TagWidget } from "./TagWidget";

test('render Tag widget', () => {
  render(
    <MemoryRouter>
      <TagWidget />
    </MemoryRouter>
    );
  expect(screen.getByText("Loading...")).toBeInTheDocument();
});
```

Notes:

1. Because `TagWidget` contains `Link` from `react-router-dom`, so we wrap it using `MemoryRouter` to avoid error.
2. We `render` the component, and check if there is `Loading` text in the document.

```
$ docker-compose exec frontend bash
$ yarn test

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        18.144s
Ran all test suites.
```

20.6 Test Ajax

As you know, in `TagWidget`, we use `Ajax` to query data and save it to `state` of the component.

To test the behavior, we need to do two things

1. Mock data
2. Let the test wait for the component to finish loading

As a test framework, `jest` provides a simple way for us to create `mock` function. Mock Functions[59]

And we would use `Async/Await` to make testing asynchronous code possible. Testing Asynchronous Code[60]

```
import React from "react";
import { render, screen, wait} from "@testing-library/react";
import { MemoryRouter } from "react-router-dom";
import axios from 'axios';

import { TagWidget } from "./TagWidget";

jest.mock('axios');

test('render Tag widget', async () => {
  const resp = {
    data: {
      results: [
        {
          slug: "wagtail",
          name: "Wagtail",
        },
```

[59] https://jestjs.io/docs/en/mock-functions
[60] https://jestjs.io/docs/en/asynchronous#asyncawait

```
        {
          slug: "django",
          name: "Django",
        },
        {
          slug: "react",
          name: "React",
        },
      ],
    }
  };
  axios.get.mockResolvedValue(resp);

  render(
    <MemoryRouter>
      <TagWidget />
    </MemoryRouter>
  );
  expect(screen.getByText("Loading...")).toBeInTheDocument();

  await wait(() => expect(axios.get).toHaveBeenCalled());

  await wait(() => expect(screen.getByText("Wagtail")).toBeInTheDocument());
  const el = screen.getByText("Wagtail");
  expect(el.tagName).toEqual('SPAN');
  expect(el).toHaveClass('badge badge-secondary');

  resp.data.results.map((tag) =>
    expect(screen.getByText(tag.name)).toBeInTheDocument()
  );

});
```

Notes:

1. We use `async` in front of the test function, so we can use `await` in it.
2. We use `jest.mock` to mock the `axios` modules.
3. In the `arrange` state, we create mock data and make it work by using `axios.get.mockResolvedValue`. (This would make the `axios.get` method return the mock data)
4. In `assert` stage, we use `await and wait` to let `jest` wait and keep running after `expect` statement return `true`

```
$ docker-compose exec frontend bash
$ yarn test

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        22.735s, estimated 27s
Ran all test suites.
```

20.7 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/src/components/TagWidget.test.js*

```
import React from "react";
import { render, screen, wait} from "@testing-library/react";
import { MemoryRouter } from "react-router-dom";
import axios from 'axios';

import { TagWidget } from "./TagWidget";

jest.mock('axios');

test('render Tag widget', async () => {
  const resp = {
    data: {
      results: [
        {
          slug: "wagtail",
          name: "Wagtail",
        },
        {
          slug: "django",
          name: "Django",
        },
        {
          slug: "react",
          name: "React",
        },
      ],
    }
  };
  axios.get.mockResolvedValue(resp);

  const { asFragment } = render(
    <MemoryRouter>
      <TagWidget />
    </MemoryRouter>
  );
  expect(screen.getByText("Loading...")).toBeInTheDocument();

  await wait(() => expect(axios.get).toHaveBeenCalled());

  await wait(() => expect(screen.getByText("Wagtail")).toBeInTheDocument());
  const el = screen.getByText("Wagtail");
  expect(el.tagName).toEqual('SPAN');
  expect(el).toHaveClass('badge badge-secondary');

  resp.data.results.map((tag) =>
    expect(screen.getByText(tag.name)).toBeInTheDocument()
  );

  expect(asFragment()).toMatchSnapshot();

});
```

1. `asFragment` can help us get `DocumentFragment` of our component
2. And jest check if if match the snapshot at the end of the test.

```
$ docker-compose exec frontend bash
$ yarn test

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
```

```
Time:        22.735s, estimated 27s
Ran all test suites.
```

Now you would see *frontend/src/components/**snapshots**/TagWidget.test.js.snap* is generated

And it contains something like this

```
exports[`render Tag widget 1`] = `
<DocumentFragment>
  <div
    class="card my-4"
  >
    <h5
      class="card-header"
    >
      Tags
    </h5>
    <div
      class="card-body"
    >
      <a
        href="/tag/wagtail"
      >
        <span
          class="badge badge-secondary"
        >
          Wagtail
        </span>

      </a>
      <a
        href="/tag/django"
      >
        <span
          class="badge badge-secondary"
        >
          Django
        </span>

      </a>
      <a
        href="/tag/react"
      >
        <span
          class="badge badge-secondary"
        >
          React
        </span>

      </a>
    </div>
  </div>
</DocumentFragment>
`;
```

Notes:

1. If you change HTML in `TagWidget`, the snapshot test will fail, this can make sure you would not bring unexpected change to the HTMl of the component.
2. When talking about `snapshot test for React component`, many oneline resources[61] would also

[61] https://jestjs.io/docs/en/snapshot-testing

use Test Renderer[62], you should konw it is diffrenet with `@testing-library/react`.

[62] https://reactjs.org/docs/test-renderer.html

Chapter 21

Unittest React Component (Part 2)

21.1 Objectives

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

1. Use `axios-mock-adapter` to mock requests in test.
2. Generate code coverage report

21.2 Test Filter Function

Create *frontend/src/App.test.js*

```
import React from 'react';
import { render, screen, wait, fireEvent } from "@testing-library/react";
import { within } from '@testing-library/dom';
import { MemoryRouter } from "react-router-dom";

import MockAdapter from "axios-mock-adapter";
import axios from "axios";
import { mockPost, mockCategory, mockTag } from "./stories/mockUtils";

import App from './App';


test('Test Category Link', async () => {
  const mock = new MockAdapter(axios);
  mockPost(mock);
  mockCategory(mock);
  mockTag(mock);

  render(
    <MemoryRouter initialEntries={[ '/' ]}>
      <App/>
    </MemoryRouter>,
  );

  const elTag = screen.getByText("Categories");
  expect(elTag.tagName).toEqual('H5');
  expect(elTag).toHaveClass('card-header');
  const { getByText } = within(elTag.parentNode);

  await wait(() => expect(getByText("Programming")).toBeInTheDocument());
```

```
  const el = getByText('Programming');

  fireEvent.click(el);

  await wait(() => expect(screen.getByText("Love React 1")).toBeInTheDocument());
  await wait(() => expect(screen.getByText("Love React 3")).toBeInTheDocument());

});
```

Notes:

1. Here we we use `axios-mock-adapter` to help us mock response for `axios.get`, which can make us reuse the mock data in `mockUtils`
2. The test would wait and check if there is `Programming` text appare in the Category widget. If it exists, it would click the link (`fireEvent.click(el)`)
3. After it click the link, it would wait and check there is `Love React 1` and `Love React 3` in the new page. (It would check if the category filter function work as expected.)

```
$ docker-compose exec frontend bash
$ yarn test

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   1 passed, 1 total
Time:        8.305s, estimated 36s
Ran all test suites.
```

Let's add test to *frontend/src/App.test.js*

```
test('Test Tag Link', async () => {
  const mock = new MockAdapter(axios);
  mockPost(mock);
  mockCategory(mock);
  mockTag(mock);

  render(
    <MemoryRouter initialEntries={[ '/' ]}>
      <App/>
    </MemoryRouter>,
  );

  const elTag = screen.getByText("Tags");
  expect(elTag.tagName).toEqual('H5');
  expect(elTag).toHaveClass('card-header');
  const { getByText } = within(elTag.parentNode);

  await wait(() => expect(getByText("React")).toBeInTheDocument());
  const el = getByText('React');

  fireEvent.click(el);

  await wait(() => expect(screen.getByText("Love React 2")).toBeInTheDocument());
  await wait(() => expect(screen.getByText("Love React 4")).toBeInTheDocument());

});
```

Notes:

1. The logic is very similar with the above test

```
$ docker-compose exec frontend bash
$ yarn test
```

```
Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   1 passed, 1 total
Time:        22.562s, estimated 34s
Ran all test suites.
```

21.3 Test Pagination

Let's add test to *frontend/src/App.test.js*

```
test('Test Pagination', async () => {
  const mock = new MockAdapter(axios);
  mockPost(mock);
  mockCategory(mock);
  mockTag(mock);

  render(
    <MemoryRouter initialEntries={[ '/' ]}>
      <App/>
    </MemoryRouter>,
  );

  await wait(() => expect(screen.getByText("Love React 1")).toBeInTheDocument());

  const el = screen.getByText("Next");

  fireEvent.click(el);

  await wait(() => expect(screen.getByText("Love React 3")).toBeInTheDocument());
  await wait(() => expect(screen.getByText("Love React 4")).toBeInTheDocument());

});
```

```
$ docker-compose exec frontend bash
$ yarn test

Test Suites: 2 passed, 2 total
Tests:       4 passed, 4 total
Snapshots:   1 passed, 1 total
Time:        26.904s, estimated 37s
Ran all test suites.
```

21.4 Test PostPage

Let's add test to *frontend/src/App.test.js*

```
test('Check Post Link', async () => {
  const mock = new MockAdapter(axios);
  mockPost(mock);
  mockCategory(mock);
  mockTag(mock);

  render(
    <MemoryRouter initialEntries={[ '/' ]}>
      <App/>
```

```
    </MemoryRouter>,
  );

  await wait(() => expect(screen.getByText("Love React 1")).toBeInTheDocument());

  // click post link
  const el = screen.getByText("Love React 1");
  fireEvent.click(el);

  // check if content can render
  await wait(() => expect(screen.getByText("The Zen of Wagtail")).toBeInTheDocument());

});
```

Notes:

1. Here we wail for the `Love React 1` appare on the page and click the link.
2. And then we check if the `PostDetail` page is working as expected.

```
$ docker-compose exec frontend bash
$ yarn test

Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   1 passed, 1 total
Time:        23.821s
Ran all test suites.
```

21.5 Test Coverage

Test coverage report can print stats about the test, which can give us confidence.

Notes: If you see `Nothing was returned from render. This usually means a return statement is missing` error, please find solution in *Frontend FAQ*

```
$ npm run test -- --coverage --watchAll=false


-----------------------------------|----------|----------|----------|----------|-------------------|
File                               |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-----------------------------------|----------|----------|----------|----------|-------------------|
All files                          |    56.12 |    48.48 |    62.86 |     55.9 |                   |
 src                               |     2.33 |        0 |     5.56 |     2.33 |                   |
  App.js                           |       50 |      100 |       50 |       50 |                17 |
  index.js                         |        0 |      100 |      100 |        0 |              7,17 |
  serviceWorker.js                 |        0 |        0 |        0 |        0 |... 32,133,135,138 |
 src/components                    |      100 |      100 |      100 |      100 |                   |
  BlogPage.js                      |      100 |      100 |      100 |      100 |                   |
  CategoryWidget.js                |      100 |      100 |      100 |      100 |                   |
  Footer.js                        |      100 |      100 |      100 |      100 |                   |
  PostDetail.js                    |      100 |      100 |      100 |      100 |                   |
  PostPage.js                      |      100 |      100 |      100 |      100 |                   |
  PostPageCard.js                  |      100 |      100 |      100 |      100 |                   |
  PostPageCardContainer.js         |      100 |      100 |      100 |      100 |                   |
  SideBar.js                       |      100 |      100 |      100 |      100 |                   |
  TagWidget.js                     |      100 |      100 |      100 |      100 |                   |
  TopNav.js                        |      100 |      100 |      100 |      100 |                   |
 src/components/StreamField        |    91.67 |    85.71 |      100 |     91.3 |                   |
  ImageCarousel.js                 |      100 |      100 |      100 |      100 |                   |
```

```
  ImageText.js                      |      100 |      100 |      100 |      100 |                   |
  StreamField.js                    |    89.47 |    83.33 |      100 |    88.89 |             14,35 |
  ThumbnailGallery.js               |      100 |      100 |      100 |      100 |                   |
 src/stories                        |    32.26 |      100 |       25 |    32.26 |                   |
  App.stories.js                    |        0 |      100 |        0 |        0 | 16,17,18,19,20,22 |
  CategoryWidget.stories.js         |        0 |      100 |        0 |        0 |       15,16,17,19 |
  PostDetail.stories.js             |        0 |      100 |        0 |        0 |       16,17,18,20 |
  PostPage.stories.js               |        0 |      100 |        0 |        0 |... 31,32,33,34,36 |
  PostPageCard.stories.js           |        0 |      100 |        0 |        0 |       15,16,17,19 |
  PostPageCardContainer.stories.js  |        0 |      100 |        0 |        0 |... 22,36,37,38,40 |
  TagWidget.stories.js              |        0 |      100 |        0 |        0 |       15,16,17,19 |
  mockUtils.js                      |      100 |      100 |      100 |      100 |                   |
-----------------------------------|----------|----------|----------|----------|-------------------|
```

Chapter 22

Integrate Frontend App with REST API

22.1 Objective

In this chapter, we will config to make our frontend app work with our REST API.

22.2 Index.js

Considering we already have:

1. REST API which is working and the media files can be fetched through Django dev server.
2. `App.js` which can work with the REST API.

Now let's update `app/frontend/src/index.js`, which is the entry file of our frontend APP.

```
import React from "react";
import ReactDOM from "react-dom";
import { BrowserRouter } from "react-router-dom";
import App from "./App";
import "./index.scss";

ReactDOM.render(
  <React.StrictMode>
    <BrowserRouter>
      <App/>
    </BrowserRouter>
  </React.StrictMode>,
  document.getElementById("root")
);
```

1. `index.scss` contains style code for our app (for now, it only contains `bootstrap`)
2. We use `BrowserRouter` to wrap our `App` component instead of `MemoryRouter`, so you can check the route change through the address bar of web browser.
3. We use `ReactDOM.render` to render component into the DOM element.

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

If you check http://127.0.0.1:3000/ in your browser, you will see

1. The frontend app `can not` fetch the data from REST API.
2. If you check in `devtools`, you will see the AJAX requests sent to `http://127.0.0.1:3000` all get `404` error.

But wait, the REST API is working on `http://127.0.0.1:8000`, the port number is not correct here.

Let's fix it.

22.3 Proxying API Requests

Some people might think we can add domain and port number to the `axios` request to make it work, but there is another way to solve this in an elegant way.

From CRA Doc[63]

> To tell the development server to proxy any unknown requests to your API server in development, add a `proxy` field to your package.json

Let's update *frontend/package.json*

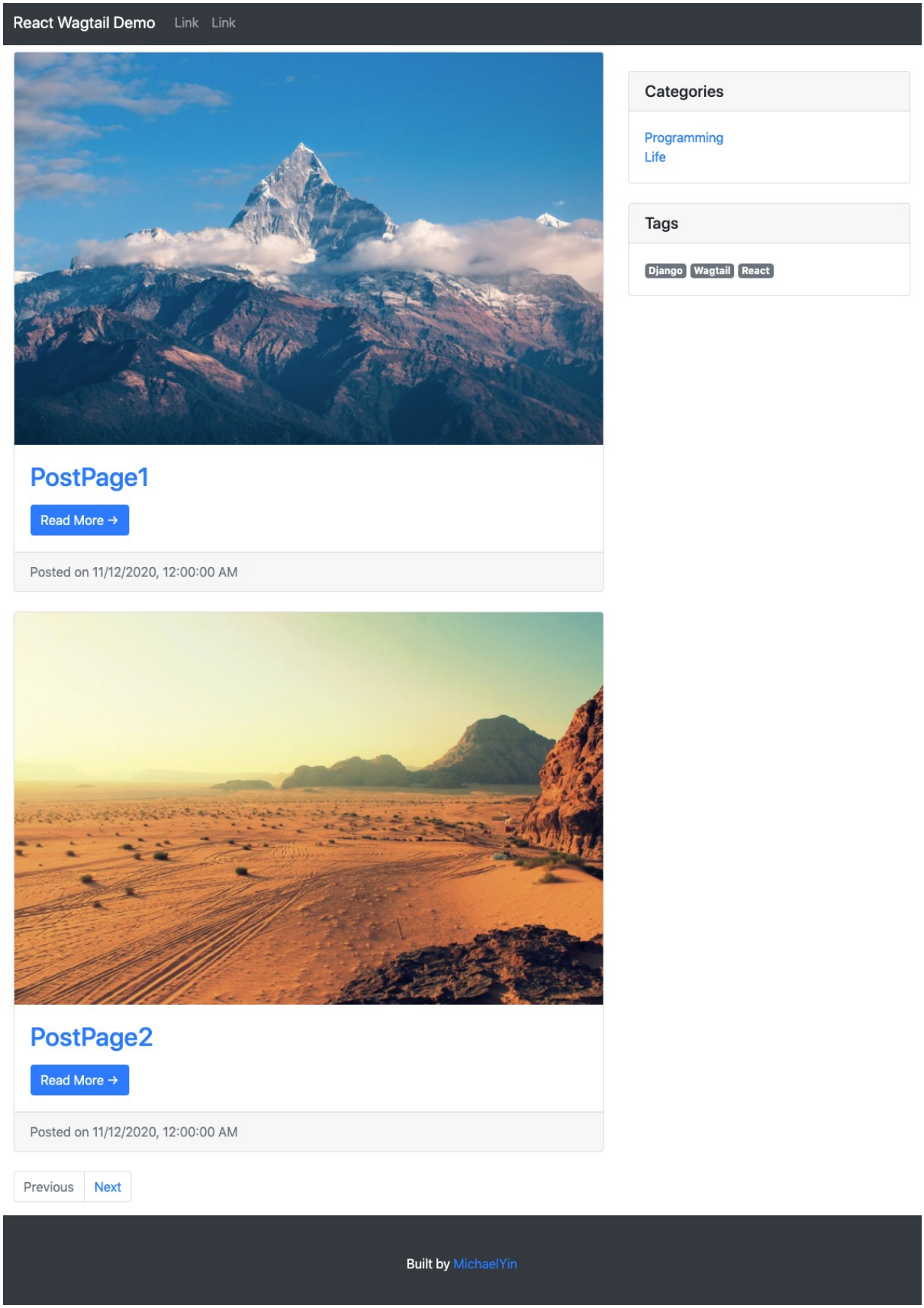
```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject",
  "storybook": "start-storybook -p 6006 -s public",
  "build-storybook": "build-storybook -s public"
},
"proxy": "http://web:8000",
"eslintConfig": {
  "extends": "react-app"
},
```

As you can see, we add `proxy` which has value `http://web:8000`, and the CRA dev server would proxy the REST API request to `http://web:8000` and we can still write relative url in our frontend code, which is clean.

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

Now if you check http://127.0.0.1:3000/ you will see it can work without problem.

[63] https://create-react-app.dev/docs/proxying-api-requests-in-development/


React Wagtail Demo
Link
Link
Categories
Programming
Life
Tags
Django
Wagtail
React
PostPage1
Read More →
Posted on 11/12/2020, 12:00:00 AM
PostPage2
Read More →
Posted on 11/12/2020, 12:00:00 AM
Previous
Next
Built by MichaelYin

Chapter 23

Add Preview Support to React project

23.1 Objective

1. Make Wagtail preview work with the frontend app.

23.2 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 frontend app, the url contains the `token` in querystring.
3. The frontend app use the `token` in querystring to send Ajax request to get the `draft page content`, and display it.

So this is the basic workflow of `wagtail-headless-preview` and let's make it work in our project.

23.3 Wagtail headless preview

Update *requirements.txt*

```
# other packages
wagtail-headless-preview==0.1.4
```

Update *react_wagtail_app/settings.py*

```
INSTALLED_APPS = [
    # other packages
    "wagtail_headless_preview",
]


HEADLESS_PREVIEW_CLIENT_URLS = {
    "default": "http://localhost:3000/",
}
```

Notes:

1. In `HEADLESS_PREVIEW_CLIENT_URLS` we tell `wagtail-headless-preview` the domain of frontend app.

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


class BlogPage(BasePage):
    # for brevity


class PostPage(BasePage):
    # for brevity

    def get_preview_url(self, token):
        return urllib.parse.urljoin(
            self.get_client_root_url(),
            f"post/{self.pk}/"
            + "?"
            + urllib.parse.urlencode(
                {"content_type": self.get_content_type_str(), "token": token}
            ),
        )
```

Notes:

1. We create a `BasePage` class, which inherit `HeadlessPreviewMixin`.
2. `PostPage` and `BlogPage` inherit the above `BasePage`.
3. We overwrite `get_preview_url` method to generate the `preview` post url according to the `React Routes` in our frontend app.
4. `get_client_root_url` would return the value we defined in `HEADLESS_PREVIEW_CLIENT_URLS` ()
5. So if the post has pk `4`, the `get_preview_url` would return something like `http://localhost:3000/post/4/?content_type=blog.postpage&token=xxxxxxx`, people can only view the `preview` content if the `token` is correct.

23.4 Rest API

Update *blog/api.py* to consume the `?content_type=blog.postpage&token=xxxxxxx`

```
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()
        serializer = self.get_serializer(page)
        return Response(serializer.data)

    def detail_view(self, request, pk):
```

```
        page = self.get_object()
        serializer = self.get_serializer(page)
        return Response(serializer.data)

    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


cms_api_router = WagtailAPIRouter("wagtailapi")
cms_api_router.register_endpoint("pages", PagesAPIViewSet)
cms_api_router.register_endpoint("images", ImagesAPIViewSet)
cms_api_router.register_endpoint("documents", DocumentsAPIViewSet)
cms_api_router.register_endpoint("page_preview", PagePreviewAPIViewSet)
```

Notes:

1. We create an endpoint `api/cms/page_preview`
2. 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
$ docker-compose exec web bash
(container) $ ./manage.py migrate
```

23.5 PostDetail Component

Update *frontend/src/components/PostDetail.js*

```
class PostDetail extends React.Component {

  componentDidMount() {
    const pk = this.props.match.params.id;

    // convert querystring to dict
    const querystring = this.props.location.search.replace(/^\?/, '');
    const params = {};
    querystring.replace(/([^=&]+)=([^&]*)/g, function (m, key, value) {
      params[decodeURIComponent(key)] = decodeURIComponent(value);
    });

    if (params.token) {
      // preview
      axios.get(`/api/cms/page_preview/${pk}/${this.props.location.search}`)
        .then((res) => {
```

```
          const post = res.data;
          this.setState({
            post,
            loading: false
          });
        });
    } else {
      axios.get(`/api/cms/pages/${pk}/`).then((res) => {
        const post = res.data;
        this.setState({
          post,
          loading: false
        });
      })
    }
  }

}
```

Notes:

1. As we know, `this.props.location.search` contains info about the querystring of URL.
2. We check if querystring contains `token`, if it does, we send AJAX request to `api/cms/page_preview`, the request URL contains `content_type` and `token` passed by Wagtail admin.

Now if you check `draft` in Wagtail admin, it can work with our `React app`

23.6 Live view

If you click `View live` button, you will get `template does not exist` error, let's fix it to make it work.

```
from django.http.response import HttpResponseRedirect


class BlogPage(BasePage):

    def serve(self, request, *args, **kwargs):
        return HttpResponseRedirect(self.get_client_root_url())


class PostPage(BasePage):

    def serve(self, request, *args, **kwargs):
        return HttpResponseRedirect(
            urllib.parse.urljoin(self.get_client_root_url(), f"/post/{self.pk}")
        )
```

Notes:

1. In `serve` method, we return `HttpResponseRedirect` to redirect user to the relevant frontend url.

23.7 Conclusion

1. The `serve` method can help redirect visitor to relevant frontend component.
2. The `HeadlessPreviewMixin.get_preview_url` can help pass `token` and `content_type` to the frontend app, and then frontend app can use them to send AJAX to REST API to fetch `preview page data`

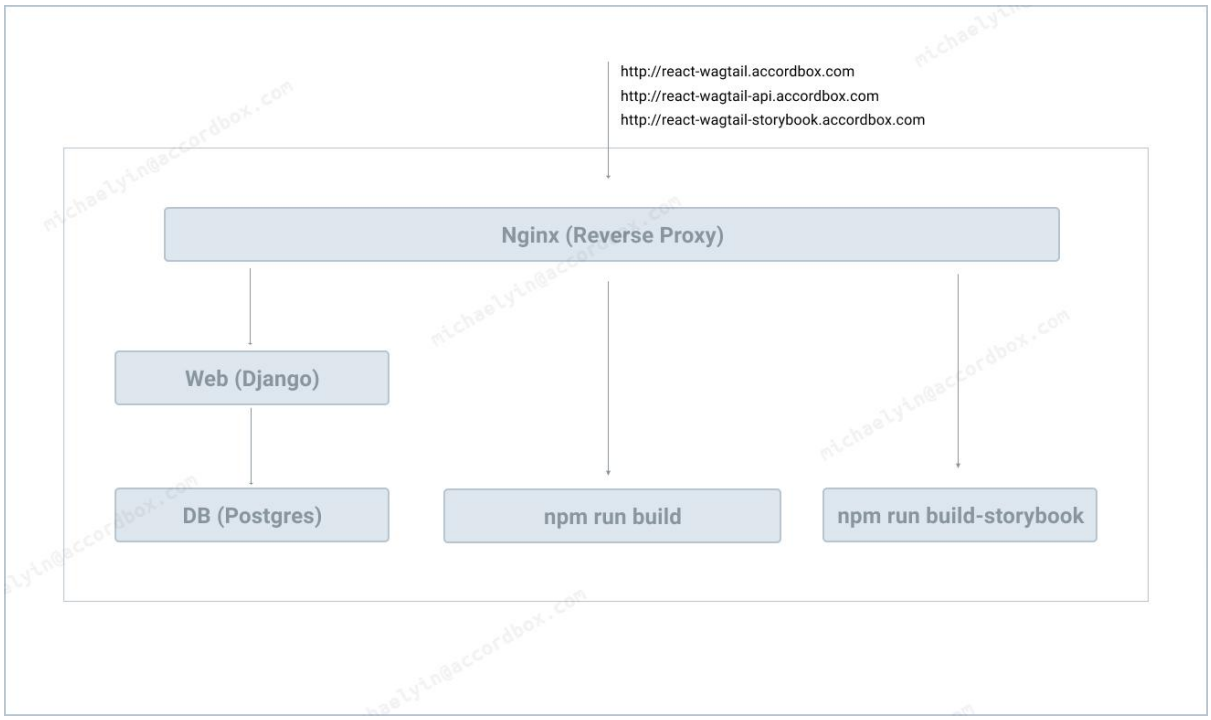
Chapter 24

Deploy REST API

24.1 Objective

In this chapter, we will learn how to deploy our REST API to DigitalOcean[64] with Docker Compose.

24.2 Workflow



24.3 Compose File

Let's create *docker-compose.prod.yml*, the `prod` means this compose file is for our `production` app.

[64] https://www.digitalocean.com/

```yaml
version: '3.7'

services:

  nginx:
    build:
      context: .
      dockerfile: ./compose/production/nginx/Dockerfile
    volumes:
      - staticfiles:/app/static
      - mediafiles:/app/media
    ports:
      - 80:80
    depends_on:
      - web

  web:
    build:
      context: .
      dockerfile: ./compose/production/django/Dockerfile
    command: /start
    volumes:
      - staticfiles:/app/static
      - 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:
  staticfiles:
  mediafiles:
```

Notes:

1. Here we create 3 services, `nginx` is `reverse proxy` for `web` service, we only need to expose `80` port for `nginx`
2. For `web` service, we created `staticfiles` and `mediafiles` docker volumn to store the assets.
3. All env variables are stored in `.env/.prod-sample`

Create `production` directory under the `compose`, and then create sub directory `django` and `nginx`.

So you would have file structure like this.

```
├── compose
|   ├── local
|   |   ├── django
|   |   └── node
|   └── production
|       ├── django
|       └── nginx
```

24.4 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*:

```
upstream hello_django {
    server web:8000;
}


server {
    listen 80;
    location / {
        proxy_pass http://hello_django;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
        client_max_body_size 20M;
    }
    location /static/ {
        alias /app/static/;
    }
    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/static/` and `/app/media/` point to the docker volume `staticfiles` and `mediafiles`, where Nginx would find files.

24.5 Web Service

Before we start, let's take a look at the file strcutures

```
├── compose
|   └── production
|       ├── django
|       |   ├── Dockerfile
|       |   ├── entrypoint
|       |   └── start
```

24.5.1 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/static
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/static`, `RUN mkdir /app/media` combined with `RUN chown -R django:django /app` to solve the `permission denied` problem.

24.5.2 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[65].

24.5.3 Start script

Update *compose/production/django/start*

```
#!/bin/bash

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

python /app/manage.py collectstatic --noinput
python /app/manage.py migrate

/usr/local/bin/gunicorn react_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[66]
2. We use gunicorn to run Django app

[65] https://stackoverflow.com/a/39082923/2371995
[66] https://docs.djangoproject.com/en/3.1/howto/static-files/deployment/

24.6 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
```

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

Update *react_wagtail_app/settings.py* to read the above `SECRET_KEY`, `DEBUG` and `ALLOWED_HOSTS` environment variables:

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

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = int(os.environ.get("DEBUG", default=1))

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

24.6.1 Config

Update the static and media file config in *react_wagtail_app/settings.py*:

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

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

`STATIC_URL` and `MEDIA_URL` should match the above Nginx `location` config.

Add `gunicorn` to *requirements.txt*

```
django==3.1
wagtail==2.10.2
wagtail-headless-preview==0.1.4
psycopg2-binary
djangorestframework

factory-boy==2.12.0
wagtail-factories==2.0.0
coverage

gunicorn
```

24.7 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 react-wagtail-prod build
$ docker-compose -f docker-compose.prod.yml -p react-wagtail-prod up -d
$ docker-compose -f docker-compose.prod.yml -p react-wagtail-prod logs -f

web_1    | [INFO] Starting gunicorn 20.0.4
web_1    | [INFO] Listening at: http://0.0.0.0:8000 (12)
web_1    | [INFO] Using worker: sync
web_1    | [INFO] Booting worker with pid: 14
```

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 react-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[67] for more details.
3. If you visit http://localhost/cms-admin in your browser, you will see Wagtail admin login page. This means the setup was correct.

24.8 Docker Ignore

Next, let's check the source code in the web container

```
$ docker-compose -f docker-compose.prod.yml -p react-wagtail-prod run --rm web bash

(container)$ ls media
images  original_images
```

Here we see some problems, the `media` directory was copied to `/app` when docker build image.

So we should tell docker to `ignore` it. dockerignore-file[68] can help us solve this problem

Create *.dockerignore*

```
db.sqlite3
node_modules
/media
```

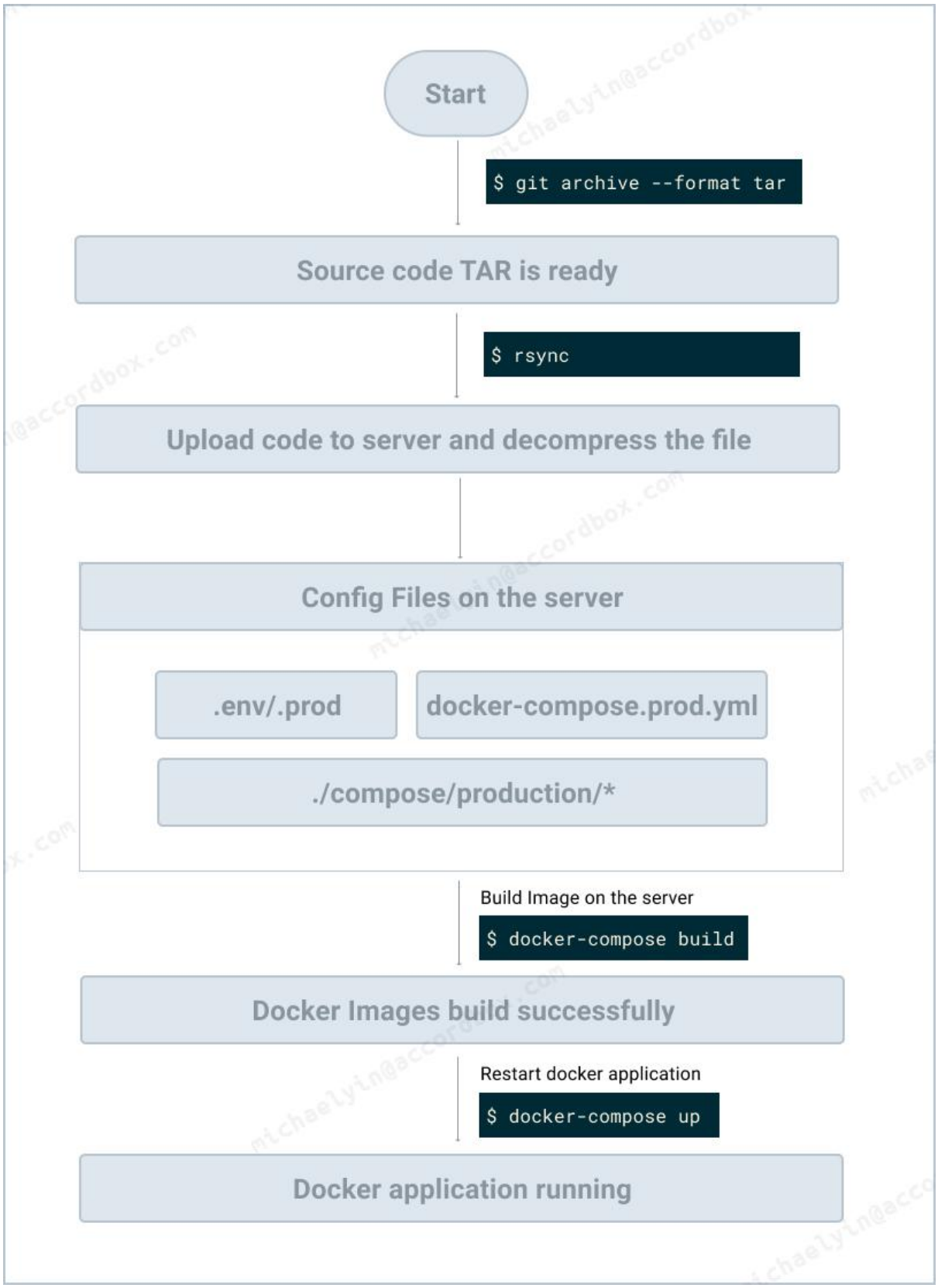
Now if you build the image, rerun the application and check, `media` directory should not contains the local media files.

```
# cleanup
$ docker-compose -f docker-compose.prod.yml -p react-wagtail-prod stop
# delete containers and volumes
$ docker-compose -f docker-compose.prod.yml -p react-wagtail-prod down -v
$ docker-compose -f docker-compose.prod.yml -p react-wagtail-prod ps
```

[67] https://docs.docker.com/compose/reference/envvars/#compose_project_name
[68] https://docs.docker.com/engine/reference/builder/#dockerignore-file

```
Name   Command   State   Ports
-----------------------------
```

24.9 Deploy to DigitalOcean



In this section:

1. `(local)$` means that the command should be ran on your local environment

2. `(server)$` means that the command should be ran on the remote server.

24.9.1 Server Setup

First, sign up for a DigitalOcean account[69] (if you don't already have one), and then generate[70] 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[71], 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":"react-wagtail-project","region":"sfo2","size":"s-2vcpu-4gb","image":"docker-20-04"}' \
    "https://api.digitalocean.com/v2/droplets"

# check status
curl \
    -H 'Content-Type: application/json' \
    -H 'Authorization: Bearer '$DIGITAL_OCEAN_ACCESS_TOKEN'' \
    "https://api.digitalocean.com/v2/droplets?react-wagtail-project"
```

Notes:

1. We use `react-wagtail-project` as the `Droplet` name, you can modify it.
2. When the `Droplet` is available, you should receive an email which contains the login credentials.

24.9.2 Config SSH

Notes: This section assume you have no experience with SSH, and it would help you get it work quicikly.

```
(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
```

[69] https://m.do.co/c/b585bd8722ec
[70] https://www.digitalocean.com/docs/apis-clis/api/
[71] https://marketplace.digitalocean.com/apps/docker

```
VXL9Pdbo8X7PtCmvdD/lvuhcg8iFhwJR8YqxeZhRvds5PzwIhYx9/n7f3y6goR0s
8J71z47xZs6phQD96o3dG692E8gUBbt525p08+ysOQBLbv8DTdv0xoCOkV83I2z1
...
-----END RSA PRIVATE KEY-----'
```

Add the key to your ssh-agent[72]:

```
(local)$ ssh-add - <<< "${PRIVATE_KEY}"

Identity added
```

To test, run:

```
(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[73]
2. To keep your server secure, if you can log in using the SSH private key, you should disable SSH password login.

24.9.3 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:

[72] https://en.wikipedia.org/wiki/Ssh-agent
[73] 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

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...
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 http://<YOUR_INSTANCE_IP>/cms-admin/[74], 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.

24.9.4 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/react-wagtail-project.conf*:

```
[program:react-wagtail-project]
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
react-wagtail-project                        STARTING
```

[74] http://%3CYOUR_INSTANCE_IP%3E/cms-admin/

Now the containers should automatically run on boot. We can also restart the containers via the `supervisorctl restart react-wagtail-project` 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 react-wagtail-project
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...

...

react-wagtail-project: stopped
react-wagtail-project: started
Build complete.
```

24.10 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 http://<YOUR_INSTANCE_IP>/cms-admin/[75] and setup your site, upload images and create pages.

[75] http://%3CYOUR_INSTANCE_IP%3E/cms-admin/

24.11 Config DNS

To make the site work with domain, let's create DNS records to point to the IP address of the server.

Here I config `react-wagtail-api.accordbox.com` (you need change it) to point to the IP of the server, after waiting for some minutes, let's run test.

```
$ curl http://react-wagtail-api.accordbox.com/

<!DOCTYPE HTML>
<html>
    <head>
        <title>Welcome to your new Wagtail site!</title>
    </head>
    <body>
        <h1>Welcome to your new Wagtail site!</h1>
    </body>
</html>
```

Please remember to config `Wagtail site hostname` to make the generated url from REST API has the correct hostname.

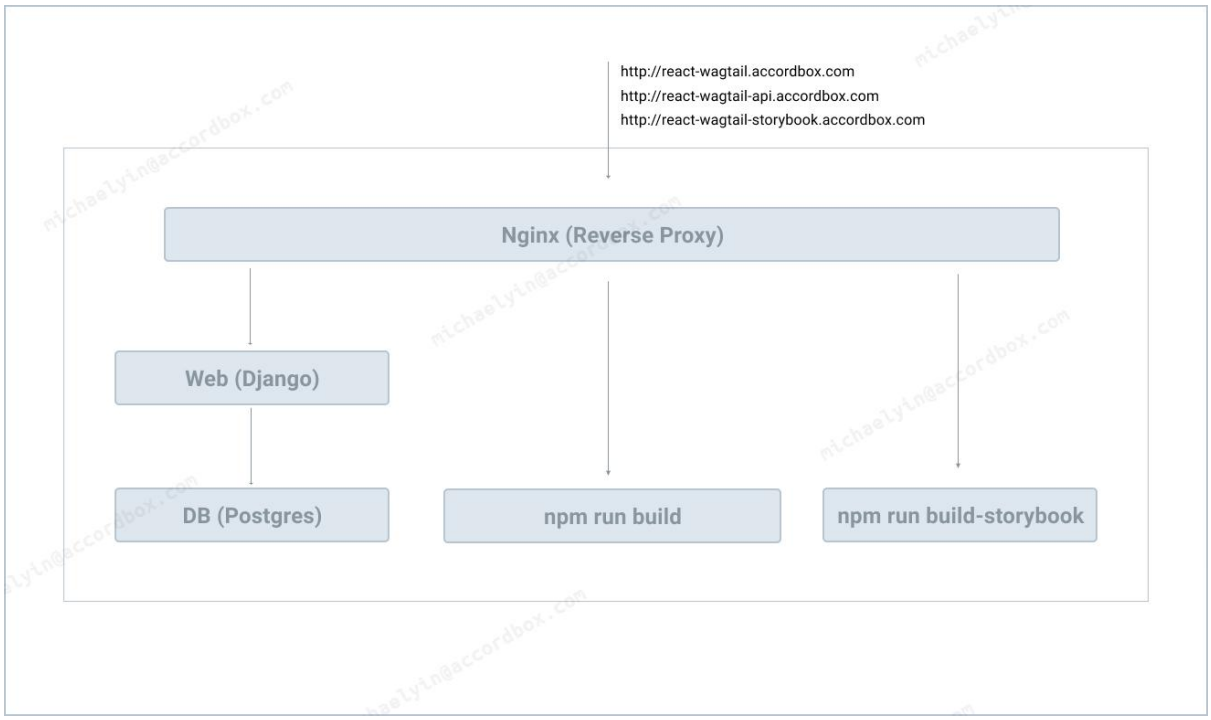
Chapter 25

Deploy Storybook

25.1 Objectives

In this chapter, we will learn how to deploy Storybook to DigitalOcean[76] with Docker Compose.

25.2 Workflow



25.3 Config DNS

Before we start, config `react-wagtail-storybook.accordbox.com` (please use a different domain you own) to point it to the IP of the server. So we do not have to wait after a while.

[76] https://www.digitalocean.com/

25.4 Workflow

Below is our workflow to deploy the Storybook.

1. Install the frontend dependency packages and build the storybook using `yarn build-storybook`
2. Copy the built static files to nginx image.
3. Use nginx to serve the storybook site just like a static website.

25.5 DockerFile

Before we start, let's take a look at docker multi-stage builds[77]

> With multi-stage builds, you use multiple FROM statements in your Dockerfile. Each FROM instruction can use a different base, and each of them begins a new stage of the build. You can selectively copy artifacts from one stage to another, leaving behind everything you don't want in the final image.

What we want is the final static storybook assets, not packages in the `node_modules`,

Let's update *compose/production/nginx/Dockerfile*

```
FROM node:12-stretch-slim as frontend-builder

WORKDIR /app/frontend

COPY ./frontend/package.json /app/frontend
# COPY ./frontend/yarn.lock /app/frontend
COPY ./frontend/package-lock.json /app/frontend

ENV PATH ./node_modules/.bin/:$PATH

# RUN yarn install
RUN npm install

COPY ./frontend .

# build storybook
RUN yarn build-storybook

##################################################

FROM nginx:1.19.2-alpine

RUN rm /etc/nginx/conf.d/default.conf
COPY ./compose/production/nginx/nginx.conf /etc/nginx/conf.d
# copy storybook
COPY --from=frontend-builder /app/frontend/storybook-static /usr/share/nginx/html/storybook-static
```

Notes:

1. For the first build stage, we assign it a name `frontend-builder`
2. In the first build stage, we install frontend denendpency packages and use `yarn build-storybook` to build `storybook`, after this command is finished, the built assets would be available in `/app/frontend/storybook-static`
3. In the second build stage, when building image for nginx service, we copy `storybook-static` from the first stage and put it in `/usr/share/nginx/html/storybook-static`

[77] https://docs.docker.com/develop/develop-images/multistage-build/#use-multi-stage-builds

4. `--from` has the value of the `build stage`, here the value is `frontend-builder`, which we set in `FROM node:12-stretch-slim as frontend-builder`
5. Next we can config nginx serve storybook like normal static site (HTML, JS and other assets).

25.6 Nginx

Let's update *compose/production/nginx/nginx.conf*

```
upstream hello_django {
    server web:8000; }

server {
    listen 80;
    server_name react-wagtail-api.accordbox.com;
    location / {
        proxy_pass http://hello_django;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
        client_max_body_size 20M;
    }
    location /static/ {
        alias /app/static/;
    }
    location /media/ {
        alias /app/media/;
    }
}

server {
    listen 80;
    server_name react-wagtail-storybook.accordbox.com;
    location / {
      root   /usr/share/nginx/html/storybook-static;
      index  index.html index.htm;
      try_files $uri $uri/ /index.html;
    }
}
```

Notes:

1. We add `server_name` to the nginx `location` so multiple sites can work on the same `80` port.
2. The REST API has domain `react-wagtail-api.accordbox.com`
3. the storybook has domain `react-wagtail-storybook.accordbox.com`
4. `/usr/share/nginx/html/storybook-static` contains the built asstes of storybook, we copy the files during the `docker build` stage.

25.7 Deploy

Now, please `git add` and `git commit` the above files, and then deploy to the server.

```
$ bash compose/auto_deploy_do.sh
```

If you visit http://react-wagtail-storybook.accordbox.com, you will see storybook of our application. Where you can check UI of our components.

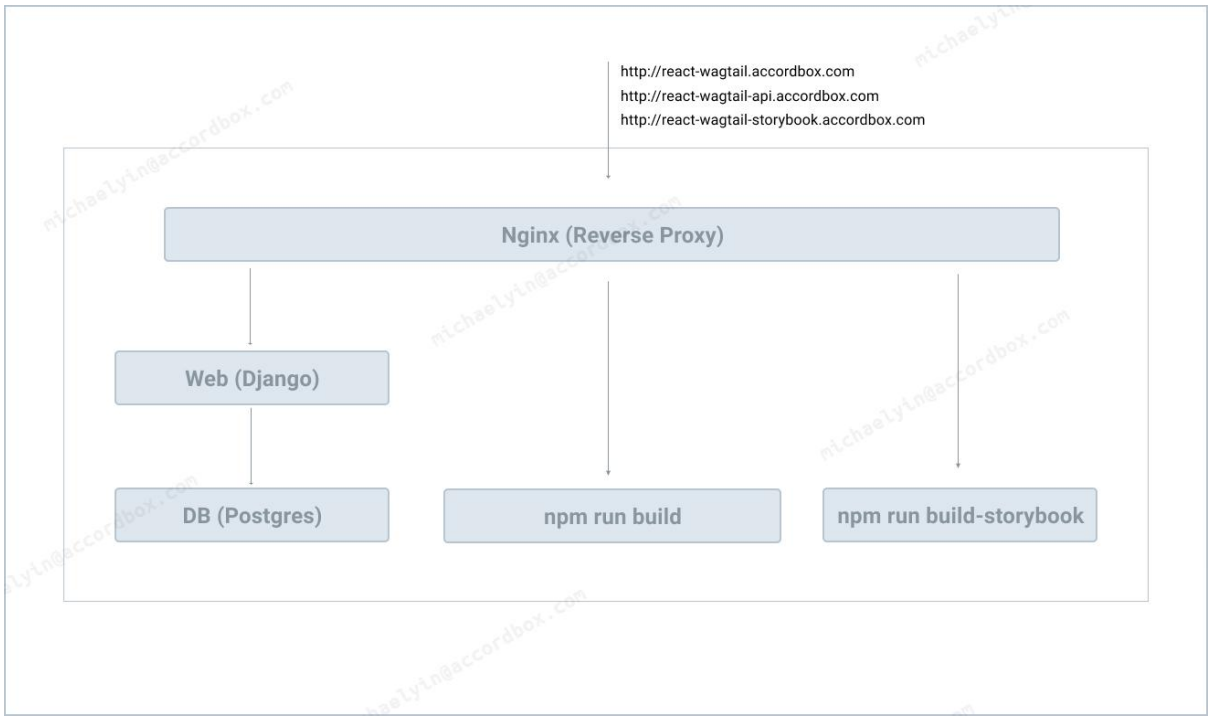
Chapter 26

Deploy React app

26.1 Objective

In this chapter, we will learn how to deploy React App to DigitalOcean[78] with Docker Compose.

26.2 Workflow



26.3 Config DNS

Before we start, please config `react-wagtail.accordbox.com` (please use a different domain you own) to point it to the IP of the server. So we do not have to wait after a while.

[78] https://www.digitalocean.com/

26.4 DockerFile

We already understand what is docker multi-stage builds[79], let's keep using it to deploy our `frontend` app.

Update *compose/production/nginx/Dockerfile*

```
FROM node:12-stretch-slim as frontend-builder

WORKDIR /app/frontend

COPY ./frontend/package.json /app/frontend
# COPY ./frontend/yarn.lock /app/frontend
COPY ./frontend/package-lock.json /app/frontend

ENV PATH ./node_modules/.bin/:$PATH

# RUN yarn install
RUN npm install

COPY ./frontend .

# build storybook
RUN yarn build-storybook

# build frontend app
RUN yarn build

##############################################

FROM nginx:1.19.2-alpine

RUN rm /etc/nginx/conf.d/default.conf
COPY ./compose/production/nginx/nginx.conf /etc/nginx/conf.d
# copy storybook
COPY --from=frontend-builder /app/frontend/storybook-static /usr/share/nginx/html/storybook-static
# copy the frontend build
COPY --from=frontend-builder /app/frontend/build /usr/share/nginx/html/build
```

Notes:

1. We run `yarn build` to build `frontend` app after `yarn build-storybook`
2. After `frontapp` is built, we copy the static assets to `/usr/share/nginx/html/build` of nginx image.

26.5 Nginx

Update *compose/production/nginx/nginx.conf*

```
upstream hello_django {
    server web:8000;
}

server {
    listen 80;
    server_name react-wagtail-api.accordbox.com;
    location / {
        proxy_pass http://hello_django;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

[79] https://docs.docker.com/develop/develop-images/multistage-build/#use-multi-stage-builds

```
        proxy_set_header Host $host;
        proxy_redirect off;
        client_max_body_size 20M;
    }
    location /static/ {
        alias /app/static/;
    }
    location /media/ {
        alias /app/media/;
    }
}

server {
    listen 80;
    server_name react-wagtail-storybook.accordbox.com;
    location / {
      root   /usr/share/nginx/html/storybook-static;
      index  index.html index.htm;
      try_files $uri $uri/ /index.html;
    }
}

server {
    listen 80;
    server_name react-wagtail.accordbox.com;
    location / {
      root   /usr/share/nginx/html/build;
      index  index.html index.htm;
      try_files $uri $uri/ /index.html;
    }
}
```

Notes:

1. We need `try_files $uri $uri/ /index.html;` to make `react-router` work with url after browser refresh. Github issue[80]

Now, please `git add` and `git commit` the above files, and then deploy to the server.

```
$ bash compose/auto_deploy_do.sh
```

If you visit http://react-wagtail.accordbox.com, you will see the `Ajax` requests all fail.

Let's figure out why this happen.

26.6 API Domain

When we develop the frontend app on local env, we learned we can Proxying API Requests in Development[81] by adding `proxy` field to our `package.json`

For production apps, `frontend` app and `REST API` app are deployed on different domains, so we should tell `frontend` app to send Ajax request to the `REST API` app.

`Create React App` has already provided solution for us Adding Custom Environment Variables[82]

1. We can add the REST API domain to the ENV file
2. `CRA` would write the value to the final bundles when building. (This happen in `docker build` stage)

[80] https://github.com/react-boilerplate/react-boilerplate/issues/1480
[81] https://create-react-app.dev/docs/proxying-api-requests-in-development/
[82] https://create-react-app.dev/docs/adding-custom-environment-variables/

Create *frontend/.env.production*

```
REACT_APP_API_URL=http://react-wagtail-api.accordbox.com
```

Here we add a env `REACT_APP_API_URL`

And then let's update *frontend/src/index.js*

```
import React from "react";
import ReactDOM from "react-dom";
import axios from 'axios';
import { BrowserRouter } from "react-router-dom";
import App from "./App";
import "./index.scss";

if (process.env.REACT_APP_API_URL) {
  axios.defaults.baseURL = process.env.REACT_APP_API_URL;
}

ReactDOM.render(
  <React.StrictMode>
    <BrowserRouter>
      <App/>
    </BrowserRouter>
  </React.StrictMode>,
  document.getElementById("root")
);
```

1. If the `REACT_APP_API_URL` is defined in env, we would set it to `axios.defaults.baseURL`
2. So this would work on all `axios` requests (because we set it in the top of `index.js`)

26.7 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

By default, if your Django app is serving on `react-wagtail-api.accordbox.com`, the Ajax requests from `react-wagtail.accordbox.com` would get `forbidden` error because of Same-origin policy[83]

We can change this behavior by using django-cors-headers[84]

Add it to *requirements.txt*

```
django==3.1
wagtail==2.10.2
wagtail-headless-preview==0.1.4
psycopg2-binary
djangorestframework

factory-boy==2.12.0
wagtail-factories==2.0.0
coverage

gunicorn
django-cors-headers
```

Update *react_wagtail_app/settings.py*

[83] https://en.wikipedia.org/wiki/Same-origin_policy
[84] https://github.com/adamchainz/django-cors-headers

```
INSTALLED_APPS = [
    # code omitted for brevity
    "corsheaders",
]


MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'corsheaders.middleware.CorsMiddleware',
    # code omitted for brevity
]

CORS_ORIGIN_ALLOW_ALL = True
```

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 change to only allow specific origins later)

Now, please `git add` and `git commit` the above files, and then deploy to the server.

```
$ bash compose/auto_deploy_do.sh
```

To quickly check if `django-cors-headers` is working as expected. We can test on local env by using command below

```
$ curl -H "Origin: http://react-wagtail.accordbox.com" --verbose \
  http://react-wagtail-api.accordbox.com/api/cms/pages/

# will see response header like this if django-cors-headers is working

Access-Control-Allow-Origin: *
```

If we visit http://react-wagtail.accordbox.com, we will see the `Ajax` request now is working as expected.

But we also find one problem, the images can not display.

If we check them in `devtool`, we see the problem.

1. The JSON data from the `REST API` has `relative image url`
2. So the web browser would try to download the image from the `react-wagtail.accordbox.com/media` instead of the `react-wagtail-api.accordbox.com/media`

Let's fix it in the next section.

26.8 Media Domain

By default, if we use Django's default file storage (`django.core.files.storage.FileSystemStorage`), the image url would be a relative url like this `/media/images/photo-1506765515384-028b60a970df.width-800.jpg`

The image url does not have `domain` value, which might cause the image not load in some cases.

There are some ways to solve this problem.

1. Use `Object Storage` such as `AWS S3` for production site, and the image url would be `absolute url`, this is the most recommended solution.
2. Modify `serializer` to convert the relative url to absolute url in JSON data.

3. Create some rules to rewrite the URL. (For examle, in Nginx)

Here I chose to use Nginx config because it is simple in this case.

26.9 Nginx

Update *compose/production/nginx/nginx.conf*

```
server {
    listen 80;
    server_name react-wagtail.accordbox.com;
    location / {
      root   /usr/share/nginx/html/build;
      index  index.html index.htm;
      try_files $uri $uri/ /index.html;
    }

    location /media/ {
      return 301 http://react-wagtail-api.accordbox.com$request_uri;
    }
}
```

Notes:

1. On `react-wagtail.accordbox.com` site, all `media` requests `http://react-wagtail.accordbox.com/media/xxx` would be redirected to `react-wagtail-api.accordbox.com`

Now, please `git add` and `git commit` the above files, and then deploy to the server.

```
$ bash compose/auto_deploy_do.sh
```

Now everything would work as expected.

26.10 Live View from Wagtail Admin

To make editor can check `preview` and `live` version of the `PostPage`.

Update *react_wagtail_app/settings.py*

```
HEADLESS_PREVIEW_CLIENT_URLS = {
    "default": os.environ.get("FRONTEND_BASE_URL", "http://localhost:3000/"),
}
```

Add `FRONTEND_BASE_URL` to *.env/.prod-sample*

```
FRONTEND_BASE_URL=http://react-wagtail.accordbox.com
```

Now, please `git add` and `git commit` the above files, and then deploy to the server.

```
$ bash compose/auto_deploy_do.sh
```

Now if you click `Live` button or `View Draft` button, you should be able to redirected to the correct url which contains the page content.

Chapter 27

REST API FAQ

27.1 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
```

27.2 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
```

Chapter 28

Frontend FAQ

28.1 Module not found: Error: Can't resolve

Please note that the docker container has its own `node_modules` directory when we develop our app. (Because of `/app/frontend/node_modules` in `docker-compose.yml`)

So when you add some package to one container `node_modules`, the other container might not be synced.

Since you know the cause of the problem so you can solve in this way.

```
# please update command if you use npm install
$ docker-compose exec frontend yarn install
$ docker-compose exec storybook yarn install
```

Another approach is

1. Rebuild docker image.
2. `docker-compose stop XXX` and `docker-compose rm XXX` the service container.
3. `docker-compose up -d` (it will mount `node_modeuls` from recently built docker image)

28.2 Nothing was returned from render. This usually means a return statement is missing

This is the problem of the dependency package of `CRA`, and the relevant issue is https://github.com/facebook/create-react-app/issues/8689

I solved this by rollback the `react-scripts` to `3.4.0` and use `npm install` to install the dependency packages. (https://github.com/facebook/create-react-app/issues/8689#issuecomment-602233612)