CSC-40098

MSc Project

Clinical Guidelines Authoring Platform for a Health Service

Dan Bayford

Keele University

Abstract

Clinical guidelines are a resource for medics to refer to when treating their patients. Having access to concise, up to date medical information has been shown to improve clinical outcomes.

In recent years, the Bedside Clinical Guidelines Partnership (BCGP) project has attempted to digitise these records to allow for clinicians to access them via platforms such as smartphones and tablets, rather than relying on hard copies stored at certain locations within the hospital. The current proposed frontend solution for the platform wraps static markup created with standard web technologies such as HTML, CSS, and JavaScript, and presents the user with a browser-like experience.

This project proposes that the Wagtail content management system (CMS) could be used as the platform to create, edit and store these digitised guidelines, and implements a prototype that was then used as the data source to build static versions of the guidelines that the current frontend solution relies upon.

It was found that Wagtail could implement all the broad project requirements, and therefore could be considered a viable alternative for the wider project in place of the current, bespoke solution being developed.

Acknowledgements

I would like to thank my project supervisor Beran Necat for assisting me throughout this project. Any issues I had or clarifications I required were always responded to quickly and with useful advice or guidance. My fortnightly reporting in with him kept the project on a rapid but manageable pace and allowed me to meet all my deliverables within the project timescale.

I would also like to thank Professor Ed de Quincey, my contact at the University for the Bedside Clinical Guidelines Partnership (BCGP) project. Professor de Quincey really helped me in building out the project requirements from the rather brief statement from the project module, and clarified much of the functionality required from a potential solution, as well as supplying me with feedback so I could assess the success of the project.

Finally, I'd like to thank my wife, Ashlee. She's put up with me spending hours on this project and the wider course over the last couple of years. Worse than that, she has then been subjected to me attempting to talk to her about some of the finer points of computer science. I promise I'm finished now.

Table of Contents

1 Introduction

This development project is an adaptation of a project previously offered on the Keele University Computer Science MSc pathway:

"JM1 - Developing an authoring platform to create and edit existing clinical guidelines."

The original project brief was as follows:

"As part of an ongoing research study at the School of Computing and Mathematics, we have developed a mobile device application to deliver clinical information to NHS clinicians. This MSc project aims to build a prototype clinical guideline authoring and editing platform based on existing web frameworks. The goals are to:

- *produce a web- based application where users can utilise CRUD functions (Create, Read, Update, Delete) with existing guideline documents*
- *implement a multi-level user admin system*
- *utilise HTML, CSS, JS, and other web-based languages*
- *test the system to ensure it meets usability guidelines*
- *conduct literature review/research/testing to ensure the system is efficient"*

Although the opportunity to work on the original project was not available to students at the start of this project module, a proposal was submitted and accepted to create a prototype platform build on the Wagtail (*Wagtail CMS, n.d.*) content management system.

2 Existing Project Background

2.1 Clinical Guidelines

When a clinician treats a patient, they apply logical steps based on their current medical knowledge of a suspected condition. The potential scope of conditions that some clinicians might treat is vast, and if wider medical science has evolved since the clinician was last instructed on current best practice for a given condition, the care given to a patient may not be to the highest standard. Clinical guidelines are:

'Systematically developed statements to assist practitioner and patient decide about appropriate healthcare for specific clinical circumstances (Pereira et al, 2022)'

These guidelines combine peer-reviewed evidence and current considered best practices to present to the treating clinician accurate medical information. They are maintained by specialists who keep abreast of the changing literature and techniques across various clinical fields. The guidelines also promote a consistency of treatment across a medical organisation. However, clinical guidelines published by the likes of the National Institute for Health and Clinical Evidence (NICE) and other appropriate bodies are often extremely comprehensive and not best suited for quick reference in a potentially acute medical situation on a hospital ward.

2.2 The Bedside Clinical Guidelines Partnership (BCGP)

In 1993, the North Staffordshire Hospital (now University Hospital of North Staffordshire) made a review of how it delivered acute medical care (*Smith et al., 1998*). One of the reviews findings were that faster clinical decision making could potentially save up to 4,000 bed-days per annum and recommended that a more practical version of clinical guidelines be made available as a resource for its clinicians.

After a few years of development, the first digital version of the guidelines was released in 1996, before being offered to nearby trusts on a subscription basis from 1997. The West Mercia Clinical Guidelines Partnership was established in 1998, later renamed in 2004 to the current Bedside Clinical Guidelines Partnership (*Pantin et al., 2006*). The current board consists of consultant physicians, librarians, coordinators, and developers who meet regularly to discuss the development and improvement of guidelines based on feedback, new clinical evidence, and current literature.

2.3 Guideline Process

The current guideline process involves several steps and multiple users. The users involved in the actual guideline generation, moderation and approval stages would include:

- *Librarians* – monitor advances in medical evidence from sources such as Medline, PubMed and NICE, collating any relevant information into spreadsheets. Before a new release of the platform, they submit their findings to the secretariat.

- *Secretariat* – the secretariat acts as the process administer. They receive new evidence from the librarians (as mentioned above), distribute this new information to the relevant clinical author(s), field any queries on it and submit the updated guidelines to the clinical quality controllers (QCs) for final approval, highlighting any changes.
- *Clinical Authors* – each guideline has a responsible clinical author, usually a doctor, who will integrate the new evidence into the existing guideline Word document and once satisfied, return it to the secretariat.
- *Clinical QCs* – the clinical QC is normally a senior doctor in the appropriate field who will review the amended guideline and either give a final approval for publication or reject the changes.

Beyond the above, there would also likely be a requirement for access to clinical developers and external quality and safety auditors, although they would not generally be involved in the core operation of the platform.

Figure 1 shows the current workflow for updating an existing medical guideline:

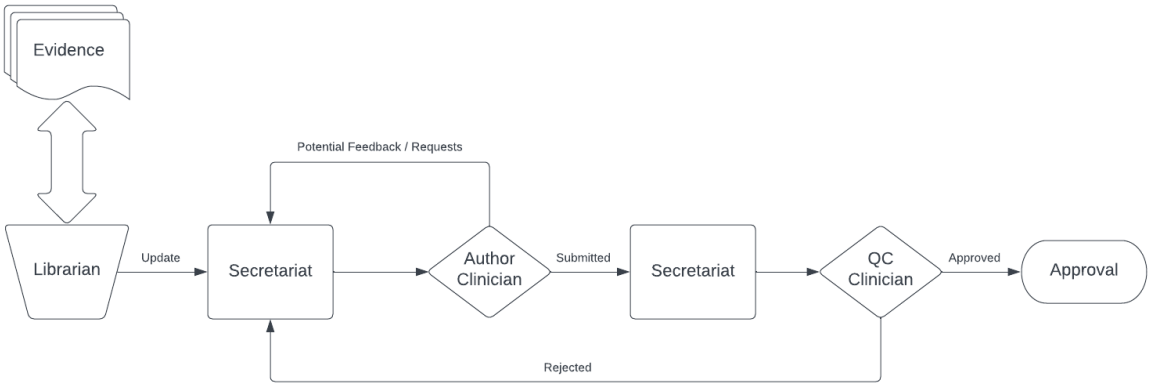

Figure 1 – Clinical Guideline Workflow

The process for creating a new guideline is similar, except the clinical author would create a

new Word document from the supplied evidence, rather than editing an existing one.

2.4 Guideline Structure

A guideline itself can be made up from various different types of content (*Mitchell et al., 2021*). The current list of supported content types include:

- Text
- Images
- Videos
- Labels (information, warning, and danger)
- Calculators
- Flowcharts

The guidelines do not follow a fixed structure, so two different guidelines can have completely different content types in the main body of the report.

2.5 Current Frontend Solution

The current frontend solution (*Mitchell et al., 2020*) is mobile based and requires the guidelines to be constructed from HTML, CSS, and vanilla JavaScript, potentially with no internet connection when being used. Each guideline is its own HTML file, linked to the appropriate styling and scripts. A native application then ‘wraps’ these files and allows navigation between them, in a manner similar to a web browser.

3 Project Deliverables

In consultation with Professor Ed de Quincy, the BCGP project lead at the University, the following key deliverables were agreed upon for a potential solution:

- A dynamic main content type as the core of each guideline, containing options for:
    - WYSIWYG (‘*what you see is what you get’*) editor
    - Images
    - Embedded video
    - Various custom content blocks (for example, Labels, Calculators)
- An option for extra regional content on each guideline
- Live preview of editing guidelines within the CMS
- Multiple user levels within the CMS (author, QC, developer) and an approval system
- A diffing functionality between different versions of the same guideline
- Search functionality available at API layer

It was felt that a solution with these deliverables would meet the requirements from the original project brief. In addition, by implementing a headless, client agnostic architecture, the proposed solution would allow for future development of the project to potentially include web based and native application front end solutions. As per Mitchell (*2022, pp 85*), there are also limitations as to what technologies the eventual front end would want to support to maintain cross platform support, and a JSON API interface would likely be the most flexible solution.

4 Proposed Solution

This project proposes the use of the Wagtail CMS to create the content authoring platform. Wagtail offers, either as standard or via configuration, all of the required functionality specified in the project deliverables:

- Mixed and dynamic content nodes
- Live preview functionality
- Configurable moderation workflows
- Diffing functionality across the history of content nodes
- Search functionality
- The ability to expose an API

The NHS already uses Wagtail for some of its own content management (*NHS Digital, 2018*), so it is already a solution that they are aware of and have a degree of confidence and experience in. Since Wagtail extends the Django (*Django, 2019*) web framework, it also benefits from the frameworks existing advantages (*MDN Web Docs, 2023*), such as:

- Secure, with regular release patches
- Consistent release cycle and long term supported (LTS) versions for production (*Download Django, n.d.*)
- Highly scalable
- 'Batteries included' philosophy which provides a lot of common web application functionality (user management, sessions, authentication) with the core install
- Highly configurable with a large ecosystem of third-party libraries
- Highly configurable to developers who can write Python

Another advantage of using an already existing solution such as Wagtail and then configuring it to the project requirements is the existing community around the platform. As well as the developer documentation (*docs.wagtail.com, 2023*), there is also dedicated editor documentation (*Wagtail Guide, 2023*) for the eventual end users to refer to, created and maintained by the Wagtail core team.

Although the core deliverable of the project is the CMS platform itself, it will also provide static builds of the front-end generated using the exposed API as a means to prove that the headless CMS solution is a viable option for the project. This will be build using the Next.js (*nextjs.org, 2023*) framework, a production framework that extends the React UI library (*Meta Open Source, 2023*). Next.js offers a static site generation (SSG) build command, and therefore could be used to build the client-side markup from the exposed Wagtail API.

There will also be a documentation solution for the exposed API, so a future developer would be able to investigate the appropriate data structure. The Swagger UI framework (*Swagger,io, 2019*) will be used to generate the appropriate schema and provide an interface.

The entire CMS will be developed and deployed within Docker (*Docker, 2018*) containers. By isolating the Python codebase within Docker containers, the surrounding environment can be controlled and made easily repeatable, which is highly desirable in an application that may be deployed in multiple instances for the different NHS regions that may use it.

5 Technical Background Discussion

5.1 Architecture and Stack Overview

The proposed CMS solution is Wagtail CMS, which is based upon the Django web server framework. The entire back end of the solution will be developed and deployed inside Docker containers and hosted on a virtual private server (VPS). For the prototype version, the database will be a dedicated Docker service.

The CMS API layer will be documented using Swagger UI, a tool that can generate a schema of a database and allow front end developers to explore the data structures exposed by the API via a browser based interface.

The front end client proof-of-concept will be built using Next.js. The final build will be entirely static, which is a requirement of the current mobile frontend solution.

There now follows a brief explanation of the relevant technologies to help understand the application architecture.

5.2 Django Web Framework

Django is an open-source Python web framework designed to build secure and scalable web-based applications. It is used in at least some capacity by many extremely large tech companies such as Instagram, Spotify, and YouTube (*StackShare, n.d.*). Although initially developed as a traditional web server using the model view template (MVT) pattern to deliver

server rendered web pages, it is also capable of acting as an API to a headless client, as is the case with this project.

Essentially, Django routes HTTP requests to an appropriate view (what would traditionally be called the controller in an MVC pattern), which should generate some sort of HTTP response. This may include HTML, JSON, or XML, with or without data from a database. The Django model, a Python class, is the interface between the view and the database. The models make use of the Django object relational mapper (ORM) to allow the developer to interact with the database through the model via Python objects known as QuerySets, rather than directly writing SQL.

Figure 2 explains the basic request and response cycle. Note that the web server gateway interface (WSGI) connector converts the HTTP request to an HTTPRequest object for Django to process, and that the view instantiates an HTTPResponse object to send back to the WSGI. For clarity, the various layers of Django middleware have been omitted.

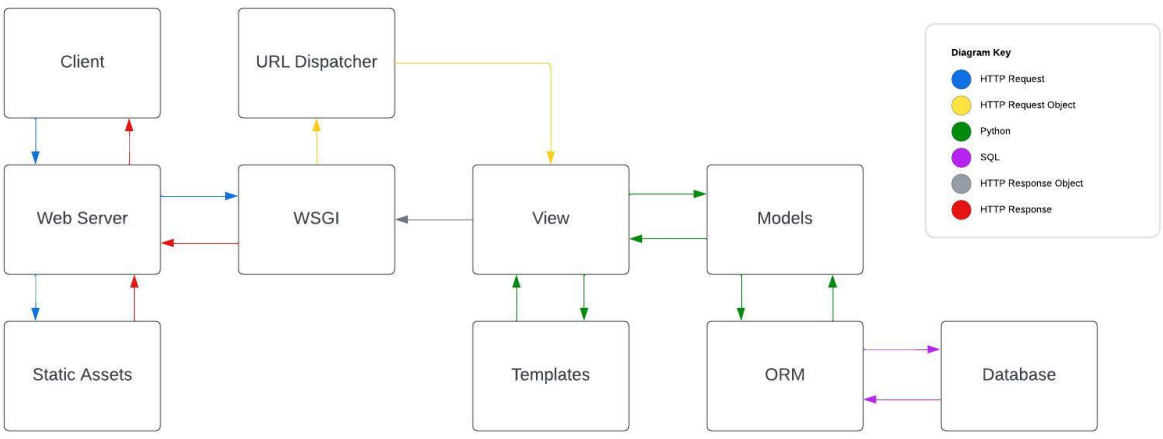

Figure 2 – Django Request and Response Cycle

The web server would likely be either Apache or Nginx, although anything that can interface with the WSGI layer can be used. For performance reasons, any static assets such as CSS or JavaScript are usually served directly from the web server without ever reaching Django. Requests for Django are routed via a WSGI interface such as Gunicorn (*gunicorn.org, n.d.*), and then the Django URL dispatcher system would determine the appropriate view to handle the incoming request. Views must then either return a response or throw an exception. Architecturally, Django consists of a single top level *project* constructed of various *applications* (Python modules) to add functionality, as shown in Figure 3.

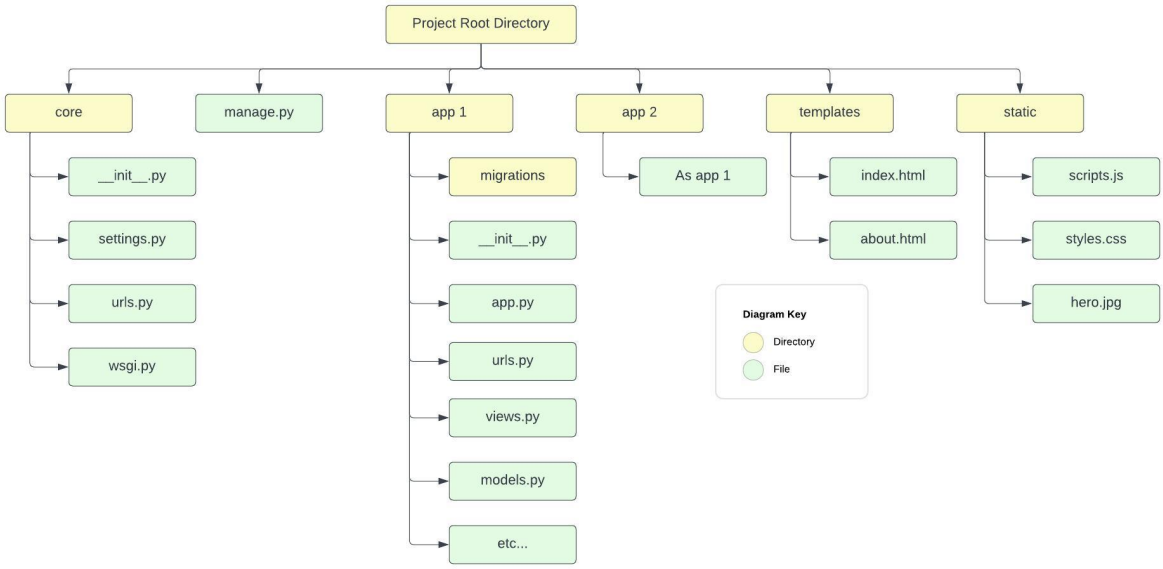

Figure 3 – Django File System Architecture

The directory containing the *settings.py* file is by default named the same as the project, but developers often rename it to something more appropriate such as *core* or *settings*, as in the diagram. This directory contains the *settings.py* file (the main project configuration file), the top level URL dispatcher and the WSGI interface.

The *manage.py* file contains a Python script that exposes various Django management commands in the terminal, such as running the development server, creating migration files for data models and collecting static assets for deployment.

Each individual application can be made up of any number of directories and files, depending on the functionality. The only required files are the *init.py* file so that Python can treat the app as a Python module and import it, and the *app.py* file which is used to register the app inside the main *settings.py* file. It is common to have an application level *urls.py* file that the root URL dispatcher can point at and then it will handle the routes for its own application. The *migrations* directory holds the migration files for the relevant Django model.

Top level directories for templates and static assets also help organise the project - the Django templating engine will then be able to look directly into a dedicated directory when searching for HTML files, and all the static assets can be collected for the main web server via a single management command when ready to deploy.

The core Django installation comes with applications and middleware for such common web service requirements as authentication, authorization, sessions, static file management, and more (*Django Project Contrib, n.d.*). Any further functionality can then be added to the project via custom applications that are registered in the main *settings.py* file.

5.3 Wagtail CMS

Wagtail is an open-source CMS that is built on top of the core web framework functionality that Django offers. Like Django, it is also used by some extremely large companies such as NASA, Mozilla, and the NHS (*Wagtail CMS, n.d.*).

As mentioned, Django is essentially a collection of apps making up a larger project, and each app can contain various views, templates, and models. Wagtail adds several extra apps to the basic Django installation, such as:

- *wagtail.core* - the core functionality of the CMS, including the Page model
- *wagtail.admin* - the Wagtail admin interface GUI
- *wagtail.documents* - support for managing uploaded documents
- *wagtail.images* - support for managing uploaded images
- *wagtail.snippets* - small, repeatable content that doesn't justify an entire Page model
- *wagtail.users* - user management and permissions functionality
- *wagtail.contrib.forms* - support for managing submitted forms
- *wagtail.contrib.modeladmin* - support customising the admin interface
- *wagtail.search* - support for searching the content of the Wagtail application

Once installed, a certain namespace is configured in the Django root URL dispatcher and then Wagtail will serve any requests that match that URL pattern.

The core of the Wagtail installation is a comprehensive Page model and the associated hierarchy management. A Wagtail Page model is a Python class that essentially holds some sort of content (usually an individual page on the site, such as the 'About' page) and

exposes several methods that allow Wagtail to organise it with respect to the rest of your content in a tree data structure (Figure 4). At the root of the site is a home page (usually at the '/' URL), and then various child nodes descend from here. Each node in the tree is an instance of a Page model. When Wagtail receives an HTTP request, it will inspect the URL and retrieve the appropriate Page model for the response.

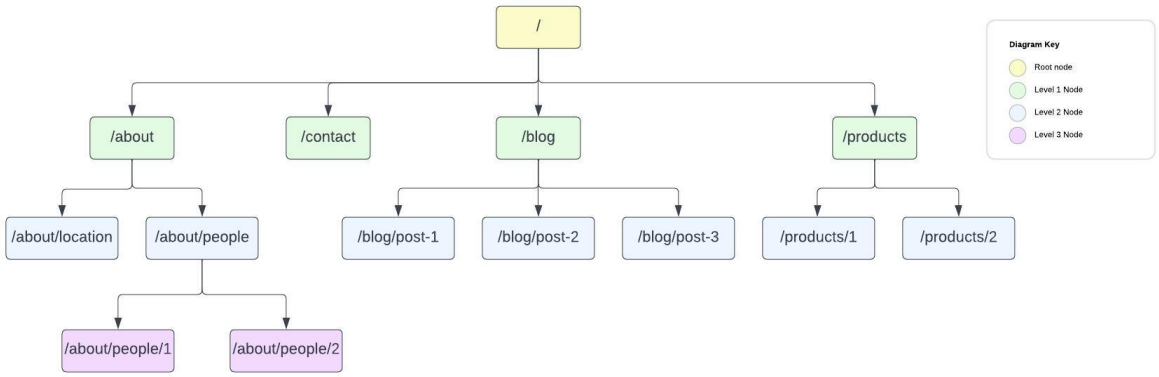

Figure 4 – Wagtail Tree Data Structure

Internally, Wagtail uses the *django-treebeard* library (*django-treebeard-readthedocs.io, n.d.*) to manage the data structure.

Fundamental to understanding how Wagtail works are two methods on the Page model:

- *route* - starting from the root node, this inspects the incoming request URL and determines if this particular Page instance can serve a response or if it needs to pass the request further down to its child node(s)
- *serve* - once a request reaches the appropriate Page instance for its URL, then the serve method will construct the response from that particular instance using its associated information in the database

Wagtail also includes an admin interface that is a much more advanced and capable version of the default Django admin interface, and allows a user to interact with the Page models, set moderation workflows and live preview the site, amongst other things.

5.4 Django Rest Framework

Although traditionally used to generate HTML templates, Wagtail can also function as an API via the open-source Django Rest Framework (*Christie, 2011*), or DRF. Much of DRF's functionality is abstracted away when installed and configured for use in Wagtail, but it is a standard Django library (i.e. not limited to Wagtail). DRF essentially acts as a further step in the resolution of an HTTP request by serialising and deserialising between the request and response body JSON and the Python data structures required by the views. It uses a dedicated Serializer class to perform this conversion, which is then referenced in the appropriate view when required. The Serializer class can then be customised to only expose certain fields, mark certain fields as read only, add access permissions and so on.

5.5 Docker

Docker is a containerisation software that allows developers to create, share and deploy consistent environments for their applications. It is similar to a virtual machine (VM) in that it acts as an isolating 'wrapper' around applications, but it is much more lightweight since it

does not virtualise the underlying operating system - the applications run on a dedicated daemon process known as the Docker Engine.

The main concepts behind Docker are images and containers. Images are the ‘blueprints’ for a container, defined in a Dockerfile. The Dockerfile will have multiple steps on how to build a particular image, copying in certain local files, installing packages, and running commands. Once an image is built and you want to run your Dockerised application, you instantiate ‘containers’ from these images - the containers are then the running instances of your application.

Docker is especially useful for Python applications as it removes the need for setting up, configuring, and maintaining virtual environments (*Python.org, 2019*) across multiple machines. When installing Python packages such as Django, it is recommended to install it within a virtual environment, managed by a Python package manager such as pip or Poetry, so as not to conflict with other applications - you may be running different versions of Django between different projects on the same machine, or even different versions of Python itself. The use of Docker removes this potential issue as the images are self-contained and can use whatever dependencies are specified in their Dockerfiles, installed and isolated from the rest of the machine.

5.6 Swagger UI

Since the CMS solution has a headless architecture, it will require a separate frontend. For this project, Next.js will be used (see below) to generate the static files required by the existing native mobile wrapper application. However, during development, it will be necessary to examine and understand the data structures that the CMS outputs for a given URL. This could be done with a traditional HTTP client such as Postman (*Postman, 2021*) or similar, but there are also libraries that can give a better developer UX.

The Swagger UI library is an open-source tool that provides a user-friendly interface for visualising and interacting with APIs. Once installed, Swagger UI will generate an OpenAPI Specification (*Swagger.io, 2020*) schema of your data and store it in a JSON or YAML file. You also register an extra URL on your site to point at the appropriate Swagger functionality. This schema and URL can then be used via a web browser to explore and interact with your data in the same way your eventual frontend solution would.

5.7 Next.js

Next.js, often referred to as just Next, is an open-source production framework for building web applications. It is based on the React JavaScript UI library and offers extra functionality on top of core React such as integrated routing, serverless APIs and automatic code splitting. Next is highly configurable and can serve individual pages as either server side rendered

(SSR), client side rendered (CSR) as in a traditional single page app (SPA) or, as will be the case in this project, generating a static build via a static site generation (SSG) process. React (and therefore Next) uses the concept of discrete components to build out various sections of a UI using JavaScript functions (or, previously, class instances). This makes it particularly effective for building dynamic client applications with variable data structures, as in this project. It is not necessary to build out lots of individual HTML templates to cover all the possible data combinations - rather, the developer can create logic inside the components to return what is known as JSX (JavaScript XML - an abstraction over HTML), and then the returned markup for a given component will depend on what data is passed into the component as arguments. Using this pattern, the CMS editor can create content in any structure they require (within the Wagtail Page model limitations) - the front end will simply format it as required during the build stage.

6 Development and Functionality

6.1 Project Management

For project management, the various tasks and features were divided into two week sprints. The content of these sprints was determined after analysing the proposed project schedule from the Gantt chart submitted during CSC-40100.

Once identified, these sprints were organised and tracked using a Trello card system. The tasks in Trello were organised using various labels and the board updated as sprints were completed and functionality or deliverables were achieved. Any temporary roadblocks or waiting periods were controlled via various columns of the Trello.

Fortnightly updates were also sent to the project supervisor. Details of these updates can be found in *Summative Assessment 3 – Progress Reporting*.

During project planning, certain contingency plans were proposed in the event that the intended functionality was found to not be achievable, either due to time constraints or limitations in the proposed solutions. In the event of a minor time constraint, it was proposed to drop the Swagger integration due to the alternative deliverable of a Postman HTTP client collection to explore the API, and in the event of a major time constraint it was proposed to drop the front end client build as the core project deliverable was the CMS platform. There was also the option to drop the Swagger integration if it was found not to be technically feasible, as there was limited information available during the project planning stage. In the

event, the project ran ahead of schedule, the Swagger integration was successful, and these contingencies were not required.

6.2 Project Architecture and Deployment

The CMS was developed and deployed inside a Docker container. This allowed for granular control of the immediate environment that the server runs in and allows the CMS to eventually be deployed on any service that can run Docker. Containerisation is especially useful for a Python based framework such as Django, as it allows you to not use a virtual environment and you can also specify the exact libraries to use during a build step via a *requirements.txt* manifest or similar form within the Dockerfile

Django comes with a built in development server, but it is not recommended for production. The standard pattern is to use the Gunicorn WSGI server interface to sit in front of the Django instance, and then have a further web server in front of this. Gunicorn can then use multiple worker processes (the number of which depending on CPU resources) to call individual instances of the Django bootstrap, and a web server such as Nginx can sit in front of Gunicorn and handle any static requests (stylesheets, images) to reduce load. Gunicorn is available as a Python library so sits inside the CMS Docker container, but Nginx is added as an extra Docker service.

For a full production deployment, it is likely that a hosted database solution such as AWS

RDS (*AWS, n.d.*) or similar would be used, as it abstracts much of the administration of running a database day-to-day (backups, security patche). However, for this project, another Docker container was used to run a Postgres service for the server to interface with. In total, the CMS has three Docker containers. To orchestrate this, a *docker-compose.prod* YAML file (Figure 5) was used to configure them and ensure they interface correctly.

docker-compose.prod.yml

Guidelines > docker-compose.prod.yml

```yaml
version: '3.8'

services:
  web:
    build:
      context: .
      dockerfile: Dockerfile.prod
    command: gunicorn app.wsgi:application --bind 0.0.0.0:8000
    volumes:
      - static_volume:/home/app/web/staticfiles
      - media_volume:/home/app/web/mediafiles
    expose:
      - 8000
    env_file:
      - ./.env.prod
    depends_on:
      - db
  db:
    image: postgres:13.0-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    # Have to expose to allow pgAdmin access
    ports:
      - 5432:5432
    env_file:
      - ./.env.prod.db

  nginx:
    build: ./nginx
    volumes:
      - static_volume:/home/app/web/staticfiles
      - media_volume:/home/app/web/mediafiles
    ports:
    # HOST:CONTAINER
      - 80:80
    depends_on:
      - web

volumes:
  postgres_data:
  static_volume:
  media_volume:
```

Figure 5 – Production docker-compose.prod.yml

In Figure 5, the three services and their interfaces can be seen. Inside the Docker network, the web (Django) service is exposed on port 8000. The Nginx configuration listening on external HTTP port (80) is proxying requests from there to port 8000 within its *nginx.conf* file.

The web service contains a script that delays the start of the server until the database is ready to accept connections – a common issue in a Docker system is that the container may be running but the internal process may not actually be ready, causing synchronisation problems (*Docker Documentation, 2020*).

Note the use of Docker volumes to persist certain data even if the stack is stopped (*postgres_data* for the database, *static_volume* for static assets such as JavaScript and CSS, and *media_volume* for images and video). These volumes are binds between the host file system and the Docker containers, essentially making them mirror each other. The web service and the Nginx service share access to the *static_volume* and the *media_volume*, so the CMS can control the content and the web server can serve it.

Volumes also make it possible to develop inside running Docker containers by mirroring the source code on the development machine into the container. This is exactly how this project was developed, using a separate *docker-compose.dev.yml* file to map volumes appropriately during development.

The full architecture of the project and the interfaces between the various services and volumes can be seen in Figure 6:

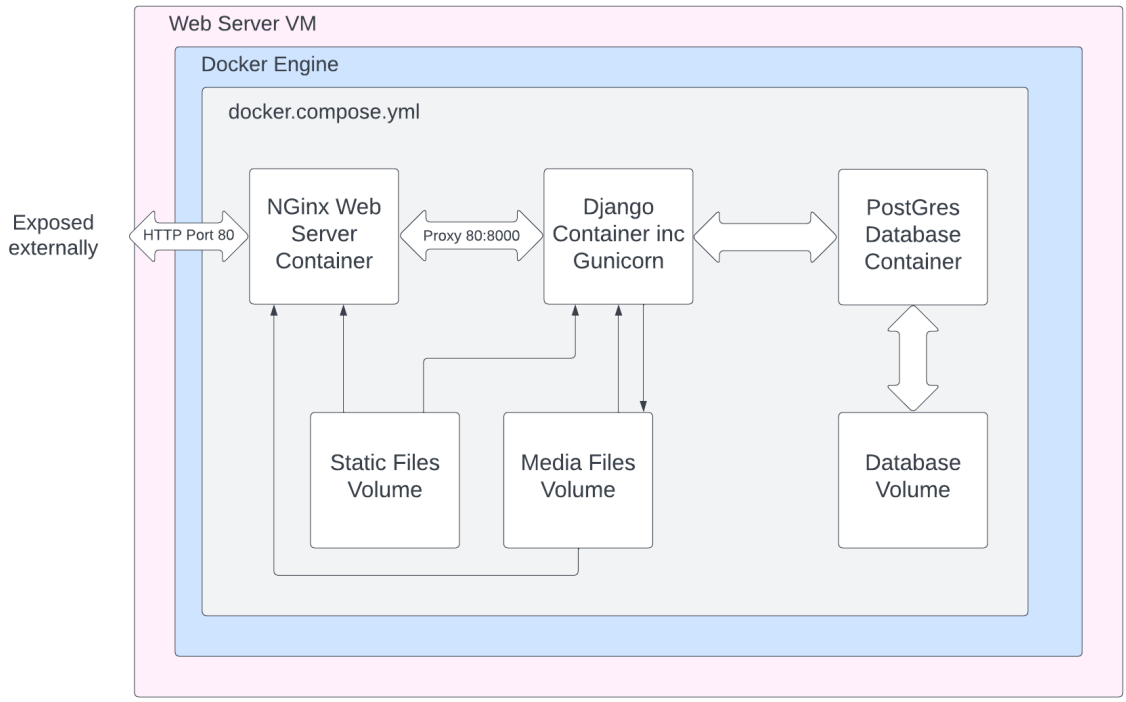

Figure 6 – Project Docker Architecture

By using this pattern, the entire stack is highly repeatable, and, in the case of the Guidelines project, it would be trivial to quickly install and run another instance of the software for another NHS Trust if required – once the codebase is available on the hosting service, simply running the orchestration file with a build argument would create a new instance of the Guidelines.

Another advantage of Docker is that, in future, it would be possible to add further functionality and orchestrate it as a service. For example, it would be possible to have an ElasticSearch (*Elastic.co, 2019*) service to improve search functionality, or a Celery (*docs.celeryq.dev, n.d.*) worker service to hand off any blocking tasks from the main Django execution threads (sending emails, for example).

6.3 Project Functionality

6.3.1 Dynamic Content

As mentioned, a Wagtail Page model is a highly customised Django model, which is in itself a Python class. Each model is generally represented by a table in a database, and each instance of the class is represented by a table row. So, for example, Wagtail will create and manage a table in the application database called *wagtailcore_pages*, and each row in this table will contain the meta data for a particular page instance.

The Wagtail Page model inherits from the *django.db.models.Model* class, the base class for all Django models. The Django base class adds functionality such as database operations, field definitions and querying capabilities. The Wagtail subclass then adds methods and properties to assist with the management of the Wagtail tree structure, URL routing, content types and so on.

Beyond this, the developer is free to add extra fields and methods to their model to store custom data or add functionality. Wagtail offers a large amount of basic field types in which to store model data (such as TextField, URLField, ImageField, ForeignKey), and the various super classes will handle the creation of appropriate database columns and migrations on a schema change.

A core feature of Wagtail is the concept of a StreamField as a field type on a page. A StreamField is a special type of field on a model that can store mixed content data. For

example, a particular StreamField could be defined as being constructed of TextFields, URLFields and ImageFields. Now, when an editor adds this particular StreamField to a page instance in the admin area, they will create a StreamField that can be made up of any combination of TextFields, URLFields and ImageFields, in any order and with as many individual instances of the sub-fields as required, within the constraints of the particular StreamField configuration. The content can also be rearranged at a later date, have extra fields added or remove others.

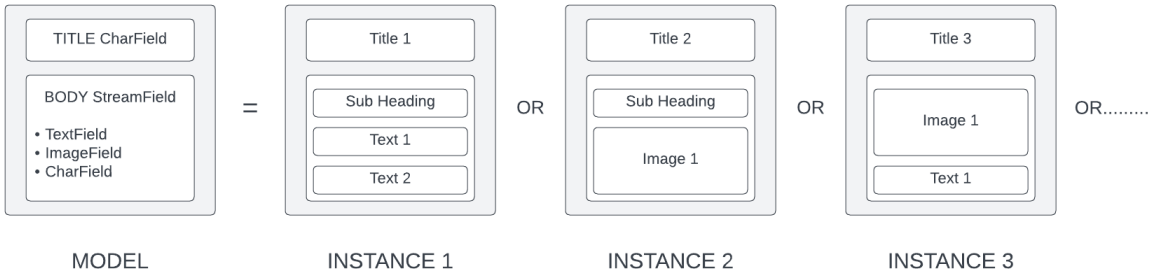

Figure 7 – Wagtail StreamField Pattern

Wagtail stores these StreamFields, along with the rest of the page content, in a table related to the *wagtailcore_pages* table as JSON. By storing the information as JSON, it is possible to rearrange it as required by the editor, and there are few constraints on its structure – if it is saved in a valid JSON structure, the database can store it.

Data Output | Explain | Messages | Notifications

	page_ptr_id [PK] integer	body jsonb
1	5	[{"id": "c3fdac2e-e5af-4f1b-beb3-88e1a4ad104c", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "9a4091c4-9988-4034-a3ca-541f91b10ce2", "type": "guideline", "value": {"sub": {"subhea
2	9	[{"id": "e1d9902e-a651-46f6-a038-f0af15a65a52", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "367bf482-0d6e-4fce-9a2d-83794bca56ed", "type": "guideline", "value": {"sub": {"subhea
3	10	[{"id": "d3eb715a-ea8d-4b82-ab40-d50cadc9c5ec", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "6ea77d6c-7ed2-460a-8232-fd1c5d8d333a", "type": "guideline", "value": {"sub": {"subhe
4	11	[{"id": "f566f70f-7549-425e-8875-189fd81a982b", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "8fb844df-f74f-4b14-a2cd-fb22c6172ad6", "type": "guideline", "value": {"sub": {"subhead
5	12	[{"id": "bafce630-a3dc-413a-a3f4-c987606b9e1d", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "b7f04b29-7306-455a-9e0b-bc5062f59121", "type": "guideline", "value": {"sub": {"subhea
6	13	[{"id": "c8485761-1bf4-41e3-a4cb-ae0458ea85bd", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "9c14ecf1-753b-4e87-9ed2-1d1d3b37390f", "type": "guideline", "value": {"sub": {"subhe
7	14	[{"id": "b01bcd54-e8f1-4458-a567-59c0fa910ccc", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "c34dfe99-0c71-43fc-800d-9e2f9495c55e", "type": "guideline", "value": {"sub": {"subhea
8	15	[{"id": "904083fe-f07e-48ac-86b7-2cb9e99ec596", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "bc4b17a3-b453-47ea-891e-d92a073694a1", "type": "guideline", "value": {"sub": {"subhe
9	16	[{"id": "8c92a623-0db6-44aa-b81b-d1c39d57b9f9", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "4c0ec681-fac4-4c43-8d13-a63eed8874f9", "type": "guideline", "value": {"sub": {"subhe
10	17	[{"id": "f33427a5-87c6-4d0b-bcd1-9153ff1754d1", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "8f51af1b-e35c-4975-b4cf-716f988c0c39", "type": "guideline", "value": {"sub": {"subhead
11	18	[{"id": "3866a027-465f-4ad2-85cd-3bc87d2573a4", "type": "main", "value": {"mainheading": "Main Heading 1"}}, {"id": "a5a14ee6-22c5-4eaa-b2c9-5e2427f6693d", "type": "guideline", "value": {"sub": {"subhe

Figure 8 – JSON Page Body Content

Note that adding or removing entire new fields is possible but would require a database migration (handled by Django management command system) to maintain synchronisation with the application models.

Figure 9 shows two instances of the same Page model but with different content structures:

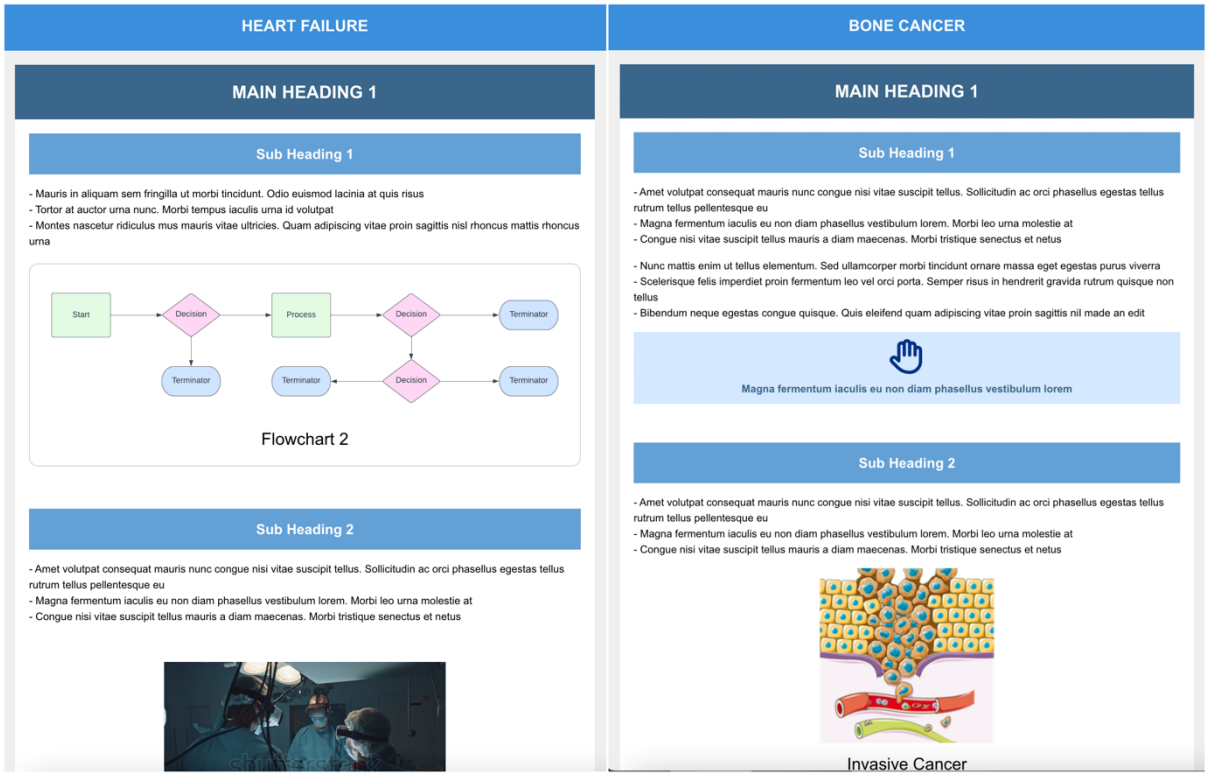

Figure 9 – Example of Mixed Content

By default, Wagtail offers the following content types for a StreamField:

- *RichTextBlock* - WYSIWYG editor
- *Charfield* - for titles, subheadings etc
- *ChoiceBlock* - for dropdown menus
- *ImageChooserBlock* - for images
- *VideoChooserBlock* - for media

Beyond these, the project was extended, either via third party libraries or within code, to provide the following extra content types:

- *CalculatorBlock* – select appropriate medical calculator
- *FlowChartBlock* – select appropriate medical flowchart
- *LabelBlock* – select appropriate label
- *TrustSpecificContentBlock* – select trust specific content block
- *MathBlock* – select an equation block (supports MathJax syntax)
- *TableBlock* – select an HTML table

For content such as the CalculatorBlock and the FlowChart block, the content types were simple ChoiceFields that allows the editor to pick a certain medical calculator or flowchart. Within the live preview, this value could then be used on the page context within the templates to render the appropriate markup. On a frontend build, it could be used to return the appropriate component (in a React based framework such as Next.js, for example).

The LabelBlock had two fields – one a ChoiceField for the label type (Information, Warning or Danger) and a TextField to add label text.

The TrustSpecificContent block had two fields – one a ChoiceField to select the appropriate

NHS Trust, and a TextField to add Trust specific content. This could easily be expanded to include other fields to increase functionality.

The MathBlock used an external library (*Ramm, 2022*), again showing the advantages of using already existing solutions and extending existing functionality.

The TableBlock is available from Wagtail core with some extra configuration (*docs.wagtail.ord – TableBlock, n.d.*)

6.3.2 Regional Content

The current solution involves individual NHS Trusts manually updating their own HTML with regional content, but this has several issues:

- Inconsistent markup with the rest of the guidelines
- Laborious
- No centralised moderation
- Effectively multiple versions of the guidelines in existence and issues with the associated version controlling
- Doesn't take advantage of CMS live preview or diffing functionality
- Doesn't take advantage of the build steps of modern frameworks (i.e. minification)

This project suggests having a dedicated content type available within the mixed content body where trust specific information could be entered and stored with the rest of the guideline content. As an example, the Next.js frontend has a *TrustSpecific.js* component that checks a build argument for a given context and, if it matches, outputs the content for the

markup. If the build argument for a given piece of content does not match, it returns null.

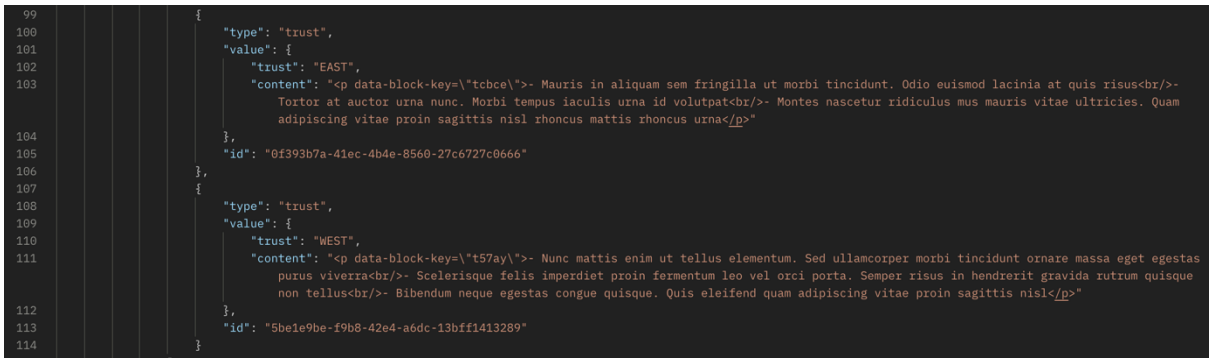

Figure 10 – Returned CMS JSON for 'Bone Cancer'

Figure 11 – Client Side Filtering by NHS Trust

Figure 12 – Example of Filtered NHS Eastern Trust Build

This solves many of the previously mentioned issues, such as maintaining a consistent markup structure, removing the requirement to manually edit HTML files, allowing the live preview and diffing functionality to be leveraged and keeping the content within the content version control framework. On a build step before a release, build arguments could be used to output specific NHS Trust versions of the Guidelines, filtering out irrelevant information.

Since Wagtail also has advanced moderation and workflow capability (see *6.3.4 - Multiple User Levels and Moderation System*), it would also be possible to configure workflows to have appropriate users at various Trusts view and approve the Trust specific content before publication.

6.3.3 Live Preview

In a standard Wagtail installation, Wagtail itself would generate the web pages for requests in much the same way that Django would – a request is received via the URL dispatcher, any extra data is recovered from the query string or request body, and web pages are generated using a combination of templates, CSS, scripts and, if required, data from the database. In such cases, it would be trivial to add a live preview functionality because Wagtail would use the same templates that are used to serve the HTTP requests and could determine which one to use via the page URL in the associated database table.

However, this project has been configured to run headless and return JSON for HTTP requests. For this reason, it was necessary to also create individual templates for each of the provided content types and to explicitly point the various block content types to the appropriate templates in code. There is also a *guideline.html* template that the overall GuidelinePage model points to, which receives the page data context and then iterates through, returning the appropriate block content for each content type in the page data.

models.py

Guidelines > server > guidelines > models.py > ...

```python

class Guideline(Page):

    # Can only be the child of a GuidelineIndexPage
    parent_page_types = ["guidelines.GuidelineIndexPage"]

    # HTML template for content
    template = "guideline.html"
```

Figure 13 – The Guideline Model Pointing to guideline.html

guideline.html

Guidelines > server > templates > guideline.html

```html
{% load static %}
{% load wagtailcore_tags %}
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Medical Guideline</title>
    <link rel="stylesheet" href="{% static 'css/reset.css' %}">
    <link rel="stylesheet" href="{% static 'css/guideline.css' %}">
</head>
<body>
    <header>{{ page.title }}</header>
    <main>
        {% for block in page.body %}
            {% if block.block_type == 'main' %}
                {% include 'blocks/mainheading.html' %}
            {% elif block.block_type == 'guideline' %}
                <div class="guideline-block">
                {% for item in block.value %}
                        {% if item == 'sub' %}
                            {% include 'blocks/subheading.html' %}
                        {% elif item == 'info' %}
                            {% for subblock in block.value.info %}
                                {% include_block subblock %}
                            {% endfor %}
                        {% endif %}
                {% endfor %}
                </div>
            {% endif %}
        {% endfor %}
    </main>
</body>
</html>
```

Figure 14 – Django Template for guideline.html

Figure 15 – Example of a Content Subblock

The *guideline.html* in Figure 14 shows how Wagtail can then dynamically generate the appropriate HTML to use in the live preview functionality.

- The GuidelineModel is instantiated on a request for the page preview
- The model points to the appropriate template and passes a page context object (containing the appropriate information from the database)
- During rendering, the templating engine has access to this context as a Python dictionary, as can be seen by the various steps of the template inspecting for properties on the page dictionary
- The *guideline.html* file will then return appropriate sub blocks of HTML (such as *subheading.html* in Figure 15), passing in further context as required

To prevent confusion and potential errors for unavailable templates, the live preview functionality was also explicitly disabled on the root page model and the sub-section models. Figure 16 shows an example of the live preview functionality from within the CMS. Note within the panel that the preview can also be changed between mobile, tablet and web views to confirm the responsive behaviour of the page:

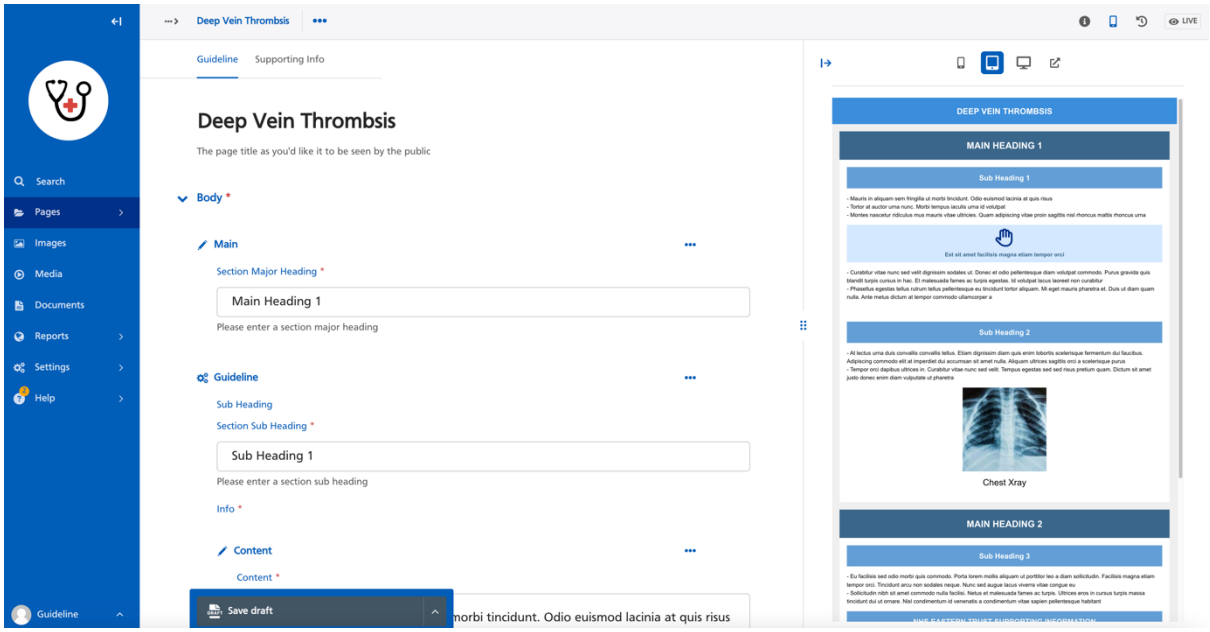

Figure 16 – Live Preview Functionality

6.3.4 Multiple User Levels and Moderation System

Wagtail is, at its core, a CMS used for managing large amounts of information. Large companies such as NASA and Google (*Wagtail CMS – Who Uses Wagtail?, n.d.*) use Wagtail precisely because it allows them to control large amounts of content in a structured way. When managing large amounts of information, there are several things to consider. Firstly, who has authorisation to make certain changes? Under the principle of least privilege, or PoLP (Saltzer, 1974), it is generally considered bad practice to allow blanket access to every user – beyond any potential malicious actions, there is also the possibility of users being overwhelmed by the amount of information presented to them if they are not a more experienced user. By restricting what a particular user can see, you reduce the potential

for confusion.

With a CMS, there is also often the need for a particular piece of content to go through a moderation process before it is considered available. In the Guidelines project, for example, this could include certain medical related steps that would need to be approved on a given piece of content – perhaps a particular field expert would need to review any edits a more junior colleague made to a guideline before it could be considered acceptable for clinical use.

A full user authentication and authorisation system, with various levels of access and all the related functionality (i.e., registration, log in, password reset), would be a large project in itself. Trying to integrate it into a custom moderation system would increase the complexity even further. Here is yet another example of why extending an existing, proven solution is often much more practical than developing from the beginning.

To understand the user and moderation process in Wagtail, there are four concepts to understand:

- Users
- Groups
- Tasks
- Workflows

A *user* is an individual account within the CMS, containing information such as the users name and email address. Users can be assigned to groups.

A *group* is a collection of users with the same privileges. Once a group is created, Wagtail allows you to grant certain actions that the group is allowed to perform within the CMS. Some examples of these actions include:

- Access to the admin area
- Object permissions – editing other users, groups, tasks
- Page permissions – editing the Page instances (i.e., the content)
- Image permission – editing images

Figure 17 is an example of the current Editor permissions, showing that members of the 'Editor' group can Add, Edit and Delete Pages, but cannot Bulk Delete, Lock or Unlock. Note the above Page Permissions in Figure 17 are on the 'Root' Page, which means the permissions apply to all Page instances on the site – this can be adjusted to be more granular and give a particular group access to only a subtree of the overall Page tree.

For moderation steps, Wagtail uses the concepts of *tasks* and *workflows*. A workflow is a complete moderation pathway that includes one or more tasks in sequence.

A task is an individual stage of moderation. For example, a site admin may want to create a moderation step where, to approve a cancer related guideline, a senior cancer specialist has to give medical approval. To implement this, a task is created and associated with a group containing the senior cancer specialists. When the content reached this stage of moderation,

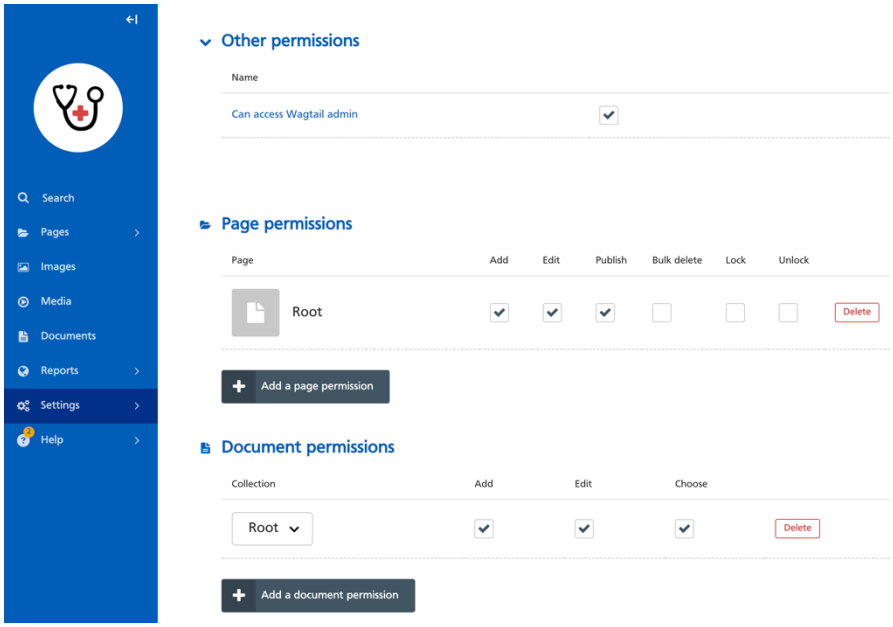

Figure 17 – Example of 'Editor' Group Permissions

all members would receive a notification and it would have to be approved by a member of this group to continue its publishing process.

Workflows are combinations of tasks ordered in an appropriate way. Continuing the cancer example above, there may be three tasks in the wider workflow – the initial editor task for making the appropriate changes, the specialist task for approving the changes, and a final moderator task to release to published. The content would have to pass all three stages of moderation, in order, to be accepted. At each stage, if rejected, the content would move back a step in moderation and would have to be resubmitted.

As well as making the workflows highly configurable, this pattern also means changes to these workflows can be made inside the admin GUI, rather than in code, meaning a moderately advanced user with appropriate permissions could reconfigure as necessary without requiring a code redeployment.

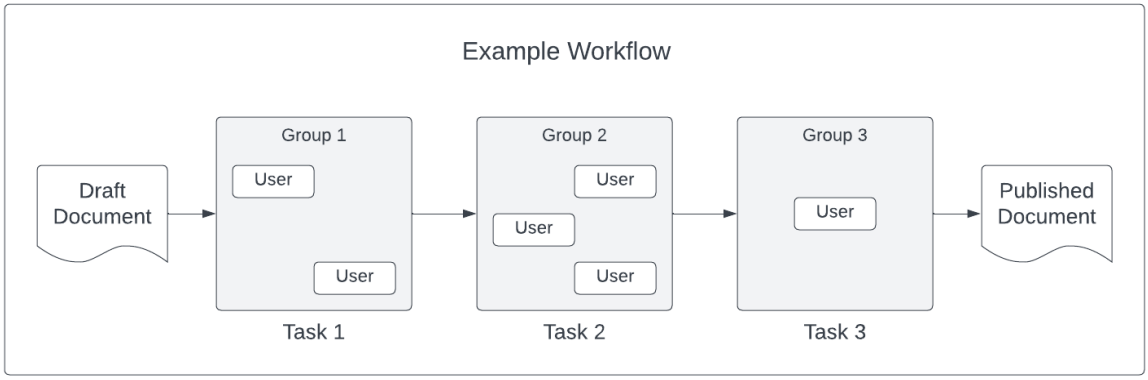

Figure 18 – Wagtail Moderation System

6.3.5 Diffing Functionality

For every node on the content tree, Wagtail stores an entry in the *watailcore_page* table within the database. It contains information on the node such as position in the tree, the number of children it has, the node page title, page content and its site URL.

	id [PK] integer	path character varying (255)	depth integer	numchild integer	title character varying (255)	slug character varying (255)	live boolean
1	1	0001	1	1	Root	root	true
2	3	00010001	2	4	Clinical Guidelines	clinical-guidelines	true
3	4	000100010001	3	3	Cancer and Neoplasms	cancers	true
4	5	0001000100010001	4	0	Bone Cancer	bone-cancer	true
5	6	000100010002	3	3	Cardiovascular	cardiovascular	true
6	7	000100010003	3	3	Neurological	neurological	true
7	8	000100010004	3	2	Respiratory	respiratory	true
8	9	0001000100010002	4	0	Leukemia	leukemia	true
9	10	0001000100010003	4	0	Pancreatic Cancer	pancreatic-cancer	true
10	11	0001000100030001	4	0	Alzheimers Disease	alzheimers-disease	true
11	12	0001000100030002	4	0	Cerebral Aneurysm	cerebral-aneurysm	true
12	13	0001000100030003	4	0	Epilepsy	epilepsy	true
13	14	0001000100020001	4	0	Deep Vein Thrombsis	deep-vein-thrombsis	true
14	15	0001000100020002	4	0	Heart Failure	heart-failure	true
15	16	0001000100020003	4	0	Stroke	stroke	true
16	17	0001000100040001	4	0	Cystic Fibrosis	cystic-fibrosis	true
17	18	0001000100040002	4	0	Pneumonia	pneumonia	true

Figure 19 – wagtailcore_page database table

When content is changed via the CMS, Wagtail will not only update the relevant entry in the table but will also create a copy of the original content and make an entry into the *wagtailcore_revision* table.

For example, from Figure 19, it can be seen that the 'Bone Cancer' page has a primary key of 5. Referencing the *wagtailcore_revision* table in Figure 20, it can be seen that there are multiple entries with an *object_id* field of 5 within the content column of the table

	id [PK] integer	object_str text	object_id character varying (255)	content jsonb
1	1	Clinical Guidelines	3	{"pk": 3, "live": true, "path": "00010001", "slug": "clinical-guidelines", "depth": 2, "owner": 1, "title": "Clinical Guidelines", "locale": 1, "locked": fa
2	2	Cancers	4	{"pk": 4, "live": true, "path": "000100010001", "slug": "cancers", "depth": 3, "owner": 1, "title": "Cancers", "locale": 1, "locked": false, "expired": f
3	3	Bone Cancer	5	{"pk": 5, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"c3fdac2e-e5af-4f1b-beb3-88e1a4ad104c\"}
4	4	Bone Cancer	5	{"pk": 5, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"c3fdac2e-e5af-4f1b-beb3-88e1a4ad104c\"}
5	5	Cancer and Neoplasms	4	{"pk": 4, "live": true, "path": "000100010001", "slug": "cancers", "depth": 3, "owner": 1, "title": "Cancer and Neoplasms", "locale": 1, "locked": fa
6	6	Cardiovascular	6	{"pk": 6, "live": true, "path": "000100010002", "slug": "cardiovascular", "depth": 3, "owner": 1, "title": "Cardiovascular", "locale": 1, "locked": fals
7	7	Neurological	7	{"pk": 7, "live": true, "path": "000100010003", "slug": "neurological", "depth": 3, "owner": 1, "title": "Neurological", "locale": 1, "locked": false, "e
8	8	Respiratory	8	{"pk": 8, "live": true, "path": "000100010004", "slug": "respiratory", "depth": 3, "owner": 1, "title": "Respiratory", "locale": 1, "locked": false, "expi
9	9	Bone Cancer	5	{"pk": 5, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"c3fdac2e-e5af-4f1b-beb3-88e1a4ad104c\"}
10	10	Leukemia	9	{"pk": 9, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"e1d9902e-a651-46f6-a038-f0af15a65a52\"
11	11	Pancreatic Cancer	10	{"pk": 10, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"d3eb715a-ea8d-4b82-ab40-d50cadc9c5ec
12	12	Bone Cancer	5	{"pk": 5, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"c3fdac2e-e5af-4f1b-beb3-88e1a4ad104c\"}
13	13	Bone Cancer	5	{"pk": 5, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"c3fdac2e-e5af-4f1b-beb3-88e1a4ad104c\"}
14	14	Bone Cancer	5	{"pk": 5, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"c3fdac2e-e5af-4f1b-beb3-88e1a4ad104c\"}
15	15	Alzheimers Disease	11	{"pk": 11, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"f566f70f-7549-425e-8875-189fd81a982b\
16	16	Cerebral Aneurysm	12	{"pk": 12, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"bafce630-a3dc-413a-a3f4-c987606b9e1d\
17	17	Cerebral Aneurysm	12	{"pk": 12, "body": "[{\"type\": \"main\", \"value\": {\"mainheading\": \"Main Heading 1\"}, \"id\": \"bafce630-a3dc-413a-a3f4-c987606b9e1d\

Figure 20 – wagtailcore_revision database table

In this way, Wagtail can keep a record of all the revisions to a particular piece of content. This allows for the implementation of a diffing view between revisions from within the CMS. Removed content is then highlighted in red, and new content is highlighted in green, as shown in Figure 21.

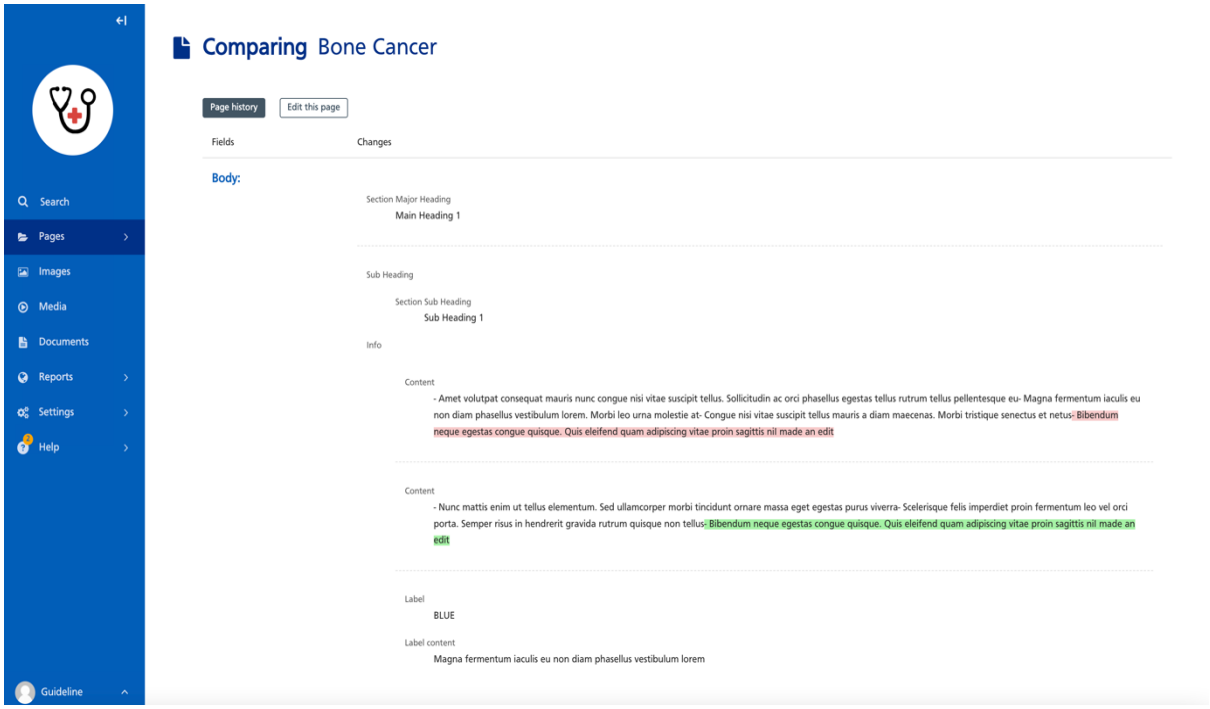

Figure 21 – Example of Diff View

6.3.6 Search Functionality

The search implementation is another example of how the underlying Django platform can be utilised to extend the functionality of the existing Wagtail site.

As mentioned in *5.2 – Django Web Framework*, a Django server is essentially a collection of Python modules known as apps registered in the main *settings.py* file. The root URL dispatcher can then inspect incoming URLs and call the appropriate functions, which should then return an HTTP response. The search functionality is one such extra app, where the URL dispatcher will pass any request URLs matching the */api/v2/search* pattern to the *search* app, and ultimately the below *search* function within its *views.py* file.

views.py M

Guidelines > server > search > views.py > search

```python
import json
from wagtail.models import Page
from django.http import HttpResponse
from guidelines.models import Guideline


def search(request):

    # Get search params
    search_query = request.GET.get('q', None)

    if search_query:
        guideline_results = Guideline.objects.live().search(search_query)
        guideline_result_ids = [p.page_ptr.id for p in guideline_results]
        search_results = Page.objects.live().filter(id__in=guideline_result_ids)

    else:
        search_results = Page.objects.none()

    # Extract page info from model instance
    page_data = []
    for page in search_results:
        page_data.append({
            'title': page.title,
            'id': page.id,
        })

    # Convert results to JSON
    data = {
        'search_query': search_query,
        'search_results': page_data,
    }
    json_data = json.dumps(data)

    return HttpResponse(json_data, content_type='application/json')

```

Figure 22 – Search app views.py

Note that this search app sits outside of the Wagtail URL space, and the Wagtail site is not even aware of it – it is a Django app reading from the Wagtail database tables via the Django ORM.

Also note the Guideline database tables only contain the page content, whereas the Page tables also contain the Wagtail specific information (moderation status, node position in tree, URLs). Searching the Guideline table rather than Page directly prevents any of these extra fields from polluting the search results.

The above search function then processes the request URL as follows:

- It attempts to extract any search parameters from the incoming URL query string
- If a search query is found, the function checks the *guidelines_guideline* table and extracts any 'live' (i.e., published) entries that contain the search parameters
- Using the Page IDs extracted in the previous stage, the function then extracts the full Page entry from the *wagtailcore_page* table
- The Page title and id fields are extracted from the results – note any further information from the Page instance could be added here if required
- The data is still in the Django QuerySet format used for processing with Python, and so it is converted to JSON
- An HTTPResponse is returned from the function, containing the search results as a JSON payload

Using the Postman HTTP client, the results of a search for cancer can be seen below:

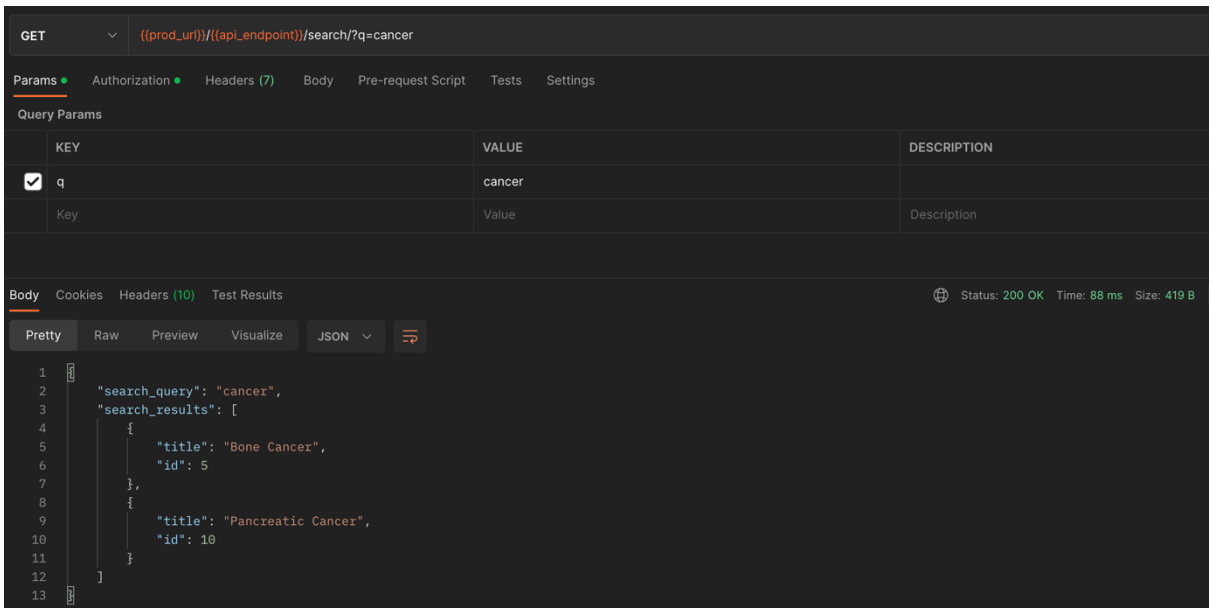

Figure 23 – Example of Search API Response for 'cancer'

6.4 Project Deliverables

6.4.1 Hosted CMS

Once successfully tested locally, the CMS was deployed to a DigitalOcean Droplet (*Droplets, 2023*). A Droplet is a Linux based virtual machine and can be configured to run the Docker engine required for this project. Once provisioned, it can be accessed remotely over an SSH session.

The codebase was developed using Git version control and hosted on a private GitHub repository. Once the Droplet was configured to have the appropriate SSH key to access the repository, the codebase was cloned to the Droplet and the Docker orchestration file (the *docker-compose.prod.yml* from Figure 5) executed. This then builds the Docker stack, either building the appropriate images from the local Dockerfile (as it the case for the Django and Nginx services) or pulls the appropriate image from a public repo (as with the PostgreSQL service).

Once built, a new Django superuser can be created to allow access to the admin area, and the Droplet firewall can be configured to allow HTTP access on port 80.

As of submission of this report, the hosted CMS is available at http://134.209.185.104/admin

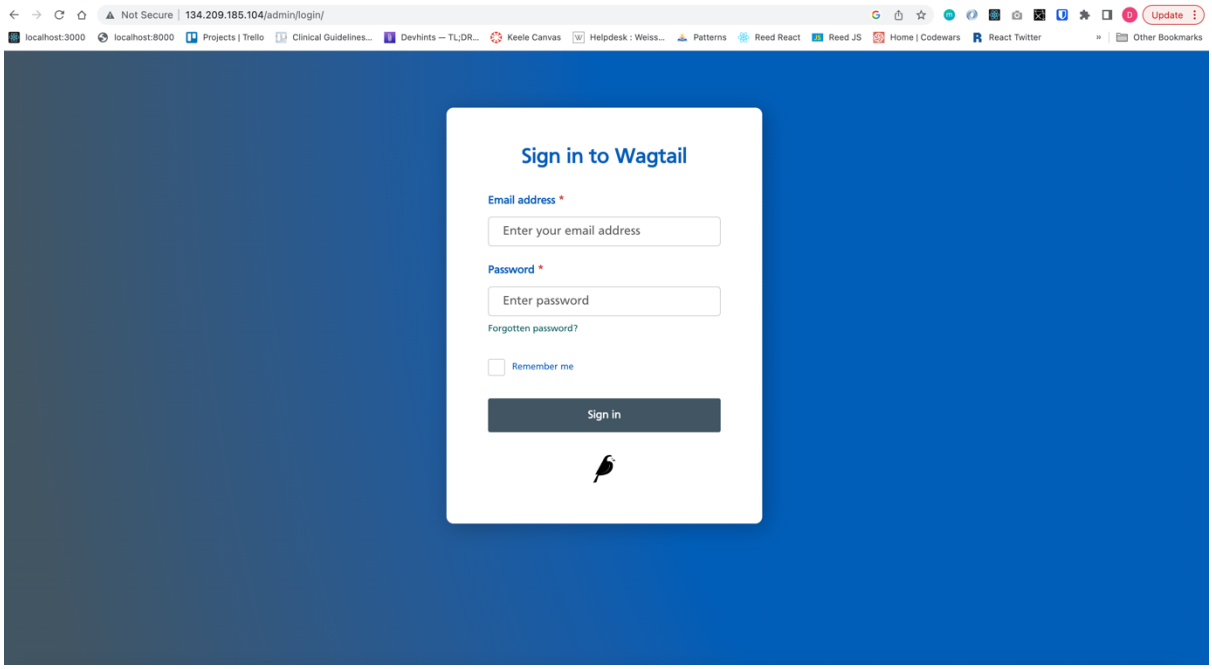

Figure 24 – Hosted CMS Login

Note that HTTPS was not configured for the purpose of the prototype, but this could also be added either manually or as another service from within the Docker orchestration file.

6.4.2 Exposed API

The CMS API is available to an HTTP client such as Postman (or even for terminal *cURL* requests) at the */api/v2* path of the URL. However, this project used the Swagger UI library to generate a browser based interface to allow developers to explore the API.

As of submission of this report, the Swagger interface is available at

http://134.209.185.104/api/v2/docs

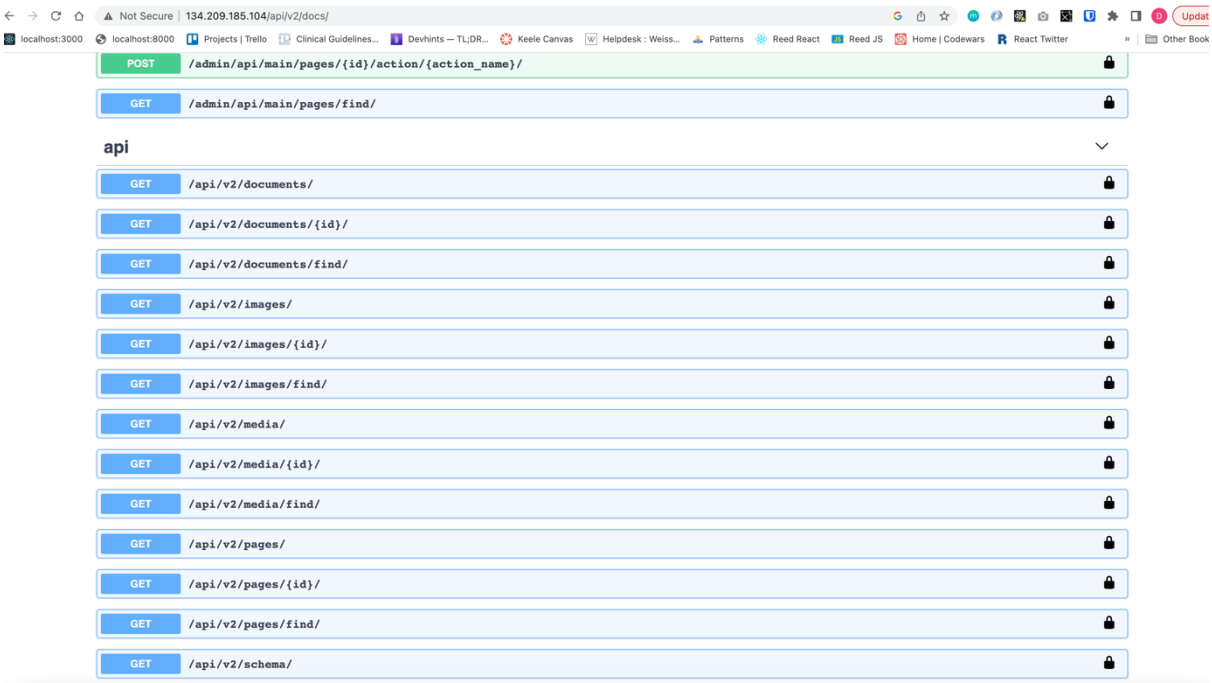

Figure 25 – Swagger UI Interface

6.4.3 Static Builds

Ultimately, the Wagtail CMS has to act as a data store for the eventual frontend solution. The current frontend is a native mobile app that wraps static markup. To prove that the Wagtail solution could be used as the backend for the existing setup, a static build was generated using Next.js, an extension of the React UI library. This static build could then be loaded into the existing wrapper application.

React creates an application by rendering various UI components. These components are essentially functions that return different parts of an overall UI and can be configured via the use of *props* (analogous to the arguments you would pass to a normal function) and *state* (information the application needs to maintain between renders). Traditionally, React is used

as a SPA and so the UI is created dynamically on the client machine via JavaScript interacting with the DOM, but Next.js allows for a static build to be generated via the WebPack build tool (or, in more modern versions of Next.js, TurboPack).

Since the individual Guideline pages have a dynamic structure, a component based solution such as React is well suited to generating the markup. It is not necessary to create multiple pages to reflect the different structures of the pages – a single React higher order component can receive the JSON for the appropriate Guideline from the API, and then conditionally render the content based on what is present in the payload.

Figure 26 shows a simplified example of how the *Guildeline.js* component was implemented:

```
1   import { MainHeading, SubHeading, Content, Label  } from '../../components';
2
3   const Guideline = ({ pageData }) => {
4
5       let components = [];
6
7       pageData.forEach((block) => {
8           if (block.type === 'MainHeading') {
9               components.push(<MainHeading props={block.data}/>)
10          }
11          if (block.type === "SubHeading") {
12              components.push(<SubHeading props={block.data}/>)
13          }
14          if (block.type === 'Content') {
15              components.push(<Content props={block.data}/>)
16          }
17          if (block.type === "Label") {
18              components.push(<Label props={block.data}/>)
19          }
20      })
21
22      return <div>{[...components]}</div>
23
24  }
25
26  export default Guideline;
```

Figure 26 – Simplified Example of Guideline Component

- Various other components are imported at the top of the file
- The '*pageData*' object (i.e., the appropriate JSON from the CMS) is passed down into the Guideline component as props (arguments)
- An empty array '*components*' is initialised
- The '*pageData*' object is iterated with a *forEach* loop
- On each iteration, the type of content is inspected, and then the appropriate component is called, receives props, and is pushed into the '*components*' array
- The Guideline component returns a spread of the '*components*' array

The effect of this pattern is that the Guideline will return a collection of sub-components relative to the received *pageData* prop contents. These sub-components will then generate the more detailed views, depending on the props they receive. This allows a single file to generate static markup for any particular combination of content types.

Two copies of the static build are to be submitted with this report – two are required to confirm that the trust specific filtering integrated to the Next build step is working as expected. Below shows the output from the Next.js build process:

```
info  - Linting and checking validity of types
info  - Creating an optimized production build
info  - Compiled successfully
info  - Collecting page data
info  - Generating static pages (14/14)
info  - Finalizing page optimization

Route (pages)                                 Size     First Load JS
┌ ● /                                         600 B          96.4 kB
├   /_app                                     0 B            74.3 kB
├ ● /[category]/[guideline] (3311 ms)         7.08 kB         103 kB
├   ├ /cardiovascular/stroke (603 ms)
├   ├ /respiratory/pneumonia (443 ms)
├   ├ /neurological/epilepsy (328 ms)
├   ├ /cardiovascular/deep-vein-thrombsis (301 ms)
├   ├ /cancers/bone-cancer
├   ├ /cancers/leukemia
├   ├ /cancers/pancreatic-cancer
├   └ [+4 more paths]
└ ○ /404                                      182 B          74.4 kB
+ First Load JS shared by all                 76.5 kB
  ├ chunks/framework-cda2f1305c3d9424.js      45.2 kB
  ├ chunks/main-2f7c9761d58c33d5.js           27.2 kB
  ├ chunks/pages/_app-aeaf2e663c92c00a.js     287 B
  ├ chunks/webpack-6ec97db2df7d879c.js        1.57 kB
  └ css/a1ef2fa54a4aea4a.css                  2.22 kB

○  (Static)  automatically rendered as static HTML (uses no initial props)
●  (SSG)     automatically generated as static HTML + JSON (uses getStaticProps)
```

Figure 27 – Next.js Build Process

In the build process of Figure 27, it can be seen that Next has used the *[guideline]* component to generate all of the Guideline pages. *_app* is the main index page, and then all the CSS and JS assets have been combined, minified, and chunked for performance via TurboPack. This professional build process is a significant advantage of using a frontend framework to generate the static build, rather than manually writing and updating HTML markup.

Figure 28 shows the 'Bone Cancer' page from two different builds – one for the 'EAST' Trust, and one for the 'WEST' Trust. Note the content is all the same, except for the trust specific content.

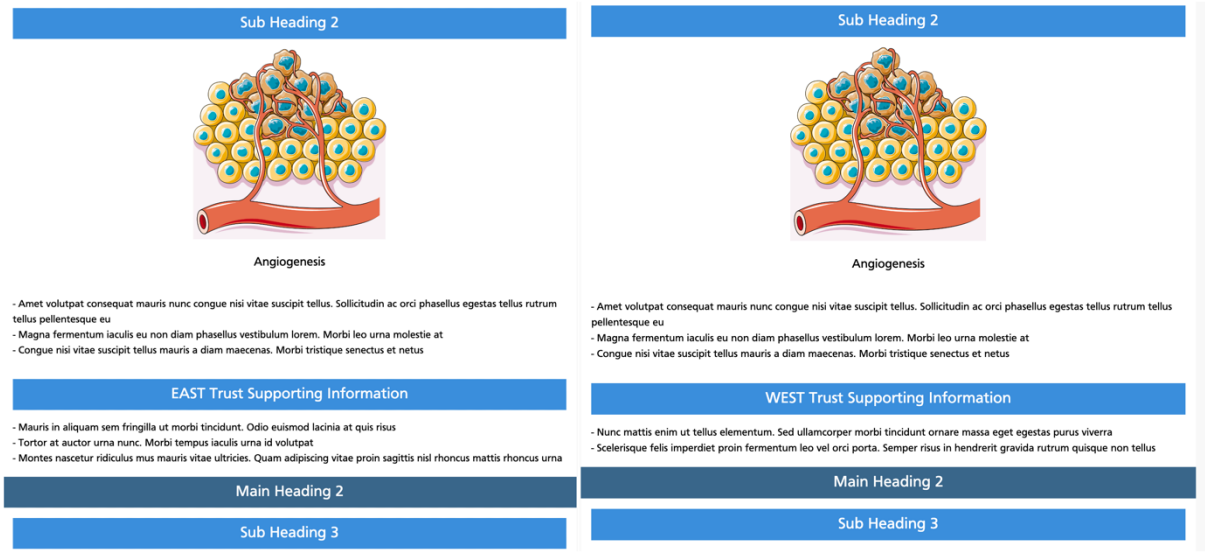

Figure 28 – Differing Trust builds

7 Testing

The project planning document provisioned for three different testing sprints:

1. CMS Testing
2. API Testing
3. Frontend Testing

All test sprints were scheduled to last for a week to give sufficient time to confirm functionality. The priority was to get the CMS developed and deployed, as no substantial testing could be started until that stage – the API testing required a populated database to test against, and the frontend required an API to read data from.

The testing strategy used was general user acceptance testing, with the developer using the CMS, the Swagger interface, and the client builds to confirm that the system was behaving as expected. This is acceptable for a proof of concept project with a single developer, but would likely not be the most efficient testing strategy if the project was taken forward (see *8.2.1 – Improved Testing Strategy*).

The Next.js frontend also comes with a built in linting library to assist with identifying potential code issues during the development phase.

7.1 Identified Issues

During testing, a potential issue was discovered relating to serving images and media over HTTP.

The CMS was hosted on a DigitalOcean VPS on an HTTP URL. During development of the frontend, this was not an issue as the Next development server hosted the images and media without error.

However, after building, the original plan was to deploy the static frontends to a deployment service such as Netlify or Vercel to allow for easier marking. This caused an issue as these platforms were enforcing a no mixed-content policy *(web.dev, 2020)* – this is an enhanced security feature that prevents HTTP assets being loaded on an HTTPS page. It prevents, as an example, cookies set on an HTTPS domain being sent over an HTTP connection. For this reason, the images and media hosted on the CMS would not load.

There are solutions for this – for example, the images and media could be moved into the build step and therefore hosted on the same domain, or the CMS configured to serve over HTTPS, but both of these options would have involved considerable work not involved with the core project deliverables.

It was agreed with the project supervisor that a copy of the static build could be hosted locally on the marking member of staff's machine using a local web server. As a development server, this would get around the HTTPS only enforcement.

8 Conclusions and Potential Future Development

8.1 Project Feedback

Project feedback was received from Professor Ed de Quincey, the project contact at the University (see *Appendix* for email).

Professor de Quincy thought that the platform was a viable solution to the original brief, with a simple enough interface to allow people with basic IT skills to be able to use it as intended. This was the key aim of the project and shows that Wagtail could be a potential platform for the Medical Guidelines project.

In particular, Professor de Quincey liked the live preview functionality that comes as standard with Wagtail, and has asked the project developer for the actual platform to recreate something similar.

The identified potential issues included some of the admin UI not being entirely intuitive, and that the report structure was not quite matching of the actual guidelines. However, the admin UI could have been improved with further development time and user testing feedback, and limited access to actual guidelines had restricted how accurately the prototype versions could be matched up – since the Wagtail models can essentially be structured in any way the developer requires, I feel confident that they could be an exact match if the full brief was made available.

8.2 Ideas for Future Development

8.2.1 Improved Testing Strategy

For a proof of concept, the current lack of robust testing beyond basic functionality is not a huge issue, but if the project was taken forward as a viable commercial solution then a more complete testing strategy should be implemented.

Django comes with the Python *unittest* testing framework as standard for writing tests, although PyTest (*docs.pytest.org, n.d.*) is another popular testing library. Wagtail itself is a well tested framework and so the core functionality of the CMS should be reliable, but it would increase the robustness of the platform if sufficient unit tests were written against the API. PyTest allows you to mock HTTP requests and use test databases, so it is possible to write tests covering the normal CRUD operations of your content and make the appropriate assertions against them.

For the frontend, a possible solution would be the Cypress (*Cypress, n.d.,*) testing library. Cypress is technology agnostic and, as long as the application can run in a browser, Cypress can run tests against it. Cypress is mainly used for end to end testing for entire user journeys such as authorisation, navigation, and interactions, but can also run unit and integration tests if required.

Both PyTest and Cypress can be integrated into a CI/CD pipeline such as GitHub Actions or AWS CodePipeline to confirm functionality before an application build or deploy.

8.2.2 Monolithic Codebase

This project was primarily about proposing Wagtail as a viable solution for the content authoring requirement of the wider Clinical Guidelines project. Next.js has been used to prove that a headless Wagtail configuration could supply JSON for an already existing frontend client.

However, Wagtail is at its core a web server, serving its own HTML templates in response to web requests. Traditionally, it runs in a Python environment on a server and dynamically creates the required pages in response to HTTP requests. During research for this project, the *wagtail-bakery* package (*Github, 2023*) was discovered which allows a developer to create a static export of the entire Wagtail site. This could potentially entirely remove the need to support a frontend client, and run the entire project via Wagtail using the standard templating system. As well as removing the frontend client, it would also remove the duplicated work of creating separate templates for the live preview as Wagtail would simply use the production templates instead.

8.2.3 Managed Services

For the proof of concept, the Docker services include a PostgreSQL database with an associated volume. The CMS images and media files are also stored on volumes, served via the Nginx reverse proxy.

For a production environment, a managed database service would be preferable. This would remove all database administration tasks such as backups, patching and performance monitoring to a third party, specialised service.

Similarly, the static assets could be moved to a third-party service and even served via a CDN if required. This removes further unnecessary load from the web server and allows it to concentrate on handling core business logic. Like the database third party services, a managed file storage service such as AWS S3 or DigitalOcean Spaces would also likely offer backup functionality and other optimizations.

8.2.4 Stack Review

In hindsight, the Swagger interface, although still useful, was perhaps not entirely necessary for this project. Wagtail abstracts much of the URL management from the user, and since the frontend is essentially static without much mutation activity towards the server (i.e., POST, PUT and DELETE requests), there is not a large number of useful endpoints exposed via the Swagger interface. In this instance, a comprehensive Postman collection or similar would have likely been sufficient. Although it could be argued that it may as well be used since it is already installed, it is generally considered good practice to remove unnecessary libraries in a production environment. This reduces the surface area of the codebase and means one less library to maintain.

8.3 Conclusions

The core requirement of the initial project brief was to '*produce a web- based application where users can utilise CRUD functions … with existing guideline documents.'* The Wagtail solution delivers this - users can create and edit Guidelines within the constraints of the appropriate Page model. A combination of core Wagtail content types, custom content types and third party libraries mean that any realistic combination could be delivered to match the requested guideline structure.

Another requirement was to '*implement a multi-level user admin system.*' As a CMS already used by many major companies to manage their own content, Wagtail offers a sophisticated, out of the box solution involving management of user accounts, delegation of various levels of privilege and highly configurable workflows to assist in the management of content from draft to publishing, including the necessary moderation steps required for complex content types with multiple stakeholders. Such a system would be highly complex to implement natively, and it would save large amounts of development and testing time by using an already proven configuration such as that offered by Wagtail.

The brief also required that the system be tested '*to ensure it meets usability guidelines.*' Whilst some of the feedback suggested that there could be improvements made in this area, the Wagtail admin is highly customisable, and with extra UI/UX research, user testing and feedback, and further development time, the editing interface could be made more intuitive

and better suited to the specific project needs. As previously mentioned, there is also comprehensive existing documentation for the editor interface, separate from the more involved developer documentation.

In conclusion, the Wagtail platform is clearly a potential solution for the initial project specification - it delivers on the core requirements, and even has potential advantages over the bespoke solution currently in development.

Primarily, as an existing solution with much of the core functionality already proven, the development time would be much less. Wagtail has already had many hours of development and testing time dedicated to it. There are likely to be many complexities involved in creating, from scratch, a feature such as a moderation system and it would be quite difficult to predict exactly how long and how much resource that might take to realise.

There is also the advantage of using a well-maintained solution backed by a large existing community and prominent technology companies – beyond simple functionality, there is also the extra work that has gone into other areas of the platform that might not be considered or have the resource for a bespoke solution, such as third party libraries to extend functionality, accessibility testing (*Wagtail Accessibility, 2023*) and existing test suites.

Finally, there is the issue of institutional knowledge – when building a complex, bespoke software solution, a project (especially a smaller one) is often vulnerable to losing certain key members. The more complex the solution, the harder it may be to replace them and the longer amount of time it may take to get new project members or consultants onboarded. With an

open-source solution such as Wagtail, any Wagtail (or even Django) developer can quickly get up to speed with the application and make useful contributions,.

The obvious major advantage of a fully bespoke platform would be that it can essentially be customised to a customer's exact needs, but for the case of what is ultimately a variation of a very common existing requirement on the Internet – a content repository with a moderately complex moderation workflow and several user levels – it could be argued that any potential custom application is simply duplicating a solution that has already been created and proven.

References

Amazon Web Services, Inc. (n.d.). *Free Databases - AWS*. [online] Available at: https://aws.amazon.com/free/database/ [Accessed 20 Jun. 2023].

Christie, T. (2011). Home - Django REST framework. [online] Django-rest-framework.org. Available at: https://www.django-rest-framework.org/.

Cypress (n.d.). *Open-Source E2E Testing Tools & UI Testing Framework | cypress.io*. [online] Available at: https://www.cypress.io/app/ [Accessed 28 Jun. 2023].

Django (2019). The Web framework for perfectionists with deadlines | Django. [online] Djangoproject.com. Available at: https://www.djangoproject.com/.

Django Project Contrib. (n.d.). Django. [online] Available at: https://docs.djangoproject.com/en/4.2/ref/contrib/ [Accessed 20 Jun. 2023].

django-treebeard.readthedocs.io. (n.d.). django-treebeard — django-treebeard 4.0.1 documentation. [online] Available at: https://django-treebeard.readthedocs.io/en/latest/ [Accessed 19 Jun. 2023].

Docker Documentation. (2020). *Control startup and shutdown order in Compose*. [online] Available at: https://docs.docker.com/compose/startup-order/.

docs.celeryq.dev. (n.d.). *Celery - Distributed Task Queue — Celery 5.2.7 documentation*. [online] Available at: https://docs.celeryq.dev/en/stable/index.html.

docs.pytest.org. (n.d.). *pytest: helps you write better programs — pytest documentation*. [online] Available at: https://docs.pytest.org/en/7.3.x/.

docs.wagtail.org. (n.d.). *TableBlock — Wagtail Documentation 5.0.1 documentation*. [online] Available at: https://docs.wagtail.org/en/stable/reference/contrib/table_block.html [Accessed 20 Jun. 2023].

docs.wagtail.org. (2023). Welcome to Wagtail's documentation — Wagtail Documentation 5.0.1 documentation. [online] Available at: https://docs.wagtail.org/en/stable/ [Accessed 19 Jun. 2023].

Download Django (n.d.) | Django. [online] Available at: https://www.djangoproject.com/download/.

Droplets (2023) *Digitalocean.com*. Available at: https://www.digitalocean.com/products/droplets (Accessed: June 28, 2023).

Elastic.co. (2019). *Open Source Search: The Creators of Elasticsearch, ELK Stack & Kibana | Elastic*. [online] Available at: https://www.elastic.co/.

GitHub. (2023). *Wagtail-bakery*. [online] Available at: https://github.com/wagtail-nest/wagtail-bakery [Accessed 28 Jun. 2023].

gunicorn.org. (n.d.). Gunicorn - Python WSGI HTTP Server for UNIX. [online] Available at: https://gunicorn.org/.

MDN Web Docs. (n.d.). Django introduction. [online] Available at: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction.

Meta Open Source (2023). React. [online] react.dev. Available at: https://react.dev/.
Mitchell, James Anthony (2022) User-centred design of bedside clinical guidelines for mobile devices. Doctoral thesis, Keele University.

Mitchell, J.E., Ed de Quincey, Pantin, C.F.A. and Naveed Mustfa (2020). The Development of a Point of Care Clinical Guidelines Mobile Application Following a User-Centred Design Approach. doi:https://doi.org/10.1007/978-3-030-49757-6_21.

Mitchell, J.E., Ed de Quincey, Pantin, C.F.A. and Naveed Mustfa (2021). 15 Usability Recommendations for Delivering Clinical Guidelines on Mobile Devices. 2021. DOI: 10.14236/ewic/HCI2021.7

nextjs.org. (n.d.). Next.js by Vercel - The React Framework. [online] Available at: https://nextjs.org/.

NHS Digital. (2018). NHS.UK content migration update. [online] Available at: https://digital.nhs.uk/blog/transformation-blog/2018/nhs.uk-content-migration-update.

npm. (2020). *npx*. [online] Available at: https://www.npmjs.com/package/npx [Accessed 8 Jul. 2023].

Pantin, C., Mucklow, J., Rogers, D., Cross, M. and Wall, J. (2006). Bedside clinical guidelines: the missing link. Clinical Medicine, 6(1), pp.98–104. doi:https://doi.org/10.7861/clinmedicine.6-1-98.

Pereira, V.C., Silva, S.N., Carvalho, V.K.S., Zanghelini, F. and Barreto, J.O.M. (2022). Strategies for the implementation of clinical practice guidelines in public health: an overview of systematic reviews. Health Research Policy and Systems, [online] 20(1). doi:https://doi.org/10.1186/s12961-022-00815-4.

Postman (2021). *Postman | The Collaboration Platform for API Development*. [online] Postman. Available at: https://www.postman.com/.

Python.org. (2019). venv — Creation of virtual environments — Python 3.8.1 documentation. [online] Available at: https://docs.python.org/3/library/venv.html.

Ramm, J. (2022). *WagtailMath*. [online] GitHub. Available at: https://github.com/JamesRamm/wagtailmath [Accessed 20 Jun. 2023].

Saltzer, J. H. (1974) “Protection and the control of information sharing in multics,” *Communications of the ACM*, 17(7), pp. 388–402. doi: 10.1145/361011.361067.

Smith, H., Pryce A, Carlisle, L., Jones, J.R., Scarpello, B. and Pantin, A. (1998). Appropriateness of acute medical admissions and length of stay. 31(5), pp.527–32.

StackShare. (n.d.). Why developers like Django. [online] Available at: https://stackshare.io/django.

Swagger.io. (2019). The Best APIs are Built with Swagger Tools | Swagger. [online] Available at: https://swagger.io/.

Swagger.io. (2020). *OpenAPI Specification - Version 3.0.3 | Swagger*. [online] Available at: https://swagger.io/specification/.

Wagtail Accessibilty. (2023). *Introducing Wagtail’s new accessibility checker*. [online] Available at: https://wagtail.org/blog/introducing-wagtails-new-accessibility-checker/ [Accessed 8 Jul. 2023].

Wagtail CMS. (n.d.). Django Content Management System. [online] Available at: https://wagtail.org/.

Wagtail CMS. (n.d.). Who Uses Wagtail. [online] Available at: https://wagtail.org/who-uses-wagtail/.

Wagtail Guide. (n.d.). Wagtail User Guide. [online] Available at: https://guide.wagtail.org/en-latest/.

web.dev. (n.d.). *What is mixed content?* [online] Available at: https://web.dev/what-is-mixed-content/ [Accessed 8 Jul. 2023].

Appendix

From: Ed de Quincey <****@********>
Date: Friday, 30 June 2023 at 14:18
To: Daniel Bayford <****@********>
Cc: Beran Necat <****@********>

Subject: Re: Deployed Wagtail CMS

Here you go:

1. From your initial exploratory use, would you see Wagtail as a potential solution for the content authoring platform?

Yes. I think this looks simple enough to use for people with basic IT skills. There are some small usability tweeks that would be needed to make certain interactions clearer (that I assume are part of Wagtail) but overall this supports the key functionality and processes that would be needed.

2. What in particular do you like about the platform?

I thought the side by side editing was the best feature and is something I've shown the developer of the CMS we are currently finalising and asked him to make something similar.

3. Are there any obvious issues?

The only issues relate to not quite fitting the structure of the guidelines but that is due to you not having access to the full set and I imagine would be trivial to include with more time and feedback from us.

Best wishes,

Ed

Professor Ed de Quincey SFHEA, FBCS

School of Computer Science and Mathematics
Keele University