RIS Backend Dokumentation

Release 1.0

Maximiliano Santander

30.12.2024

Hauptinhalt

Willkommen zur Dokumentation des RIS Backend-Projekts. Diese Dokumentation bietet einen Überblick über die Planung, den Entwurf, die Implementierung und die Tests des Projekts.

KAPITEL 1

Inhaltsverzeichnis

1.1 Projektplanung und Analyse

1.1.1 Einleitung

Projektumfeld

- Ich mache zurzeit eine Umschulung zum Fachinformatiker für Anwendungsentwicklung beim Städtische Berufsschule III Matthäus Runtinger in Regensburg.
- Aktuell absolviere ich meine Ausbildung als Fachinformatiker für Anwendungsentwicklung bei der Firma RIS Web- & Software-Development GmbH & Co. KG in Regensburg, nachfolgend RIS genannt.
- Der Betrieb beschäftigt 13 Mitarbeiter. RIS Development ist ein offizieller Servicepartner der E-Commerce-Software von JTL.
- Neben dem Support für die JTL-Software werden auch individuelle Komponenten für den Webshop oder die Warenwirtschaft (WAWI) entwickelt. Diese reichen vom Erstellen von Template-Vorlagen für Dokumente und Webdesign bis hin zu erweiterten Funktionalitäten in Form von Shop-Plugins.
- Es werden auch individuelle Lösungen für Kunden außerhalb der E-Commerce-Software erstellt. Der Umfang reicht vom Entwickeln von Corporate Designs, Social-Media-Marketing, Neugestaltung des Website-Designs, Erstellung von Web-Applikationen bis hin zur Umsetzung nativer Software.

Projektziel

Das Ziel des Projekts ist der Aufbau und die Implementierung eines Backends für RIS. Moderne Technologien wie Django, Laravel oder Node.js werden in Betracht gezogen. Dieses Backend soll folgende Anforderungen erfüllen:

1. **Volle Kontrolle**: - Alle Backend-Funktionen sollen feinjustiert werden und unabhängig von allgemeinen Lösungen wie WordPress arbeiten. - Die Nutzung von Templates und Plugins, die oft mit Abogebühren verbunden sind, soll vermieden werden.
2. **Verschlankung**: - Das Backend soll gezielt auf die spezifischen Bedürfnisse von RIS zugeschnitten werden, um unnötige Funktionen zu vermeiden und die Bedienbarkeit zu vereinfachen.
3. **Effiziente Inhaltspflege**: - Die Verwaltung von Inhalten soll ohne die Einschränkungen und Komplexität eines WordPress-Templates erfolgen, wodurch ein direkter und effizienter Arbeitsprozess ermöglicht wird.
4. **Performance**: - Die Website soll serverseitig gerendert werden, und die Seiteninhalte sollen vollständig im Cache gespeichert werden, um mit neuer Server-Hardware optimale Ladegeschwindigkeiten zu erreichen.

Ausgangssituation

Die aktuelle Website von RIS Web- & Software-Development GmbH & Co. KG, im Folgenden RIS genannt, basiert auf WordPress und erfüllt ihre grundlegende Funktion. Jedoch sind wir mit WordPress auf allgemeine Lösungen angewiesen, die nicht immer unseren spezifischen Anforderungen entsprechen:

- WordPress erfordert die Nutzung allgemeiner Plugins, die oft nicht an die Anforderungen von RIS angepasst sind.
- WordPress enthält viele Funktionen, die nicht benötigt werden, was unnötigen Overhead beim Lernen und Bedienen verursacht.
- Die Verwaltung von Inhalten in WordPress mit Templating hat sich als umständlich und unvorhersehbar erwiesen.

1.1.2 Abgrenzung des Aufgabenbereichs

Der Fokus des Projekts liegt auf der Entwicklung des Backends. Frontend-Aufgaben sowie Optimierungen in Bereichen wie Blogs sind nicht Teil des Projekts, da ein Kollege sich um das Frontend kümmern wird. Außerdem werden die Sicherheitsmaßnahmen als Standard angesehen, da nur minimale Daten von den Kunden abgefragt werden (z.B. keine Zahlungsinformationen).

1.1.3 Projektplanung

Projektphasen

- Das Projekt beginnt am 11. November 2024 und endet am 6. Januar 2025, was insgesamt etwa 8 Wochen umfasst. Innerhalb dieses Zeitraums werden die verschiedenen Phasen des Projekts parallel zu den regulären Aufgaben des Unternehmens entwickelt.

- Die projektbezogenen Tätigkeiten werden so geplant, dass die 80 Stunden für das Projekt effizient genutzt werden, um die Ziele fristgerecht zu erreichen.

Grobe Zeitplanung

Projektphase	Geplante Zeit
Projektplanung/Analyse	8 h
Entwurf	16 h
Implementierung	36 h
Test/Durchführung	8 h
Dokumentation	8 h
Abnahme	4 h
Gesamt	**80 h**

Detallierter Zeitplanung

Ressourcenplanung

Software:

- Framework: Django
- Django Tools: z.B. Pillow
- DB-Software: SQLite für Entwicklung, PostgreSQL für Produktion
- IDE: PyCharm
- Cache-Tools: Redis, Middleware
- Test-Software: PyTest Django, PyTest, FactoryBoy (Testdaten Erstellung), pdoc
- Versionverwaltung: GitLab

Hardware: - Laptop

1.1.4 Analysephase

Auswahl der Technologieplattform

Für das Projekt wurden moderne Technologien wie Django, Laravel und Node.js in Betracht gezogen, wobei wir uns nach gründlicher Recherche für Django entschieden haben. Dieses Framework bietet robuste und skalierbare Backend-Lösungen und passt ideal zu den Anforderungen unseres Projekts. Ich und der Rest unseres Entwicklungsteams haben bereits umfassende Kenntnisse in Python sowie in JavaScript, HTML und CSS.

Vorteile von Django

- **Caching**: Ein flexibles Caching-System unterstützt Technologien wie Middleware Cache und Redis (Backend Cache), was die Anwendungsperformance erheblich steigert.

- **Vollständiges Framework**: Django ist ein „batteries-included“ Framework mit integrierten Funktionen wie Authentifizierung, Administrationsoberflächen und einem leistungsstarken ORM, was die Entwicklung effizienter gestaltet.
- **Große Community und Dokumentation**: Die aktive Community und die umfassende Dokumentation erleichtern die Problemlösung und kontinuierliche Wartung.

In der Entwicklungsphase nutzen wir SQLite wegen seiner Einfachheit und weil es schon by Default in Django konfiguriert ist. In der Produktionsphase setzen wir PostgreSQL ein, um von den erweiterten Funktionen, der Leistung und der Skalierbarkeit zu profitieren, die unser Projekt benötigt.

Ist-Analyse

- Das Projekt findet im Rahmen der Webentwicklung bei der RIS statt. Ziel ist es, ein neues, maßgeschneidertes Backend für die Unternehmenswebsite zu entwickeln, um die Abhängigkeit von WordPress zu vermeiden.
- Die bestehende Website basiert auf WordPress, was zu Problemen wie unnötigem Overhead, langsamerer Performance und umständlicher Inhaltsverwaltung führt. Die derzeit verwendeten Plugins und Vorlagen entsprechen nicht den spezifischen Anforderungen von RIS.

Wirtschaftlichkeitsanalyse

Eine klassische Wirtschaftlichkeitsanalyse, inklusive Projektkosten und Amortisationsdauer, wurde für dieses Projekt aus folgenden Gründen nicht durchgeführt:

Ausbildungskontext - Projekt im Rahmen der Berufsausbildung - Fokus auf Kompetenzerwerb statt wirtschaftlichem Nutzen - Keine zusätzlichen Personal- oder Ausbildungskosten

Ressourcen und Kosten - Nutzung vorhandener Infrastruktur (Hardware/Software) - Einsatz von Open-Source-Technologien (Django, Python) - Keine externen Ressourcen oder Lizenzen erforderlich - Integration in bestehende IT-Systeme

Projektrahmen - Entwicklung durch Auszubildenden - Teil der regulären Ausbildungszeit - Keine zusätzlichen Betriebskosten

Die Projektbewertung konzentrierte sich stattdessen auf:

- Technische Umsetzung
- Code-Qualität
- Dokumentation
- Lernerfahrung

1.1.5 Vorgehensmodelle

Für dieses Projekt wurde das Wasserfallmodell gewählt, da es optimal zu den folgenden Projektbedingungen passt:

- **Klare Zeitplanung**:

  - Fester Zeitrahmen von 80 Stunden
  - Klar definierte Projektphasen mit zugewiesenen Zeitbudgets
- **Einzelentwickler-Projekt**:
  - Keine Notwendigkeit für komplexe Team-Koordination
  - Direkte Kontrolle über alle Entwicklungsphasen
  - Vereinfachte Entscheidungsprozesse
- **Vordefinierte Ressourcen**:
  - Festgelegte technische Infrastruktur
  - Bereits bestimmte Entwicklungswerkzeuge
  - Klare Hardware-Anforderungen
- **Spezifisches Projektziel**:
  - Eindeutig definierter Funktionsumfang
  - Klare Abgrenzung der Anforderungen
  - Vorhersehbare technische Umsetzung

Diese Rahmenbedingungen ermöglichen eine sequentielle Abarbeitung der Projektphasen, wie sie im Wasserfallmodell vorgesehen ist.

1.1.6 Make or Buy-Entscheidung

- Als weborientiertes Entwicklungsunternehmen war die Entscheidung klar, eine fertige Website-Lösung zu kaufen, kam nicht in Frage. Stattdessen haben wir uns dafür entschieden, ein eigenes Backend zu entwickeln, um volle Kontrolle über alle Funktionen und Anpassungen zu haben. Diese Entscheidung geht jedoch über eine einfache Neugestaltung hinaus. Es ging darum, eine Grundlage für eine effiziente, skalierbare und langfristig wartbare Lösung zu schaffen, die den spezifischen Anforderungen von RIS gerecht wird.

1.1.7 Nutzwertanalyse

Kriterium	Gewichtung	Django	Laravel	Node.js
Template-System & Caching	30%	10 (3.0)	7 (2.1)	6 (1.8)
Entwicklungsgeschwindigkeit	25%	9 (2.25)	8 (2.0)	7 (1.75)
Database Integration & ORM	25%	9 (2.25)	7 (1.75)	6 (1.5)
Skalierbarkeit	20%	8 (1.6)	7 (1.4)	9 (1.8)
Gesamtwertung	**100%**	**9.1**	7.25	6.85

Technische Eignung:

- Besonders stark im Bereich Template-System & Caching
- Ideal für die geforderte serverseitige Renderung

Teamkompetenz:

- Vorhandene Python/JavaScript-Expertise außer CSS und HTML im Entwicklungsteam

Database Integration & ORM:

- Robustes ORM für komplexe Datenbankstrukturen
- Automatische Migrationen und optimierte Queries

1.1.8 Anwendungsfälle

Administratoren können:

- Inhalt erstellen, bearbeiten und löschen

Web-Users können:

- Verschiedene Formulare ausfüllen und schicken
- Web surfen

Für eine visuelle Darstellung der Anwendungsfälle, siehe *Use Case Diagramm*.

1.2 Entwurf

1.2.1 Architekturdesign

Das Backend verwendet die MTV-Architektur (Model-Template-View), die speziell für Django entwickelt wurde. Diese Architektur trennt die Datenlogik (Model), die Präsentationslogik (Template) und die Steuerlogik (View) klar voneinander.

1.2.2 MTV-Architektur

Siehe *MTV-Architektur* in den Diagrammen für weitere Details.

1.2.3 ERD-Diagramm

Die Hauptentitäten des ER-Modells sind:

- **Page**: Verwaltung von mehrsprachigen Inhalten.
- **Block**: Wiederverwendbare Inhaltsblöcke.
- **MenuItem**: Navigationselemente.

1.2.4 Systemkomponenten

Die wichtigsten Systemkomponenten umfassen:

- **Page**: Zentrale Entität für Webseiteninhalte, speichert mehrsprachige Inhalte (DE/EN) und verwaltet Meta-Informationen.
- **Block**: Enthält Template-basierte Inhaltsblöcke, sortierbare Komponenten und wiederverwendbare Strukturen.

- **MenuItem**: Verwaltet Navigationsstrukturen, template-basierte Menüelemente und sortierbare Menükomponenten.

1.2.5 Beziehungen

- **Page** (n) hat **Block** (m)
- **Page** (1) hat **MenuItem** (0..1)

Diese Struktur ermöglicht: - Flexible Seitenerstellung - Wiederverwendbare Komponenten - Mehrsprachige Inhalte - Geordnete Navigation

1.2.6 Datenmodell

Das Datenmodell umfasst die folgenden Klassen:

```
from django.db import models

class Page(models.Model):
    title = models.CharField(max_length=255)
    url_path = models.CharField(max_length=2048)
    language = models.CharField(
        max_length=2,
        choices=[('EN', 'English'), ('DE', 'Deutsch')],
        default='DE',
    )
    meta_description = models.TextField(blank=True, null=True)
    meta_keywords = models.CharField(max_length=255, blank=True,␣
↪null=True)
    is_published = models.BooleanField(default=False)
    published_at = models.DateTimeField(blank=True, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

class Block(models.Model):
    template = models.TextField()
    sorting = models.IntegerField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

class MenuItem(models.Model):
    page = models.ForeignKey('Page', on_delete=models.CASCADE)
    template = models.TextField()
    sorting = models.IntegerField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

class PageBlock(models.Model):
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    page = models.ForeignKey('Page', on_delete=models.CASCADE)
    block = models.ForeignKey('Block', on_delete=models.CASCADE)
```

1.2.7 Cache-Strategie

Das Backend implementiert eine effiziente Cache-Strategie, um die Performance zu optimieren. Es wird Redis für das Caching verwendet, um schnelle Zugriffszeiten zu gewährleisten.

```
from django.core.cache import cache
from django.db.models.signals import post_save, post_delete
from django.dispatch import receiver

@receiver([post_save, post_delete], sender=Block)
@receiver([post_save, post_delete], sender=MenuItem)
@receiver([post_save, post_delete], sender=PageBlock)
def invalidate_cache(sender, **kwargs):
    cache.clear()
```

1.3 Implementierung

1.3.1 Details der Umsetzung

Die Implementierung des Backends umfasst mehrere Kernkomponenten und Technologien, die im Folgenden beschrieben werden.

1.3.2 Page-Modell

Das Page-Modell ist die zentrale Entität für Webseiteninhalte. Es unterstützt mehrsprachige Inhalte und verwaltet Meta-Informationen.

```
from django.db import models

class Page(models.Model):
    title = models.CharField(max_length=255)
    url_path = models.CharField(max_length=2048)
    language = models.CharField(
        max_length=2,
        choices=[('EN', 'English'), ('DE', 'Deutsch')],
        default='DE',
    )
    meta_description = models.TextField(blank=True, null=True)
    meta_keywords = models.CharField(max_length=255, blank=True,␣
↪null=True)
    is_published = models.BooleanField(default=False)
    published_at = models.DateTimeField(blank=True, null=True)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

1.3.3 Block-Modell

Das Block-Modell repräsentiert wiederverwendbare Inhaltskomponenten, die in mehreren Seiten verwendet werden können.

```
from django.db import models

class Block(models.Model):
    template = models.TextField()
    sorting = models.IntegerField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

1.3.4 MenuItem-Modell

Das MenuItem-Modell verwaltet Navigationsstrukturen und verlinkt Seiten mit anpassbaren Templates.

```
from django.db import models

class MenuItem(models.Model):
    page = models.ForeignKey('Page', on_delete=models.CASCADE)
    template = models.TextField()
    sorting = models.IntegerField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

1.3.5 PageBlock-Modell

Das PageBlock-Modell verwaltet die Beziehungen zwischen Seiten und Blöcken.

```
from django.db import models

class PageBlock(models.Model):
    page = models.ForeignKey('Page', on_delete=models.CASCADE)
    block = models.ForeignKey('Block', on_delete=models.CASCADE)
```

1.3.6 Caching

Das Backend implementiert eine effiziente Cache-Strategie, um die Performance zu optimieren. Es wird Redis für das Caching verwendet.

```
from django.core.cache import cache
from django.db.models.signals import post_save, post_delete
from django.dispatch import receiver

@receiver([post_save, post_delete], sender=Block)
@receiver([post_save, post_delete], sender=MenuItem)
@receiver([post_save, post_delete], sender=PageBlock)
def invalidate_cache(sender, **kwargs):
    cache.clear()
```

1.3.7 Django Signals

Django Signals werden verwendet, um bestimmte Aktionen automatisch auszulösen, wenn Änderungen an den Modellen vorgenommen werden.

```
from django.db.models.signals import post_save
from django.dispatch import receiver
from .models import Page

@receiver(post_save, sender=Page)
def update_page_cache(sender, instance, **kwargs):
    cache_key = f"page_{instance.url_path}_{instance.language}"
    cache.set(cache_key, instance)
```

1.3.8 Software & Technologien

- **Django-Ökosystem:**
  - Django Framework
  - django-simple-history
  - django-redis
  - django-environ
  - Django Debug Toolbar
- **Datenbank & Caching:**
  - SQLite (Entwicklung)
  - PostgreSQL (Produktion)
  - Redis (Caching-Server)
- **Server & Deployment:**
  - Gunicorn (WSGI)
  - Nginx
  - WhiteNoise

  - systemd

- **Testing & QA:**
  - PyTest & PyTest-Django
  - FactoryBoy
  - Coverage.py
- **Dokumentation:**
  - Sphinx
  - sphinx-rtd-theme
  - sphinxcontrib-plantuml
- **Entwicklungstools:**
  - Python 3.8+
  - PyCharm IDE
  - Git & GitLab
  - make
- **Media & Assets:**
  - Pillow

1.4 Test und Durchführung

Diese Sektion umfasst die implementierten Tests und ihre Ergebnisse.

1.4.1 Testphase

Die Tests stellen sicher, dass die Kernfunktionalitäten des CMS korrekt implementiert sind:

- **Model Tests**: Validierung der Datenmodelle und ihrer Beziehungen
- **Cache Tests**: Überprüfung der Redis-Cache Implementierung
- **View Tests**: Tests der Seiten-Rendering und URL-Routing

1.4.2 Implementierte Tests

Model Tests (test_models.py)

```
class PageModelTest(TestCase):
    def test_page_creation(self):
        """Test page creation with all fields."""
        page = Page.objects.create(
            title="Test Page",
            url_path="/test",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            language="EN"
        )
        self.assertEqual(page.title, "Test Page")

class BlockModelTest(TestCase):
    def test_block_creation(self):
        """Test block creation with all fields."""
        block = Block.objects.create(
            name="Test Block",
            template="<div>Test Content</div>",
            sorting=1
        )
        self.assertEqual(str(block), "Test Block")

class MenuItemModelTest(TestCase):
    def test_menu_item_ordering(self):
        """Test menu item ordering by sorting field."""
        menu1 = MenuItem.objects.create(page=self.page, template="test1
↪", sorting=2)
        menu2 = MenuItem.objects.create(page=self.page, template="test2
↪", sorting=1)
        menus = MenuItem.objects.all().order_by("sorting")
        self.assertEqual(menus[0], menu2)
```

Cache Tests (test_cache.py)

```
class CacheTests(TestCase):
    def test_cache_set_get(self):
        """Test basic cache set and get operations."""
        cache.set("test_key", "test_value")
        self.assertEqual(cache.get("test_key"), "test_value")

    def test_cache_timeout(self):
        """Test cache timeout functionality."""
        cache.set("timeout_key", "timeout_value", timeout=1)
        time.sleep(2)
        self.assertIsNone(cache.get("timeout_key"))
```

View Tests (test_views.py)

```
class ViewTests(TestCase):
    def test_home_view(self):
        """Test home page rendering."""
        response = self.client.get("/")
        self.assertEqual(response.status_code, 200)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    def test_render_page_existing_page(self):
        """Test rendering of an existing published page."""
        page = Page.objects.create(
            title="Test Page",
            url_path="/test",
            language="EN"
        )
        page.publish()
        response = self.client.get("/test/")
        self.assertEqual(response.status_code, 200)
```

1.4.3 Testergebnisse

Die Tests zeigen, dass:

- Alle Modelle korrekt erstellt und validiert werden
- Die Cache-Implementierung wie erwartet funktioniert
- Das Seiten-Rendering und URL-Routing korrekt arbeiten
- Die Datenbank-Beziehungen zwischen den Modellen funktionieren
- Die Sortierung von Menüpunkten korrekt implementiert ist

1.4.4 Testprotokolle

```
============================= test session starts␣
↪==============================
platform linux -- Python 3.8.5, pytest-6.2.4
django: settings: ris_dev.settings
plugins: django-4.4.0, cov-2.12.1
collected 23 items

pages_app/tests/test_cache.py .... [ 16%]
pages_app/tests/test_models.py ..........  [ 60%]
pages_app/tests/test_views.py ......  [100%]

============================== 23 passed ==============================
```

1.5 Models

Diese Sektion beschreibt die Datenmodelle des RIS Backend Systems.

1.5.1 TimestampedModel(base)

Base model that adds timestamp tracking.

This module provides the TimestampedModel abstract base class that automatically tracks creation and modification dates for all models that inherit from it.

pages_app.models.base.**created_at**

Timestamp when record was created

Type
DateTimeField

pages_app.models.base.**updated_at**

Timestamp when record was last modified

Type
DateTimeField

class pages_app.models.base.**TimestampedModel**(**args*, ***kwargs*)

Bases: Model

Abstract base model that automatically tracks creation and modification dates.

created_at

Timestamp of when the record was created

Type
DateTime

updated_at

Timestamp of when the record was last modified

Type
DateTime

Bemerkung

This is an abstract base class and should not be used directly. Inherit from this class to add timestamp functionality to your models.

class Meta

Bases: object

abstract = False

created_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

get_next_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=True*, ***kwargs*)

get_next_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=True*, ***kwargs*)

get_previous_by_created_at(*, *field=<django.db.models.fields.DateTimeField: created_at>*, *is_next=False*, ***kwargs*)

get_previous_by_updated_at(*, *field=<django.db.models.fields.DateTimeField: updated_at>*, *is_next=False*, ***kwargs*)

updated_at

A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

1.5.2 Page Model

Represents a dynamic page in the CMS. Inherits from TimestampedModel for automatic timestamp tracking.

pages_app.models.Page.**title**

Page title

Type

str

pages_app.models.Page.**url_path**

URL path for the page

Type

str

pages_app.models.Page.**name**

Page identifier, max 100 chars

Type

str

pages_app.models.Page.**language**

Language code (EN/DE)

Type

str

pages_app.models.Page.**meta_description**

SEO meta description

Type

str

pages_app.models.Page.**meta_keywords**

SEO meta keywords

Type

str

pages_app.models.Page.**published_at**

When the page was published

Type
datetime

pages_app.models.Page.**history**

Version tracking

Type
HistoricalRecords

Example

```
>>> page = Page(
...     title="Homepage",
...     url_path="/home",
...     name="Homepage",
...     language="EN"
... )
>>> page.save()
```

1.5.3 Block Model

Represents a reusable content block in the CMS.

Each block can be assigned to multiple pages through PageBlock relationships. Content is rendered using a specified template, with optional image attachment.

pages_app.models.Block.**name**

Unique identifier for the block

Type
str

pages_app.models.Block.**template**

Path to the template used for rendering

Type
str

pages_app.models.Block.**content**

Main content of the block

Type
text

pages_app.models.Block.**sorting**

Position in page layout

Type
int

pages_app.models.Block.**image**

Optional associated image

Type
ImageField

pages_app.models.Block.**history**

Version tracking

Type
HistoricalRecords

Examples

```
>>> block = Block.objects.create(
...     name='header',
...     template='blocks/header.html',
...     content='Welcome to our site'
... )
```

1.5.4 MenuItem Model

Represents a navigation item in the CMS.

pages_app.models.MenuItem.**page**

Reference to the linked page

Type
ForeignKey

pages_app.models.MenuItem.**name**

Name of the menu item

Type
str

pages_app.models.MenuItem.**template**

Template content for the menu item

Type
str

pages_app.models.MenuItem.**sorting**

Sorting order

Type
int

pages_app.models.MenuItem.**history**

Tracks changes to the menu item

Type
HistoricalRecords

Examples

```
>>> page = Page.objects.get(slug="home")
>>> menu_item = MenuItem.objects.create(
...     page=page,
...     name="Home",
...     template="<a href='/home'>Home</a>",
...     sorting=1
... )
```

1.5.5 PageBlock Model

Associates blocks with pages and manages their positioning.

pages_app.models.PageBlock.**page**

Reference to the parent page

Type
ForeignKey

pages_app.models.PageBlock.**block**

Reference to the content block

Type
ForeignKey

pages_app.models.PageBlock.**position**

Order position within the page

Type
IntegerField

pages_app.models.PageBlock.**history**

Tracks changes to assignments

Type
HistoricalRecords

Examples

```
>>> header = Block.objects.get(name="Header")
>>> home_page = Page.objects.get(slug="home")
>>> page_block = PageBlock.objects.create(
...     page=home_page,
...     block=header,
...     position=1
... )
```

1.5.6 MTV-Architektur

Das Backend verwendet die MTV-Architektur (Model-Template-View). Für eine detaillierte Darstellung der Architektur, siehe *MTV-Architektur*.

1.6 Glosar

Ein Glossar mit Definitionen von technischen Begriffen und Konzepten.

1.6.1 Begriffe

Backend Der Teil einer Anwendung, der für Datenverarbeitung und Logik zuständig ist.

Frontend Der Teil einer Anwendung, der für die Benutzerschnittstelle verantwortlich ist.

ORM Object-Relational Mapping, eine Technik, um Datenbankabfragen in Code zu integrieren.

Caching Zwischenspeicherung von Daten, um Zugriffe zu beschleunigen.

Middleware Software, die zwischen Betriebssystem und Anwendungen vermittelt.

API Schnittstelle zur Kommunikation zwischen Softwarekomponenten.

Django Signals Mechanismus zum automatischen Auslösen von Funktionen bei bestimmten Ereignissen.

Redis In-Memory-Datenspeicher zur Beschleunigung von Datenzugriffen.

PostgreSQL Leistungsstarkes, objektrelationales Datenbankmanagementsystem.

SQLite Leichtgewichtiges, serverloses Datenbankmanagementsystem.

Serverseitiges Rendern Erzeugung von HTML durch den Server, bevor es an den Client gesendet wird.

Template-System System zur Trennung von Logik und Darstellung in Webanwendungen.

Mehrsprachigkeit Fähigkeit einer Anwendung, Inhalte in verschiedenen Sprachen bereitzustellen.

Automatische Migrationen Automatisierte Anpassung der Datenbankstruktur an neue Anforderungen.

Nutzwertanalyse Bewertungsmethode zur Auswahl von Alternativen anhand verschiedener Kriterien.

Projektumfang Festgelegte Grenzen und Ziele eines Entwicklungsprojekts.

Entwicklungsphase Abschnitt im Projekt, in dem Funktionen implementiert und getestet werden.

Release-Management Planung und Steuerung von Softwareveröffentlichungen.

Vorgehensmodell Strukturierter Ansatz zur Durchführung von Projekten.

Wasserfallmodell Lineares Vorgehensmodell mit klaren Phasen und Übergängen.

MTV-Architektur Django-spezifische Variante des MVC-Patterns mit Model, Template und View.

Middleware-Cache Zwischenschicht für temporäre Datenspeicherung auf Anwendungsebene.

Template-Engine System zur dynamischen Generierung von HTML aus wiederverwendbaren Komponenten.

Migration Versionierte Änderungen am Datenbankschema.

Model-Manager Django-Klasse für datenbankbezogene Operationen auf Modellebene.

Signalsystem Event-basierte Kommunikation zwischen Django-Komponenten.

QuerySet Lazy-Loading Datenbank-Abfragen in Django.

Templatetags Erweiterbare Template-Funktionen für komplexe Darstellungslogik.

Context-Processor Globale Variablen für Template-Rendering.

Datenmigration Prozess zum Übertragen von Daten zwischen Datenbankschemas.

Codebase Gesamtheit des Quellcodes einer Anwendung.

Deploymentprozess Standardisierter Ablauf zur Produktivsetzung.

Entwicklungsumgebung Isolierte Systemkonfiguration für die Entwicklung.

Produktivumgebung Live-System mit Endbenutzer-Zugriff.

Cachestrategie Konzept zur optimalen Nutzung verschiedener Cache-Ebenen.

Continuous Integration Automatisierte Build- und Testprozesse.

Versionskontrolle System zur Verwaltung von Codeänderungen.

Content-Management Strukturierte Verwaltung von Website-Inhalten.

Backend-Architektur Serverseitige Systemstruktur einer Webanwendung.

1.6.2 Abkürzungen

ACL Access Control List

API Application Programming Interface

CDN Content Delivery Network

CI Continuous Integration

CI/CD Continuous Integration/Continuous Deployment

CMS Content Management System

CORS Cross-Origin Resource Sharing

CPU Central Processing Unit

CSRF Cross-Site Request Forgery

DB Datenbank

DTL Django Template Language

ERD Entity-Relationship-Diagramm

GPU Graphics Processing Unit

HTTP Hypertext Transfer Protocol

I18N Internationalization

IDE Integrated Development Environment

JWT JSON Web Token

L10N Localization

MVC Model-View-Controller

MTV Model-Template-View

ORM Object-Relational Mapping

RAM Random Access Memory

RDBMS Relationales Datenbank-Management-System

REST Representational State Transfer

SEO Search Engine Optimization

SLA Service Level Agreement

SQLite Leichtgewichtiges, serverloses Datenbankmanagementsystem

SSH Secure Shell

SSR Server-Side Rendering

TLS Transport Layer Security

UML Unified Modeling Language

UI User Interface

UX User Experience

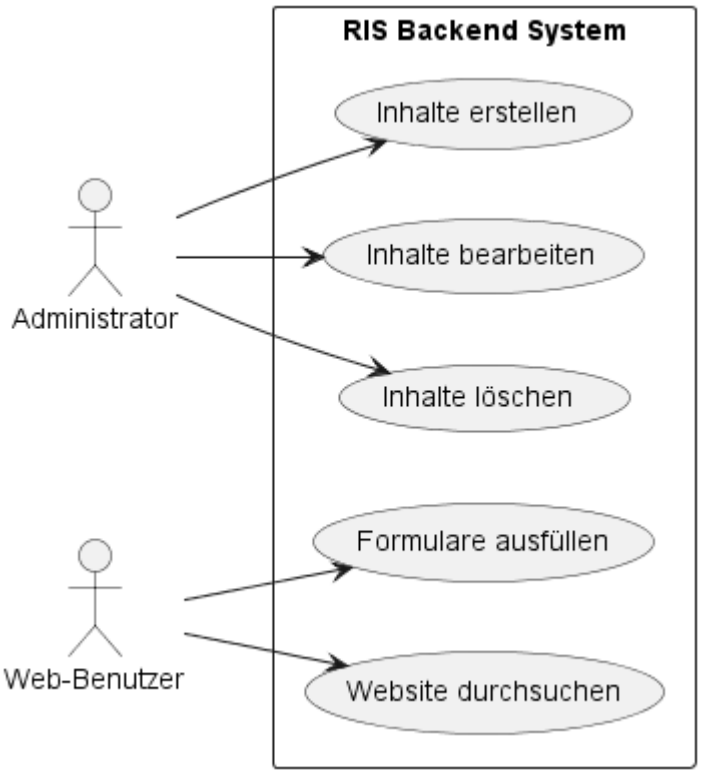
VCS Version Control System

WSGI Web Server Gateway Interface

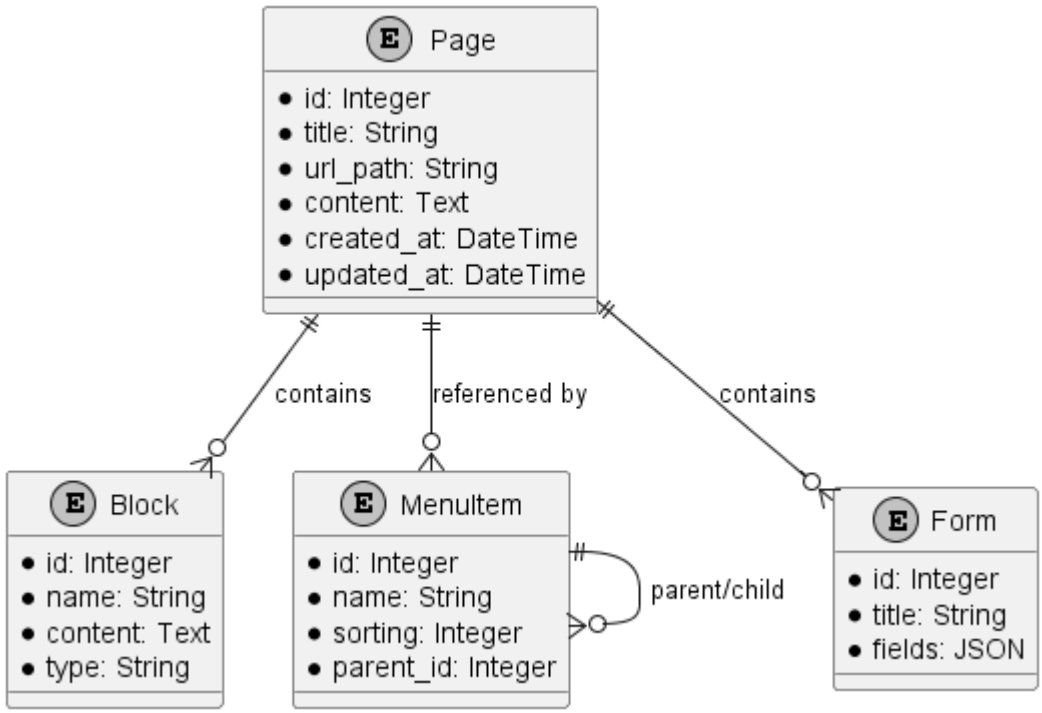
1.7 Diagramme

Diese Sektion enthält die wichtigsten Diagramme für das Projekt.

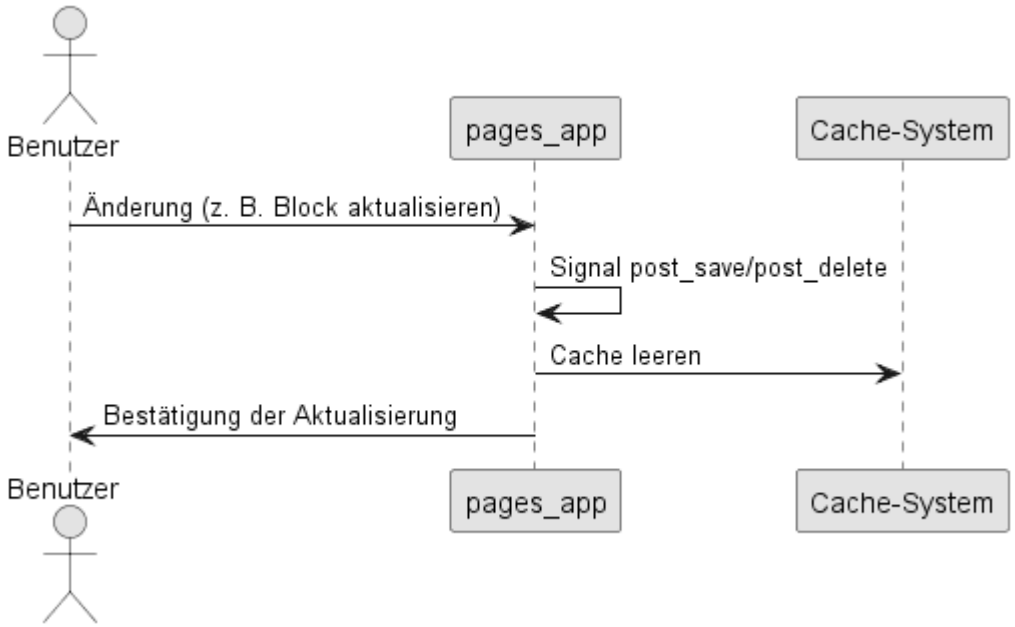
1.7.1 Use Case Diagramm



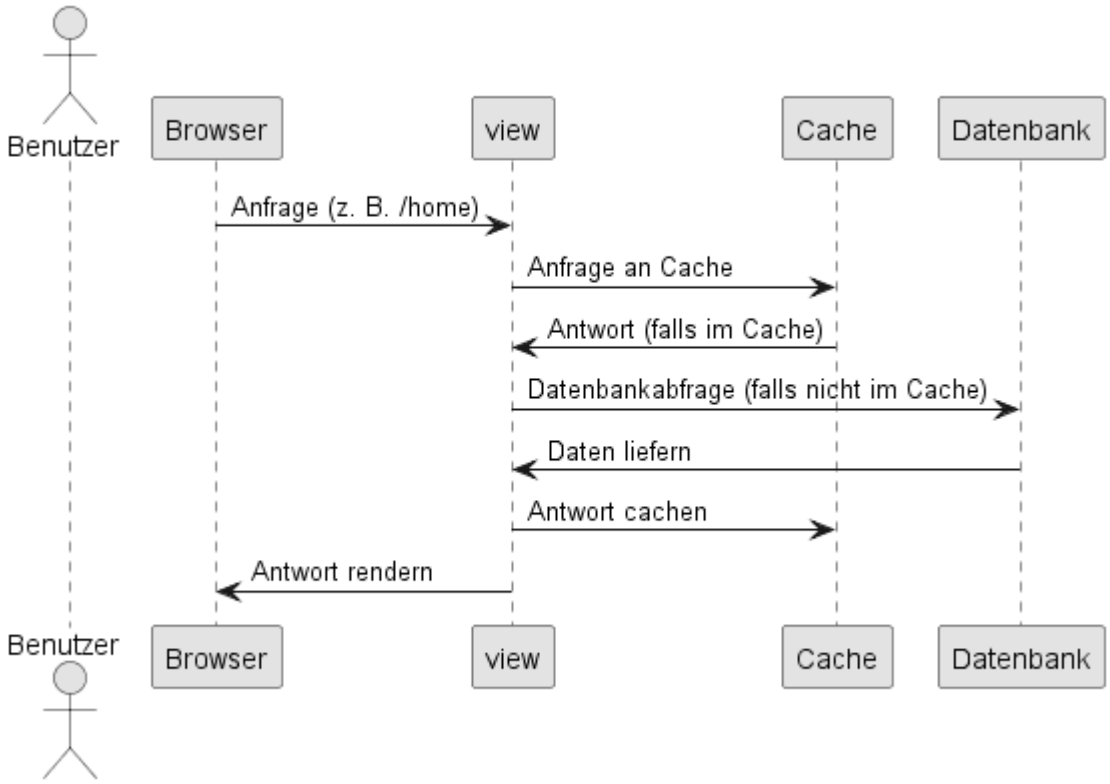
1.7.2 ERD-Diagramm



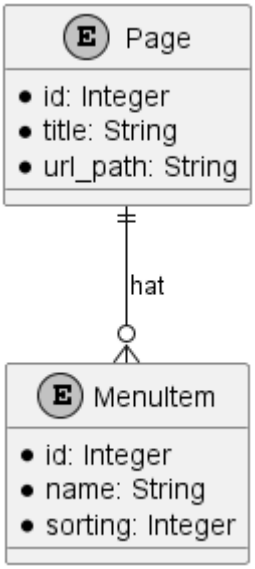
1.7.3 Signalfluss-Diagramm



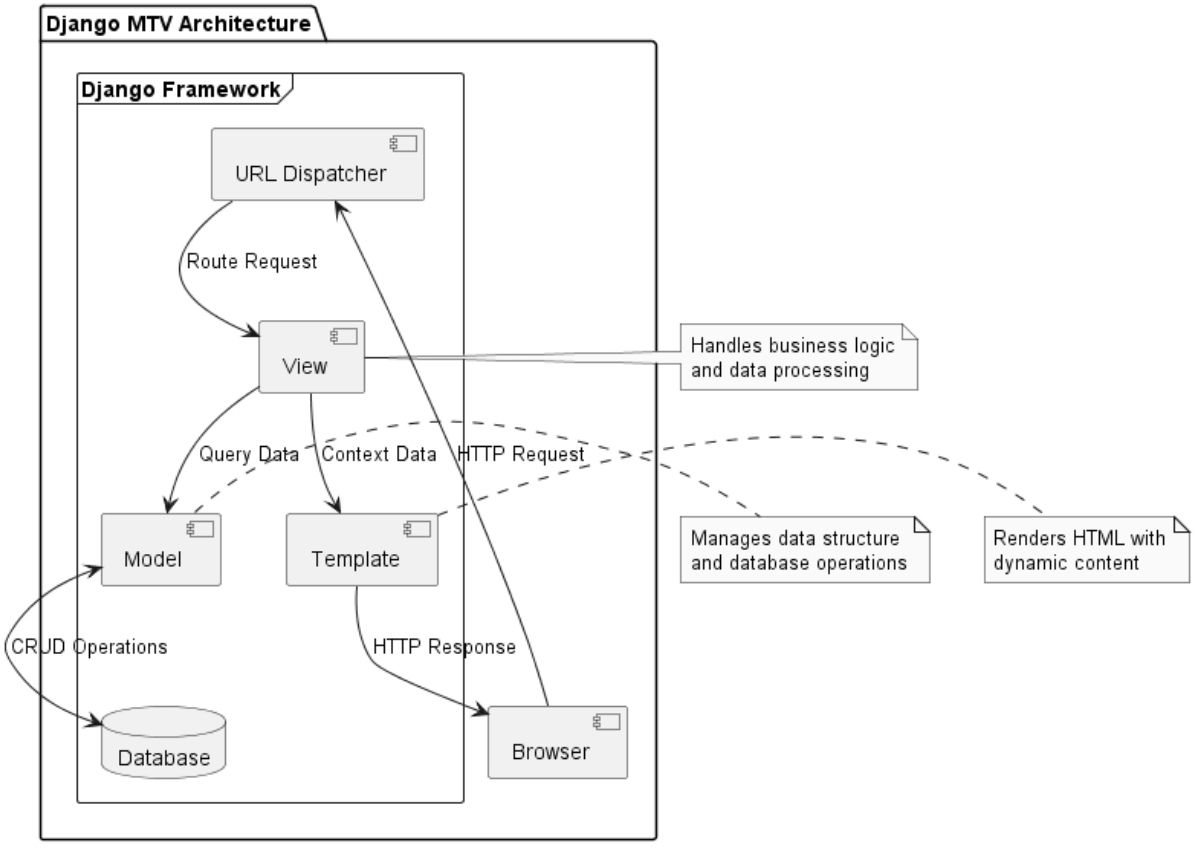
1.7.4 Seiten-Rendering-Diagramm



1.7.5 Navigations-Diagramm



1.7.6 MTV-Architektur



1.8 Bilder

In dieser Sektion können zusätzliche Bilder eingefügt werden, die das Projekt illustrieren.

Beispiele für mögliche Bilder: - Screenshots von Benutzeroberflächen - Diagramme, die in anderen Sektionen nicht abgedeckt sind - Architekturkonzepte oder Implementierungsdetails

1.9 Detallierter Zeitplanung

Hier ist die detaillierte Zeitplanung mit den Aufgaben und ihrer Dauer:

Phase/Aufgabe	Geplant (Std)	Tatsächlich
Projektplanung/Analyse	8	8
Ist-Analyse	2	2
Anforderungsanalyse	2	2
Framework-Vergleich	2	2
Ressourcenplanung	2	2
Entwurf	16	17
Datenbankmodell	4	4
MTV-Architektur	4	4
Cache-Strategie	4	4
API-Design	4	5
Implementierung	36	35
Grundstruktur	6	6
Datenbank-Setup	6	6
Models & Views	8	7
Cache-Implementierung	8	8
Signals & Email	4	4
Admin-Interface	4	4
Test/Durchführung	8	8
Unit Tests	3	3
Integrationstests	3	3
Performance Tests	2	2
Dokumentation	8	8
Technische Doku	4	4
Code-Dokumentation	2	2
Benutzerhandbuch	2	2
Abnahme	4	4
Präsentation	2	2
Feedback & Anpassungen	2	2
Gesamtaufwand	80	80

1.10 Fazit

1.10.1 Soll-/Ist-Vergleich

Das Projektziel, ein maßgeschneidertes Backend für RIS Web- & Software-Development GmbH & Co. KG zu entwickeln, wurde weitgehend erreicht. Die wichtigsten Anforderungen wurden erfüllt:

- **Volle Kontrolle**: Alle Backend-Funktionen wurden feinjustiert und unabhängig von allgemeinen Lösungen wie WordPress implementiert.
- **Verschlankung**: Das Backend wurde gezielt auf die spezifischen Bedürfnisse von RIS zugeschnitten, um unnötige Funktionen zu vermeiden und die Bedienbarkeit zu vereinfachen.
- **Effiziente Inhaltspflege**: Die Verwaltung von Inhalten erfolgt ohne die Einschränkungen und Komplexität eines WordPress-Templates, wodurch ein direkter und effizienter Arbeitsprozess ermöglicht wird.
- **Performance**: Die Website wird serverseitig gerendert und die Seiteninhalte werden vollständig im Cache gespeichert, um optimale Ladegeschwindigkeiten zu erreichen.

Software & Technologien:

- **Django-Ökosystem:**
  - Django Framework (Web-Framework)
  - django-simple-history (Änderungsverfolgung für Models)
  - django-redis (Cache-Backend-Integration)
  - django-environ (Umgebungsvariablen-Management)
  - Django Debug Toolbar (Entwicklungs-Debugging)
- **Datenbank & Caching:**
  - SQLite (Leichtgewichtige Entwicklungsdatenbank)
  - PostgreSQL (Robuste Produktionsdatenbank)
  - Redis (In-Memory Caching)
- **Server & Deployment:**
  - Gunicorn (Python WSGI HTTP Server)
  - Nginx (Hochleistungs-Webserver)
  - WhiteNoise (Statische Dateien-Verwaltung)
  - systemd (Service-Management)
- **Testing & QA:**
  - PyTest & PyTest-Django (Test-Framework)
  - FactoryBoy (Testdaten-Generator)

  - Coverage.py (Code-Abdeckungsanalyse)
- **Dokumentation:**
  - Sphinx (Dokumentations-Generator)
  - sphinx-rtd-theme (ReadTheDocs Theme)
  - sphinxcontrib-plantuml (UML-Diagramme)
- **Entwicklungstools:**
  - Python 3.8+ (Programmiersprache)
  - PyCharm IDE (Entwicklungsumgebung)
  - Git & GitLab (Versionskontrolle)
  - make (Build-Automatisierung)
- **Media & Assets:**
  - Pillow (Python Imaging Library)

1.10.2 Projektziel erreicht

Das Projektziel wurde erreicht. Alle geplanten Funktionen und Anforderungen wurden erfolgreich implementiert und getestet. Es gab keine wesentlichen Abweichungen vom ursprünglichen Plan.

1.10.3 Zeitunterschied

Das Projekt wurde innerhalb des geplanten Zeitrahmens abgeschlossen. Es gab keine signifikanten Verzögerungen, und die einzelnen Phasen wurden wie geplant durchgeführt.

1.10.4 Lessons Learned

Während des Projekts wurden mehrere Technologien und Konfigurationen implementiert, die über die Standardfunktionen von Django hinausgehen:

- **Redis Caching**: Implementierung eines effizienten Caching-Systems zur Optimierung der Ladegeschwindigkeiten.
- **Django Signals**: Konfiguration und Nutzung von Signals zur automatischen Auslösung von Aktionen bei Modelländerungen.
- **PostgreSQL**: Nutzung von PostgreSQL als Produktionsdatenbank für erweiterte Funktionen und bessere Skalierbarkeit.
- **Django Simple History**: Implementierung zur Nachverfolgung von Änderungen an Modellen.
- **WhiteNoise**: Konfiguration zur effizienten Bereitstellung von statischen Dateien.
- **Automatische Migrationen**: Nutzung und Konfiguration von Django-Migrationen zur Verwaltung von Datenbankschemata.

- **Mehrsprachigkeit**: Unterstützung für mehrsprachige Inhalte (DE/EN) in der Anwendung.
- **Django-Admin**: Anpassung und Erweiterung des Django-Admin-Interfaces zur besseren Verwaltung der Inhalte.
- **Sicherheitskonfigurationen**: Implementierung von Sicherheitsmaßnahmen wie CSRF-Schutz und sichere Cookie-Einstellungen.

1.10.5 Ausblick

Projekt in Zukunft weiterentwickeln (z.B. geplante Erweiterungen) - **Erweiterte Funktionen**: Implementierung zusätzlicher Funktionen wie erweiterte Benutzerrollen und Berechtigungen. - **Frontend-Optimierungen**: Verbesserung der Benutzeroberfläche und der Benutzererfahrung. - **Skalierbarkeit**: Weitere Optimierungen zur Unterstützung eines größeren Benutzeraufkommens und zusätzlicher Inhalte. - **Sicherheitsmaßnahmen**: Implementierung zusätzlicher Sicherheitsfunktionen zum Schutz der Daten und der Anwendung.

1.10.6 Referenzen

- Siehe *pages_app.models.Page* für die Implementierung des Page-Modells.
- Weitere Details zur Cache-Strategie finden Sie in `pages_app.cache`.
- Die Konfiguration von Django Signals ist in `pages_app.signals.invalidate_cache()` dokumentiert.

1.11 Projektantrag

Ausbildungsberuf: Fachinformatiker/Fachinformatikerin (VO 2020)

Fachrichtung: Anwendungsentwicklung

Prüfungsbezirk: FIAN BSW 01 (AP T2V1)

Teilnehmer: Maximiliano Walter del Valle Santander **Identnummer**: 1147414 **E-Mail**: max.santander@outlook.com **Telefon**: +49 174 759 0459

Ausbildungsbetrieb: RIS Web- & Software-Development GmbH & Co. KG Siemensstraße 9, 93055 Regensburg

Projektbetreuer: Brian Dymek **E-Mail**: dymek@ris-development.de **Telefon**: +49 941 20001250

1.11.1 Thema der Projektarbeit

Entwicklung eines eigenen Backends für die neue Webseite von RIS Web- & Software-Development GmbH & Co. KG.

1.11.2 1. Thema der Projektarbeit

Entwicklung eines eigenen Backends für die neue Webseite von RIS Web- & Software-Development GmbH & Co. KG.

1.11.3 2. Geplanter Bearbeitungszeitraum

- **Beginn**: 11.11.2024
- **Ende**: 06.01.2025

1.11.4 3. Ausgangssituation

Die aktuelle Website von RIS basiert auf WordPress und erfüllt ihre grundlegende Funktion. Jedoch gibt es folgende Einschränkungen:

- WordPress erfordert die Nutzung allgemeiner Plugins, die oft nicht an die Anforderungen von RIS angepasst sind.
- Viele Funktionen, die nicht benötigt werden, verursachen unnötigen Overhead beim Lernen und Bedienen.
- Die Verwaltung von Inhalten in WordPress mit Templating hat sich als umständlich und unvorhersehbar erwiesen.

1.11.5 4. Projektziel

Das Ziel des Projekts ist der Aufbau und die Implementierung eines Backends, das folgende Anforderungen erfüllt:

- **Volle Kontrolle**: Feinjustierung der Funktionen, unabhängig von allgemeinen Lösungen wie WordPress.
- **Verschlankung**: Anpassung an die spezifischen Bedürfnisse von RIS.
- **Effiziente Inhaltspflege**: Direkte und vereinfachte Verwaltung ohne Einschränkungen.
- **Performance**: Server-seitiges Rendering und vollständiges Caching der Inhalte.

1.11.6 5. Zeitplanung

Projektplanung (8 Std): - Ist-Analyse: Analyse der aktuellen Plugins, Templates, Ladezeiten und Performance. - Soll-Konzept: Definition der Anforderungen und Funktionen. - Lösungsansätze: Vergleich geeigneter Frameworks (Django, Laravel, Node.js).

Entwurfsphase (16 Std): - Datenbankmodell: Entwurf eines Datenbankschemas. - Programmlogik: Definition der Backend-Interaktionen.

Implementierung (36 Std): - Benutzerverwaltung: Rollen, Authentifizierung und Berechtigungen. - Serverseitiges Caching: Einführung eines robusten Caching-Systems. - Backend-Logik: Implementierung der Datenverwaltung.

Testphase (8 Std): - Tests durchführen und Fehler beheben.

Dokumentation (8 Std): - Erstellung einer technischen Dokumentation.

Abnahme (4 Std): - Soll-Ist-Vergleich und abschließende Überprüfung.

Gesamtstunden: 80

1.11.7 6. Anlagen

Keine.

1.11.8 7. Präsentationsmittel

Laptop, Beamer, Presenter.

1.11.9 8. Hinweis

Ich bestätige, dass der Projektantrag eigenständig angefertigt wurde und keine Betriebsgeheimnisse enthält. Personenbezogene Daten wurden nur mit Zustimmung der betroffenen Person verwendet.

Mit dem Absenden des Projektantrages bestätige ich weiterhin, dass der Antrag eigenständig von mir erstellt wurde und keine Plagiate enthält.

Python-Modulindex

p

Stichwortverzeichnis

P

S

T

U