Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

InIT Institut für angewandte
Informationstechnologie

Bachelorarbeit Informatik

Evaluation einer neuen Softwarearchitektur und Erstellung eines Prototyps

Autoren	Markus Ferber Lukas Jans
Hauptbetreuung	Prof. Jürgen Spielberger
Industriepartner	BORM-INFORMATIK AG
Datum	09.06.2023

Zürcher Hochschule
für Angewandte Wissenschaften

zh
aw School of
Engineering

Erklärung betreffend das selbstständige Verfassen einer Bachelorarbeit an der School of Engineering

Erklärung betreffend das selbstständige Verfassen einer Bachelorarbeit an der School of Engineering

Mit der Abgabe dieser Bachelorarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)
Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Bachelorarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmassnahmen der Hochschulordnung in Kraft.

Ort, Datum:	**Name Studierende:**
Schaffhausen, 09.06.2023	Markus Ferber
Märstetten, 09.06.2023	Lukas Jans

Zusammenfassung

Die BORM-INFORMATIK AG entwickelt und vertreibt eine Betriebsführungslösung mit Hauptfokus auf die Abläufe von produzierenden und projektorientierten Betrieben im holzverarbeitenden Sektor. Im Zuge der Umlagerung der Business-Logik auf den BormServer ergab sich die Notwendigkeit einer Lösung zur Verwaltung von zeitlich gesteuerten oder von Usern ausgelösten Hintergrundprozessen. Bestehende Lösungen wurden bei Bedarf individuell implementiert. Ziel dieser Arbeit war die Erarbeitung eines Konzepts für eine einheitliche, zweckbestimmte Verwaltung solcher Prozesse (Tasks) sowie die Implementation eines entsprechenden Prototyps. Dazu wurden zunächst verschiedene Taskverwaltungs-Frameworks evaluiert und für die Anwendung auf dem BormServer getestet. Diese Recherchen führten zur Auswahl des .NET-Frameworks Hangfire. Im Hinblick auf die Integration in den BormServer wurden die benötigten Systemoperationen definiert sowie eine Architektur- und eine Verteilungsübersicht entworfen. Basierend auf diesem Konzept wurde anschliessend der Prototyp implementiert. Für die Verifikation der Funktionalität wurden Unit-Tests geschrieben und Systemtests durchgeführt. Die Tasks selber werden als Python-Skripts implementiert. Task-Metadaten werden in der BORM-Datenbank gespeichert. Anhand eines einfachen Beispiels wurde die Möglichkeit der Integration von BORM-internen Python-Bibliotheken demonstriert. Mit der bestehenden Implementation ist es nun möglich, Tasks zu starten, für die zeitliche Steuerung aktiv oder inaktiv zu setzen, sowie den Status eines Tasks abzufragen. Tasks können zudem erstellt, geändert und gelöscht werden. Die Interaktion mit der Taskverwaltung erfolgt über die API-Endpoints des BormServers. Weitere Funktionalitäten, wie die Verarbeitung von Task-Resultaten sowie das Abbrechen von laufenden Tasks, wurden im Rahmen des Prototyps konzeptionell aufgezeigt. Das weitere Vorgehen beinhaltet die Umsetzung dieser Funktionalitäten sowie die Integration des Hangfire-Datenspeichers in die BORM-Datenbank.

Abstract

BORM-INFORMATIK AG develops and distributes a business management solution with the focus on the processes of manufacturing and project-oriented companies in the woodworking sector. In the course of moving the business logic to the BormServer, the need arose for a solution to manage time-controlled or user-triggered background processes. Existing solutions were implemented individually as needed. The aim of this thesis was to develop a concept for a uniform, purpose-specific control of such processes (tasks) and to implement a corresponding prototype. To this end, various task management frameworks were evaluated and tested for use on the BormServer. This research led to the selection of the .NET framework Hangfire. For the integration into the BormServer, the required system operations were defined and an architecture and a distribution overview were designed. Based on this concept, the prototype was then implemented. For the verification of the functionality, unit tests were written and system tests were performed. The tasks themselves are implemented as Python scripts. Task metadata is stored in the BORM database. A simple example was used to demonstrate the possibility of integrating BORM-internal Python libraries. With the existing implementation, it is now possible to start tasks, set them active or inactive for time control, and query the status of a task. Tasks can also be created, changed and deleted. Interaction with the task management takes place via the API endpoints of the BormServer. Further functionalities, such as the processing of task results and the cancellation of running tasks, were conceptually demonstrated within the scope of the prototype. The next steps include the implementation of these functionalities and the integration of the Hangfire data storage into the BORM database.

Vorwort

Diese Bachelorarbeit wurde von Markus Ferber und Lukas Jans im sechsten Semester ihres Bachelorstudiums in Informatik an der Zürcher Hochschule für Angewandte Wissenschaften ZHAW in Winterthur verfasst. Die Autoren bedanken sich bei ihrem Betreuer, Herrn Prof. Jürgen Spielberger, für die hilfreiche Beratung und Unterstützung. Ebenfalls herzlich danken sie den Betreuern von der Firma BORM-INFORMATIK AG, für ihre tatkräftige Mithilfe und die gelungene Zusammenarbeit bei der Durchführung der Bachelorarbeit. Zudem danken sie für das Korrekturlesen der Arbeit und das sprachliche Feedback, sowie für die Unterstützung beim Layout.

Für die Autoren stellte diese Arbeit eine gute Gelegenheit dar, ihre Kenntnisse in der Softwareentwicklung innerhalb eines bestehenden, grösseren Systems zu vertiefen. Besonders in den Bereichen Taskverwaltung sowie API-Entwicklung mit ASP.NET Core konnten viele neue Erkenntnisse gesammelt werden. Die Zusammenarbeit mit dem Industriepartner BORM-INFORMATIK AG verlief durchweg positiv und stellte eine wertvolle Zusatzerfahrung dar.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Ausgangslage . . . 1
1.1.1. Bestehende Lösungen . . . 2
1.1.2. Nachteile der bestehenden Lösungen . . . 2
1.2. Aufgabenstellung . . . 3

2. Grundlagen **4**
2.1. Git . . . 4
2.1.1. GitHub/GitLab . . . 4
2.1.2. Issues . . . 4
2.1.3. Versionsverwaltung . . . 4
2.2. Task und Task-Queue . . . 4
2.3. Message Broker . . . 4
2.4. Flask . . . 5
2.5. ASP.NET . . . 5
2.6. ADO.NET . . . 5
2.7. Python.NET . . . 5
2.8. Swagger . . . 6

3. Vorgehen **8**
3.1. Umsetzung der Aufgabenstellung . . . 8
3.1.1. Analyse der Ausgangslage . . . 8
3.1.2. Anforderungen . . . 8
3.1.3. Speichertechnologien . . . 8
3.1.4. Recherche . . . 9
3.1.5. Vorselektion und erste Tests . . . 10
3.1.5.1. Redis . . . 10
3.1.5.2. RabbitMQ . . . 11
3.1.5.3. Weitere Message-Broker . . . 11
3.1.5.4. Celery . . . 11
3.1.5.5. Python-rq . . . 11
3.1.5.6. Dramatiq . . . 11
3.1.5.7. TaskTiger . . . 12
3.1.5.8. Huey . . . 12
3.1.5.9. MRQ . . . 12
3.1.5.10. Gofer.NET . . . 12
3.1.5.11. Coravel . . . 12
3.1.5.12. Hangfire . . . 12
3.1.5.13. Weitere Taskverwaltungs-Frameworks . . . 13
3.1.6. Auswahl geeigneter Frameworks . . . 13
3.1.7. Weitere Tests der ausgewählten Frameworks . . . 13
3.1.7.1. Dramatiq auf Flask . . . 13
3.1.7.2. Celery auf Flask . . . 13
3.1.7.3. Coravel auf ASP.NET . . . 14
3.1.7.4. Hangfire auf ASP.NET . . . 14
3.1.8. Festlegung auf ein Framework und Erstellung des Konzepts . . . 14
3.1.9. Implementation des Prototyps . . . 15

3.1.10. Unit- und Systemtests . . . 16
3.1.11. Integration von BORM-internen Python-Bibliotheken . . . 16
3.1.12. Konzeptionelle Überlegungen zum Prototyp . . . 16
3.2. Arbeitsmethodik . . . 17
3.3. Git . . . 17
3.3.1. GitLab . . . 17
3.3.2. GitHub . . . 17
3.4. Pair Programming . . . 17
3.5. Zeitplanung . . . 18
3.5.1. Tatsächlicher Ablauf . . . 18

4. Resultate **19**
4.1. Konzept . . . 19
4.1.1. Systemoperationen . . . 19
4.1.2. Architekturübersicht . . . 20
4.1.3. Verteilung der Softwaremodule . . . 21
4.2. Prototyp . . . 22
4.2.1. Funktionsumfang . . . 22
4.2.1.1. Datenbank . . . 22
4.2.1.2. Worker . . . 23
4.2.1.3. Aufrufen von Python-Code via Python.NET . . . 24
4.2.1.4. Endpoints . . . 25
4.2.1.5. Integration von BORM-internen Python-Bibliotheken . . . 28
4.2.1.6. Trigger after insert . . . 29
4.2.2. Klassendiagramm . . . 30
4.2.3. Tests . . . 30
4.2.3.1. Unit-Tests . . . 30
4.2.3.2. Systemtests . . . 31
4.2.4. Weitere konzeptionelle Überlegungen . . . 36
4.2.4.1. Integration der Hangfire-Datenbank in die BORM-Datenbank . . . 36
4.2.4.2. Systemoperation Task abbrechen . . . 36
4.2.4.3. Behandlung von Task-Resultaten . . . 37
4.2.4.4. Fehlerbehandlung in Tasks . . . 39

5. Diskussion und Ausblick **40**
5.1. Diskussion . . . 40
5.1.1. Analyse und Evaluation eines geeigneten Frameworks . . . 40
5.1.2. Erstellen des Konzepts . . . 40
5.1.3. Integration von Hangfire in die BormServer-Applikation . . . 40
5.1.4. Verwendung von Python.NET zur Ausführung von Python-Skripts . . . 41
5.1.5. Implementation und Durchführung der Tests . . . 41
5.1.6. Beschränkung der implementierten Funktionalität . . . 41
5.1.7. Anwendung der reduzierten Version von SCRUM . . . 42
5.1.8. Zeitplanung . . . 42
5.2. Reflexion der Ziele und Resultate . . . 42
5.3. Ausblick . . . 43
5.3.1. Abbrechen von Tasks . . . 43
5.3.2. Create- und Update-Funktionalität für Task-Konfigurationen separieren . . . 43
5.3.3. Integration der Hangfire-Datenbank in die BORM-Datenbank . . . 43
5.3.4. Resultatbehandlung über Message Broker . . . 43
5.3.5. Ausstehende Problembehandlungen . . . 43
5.3.6. Integration in Programmverwaltung . . . 43
5.3.7. Fehlerbehandlung . . . 44

6. Verzeichnisse **45**
6.1. Literaturverzeichnis 45
6.2. Abbildungsverzeichnis 48
6.3. Tabellenverzeichnis 49
6.4. Verzeichnis der Code-Listings 50

A. Anhang **51**
A.1. Aufgabenstellung gemäss Complesis 51

1. Einleitung

Dieses Kapitel befasst sich mit der Ausgangslage und der Definition der Aufgabenstellung.

1.1. Ausgangslage

„Die BORM-INFORMATIK AG entwickelt und vertreibt eine flexible und durchgängige Betriebsführungslösung. Im Fokus stehen dabei die Abläufe von produzierenden und projektorientierten Betrieben im holzverarbeitenden Sektor. Das ERP-System "BormBusiness" als Herzstück wird ergänzt durch die CAD-Lösung "PointLineCAD", die das Erstellen, Visualisieren und Bearbeiten von 2D- und 3D-Daten in Echtzeit erlaubt. Zudem wird das Portfolio durch die Anbindung dieser Systeme ans Web ("BormLive") und Apps für mobile Geräte erweitert. Ein wichtiges Merkmal des Systems ist die hohe Flexibilität und Anpassbarkeit, wodurch individuelle Kundenbedürfnisse abgebildet werden können. [. . .] Die ERP-Desktopapplikation ist hauptsächlich in C++ geschrieben. Für das Customizing kommt die firmeneigene Sprache "BormScript" zum Einsatz. Die CAD-Applikation basiert ebenfalls auf C++ und hat ein Python-API. Das Frontend der Webanwendung ist in JavaScript implementiert. [. . .]“ A.1

Ursprünglich waren das ERP-System und die CAD-Applikation als klassische Zweischicht-Architektur implementiert worden, wobei die Desktopapplikationen direkt mit der Datenbank kommunizierten. Die Erweiterung des Portfolios um eine Web- und eine Applösung erfordert jedoch den Umstieg auf eine Dreischicht-Architektur wie in Bild 1.1 dargestellt, d. h. eine Umlagerung der Businesslogik auf einen Server (Business Logic Tier, auch BormServer genannt), der die Kommunikation mit der Datenbank (Persistence Tier) übernimmt. Die verschiedenen Applikationen (Presentation Tier) kommunizieren nur noch mit dem Server. „[. . .] Neue Business-Logik wird im "BormServer" implementiert und allen Oberflächen per API zur Verfügung gestellt. [. . .] Der BormServer nutzt Java, C# und Python, wobei letzteres ebenfalls zum Customizing verwendet werden kann. [. . .]“ A.1

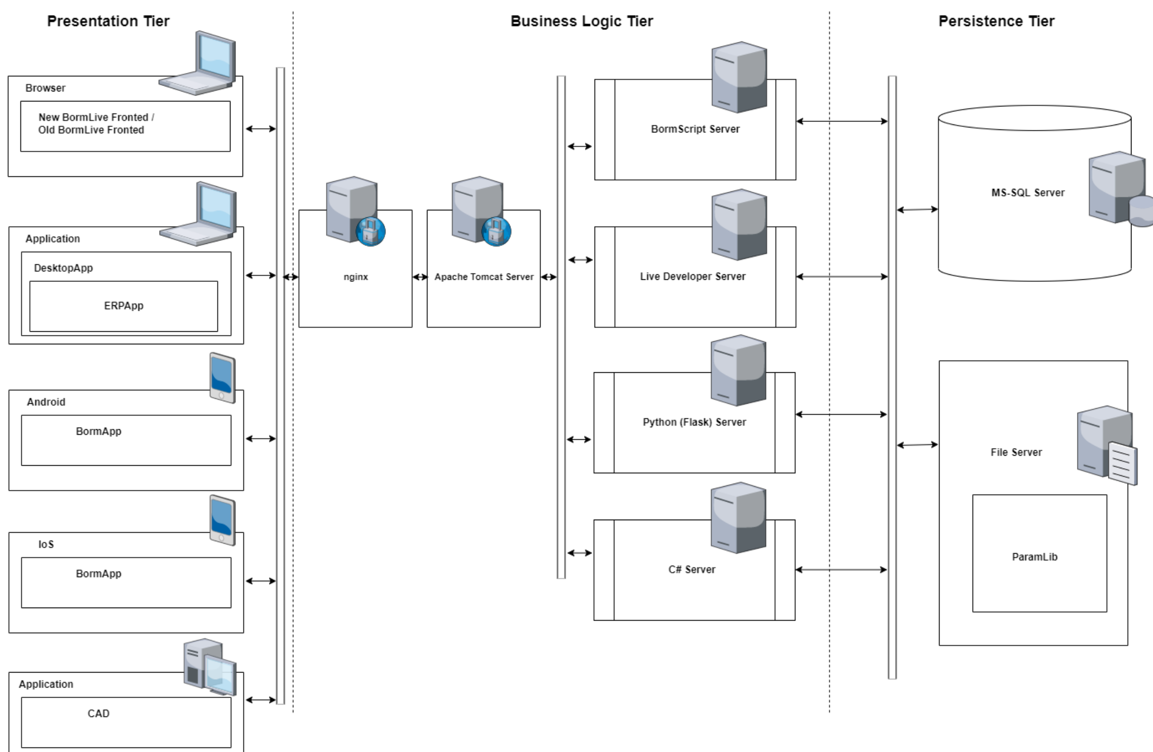

Abbildung 1.1.: Softwarearchitektur BORM [1]

In der aktuellen Implementation existiert keine Möglichkeit, um langlaufende API-Aufrufe aus einem Client-Prozess auf dem BormServer ordentlich zu verwalten. Ein Aufruf, der nicht innerhalb von 60 Sekunden beantwortet wird, löst ein HTTP-Timeout aus. „[...] Des Weiteren gibt es viele Aktionen, welche nicht durch direkte Benutzereingabe, sondern zeitlich gesteuert oder regelmässig auf dem BormServer ausgeführt werden sollen. Solche ”Tasks” sollen von Clientseite erstellt und verwaltet werden können. Ebenfalls müssen Ergebnis- oder Fehlermeldungen als Nachricht an bestimmte Benutzer gesendet werden können. [...]“ A.1

1.1.1. Bestehende Lösungen

Um die Ausführung solcher Tasks in der aktuellen Architektur zu ermöglichen, wurden bei Bedarf individuelle, behelfsmässige Lösungen implementiert, ohne dass ein einheitliches, zweckbestimmtes Konzept erstellt worden wäre. Diese beinhalten im Wesentlichen folgende Möglichkeiten:

1) Eine Python-Implementation im Flask-Server (siehe Abbildung 1.1) um eine externe API aufzurufen, wobei der Server direkt nach dem Aufruf eine Rückmeldung gibt, dass der Aufruf gestartet wurde. Danach fragt der Client den Server alle fünf Sekunden, wie der Status des Tasks ist. Der Server antwortet jeweils, dass der Task noch nicht abgeschlossen ist, bis die Antwort der externen API eingetroffen ist. Diese wird anschliessend an den Client zurückgegeben.

2) Eine ähnliche Implementation wie unter 1) beschrieben. Im Gegensatz zu dieser speichert jedoch der Server den aktuellen Zustand des Tasks in der Datenbank und meldet diesen dem Client mittels einer Benachrichtigung sobald der Task abgeschlossen ist.

3) Die oben erwähnte, firmeneigene Sprache BormScript kann unabhängig von einer Desktop-Applikation mittels eines eigens dafür erstellten Programms ausgeführt werden. Unter Verwendung der Windows-Aufgabenplanung könnten so auch zeitlich gesteuerte Tasks im Hintergrund automatisch ausgeführt werden.

4) Das Software-Portfolio enthält einen Event-Handler, dem auszuführende Tasks über Konfigurationsdateien übergeben werden.

1.1.2. Nachteile der bestehenden Lösungen

Die unter 1.1.1 aufgeführten Möglichkeiten sind zwar alle funktionsfähig und werden auch bereits eingesetzt. Bei den ersten beiden Möglichkeiten besteht jedoch das Problem, dass jeder einzelne Fall separat implementiert werden muss. Ausserdem wird durch die fortlaufende Kommunikation zwischen Server und Client unnötig viel Netzwerkverkehr erzeugt. Die dritte Option (BormScript mit Windows-Aufgabenplanung) ist für die Ausführung von Tasks auf einem Server ungeeignet, da der ausführende Benutzer ein Windows-erstellter Benutzer ist, der nicht über die notwendigen Zugriffs- und Ausführungsrechte verfügt. Des Weiteren ist diese Option sowie der Event-Handler (Option 4) UI-gesteuert und deshalb für die Benutzung auf dem Server ebenfalls nicht geeignet, da bei einer Fehlermeldung ein Fenster geöffnet wird, welches nur mit direktem Zugriff auf dem Server entfernt werden kann.

1.2. Aufgabenstellung

„[. . .] Ziel dieser Arbeit ist es, aufzuzeigen wie Hintergrundprozesse und Tasks mit dem BormServer abgebildet werden können. Folgende Punkte sind gefordert:

A) Eine Ist-Situations- und Bedarfsanalyse zu Hintergrundprozessen und Tasks im Umfeld von BormBusiness.

B) Entwicklung von Konzeptvorschlägen (SW-/HW-Architektur) für die Einbettung von Hintergrundprozessen im BormServer. Dabei ist insbesondere die Interaktion mit dem Benutzer (Direktantwort und Rückmeldung des Ergebnisses) zu berücksichtigen.

C) Entwicklung von Konzeptvorschlägen (SW-/HW-Architektur) für die Verwaltung von zeitlich gesteuerten oder regelmässigen Tasks auf dem BormServer. Hier ist ein besonderes Augenmerk auf die Administrationsoberfläche und die Notifizierung bei Erfolg oder Fehler zu richten.

D) Evaluation einer geeigneten Gesamtarchitektur. Dazu werden die Varianten aus den Punkten B) und C) gemeinsam mit der BORM-INFORMATIK AG bewertet und die jeweils beste bestimmt.

E) Ein Umsetzungsprototyp, welcher die Hauptaspekte aus den Lösungsvorschlägen aufzeigt oder beinhaltet.

“ A.1

2. Grundlagen

Dieses Kapitel beschreibt die grundlegenden Technologien welche bei BORM-INFORMATIK AG und in dieser Arbeit verwendet werden.

2.1. Git

Git [2] ist eine Open-Source-Versionsverwaltung, welche es erlaubt, Branches zu erstellen. Diese Branches sind Kopien eines Versionszustandes, welche unabhängig vom Original bearbeitet und mit dem Original wieder vereint werden können.

2.1.1. GitHub/GitLab

Der Onlineservice GitHub [3] stellt eine auf Git basierende Versionsverwaltung zur Verfügung. GitLab [4] hat im Gegensatz zu GitHub auch noch eine Self-Hosted-Variante, welche es erlaubt, den Code lokaler zu behalten.

2.1.2. Issues

Issues erlauben es, die Aufgaben in kleinere Teilprobleme aufzuteilen, um diese einzeln anzugehen. Meist wird für jedes Feature ein Issue erstellt und dieses einer Person oder Gruppe zur Bearbeitung zugeteilt. Dies erlaubt es dem Team, einen Überblick über den Fortschritt der einzelnen Tasks zu behalten und wem was zugeteilt wurde.

2.1.3. Versionsverwaltung

Die Versionsverwaltung erlaubt es zu sehen, wer wann was geändert hat. Des Weiteren ermöglicht sie auch, auf einen früheren Stand zurückzukehren, sollte dies von Nöten sein. Um ein Feature zu implementieren, erstellt man einen Feature-Branch, welcher nach Fertigstellung des Features und Abnahme ebenjenes durch eine andere Person wieder in den Main-Branch integriert wird. Dies erfolgt mittels eines Pull- respektive Merge-Requests. Während der Abnahme können andere Personen Änderungen beantragen. Dies erlaubt es den einzelnen Teammitgliedern, unabhängig voneinander zu arbeiten, ohne dass die Gefahr besteht, die Arbeit der anderen Teammitglieder zu beeinträchtigen.

2.2. Task und Task-Queue

Ein Task ist eine Aufgabe, die im Computer zur Laufzeit ausgeführt wird. Dies kann entweder in einem Prozess oder in einem Thread passieren. Solch ein Task ist meist ein Programm, welches entweder direkt oder über eine Schnittstelle gestartet wird. Tasks können auch in einer Warteschlange verwaltet werden, einer sogenannten Queue.

2.3. Message Broker

Ein Message Broker ist eine Schnittstelle, welche die Nachrichten von einem Sender an die registrierten Clients weiterleitet. Er ist auch für die Validierung sowie für das Routing zuständig. Er ermöglicht eine losere Kopplung der Programme, da diese nun nur über den Broker kommunizieren und nicht mehr wissen müssen, was alles sonst noch vorhanden ist. Ein Message Broker kann auch für das Managen von Queues verwendet werden.

2.4. Flask

Flask ist ein beliebtes Lightweight-Framework für die Erstellung von Webapplikationen. Der Fokus liegt auf einem einfachen und schnellen Einstieg, woraus auch komplexe Applikationen entwickelt werden können. Listing 2.1 zeigt eine einfache Flask-Webapplikation. Fixe Projektlayouts oder Abhängigkeiten werden nicht erzwungen. Der Entwickler kann die zu verwendenden Tools und Bibliotheken selber auswählen. Es sind viele Erweiterungen verfügbar, um externe Frameworks zu integrieren und weitere Funktionalitäten zu gewährleisten. [5]

```python
from flask import Flask, request
from markupsafe import escape

app = Flask(__name__)

@app.route('/')
def hello():
    name = request.args.get("name", "World")
    return f'Hello, {escape(name)}!'
```

Listing 2.1: Einfache Flask-Applikation

2.5. ASP.NET

ASP.NET ist ein von Microsoft entwickeltes Open-Source-Web-Framework für die Erstellung moderner Webapplikationen und -services mit .NET. .NET selber ist eine Entwicklerplattform mit verschiedenen Tools, Programmiersprachen und Bibliotheken für die Entwicklung unterschiedlicher Arten von Applikationen. ASP.NET erweitert die .NET-Plattform mit Tools und Bibliotheken spezifisch für die Webentwicklung. [6] Web-APIs können in ASP.NET mittels Controller-Klassen, die Methoden für die API-Endpoints zur Verfügung stellen, implementiert werden [7]. Mittels Services wird die Verbindung zum Backend, z.B. für Datenbankzugriffe, gewährleistet [8]. Sogenannte Hosted Services ermöglichen die Ausführung von Prozessen im Hintergrund, z.B. bei Applikationsstart [9].

2.6. ADO.NET

ADO.NET ist eine Sammlung von Klassen für die Verwendung von Datenzugriff-Services auf relationale, XML- und Applikationsdaten innerhalb des .NET-Frameworks. [10]

2.7. Python.NET

Python.NET [11] erlaubt es, in C# Python-Code einzubinden oder aufzurufen. Dafür wird Python.NET als Nuget Packet dem .NET Projekt hinzugefügt. In Listing 2.2 ist ein einfaches Python-Skript, eingebunden in C#, dargestellt.

```
static void Main(string[] args)
{
    PythonEngine.Initialize();
    using (Py.GIL())
    {
        dynamic np = Py.Import("numpy");
        Console.WriteLine(np.cos(np.pi * 2));

        dynamic sin = np.sin;
        double c = (double)(np.cos(5) + sin(5));
        Console.WriteLine(c);

        dynamic a = np.array(new List<float> {1,2,3});
        Console.WriteLine(a.dtype);

        dynamic b = np.array(new List<float> {6,5,4}, dtype:np.int32);
        Console.WriteLine(b.dtype);

        Console.WriteLine(a * b);
        Console.ReadKey();
    }
}
```

Listing 2.2: Beispiel für Pythonintegration in .NET

Es ist auch möglich, C#-Code in Python-Skripts einzubinden. Dazu muss Python.NET importiert werden. Ein Beispiel hierfür ist in Listing 2.3 zu sehen.

```
import clr
from System import String

test_string = "Hello world!"
if String.IsNullOrWhiteSpace(test_string):
    print("Invalid.")
else:
    print("Valid.")
```

Listing 2.3: C#-Code in Python

2.8. Swagger

Swagger [12] ist eine Sammlung von Softwaretools für den gesamten Zyklus der API-Entwicklung, vom Design über Dokumentation und Tests bis zum Deployment [13]. Eines dieser Tools ist Swagger UI, welches auf Basis der Spezifikation einer API eine grafische Benutzeroberfläche zum Testen der API-Endpoints zur Verfügung stellt [14]. Abbildung 2.1 zeigt ein Beispiel einer mit Swagger UI generierten Benutzeroberfläche für API-Endpoints. Über das Nuget-Paket Swashbuckle.AspNetCore.Swagger kann Swagger einfach in bestehende ASP.NET-Core-APIs integriert werden [15].

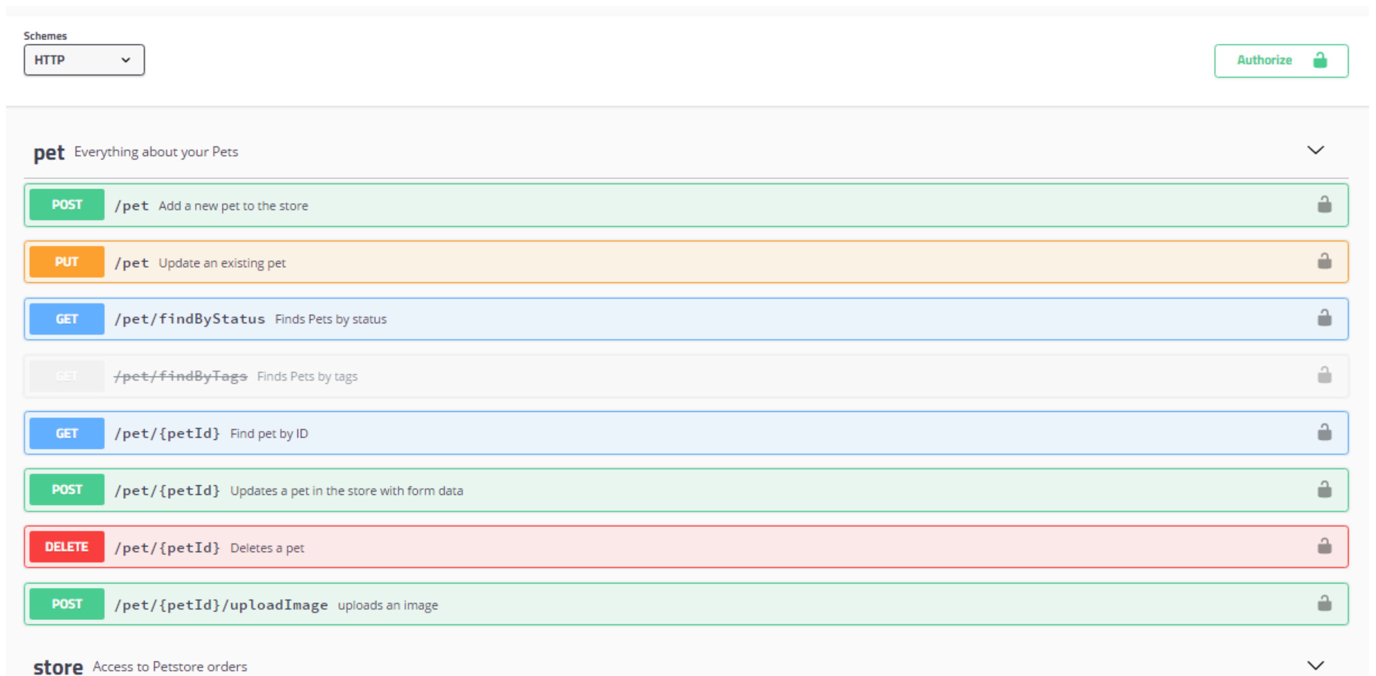

Abbildung 2.1.: Mit Swagger UI generierte API-Testoberfläche

3. Vorgehen

Dieses Kapitel beschreibt die Vorgehensweise und Methoden, welche in dieser Arbeit verwendet werden. Diese beinhalten die generelle Vorgehensweise, die Benutzung von Git Versionsverwaltung und der Zeitplan, welcher zu Beginn der Arbeit festgelegt wurde.

3.1. Umsetzung der Aufgabenstellung

In diesem Abschnitt werden die einzelnen Schritte zur Umsetzung der in Kapitel 1.2 definierten Aufgabenstellung detailliert erklärt.

3.1.1. Analyse der Ausgangslage

Der erste Schritt zur Umsetzung der Aufgabenstellung war die Analyse der Ausgangslage. Dazu wurden die in Kapitel 1.1.1 aufgeführten bestehenden Lösungen sowie ihre bisherige Verwendung in der Software zusammen mit Ingenieuren von BORM-INFORMATIK AG detailliert betrachtet und beispielhaft getestet. Dabei konnten die in Kapitel 1.1.2 genannten Nachteile dargestellt und erfasst werden. Es wurden auch die Einschränkungen, die sich durch die bestehende Architektur ergaben, besprochen.

3.1.2. Anforderungen

Um als Lösung in Frage zu kommen, musste ein Framework mit dem bestehenden BormServer kompatibel sein. Zusätzlich dazu sollten die Tasks in einer Skriptsprache verfassbar sein, da Anpassungen beim Kunden möglich sein müssen. Da Python schon verwendet wurde, war dieses auch beizubehalten. Auch Frameworks mit einem eigenen Dashboard oder einem fest integrierten Speichersystem, falls diese sich nicht einfach durch BORM-Äquivalente ersetzen lassen würden, waren nicht als Lösung zu betrachten. Da BORM-INFORMATIK AG noch kein einheitliches Messaging-System verwendete, waren Frameworks, die einen Message-Broker, der auch für andere Prozesse innerhalb des BormServers einsetzbar ist, positiv zu bewerten. Das Scheduling von Tasks sowie das automatische Starten eines Tasks, sobald Ressourcen verfügbar sind, sollte möglich sein. Während den Recherchen wurde eine weitere Anforderung bekannt: Die Worker sollten wenn möglich als Prozess erzeugt werden und nicht als Threads. Dies aus dem Grund, dass Threads auf dem Flask-Server in Verbindung mit dem SQL-Server zu Problemen führen, welche mit Prozessen nicht auftreten. Somit war ein weiteres Kriterium, ob die Worker als Thread oder als Prozess gestartet werden.

3.1.3. Speichertechnologien

Im Folgenden werden die Speichertechnologien aufgezeigt, die von den in Abschnitt 3.1.4 aufgeführten Frameworks verwendet werden.

Filesystem Bei dieser Methode werden die Daten einfach auf einem Datenspeicher wie zum Beispiel einer Festplatte abgelegt.

In Memory (Memcached, Redis) In Memory heisst, alle Daten werden im RAM gespeichert. Dies ermöglicht einen viel schnelleren Zugriff als auf eine SSD.

SQL-Datenbanken Dies sind relationale Datenbanken, bei denen alle Daten in Tabellen geschrieben sind. Für die Beziehungen zwischen Tabellen haben diese Primary und Foreign Keys. Eine Abfrage liefert als Resultat wiederum eine Tabelle.

NoSQL (MongoDB) NoSQL steht für Not only SQL. Diese Version verfolgt einen nicht-relationalen Ansatz und benötigt deshalb kein festgelegtes Tabellenschema.

3.1.4. Recherche

Nach der Analyse ging es darum, geeignete Möglichkeiten zur Taskverwaltung (Queueing, Speicherung von Tasks und Resultaten) sowie zur erforderlichen Kommunikation (Notifizierung über Erfolg/Fehler, Datenrückgabe) ausfindig zu machen, da diese beiden Features das Kernstück des zu entwickelnden Konzepts darstellen. Dies erfolgte im Wesentlichen anhand von umfangreichen Internetrecherchen. In einem ersten Schritt wurden dabei zunächst die gefundenen Möglichkeiten zusammengestellt.

Die folgenden Tabellen geben einen Überblick über die Resultate der Recherche. Tabelle 3.1 listet verschiedene Message-Brokers auf. Diese könnten sowohl für das Queue-Management als auch zur User-Notifizierung verwendet werden. Ein "X" steht für offiziellen Support in der betreffenden Technologie, ein "*" für inoffiziellen Support. In Tabelle 3.2 sind die ermittelten Taskverwaltungs-Frameworks mit unterstützten Message-Brokers und Speichertechnolgien aufgelistet. Da der BormServer bereits Python, C# und Java verwendet und bei BORM-INFORMATIK AG kein Interesse besteht, noch weitere Technologien einzubinden, wurden nur Möglichkeiten miteinbezogen, die mindestens eine der bereits verwendeten Technologien verwenden. In vielen der aufgelisteten Fälle werden noch weitere Technologien unterstützt. Die unterstützten und für dieses Projekt relevanten Technologien sind ebenfalls aus den Tabellen ersichtlich.

Framework	Python-Unterstützung	.NET-Unterstützung	Java-Unterstützung
Redis [16]	X	X	X
RabbitMQ [17]	*	X	X
Amazon SQS [18]	X	X	X
Azure Service Bus [19]	X	X	X
Apacke Kafka [20]	*	*	X
Apache ActiveMQ [21]	X	X	X
Apache Qpid [22]	X	X	X
Apache RocketMQ [23]	*	*	X
Apache Zookeeper [24]	*	*	X
Google PubSub [25]	X	X	X
IronMQ [26]	X	X	X
Beanstalk [27]	*	*	*
Nats [28]	X	X	X
NSQ [29]	X	*	*
ZeroMQ [30]	X	X	X

Tabelle 3.1.: Message-Broker

Framework	Technologie	Unterstützte Message-Broker	Unterstützte Speichertechnologien (Tasks/Resultate)
Django background tasks [31]	Python (Django)	-	Django ORM
Celery [32]	Python	RabbitMQ (Standard), Redis, Amazon SQS, Zookeeper	SQLAlchemy, Django ORM, MongoDB, Memcached, Redis, RPC (RabbitMQ)
Python-rq [33]	Python	-	Redis
Dramatiq [34]	Python	Redis, RabbitMQ, (Amazon SQS, nur als Erweiterung)	Dramatiq backend, Redis
TaskTiger [35]	Python	-	Redis
Huey [36]	Python	-	Redis, Sqlite, Dateisystem, In-Memory-Speicher
MRQ [37]	Python	-	Redis, MongoDb
KQ (Kafka Queue) [38]	Python	Apache Kafka	-
Hapless [39]	Python	-	-
Taskmaster [40]	Python	ZeroMQ	-
Kuyruk [41]	Python	RabbitMQ, (Redis, nur als Erweiterung), interne Signale	-
python-task-queue [42]	Python	Amazon SQS	Dateisystem
SimpleQ [43]	Python	Amazon SQS	-
Faktory [44]	Python, Java	-	Redis
Gofer.NET [45]	C#/.NET	-	Redis
Coravel [46]	C#/.NET	-	-
Hangfire [47]	C#/.NET	RabbitMQ (nur als Erweiterung), MSMQ (bald abgekündigt)	Redis, SQL Server, SQL Azure
Runly [48]	C#/.NET	-	-
JobRunr [49]	Java	-	Verschiedene RDBMS- und noSQL-Speicher
Quartz [50]	Java	-	Beliebige RDBMS-Speicher (Anbindung via JDBC)

Tabelle 3.2.: Taskverwaltungs-Frameworks

3.1.5. Vorselektion und erste Tests

Nach der Recherche wurden diejenigen Frameworks selektiert, die für die Realisierung der Task-Verwaltung bzw. die Kommunikation in Frage kamen. Mit diesen Frameworks wurden dann erste, vom BormServer noch völlig unabhängige Testversuche durchgeführt. Nachfolgend werden die in den Testversuchen ermittelten Erkenntnisse bzw. die Begründungen, warum gewisse Frameworks nicht näher betrachtet wurden, beschrieben.

3.1.5.1. Redis

Gemäss Dokumentation wird Redis auf Windows nicht offiziell unterstützt. Es ist jedoch möglich, Redis auf Windows unter Windows Subsystem for Linux (WSL) auszuführen [51]. Da verschiedene der getesteten Frameworks Redis entweder als Broker oder als Backend verwenden, wurde dies auch ausprobiert. Bei den Frameworks Python-rq, Dramatiq sowie Gofer.NET funktionierte der Redis-Server für die getesteten Beispielskripts, während bei Celery, Huey und MRQ merkwürdige Verbindungsfehler auftraten, zu denen auf Anhieb keine Lösung gefunden werden konnte. Die Erkenntnis aus diesen

Versuchen ist, dass die Verwendung von Redis unter Windows zwar möglich, aber nicht sehr zuverlässig ist.

3.1.5.2. RabbitMQ

Im Gegensatz zu Redis wird RabbitMQ auf Windows unterstützt. Als Vorbedingung muss lediglich eine aktuelle Version von Erlang [52] installiert werden. Der RabbitMQ-Service wird über den Desktop gestartet. Sowohl mit Dramatiq wie auch mit Celery hat RabbitMQ für die getesteten Beispielskripts einwandfrei funktioniert. Es ist daher eine gute Empfehlung für die Verwendung unter Windows.

3.1.5.3. Weitere Message-Broker

Da die getesteten Taskverwaltungs-Frameworks alle entweder Redis oder RabbitMQ oder beides unterstützen, wurden keine weiteren Message-Broker getestet. Die getesteten Beispiele haben mit Redis und RabbitMQ funktioniert. Andere Taskverwaltungs-Frameworks, die mit anderen Message-Brokern funktionieren, wurden aus anderen Gründen nicht getestet, siehe Abschnitt 3.1.5.13.

3.1.5.4. Celery

Celery gilt als renommiertes und zuverlässiges Taskverwaltungs-Framework mit breiter Unterstützung für diverse Message-Brokers und Speichertechnologien. Es verfügt über zahlreiche Scheduling-Optionen für zeitlich regelmässig auszuführende Background-Tasks. Der standardmässig verwendete Message-Broker ist RabbitMQ. Celery bietet Callback-Methoden (sogenannte Signals) an, um Informationen zum Zustand eines Tasks sowie zu Resultaten oder Fehlern in der Ausführung zu erhalten. Alternativ dazu können Resultate mittels einer der unterstützten Speichertechnologien persistiert werden. Aufgrund von Änderungen in der Concurrency-Pool-Implementation wird Celery zwar seit Version 4 unter Windows nicht mehr offiziell unterstützt. Durch die Spezifizierung des zu verwendenden Concurrency-Frameworks beim Aufruf von Celery kann es jedoch weiterhin unter Windows ausgeführt werden [53]. Für die ersten Tests wurde das Concurrency-Modul Eventlet [54] verwendet. Damit haben die Beispielskripts einwandfrei funktioniert.

3.1.5.5. Python-rq

Python-rq wurde gemäss Dokumentation mit dem Ziel entwickelt, eine einfachere Variante zu Celery in Python anzubieten [55], und hat wie dieses einen hohen Beliebtheitsgrad. Es verwendet Redis sowohl als Message-Broker wie auch als Speichertechnologie. Mit Redis unter WSL hat Python-rq für die Testskripts unter Windows funktioniert, allerdings musste der Worker dazu nicht unter Windows, sondern ebenfalls unter WSL gestartet werden. Python-rq bietet wie Celery Scheduling-Optionen an, wenn auch nicht in so zahlreicher Form. Es ist auch eine gute Möglichkeit für die Implementation einer Taskverwaltung, allerdings ist zu bedenken, dass Redis unter Windows nicht sehr zuverlässig zu funktionieren scheint, siehe Abschnitt 3.1.5.1.

3.1.5.6. Dramatiq

Dramatiq wurde wie Python-rq als Lightweight-Alternative zu Celery entwickelt [56] und hat sich als solche einige Bekanntheit erworben. Es unterstützt standardmässig RabbitMQ und Redis als Message-Broker, kann jedoch mittels einer Erweiterung auch mit AmazonSQS verwendet werden. Da bei den übrigen Tests ebenfalls RabbitMQ und Redis zum Einsatz kamen, wurde es mit diesen beiden getestet und hat für die Testskripts auch mit beiden funktioniert. Redis kann auch als Speichertechnologie verwendet werden, um Resultate abzuspeichern. Wie Celery bietet auch Dramatiq Callbacks an, um Informationen über Resultate oder Fehler in der Ausführung zu erhalten. Ein Nachteil an Dramatiq ist, dass es standardmässig nur sehr mangelhafte Unterstützung für Scheduling anbietet. Sollte Scheduling benötigt werden, können gemäss Dokumentation zusätzliche Scheduling-Libraries installiert werden, empfohlen wird APScheduler [57]. Dies wurde ebenfalls getestet und funktionierte für ein einfaches Testskript.

3.1.5.7. TaskTiger

TaskTiger ist ein Taskverwaltungs-Framework und basiert wie Python-rq auf Redis. Bei den Tests wurde zu Beginn festgestellt, dass intern die fcntl-Bibliothek verwendet wird, die unter Windows nicht unterstützt wird [58]. Daher ist TaskTiger keine Möglichkeit für die Anwendung auf dem BormServer.

3.1.5.8. Huey

Huey ist eine Task-Queueing-Library für Python. Aus diesem Grund bietet es viele Optionen wie Tasks verarbeitet werden können. Darunter auch die Option, dass fehlgeschlagene Tasks selbständig erneut versucht werden oder das Setzen von Prioritäten. Huey hat einen integrierten Support für Redis und wurde auch dafür entworfen. Es kann über die API aber auch eine andere Option implementiert werden. Es fehlt jedoch eine integrierte Unterstützung für Message-Broker und die Resultate werden in einem Result-Objekt gespeichert, welches in der Konsole ausgegeben werden kann, und müssten hier wieder abgegriffen werden. Dies wäre zwar möglich, indem beispielsweise ein Message-Broker eingebunden wird, über den die Resultate weitergeleitet werden. Beim Testen funktionierte jedoch die Verbindung mit Redis per RedisHuey nicht. Deshalb wurde es nicht weiter in Betracht gezogen.

3.1.5.9. MRQ

MRQ verspricht ein einfaches Vorgehen wie Python-rq mit einer ähnlichen Performance wie der von Celery. Es ist eine Task-Queue für Python, welche auf Redis, MongoDB und Gevent basiert. Es ist jedoch keine built-in Unterstützung für Message-Broker erkenntlich. Laut Anleitung muss ein MongoDB- sowie ein Redis-Server gestartet werden. Bei den Tests stellte sich heraus, dass das Beispiel nicht mehr funktioniert. Des weiteren konnte zwar eine Verbindung zu MongoDB aufgebaut werden, der Task konnte allerdings nicht gestartet werden und der Test lief in einen Error. Auch Anpassungen des Beispiels brachte keine Besserung. Auf GitHub steht des Weiteren, der letzte Zustand des Builds auf Error und das letztes Update war 13.12.2020 (Stand 16.05.2023). Aus diesen Gründen wurde gegen MRQ entscheiden.

3.1.5.10. Gofer.NET

Gofer.NET ist eine .NET 2.0 Lösung welche von Celery für Python inspiriert ist. Als Speichertechnologie wird Redis unterstützt. Es unterstützt das Queueing von Tasks, welche automatisch vom ersten freien Worker bearbeitet werden, wobei die Tasks persistent sind. Der Worker-Pool kann einfach durch das Hinzufügen von Nodes erweitert werden. Es fehlt, wie auch bei anderen getesteten Frameworks, eine eingebaute Unterstützung für Message-Broker. Ausserdem scheint Redis, wie oben unter 3.1.5.1 bemerkt, auf Windows nicht sehr zuverlässig zu funktionieren. Daher scheint die Verwendung von Gofer.NET ungeeignet.

3.1.5.11. Coravel

Coravel ist ein Taskverwaltungs-Framework für ASP.NET. Nebst Task-Queueing bietet es zahlreiche Schedulingoptionen an. Bei den ersten Tests liess es sich problemfrei über NuGet installieren. Um Python-Skripts in .NET ausführen zu können, wurde zudem das .NET-Python-Interface Python.NET [11] installiert, das die Ausführung von Python-Code in .NET ermöglicht. In Kombination damit konnten einfache Python-Anweisungen zeitgesteuert ausgeführt werden. Coravel unterstützt zwar standardmässig keinen Message-Broker, doch konnte RabbitMQ mittels des RabbitMQ.Client NuGet-Paketes einfach miteingebunden werden.

3.1.5.12. Hangfire

Hangfire ist eine .NET Lösung mit zahlbaren Extrafeatures. Es unterstützt in der kostenlosen Version SQL-Database und InMemory als Backend, in der kostenpflichtigen Version Redis. Die Ausführung von Python-Skripts sowie die Anbindung von RabbitMQ funktioniert ähnlich wie bei Coravel (siehe

oben 3.1.5.11). Tasks werden von einem Backgroundjob-Client in eine Queue im Backend gespeichert. Der Hangfire-Server erstellt Worker-Threads, welche die Tasks ausführen.

3.1.5.13. Weitere Taskverwaltungs-Frameworks

Weitere Taskverwaltungs-Frameworks wurden aus verschiedenen Gründen nicht getestet. Die Java-Optionen wurden nicht behandelt, da bei BORM-INFORMATIK AG keine Erweiterungen des Java-Teils des BormServers geplant sind. Eine Java-Lösung wäre somit nicht im Interesse der BORM-INFORMATIK AG. Faktory und Runly nehmen Tasks als JSON-Strings entgegen, was nicht den von BORM-INFORMATIK AG spezifizierten Anforderungen in Abschnitt 3.1.2 entspricht. Hapless ist ein CLI-Tool und fällt daher ebenfalls weg. Die Frameworks KQ, Taskmaster, Kuyruk, python-task-queue und SimpleQ bieten keine Scheduling-Unterstützung an und scheinen gemäss den Commit-Daten auf GitHub grossteils nicht mehr aktuell zu sein, weshalb sie ebenfalls keine validen Möglichkeiten darstellen.

3.1.6. Auswahl geeigneter Frameworks

Nach den in Abschnitt 3.1.5 beschriebenen ersten Tests und Vorselektionen blieben bei Python Celery und Dramatiq, bei .NET Coravel und Hangfire als mögliche Optionen für die Taskverwaltung übrig. Da Redis auf Windows nur über WSL installiert werden kann und, wie in Abschnitt 3.1.5.1 beschrieben, nicht zuverlässig zu funktionieren scheint, wurde in Absprache mit BORM-INFORMATIK AG entschieden, RabbitMQ als Message-Broker zu verwenden. Alle Frameworks, die nur Redis unterstützen, fielen dadurch weg. Durch die Verwendung der Callback-Funktionen bei Celery oder Dramatiq kann das fortlaufende Polling des Task-Zustandes eliminiert werden. In diesem Fall würde keine Speichertechnologie für die Persistierung der Resultate benötigt. Bei Coravel und Hangfire müssten die Resultate über RabbitMQ weitergeleitet werden oder zwischengespeichert werden (z.B. in der Datenbank).

3.1.7. Weitere Tests der ausgewählten Frameworks

Die vier Frameworks wurden nach der Vorselektion weiter getestet. Insbesondere musste die Verwendung von Celery und Dramatiq auf einem Flask-Server getestet werden, da der Python-Teil des BormServers ebenfalls als Flask-Server implementiert ist. Die Tests erfolgten lokal auf einem eigenen kleinen Flask-Server, losgelöst von der BormServer-Umgebung. Die Testresultate werden im Folgenden beschrieben.

3.1.7.1. Dramatiq auf Flask

Für die Integration von Dramatiq auf Flask existieren zwei zusätzliche Python-Module: Flask-Dramatiq und Flask-Melodramatiq [59]. Flask-Dramatiq liess sich zwar ohne Probleme installieren, die Background-Tasks wurden jedoch nicht ausgeführt. Mit Flask-Melodramatiq liessen sich die Tasks zwar ausführen, aber nur, wenn sie im gleichen Modul wie der Flask-Server implementiert waren. Zeitgesteuerte Tasks (unter Verwendung des empfohlenen Scheduling-Moduls APScheduler, siehe Abschnitt 3.1.5.6) wurden ebenfalls nicht ausgeführt. Ein weiteres empfohlenes Scheduling-Modul, Periodiq [60], wird gemäss aufgetretener Fehlermeldung bei erstem Test auf Windows nicht unterstützt. Die Quintessenz aus diesen Tests lautet daher, dass Dramatiq für die Task-Verwaltung in Verbindung mit dem Flask-Server keine valide Option darstellt.

3.1.7.2. Celery auf Flask

Celery kann gemäss Flask-Dokumentation einfach auf einem Flask-Server verwendet werden [61]. Bei den Tests funktionierte sowohl das Scheduling wie auch das Abrufen von Resultaten über die Callback-Funktionen sehr gut. Tasks können in einem separaten Python-Modul definiert und als "Shared Tasks" von einem anderen Modul aufgerufen werden. Der Vollständigkeit halber wurde auch das Abfragen von gespeicherten Resultaten mittels Polling getestet. Dies funktionierte jedoch überraschenderweise nicht.

Die Tasks wurden zwar erfolgreich ausgeführt, doch die Abfrage lieferte kein Resultat zurück. Verschiedene im Internet vorgeschlagene Lösungswege änderten das Resultat nicht. Recherchen suggerierten, dass die Ursache des Problems in der fehlenden Windows-Unterstützung liegen könnte (vergleiche auch Abschnitt 3.1.5.4) [62] [63] [64]. Anscheinend benutzt Celery ein Signal (SIGUSR1), welches von Windows nicht unterstützt wird [65]. Dies stellt die Verwendung von Celery trotz der erfolgreichen übrigen Tests klar in Frage. Ausserdem stellte sich heraus, dass es nicht oder nur sehr schwer möglich ist, zeitgesteuerte Tasks zur Laufzeit hinzuzufügen oder zu entfernen.

3.1.7.3. Coravel auf ASP.NET

Coravel wurde ebenfalls lokal mit einer einfachen ASP.NET-API getestet. In der App-Konfiguration, die beim Start der Anwendung ausgeführt wird, wurde ein zeitgesteuerter Test-Task, der ein minimales Python-Statement via Python.NET ausführt und den Rückgabewert auf der Konsole ausgibt, konfiguriert. Der Rückgabewert wurde ausserdem mittels RabbitMQ in eine Message-Queue publiziert. In Python wurde ein RabbitMQ-Client implementiert, der sich für diese Message-Queue registriert. Dieser Test funktionierte einwandfrei. Für nicht zeitgesteuerte Tasks existiert das IQueue-Interface, mittels dessen synchrone und asynchrone Tasks innerhalb eines API-Controllers direkt an eine Queue übergeben werden können. Dazu wurde ein eigener API-Controller erstellt. Darin wurde eine Get-Methode implementiert, die einen Task an eine Queue übergibt. Das Resultat des Tasks wurde ebenfalls via RabbitMQ publiziert. Auch dieser Test hat ohne Probleme funktioniert. Für Controller-Klassen bietet Coravel zudem einen internen Event-Listener-Mechanismus an, d.h. die Verwendung von RabbitMQ wäre innerhalb von Controller-Klassen nicht unbedingt notwendig. Bei den zeitgesteuerten Tasks ist diese Möglichkeit nicht gegeben. Die Verwendung von RabbitMQ ergibt aber ohnehin Sinn, da durch die Möglichkeit, Nachrichten in einer anderen Umgebung (z.B. in Python, wie im oben beschriebenen Test) zu empfangen, eine höhere Flexibilität gewährleistet wird. Resultate können dann auch in einem anderen Teil der Anwendung, etwa auf dem Flask-Server, bearbeitet werden.

3.1.7.4. Hangfire auf ASP.NET

Da Hangfire ähnlich wie Coravel funktioniert, wurden hier auch ähnliche Tests durchgeführt. Es wurde auch eine lokale ASP.NET-API verwendet. Auf einen Test mit RabbitMQ wurde verzichtet, da die Integration schon mit Coravel getestet wurde (siehe oben 3.1.7.3). Das Queue-Handling und Scheduling funktionieren sehr ähnlich wie in Coravel. Hangfire bietet jedoch einige zusätzliche Funktionalitäten an. So können unter anderem die Worker-Queues benannt werden. Dies erlaubt es, die Tasks in Gruppen einzuteilen und somit zum Beispiel langsamere und schnellere Tasks in separaten Prozessen auszuführen. Des weiteren können auszuführende Tasks optional mittels SQL-Server persistiert werden. Dies erlaubt die komplette Trennung von Task-Queueing/-Scheduling und Task-Ausführung.

3.1.8. Festlegung auf ein Framework und Erstellung des Konzepts

Nach diesen Tests wurde klar, dass Dramatiq, wie oben beschrieben, sich nicht eignet. Für die verbleibenden drei Frameworks wurde die nachstehende Tabelle erstellt. Die Gewichtung so wie die Vergabe der Punkte wurde zusammen mit BORM-INFORMATIK AG vorgenommen.

ENTSCHEIDUNGSMATRIX - GEWICHTET

	Kriterium 1	Kriterium 2	Kriterium 3	Kriterium 4	Kriterium 5	Kriterium 6	
KRITERIEN BESCHREIBUNG	Multi-Processing mit einfachem Handling	Dynamische Erstellung von Scheduled Tasks	Queue-Manipulation zur Laufzeit	Minimale Technologie-sprünge	Windows Long Term Support	Einfaches Resultat-Handling	
	Kriterium 1	Kriterium 2	Kriterium 3	Kriterium 4	Kriterium 5	Kriterium 6	GEWICHTETE PUNKTZAHL
GEWICHT	5	4	6	1	3	2	21
	24%	19%	29%	5%	14%	10%	100%
OPTIONEN	Kriterium 1 PUNKTE	Kriterium 2 PUNKTE	Kriterium 3 PUNKTE	Kriterium 4 PUNKTE	Kriterium 5 PUNKTE	Kriterium 6 PUNKTE	
Celery	4	1	1	5	1	5	2.29
Coravel	3	3	4	2	5	4	3.62
Hangfire	4	3	4	2	5	4	3.86

Abbildung 3.1.: Gewichtete Entscheidungsmatrix

Celery Wie schon in 3.1.7.2 beschrieben, stellte sich hier heraus, dass Celery ein Signal verwendet, welches unter Windows nicht funktioniert. Dies erschwerte bzw. verunmöglichte dynamische Erstellung von Scheduled Tasks und Queue-Manipulation. Auch sprach dies für einen schlechten Windows Long Term Support.

Coravel und Hangfire Wie aus der Tabelle zu entnehmen, fiel die Entscheidung auf Hangfire. Die zusätzlichen Features von Hangfire versprachen eine grössere Flexibilität in der Taskverwaltung. Coravel besitzt zwar eine grosse Auswahl an eingebauten Scheduling-Methoden, die meisten sind allerdings für dieses Projekt nicht relevant. So hat es unter anderem vorgefertigte Methoden für alle 15 Minuten oder alle 30 Sekunden. Für diese Arbeit jedoch sollte Cron ausreichend sein und dieses wird von beiden Frameworks unterstützt. Auch aus kommerzieller Perspektive ist die Verwendung von Hangfire gemäss Einschätzung der BORM-INFORMATIK AG besser geeignet. Bei Coravel müsste man für zusätzliche Features wie etwa die Persistierung auszuführender Tasks pro BORM-Kunde pro Jahr 250 USD bezahlen. Auch bei Hangfire gibt es bezahlte Features, allerdings für einen Fixbetrag von 4500 USD pro Jahr. Im Moment werden diese Features nicht benötigt; die Persistierung ist bei Hangfire in der Gratisversion miteinbegriffen. Sollten jedoch irgendwann bezahlte Features benötigt werden, wäre Hangfire auch hier die bessere Option.

Für die Anwendung des Hangfire-Frameworks wurde ein Konzept als Basis für die Realisierung des Prototyps erstellt. Details dazu sind in Kapitel 4.1 beschrieben.

3.1.9. Implementation des Prototyps

Basierend auf dem in Abschnitt 3.1.8 erstellten Konzept wurde anschliessend an die Konzeptfindung ein Prototyp erstellt. In Absprache mit der BORM-INFORMATIK AG wurde der Fokus dabei zunächst auf die Systemoperationen "Starten eines Tasks" und "Scheduling von Tasks" gelegt, da diese beiden die eigentlichen Kernfunktionalitäten der Taskverwaltung darstellen. Gemäss Aussage der BORM-INFORMATIK AG würde es im Umfang der vorliegenden Arbeit genügen, die beiden anderen Systemoperationen nicht zu implementieren, sondern lediglich konzeptionelle Lösungen in der Dokumentation zu beschreiben. Gleiches gilt für die Behandlung der Resultate. Da Hangfire standardmässig keinen Message-Broker unterstützt, sollte dieser Ansatz im Rahmen dieser Arbeit nicht weiterverfolgt werden, zumal die Integration eines Message-Brokers diverse weitere Anwendungsfälle in der gesamten BORM-Softwarearchitektur umfasst. Da der C#-Teil des BormServers als ASP.NET-Web-API implementiert ist, waren sämtliche Interaktionsmöglichkeiten als API-Endpoints zu implementieren. Nebst

den beiden genannten Hauptoperationen sollte es zudem möglich sein, neue Task-Konfigurationen zu erfassen sowie bestehende Task-Konfigurationen zu updaten oder zu löschen, da in Zukunft sämtliche Interaktionen vonseiten des Anwenders mit der Taskverwaltung über die API laufen soll. Eine erste Version des Prototyps wurde mit genau diesen Funktionalitäten implementiert. Die Resultate sind in Kapitel 4.2.1 beschrieben.

3.1.10. Unit- und Systemtests

Für die Verfikation des Verhaltens des Prototyps wurden sowohl Unit-Tests implementiert als auch Systemtests durchgeführt. Integratinstests waren gemäss Aussage der BORM-INFORMATIK AG nicht zwingend zu implementieren. Der Datenbankzugriff sowie der API-Controller wurden mittels Unit-Tests getestet. Die einzelnen Tests sind in Kapitel 4.2.3.1 aufgeführt. Die Systemtests wurden durchgeführt, um das Verhalten der Tasks unter verschiedenen möglichen Umständen zu untersuchen. Die zu testenden Fälle wurden gemeinsam mit der BORM-INFORMATIK AG evaluiert. Die Beschreibung der Tests mit den Ergebnissen sowie die verwendeten Testskripts sind in Kapitel 4.2.3.2 aufgeführt. Zusätzlich hätten noch Integration-Tests implementiert werden können. Da aber die Funktionalität des API-Controllers bereits über Unit-Tests sowie im laufenden Betrieb der App verifiziert wurde, wurde in Absprache mit der BORM-INFORMATIK AG darauf verzichtet.

3.1.11. Integration von BORM-internen Python-Bibliotheken

Bei der Minimalimplementation des Prototyps sowie bei den Tests war auf die Verwendung der BORM-internen Python-Bibliotheken einfachheitshalber verzichtet worden. Um die Nutzbarkeit des Prototyps für die BORM-INFORMATIK AG zu zeigen, war es nun in einem weiteren Schritt wichtig, diese Bibliotheken in den Python-Tasks zu verwenden. Wie in Kapitel 1.1.1 beschrieben, existieren für gewisse Hintergrundtasks bereits behelfsmässige Lösungen, u.a. für die Lagerbewertung im ERP. Unser ursprüngliches Ziel war, diese umzuschreiben und als Beispiel in den Prototyp zu integrieren. Dies gestaltete sich allerdings wesentlich schwieriger als erwartet, da die Lagerbewertung für den Datenbankzugriff viele weitere BORM-interne Pythonmodule importierte, welche ihrerseits via Python.NET wieder auf C#-DLLs zugreifen. Dies führte zu Fehlern beim Import dieser Module. Trotz intensiver Fehlersuche gelang es uns nicht, eine funktionierende Lagerbewertung zu implementieren. Gemäss Einschätzung der BORM-INFORMATIK AG sind die aufgetretenen Fehler im Wesentlichen auf Konflikte in den verwendeten DLLs zurückzuführen, insbesondere weil der Applikationsserver und der Prototyp der Taskverwaltung 64-Bit-kompatibel kompiliert werden, während die von den Python-Modulen importierten DLLs z.T. noch ausschliesslich 32-Bit-kompatibel sind. Die Behebung dieser Konflikte würde sich über den Bereich dieser Arbeit hinaus verzögern.

In Absprache mit der BORM-INFORMATIK AG wurde daher entschieden, einen einfachen Beispieltask zu implementieren, der eine BORM-interne Python-Bibliothek importiert, die selbst keine weiteren BORM-internen Module importiert. Es gibt solche Bibliotheken für einfache Hilfsfunktionen, z.B. für das Handling von Zeichenketten. Die Implementation selbst ist in Kapitel 4.2.1.5 näher beschrieben.

3.1.12. Konzeptionelle Überlegungen zum Prototyp

Das Resultat-Handling und die Systemoperation "Task abbrechen" konnten aus Zeit- und Komplexitätsgründen nicht mehr implementiert werden. Zu diesen und weiteren Themen, die im Verlauf der Implementation des Prototyps aufkamen, wurden aber konzeptionelle Überlegungen angestellt. Die Trennung zwischen umgesetzten und konzeptionell betrachteten Funktionalitäten erfolgte in Absprache mit der BORM-INFORMATIK AG. Die Resultate sind in Kapitel 4.2.4 beschrieben.

3.2. Arbeitsmethodik

Als Arbeitsmethodik wurde eine reduzierte Version von SCRUM [66] gewält. Diese beinhaltete im Wesentlichen die Unterteilung der Projektphase in zweiwöchige Sprints (siehe Abschnitt 3.5) sowie die Durchführung eines Startmeetings am Anfang sowie eines Reviews am Ende eines einzelnen Sprints. Eine ähnliche Vorgehensweise hatten wir bereits erfolgreich in der Projektarbeit im 5. Semester angewendet und waren daher schon damit vertraut. Die Startmeetings und Reviews wurden jeweils am Entwicklungsstandort der BORM-INFORMATIK AG in Steinhausen zusammen mit den uns betreuenden Ingenieuren durchgeführt. Zusätzlich hatten wir jede Woche am Donnerstagmorgen eine Sitzung mit Herrn Prof. Spielberger, wobei wir den aktuellen Fortschritt und allfällige Probleme besprachen. Weitere Meetings, miteinander oder auch mit Betreuern, fanden je nach Bedarf spontan statt.

3.3. Git

Für die Arbeit selbst sowie für die Dokumentation wurde Git verwendet. GitHub (siehe 2.1.1) wurde nur für die Dokumentation verwendet, da für die Programmierarbeit das GitLab (siehe 2.1.1) von BORM-INFORMATIK AG verwendet wurde.

3.3.1. GitLab

Für die Arbeit wurde von BORM-INFORMATIK AG ein Feature-Branch zur Verfügung gestellt. Dieser wurde im Zuge der Arbeit wie ein Main-Branch behandelt. Für das Ausprobieren und Implementieren der Features wurden weitere Feature-Branches erstellt, welche von diesem "Main"-Branch abhängig waren.

3.3.2. GitHub

Auf dem GitHub der ZHAW wurde ein Repository für die Dokumentation eingerichtet. Es wurden für die einzelnen Kapitel Issues (siehe 2.1.2) erstellt, um die Dokumentationsarbeit besser aufzuteilen und um einen Überblick zu behalten, was noch geschrieben werden muss. Es wurde pro Issue ein Branch erstellt. Des Weiteren erlaubte GitHub die Abnahme einzelner Texte durch den Arbeitspartner zu einem späteren Zeitpunkt, ohne dass der Arbeitsfluss beeinträchtigt wurde.

3.4. Pair Programming

Unter Pair Programming [67] versteht man eine Arbeitsweise, bei der eine Person Code schreibt während die andere Person als Co-Pilot zuschaut und den Überblick behält. Der Co-Pilot ist auch für allfällige Recherchen und Tipps während des Programmierens zuständig. Dabei sollte darauf geachtet werden, dass sich Pilot und Co-Pilot regelmässig in den Rollen abwechseln. Diese Arbeitsweise eignet sich besonders für neue bzw. unbekannte Technologie oder bei sehr komplexem Code. Es ermöglicht den Programmierern sich mit einander abzusprechen und gemeinsam eine Lösung zu finden. Des Weiteren vermindert es die Chance, einen Fehler zu schreiben, da der Co-Pilot auch den Code betrachtet und nicht so sehr ins Schreiben selbst vertieft ist. Pair Programming wurde bei der Erstellung des Prototyps fast immer verwendet, da der BORM-Code selbst schon sehr gross ist und oft etwas bezüglich Hangfire recherchiert werden musste. Auch die Erstellung des Konzepts sowie die Systemtests wurden gemeinsam durchgeführt. Die Unit-Tests hingegen sowie die Recherche wurden aufgeteilt und als Einzelarbeit durchgeführt.

3.5. Zeitplanung

Die Ausgabe der Bachelorarbeit erfolgte offiziell am 13.02.2023. Am 15.02. erfolgte die Ausgabe und Inbetriebnahme der für die Durchführung verwendeten firmeneigenen Notebooks. Der darauf folgende Zeitraum von 14 Wochen bis zum Ende der Vorlesungen am 26.05. wurde in 7 zweiwöchige Entwicklungszyklen (Sprints, siehe Abschnitt 3.2) eingeteilt. Die verbleibende Zeit bis zur Abgabe am 09.06. wurde für die Fertigstellung der Dokumentation reserviert. Zusätzlich wurden in Orientierung an der Aufgabenstellung (siehe Kapitel 1.2) die folgenden vier Meilensteine definiert:

1) Analyse beendet: 04.03. (Ende Sprint 1)

2) Konzept entwickelt: 15.04. (Ende Sprint 4)

3) Prototyp entwickelt und funktionsfähig: 26.05. (Ende Sprint 7)

4) Dokumentation fertiggestellt: 09.06. (Abgabe)

3.5.1. Tatsächlicher Ablauf

Der tatsächliche Ablauf folgte im Grossen und Ganzen dem Zeitplan. Die Sprintanfänge und -enden wurden auf die jeweiligen BORM-Meetings verschoben. Die wichtigsten Ereignisse sind aus der Abbildung 3.2 zu entnehmen. Darin sieht man auch, dass schon etwas vor dem definitiven Entscheid Hangfire zu benutzen, dies intensiver betrachtet wurde. Somit konnten Vorbereitungen für den Prototyp schon vorweggenommen werden. Ausser im Falle eines grossen Nachholen der Dokumentation wurde diese fortlaufend erledigt und wird in der Abbildung nicht erwähnt.

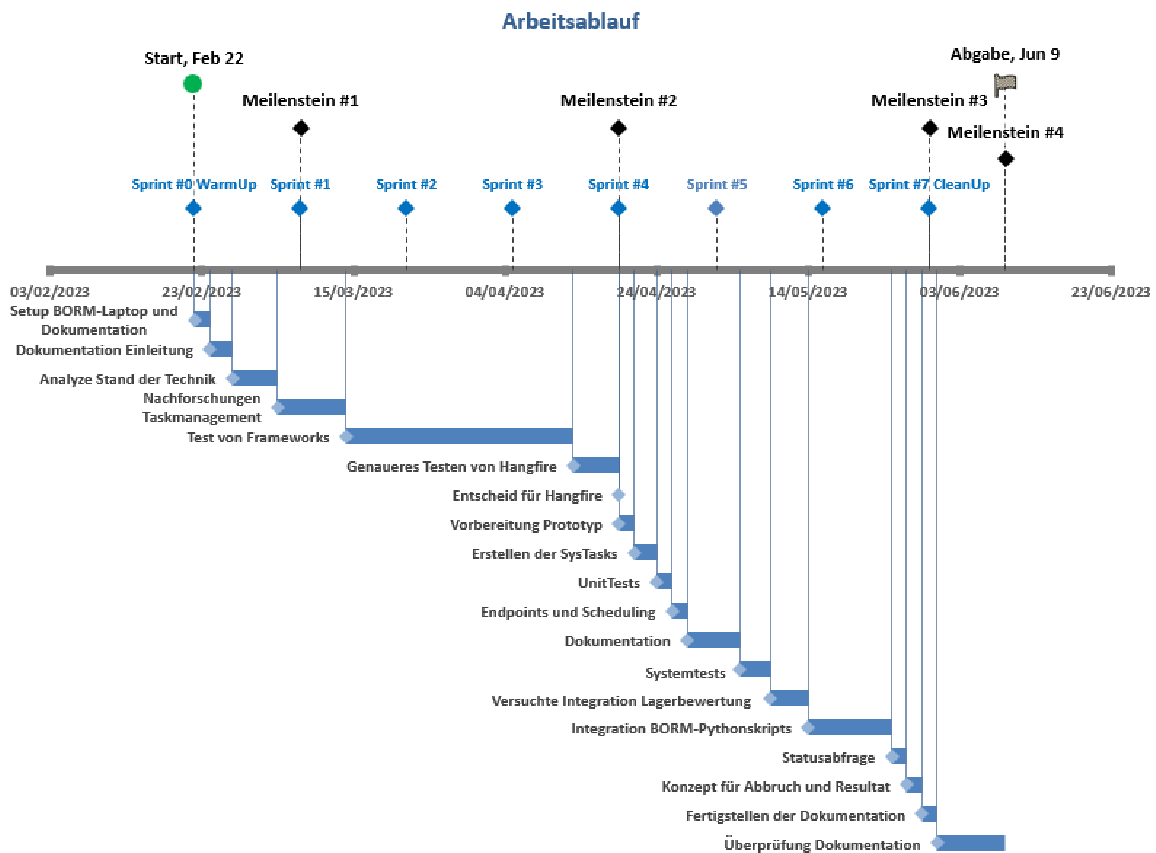

Abbildung 3.2.: Ablauf der Bachelorarbeit

4. Resultate

Dieses Kapitel befasst sich mit den Resultaten der Arbeit.

4.1. Konzept

Als Basis für die Umsetzung des Prototyps wurde ein entsprechendes Konzept erstellt. Dieses umfasst im Wesentlichen die Spezifikation der Systemoperationen, die Architekturübersicht sowie die Verteilung der einzelnen Softwaremodule auf die verschiedenen Systemkomponenten. Die folgenden Abschnitte beschreiben die verschiedenen Teile des Konzepts.

4.1.1. Systemoperationen

In Absprache mit der BORM-INFORMATIK AG wurden die einzelnen Systemoperationen, die auf der Taskverwaltung ausgeführt werden sollen, definiert. Es soll möglich sein, Tasks zu starten (Abbildung 4.1), den Status eines Tasks zur Laufzeit abzufragen (Abbildung 4.2), sowie Tasks abzubrechen (Abbildung 4.3). Ausserdem soll der Ablauf von zeitgesteuerten Tasks beim Start des Servers geplant werden (Abbildung 4.4). Es soll zudem möglich sein, zeitgesteuerte Tasks zur Laufzeit zum Ablaufplan hinzuzufügen.

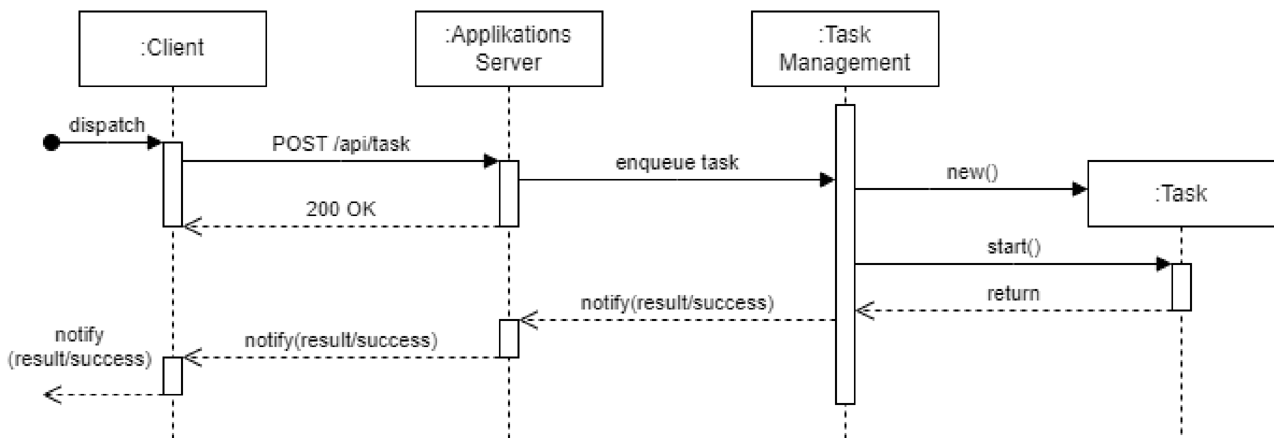

Abbildung 4.1.: Sequenzdiagramm für das Starten eines Tasks

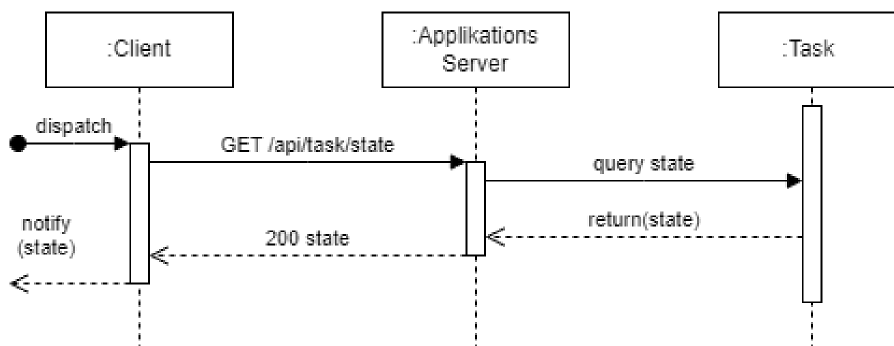

Abbildung 4.2.: Sequenzdiagramm für das Erhalten des Tasksstatus

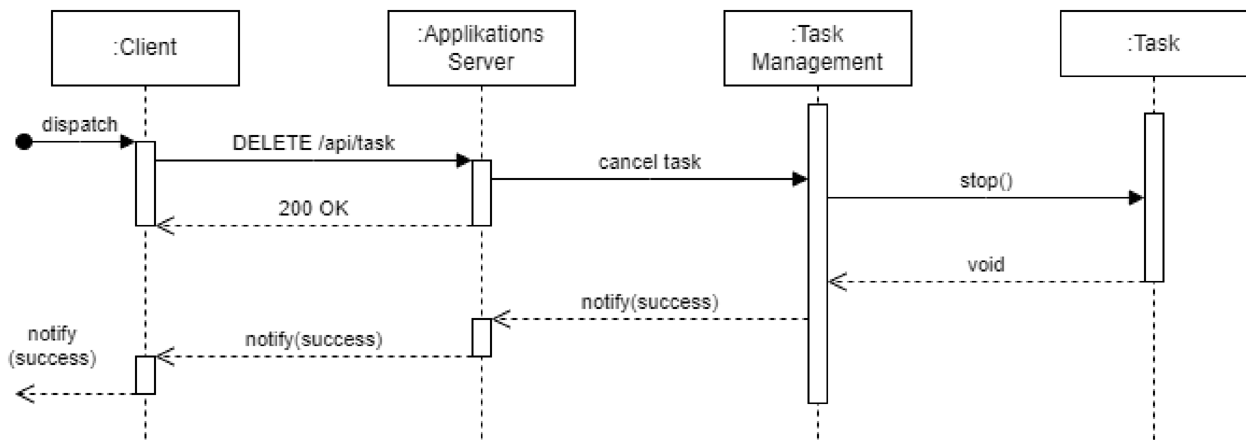

Abbildung 4.3.: Sequenzdiagramm für das Abbrechen eines Tasks

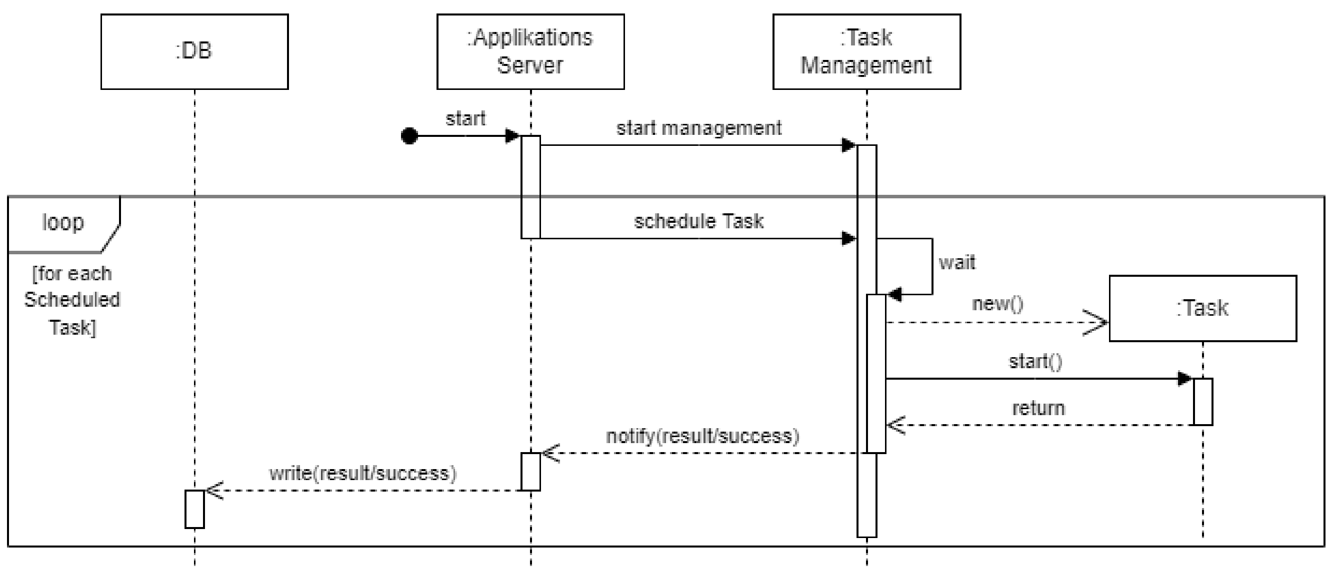

Abbildung 4.4.: Sequenzdiagramm für das Scheduling von Tasks

4.1.2. Architekturübersicht

Nach Auswahl des Frameworks (siehe Kapitel 3.1.8) wurde eine entsprechende Architekturübersicht entworfen, siehe Abbildung 4.5. Eine ausführliche Darstellung der Client-Seite ist nicht notwendig, da die Implementation der Taskverwaltung auf dem Server erfolgt. Die Client-Seite wird hier demnach einfach vollständigkeitshalber angedeutet. Auf der Server-Seite sind die verschiedenen Komponenten der Taskverwaltung dargestellt. Weitere Server-Komponenten wurden der Übersicht wegen weggelassen. In der API wird ein Hangfire-Client verwendet, auf den die API sowie die Startup-Konfigurationsklasse über Services Zugriff haben. Über diesen Client werden Tasks gestartet bzw. der Zeitplanung übergeben. Die Task-Konfigurationen (Name, Python-Dateipfad etc.) werden in der BORM-Datenbank gespeichert. Die für die Ausführung geplanten Tasks (in Hangfire-Syntax als Jobs bezeichnet) werden von Hangfire auf dem SQL-Server persistiert (dargestellt durch HangfirePersistency). Die Hangfire-Server schliesslich führen die vom Hangfire-Client in Auftrag gegebenen Jobs aus. Je nachdem ob die Jobs in die Slow-, Fast- oder Express-Queue eingereiht werden, werden sie auf dem entsprechenden Server ausgeführt. Bei dem Paket QueueMethods handelt es sich um Hilfsmethoden, mittels denen Tasks in definierte Queues enqueued werden.

Abbildung 4.6 zeigt den Workflow von Hangfire stark vereinfacht. In der Architektur wird der Hangfire-Client in der ASP.NET-API implementiert. Als Job-Storage wird der SQL-Server verwendet. Das Paket HangfireServers in der Architekturübersicht entspricht dem Hangfire Server im Workflow.

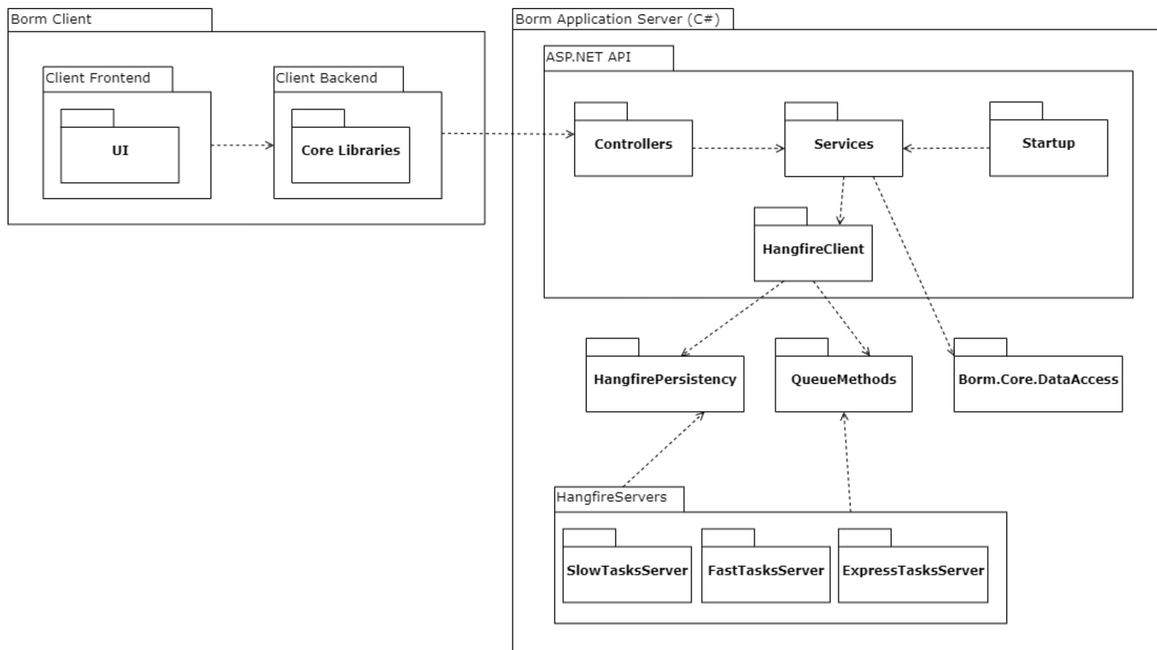

Abbildung 4.5.: Architekturübersicht mit Hangfire

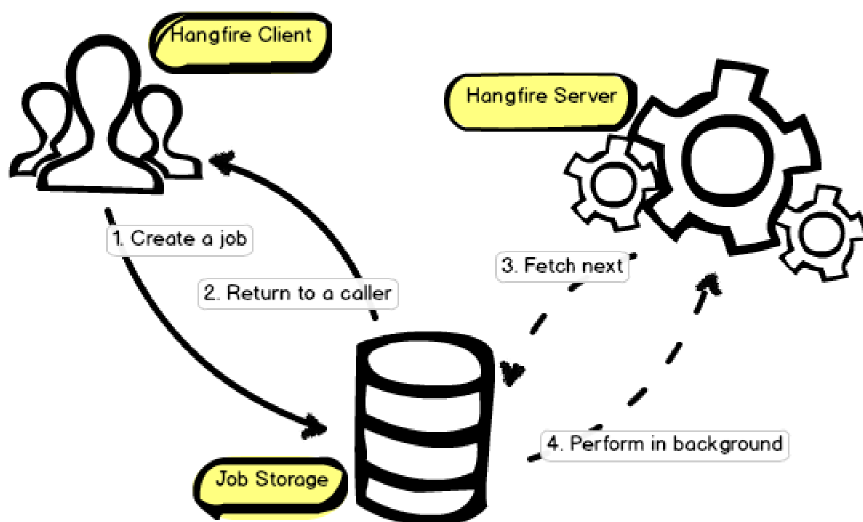

Abbildung 4.6.: Übersicht über Hangfire-Workflow [68]

4.1.3. Verteilung der Softwaremodule

Abbildung 4.7 zeigt die Verteilung der einzelnen Softwaremodule (vgl. oben Abschnitt 4.1.2) auf die verschiedenen Systemkomponenten. Auch hier ist die Client-Seite stark vereinfacht dargestellt, da sie nur insofern von Bedeutung ist, als von dort aus die durch die API zur Verfügung gestellten Endpoints angesprochen werden. Die Taskverwaltung als solche ist auf dem BormServer angesiedelt. Die Persistenzschicht besteht aus der BORM-Datenbank, in der die Task-Konfigurationen gespeichert werden, sowie einer separaten Datenbank, die von Hangfire selbst konfiguriert und zur Speicherung von Jobs verwendet wird. Gemäss Hangfire-Dokumentation ist es auch möglich, eine eigene Datenbank für die Job-Persistierung zu verwenden. Die Konfiguration (Tabellen, Constraints etc.) muss dann auch selbst durchgeführt werden [69].

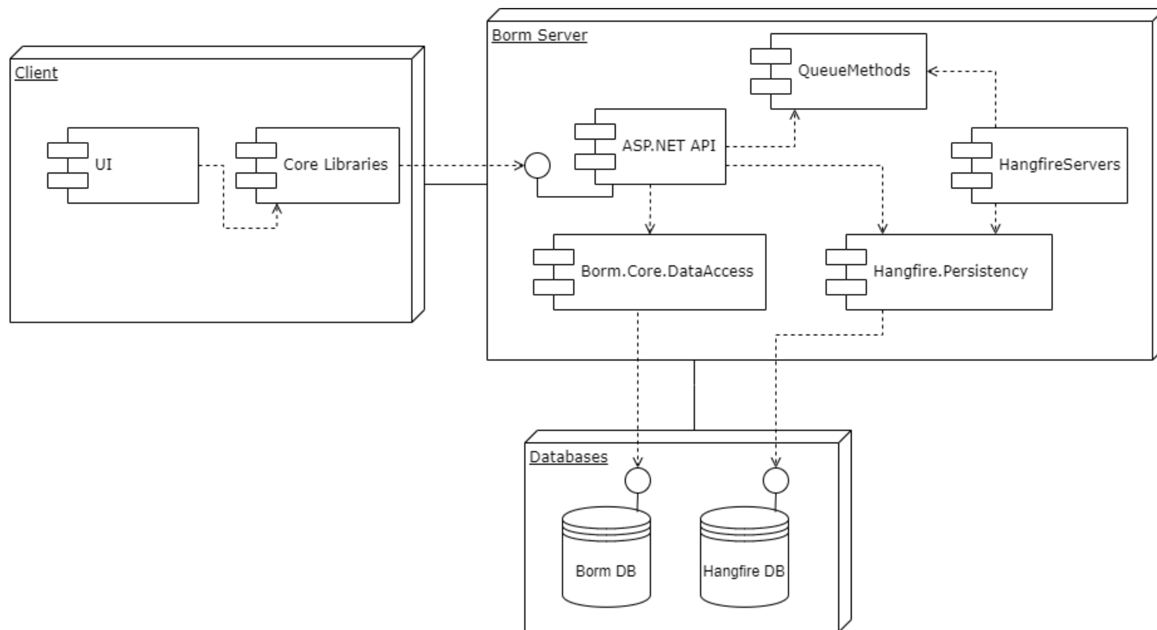

Abbildung 4.7.: Übersicht über die Verteilung der Softwaremodule

4.2. Prototyp

Im Anschluss an die Konzeptfindung wurde ein Prototyp erstellt, der in den folgenden Abschnitten näher beschrieben wird.

4.2.1. Funktionsumfang

Wie in Kapitel 3.1.9 erwähnt, lag der Fokus während der ersten Implementationsphase darauf, Tasks zu erfassen, updaten und löschen, sowie auf Scheduling und Enqueuing (Ausführen unabhängig vom Scheduling). Bezüglich Implementation wurden hierzu die Bereiche Datenbank und -zugriff, Aufrufen von Python-Code aus C#, Worker (Hangfire-Server) sowie API-Endpoints identifiziert. In den folgenden Abschnitten werden die vorgenommenen Neuimplementationen bzw. Änderungen an der bestehenden Implementation erläutert.

4.2.1.1. Datenbank

Wie in Abschnitt 4.1.2 beschrieben, sollten die Konfigurationen der einzelnen Tasks in der BORM-Datenbank gespeichert werden. Dazu wurde eine neue Tabelle in der BORM-Datenbank angelegt. BORM-intern ist es üblich, vor den eigentlichen Tabellennamen ein Präfix zu setzen, welches den Themenbereich, zu dem die Tabelle gehört, anzeigt (z. B. MAWI für Materialwirtschaft, SYS für allgemeine Einstellungen und Systemdaten). In Übereinstimmung mit dieser Konvention wurde die Tabelle SYS_TASK genannt.

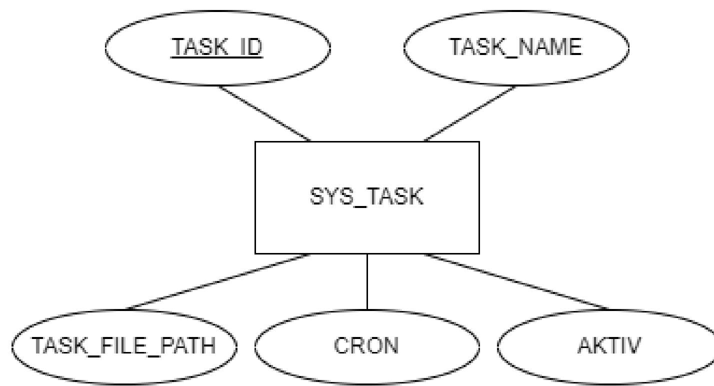

Abbildung 4.8.: Entity Relation Diagramm der SYS_TASK-Tabelle in der Datenbank

Auf dem C#-Applikationsserver existiert für Datenbankzugriffe auf bestimmte Tabellen die DLL Borm.Core.DataAccess. Die Datenbankzugriffe selbst erfolgen über ADO.NET (siehe Kapitel 2.6). Dieser DLL wurde eine neue Klasse zum Auslesen der in SYS_TASK gespeicherten Task-Konfigurationen hinzugefügt.

4.2.1.2. Worker

Die Worker wurden wie in Abbildung 4.5 dargestellt als eigenständige Prozesse implementiert. In Absprache mit der BORM-INFORMATIK AG wurden drei Server vorgesehen, die als eigenständige Prozesse laufen und über genau einen Worker-Thread verfügen. Dadurch wird sichergestellt, dass die in Kapitel 3.1.2 beschriebenen Probleme, die bei Multithreading auftreten, vermieden werden. Ein Worker kann demzufolge auch nur genau einen Job simultan ausführen. Gemäss Einschätzung der BORM-INFORMATIK AG sollte dies aber ausreichen, da scheduled Tasks nicht häufiger als ein- bis zweimal pro Tag ausgeführt werden und ein gleichzeitiges Enqueuing von mehreren Tasks auf derselben Queue, bedingt durch die im Normalfall geringe Anzahl überhaupt vorhandener Tasks (zwischen 0 und 20), selten oder gar nicht auftritt.

Anstelle dreier fast identischer Implementationen wurde ein einziges Console-Projekt so implementiert, dass die konfigurierbaren Parameter bei Programmstart als Kommandozeilenargument übergeben werden können (Listing 4.1). Das Projekt wurde Borm.Background.Worker genannt. Bei den Parametern handelt es sich um den Namen des Workers, der beliebig gewählt werden kann, sowie um die Queue, die der Worker abarbeiten soll (slow, fast oder express). Bei Start des Applikationsservers sollen dann drei Worker gestartet werden, von denen jeder je eine der drei möglichen Queues abarbeiten soll. Um einen Task in eine bestimmte Queue einreihen zu können, wurde die DLL Borm.Background.QueueMethods implementiert, welche drei Methoden zur Verfügung stellt, die je an eine der drei Queues gebunden sind (Listing 4.2). Diese Methoden können dann beim Enqueuing und Scheduling von API-Seite her aufgerufen werden (Listing 4.3).

```
1     var options = new BackgroundJobServerOptions()
2     {
3         ServerName = args[0],
4         Queues = new[] { args[1] },
5         WorkerCount = 1,
6     };
7
8     using (var server = new BackgroundJobServer(options))
9     {
10        Console.WriteLine("Hangfire Server started. Press any key to exit...");
11        Console.ReadKey();
12    }
```

Listing 4.1: Ausschnitt aus Borm.Background.Worker: Worker-Konfiguration

```
[Queue(slow)]
public static void Slow(int taskId, string taskFilePath)
{
    ExecutePythonFile(taskFilePath);
}
```

Listing 4.2: Queue-Anbindung einer Methode in Borm.Background.QueueMethods

```
switch (task.Value.Queue)
{
    case SysTask.QueueType.Slow:
        _backgroundJobClient.Enqueue(() =>
            QueueMethods.Slow(taskId, task.Value.TaskFilePath)
        );
        break;
    case SysTask.QueueType.Fast:
        _backgroundJobClient.Enqueue(() =>
            QueueMethods.Fast(taskId, task.Value.TaskFilePath)
        );
        break;
    case SysTask.QueueType.Express:
        _backgroundJobClient.Enqueue(() =>
            QueueMethods.Express(taskId, task.Value.TaskFilePath)
        );
        break;
    default:
        throw new System.Exception(
            $"Invalid queue type: {task.Value.Queue}"
        );
}
```

Listing 4.3: Enqueuing von Tasks mittels Borm.Background.QueueMethods

4.2.1.3. Aufrufen von Python-Code via Python.NET

Wie in Kapitel 3.1.2 beschrieben, sollten die auszuführenden Tasks in Python implementiert werden. Da das gewählte Framework selber nun ein C#-Framework ist, musste eine Möglichkeit geschaffen werden, um Python-Code aus C# aufzurufen. BORM-INFORMATIK AG verwendet bereits das Open-Source-Framework Python.NET (siehe Kapitel 2.7), um Funktionen aus C#-DLLs in Python zu verwenden. Da Python.NET auch die umgekehrte Funktionalität anbietet, nämlich Python-Module in C# zu importieren, war es naheliegend, dieses Framework auch hier zu verwenden. Listing 4.4 zeigt eine Minimalversion der Funktion "ExecutePythonFile", die, wie in Listing 4.2 gezeigt, von den an die verschiedenen Queues gebundenen Methoden aufgerufen wird. Der Task ist in einer Python-Datei implementiert, deren Pfad als Argument übergeben wird. Der Pfad gehört zur Task-Konfiguration, die in der BORM-Datenbank gespeichert wird (siehe Abschnitt 4.2.1.1).

```
private static void ExecutePythonFile(string taskFilePath)
{
    PythonEngine.Initialize();
    PythonEngine.BeginAllowThreads();

    using (Py.GIL())
    {
        using (PyModule scope = Py.CreateScope())
        {
            var code = System.IO.File.ReadAllText(taskFilePath);
            var scriptCompiled = PythonEngine.Compile(code, taskFilePath);
            scope.Execute(scriptCompiled);

            var result = scope.Get("result");
            System.Console.WriteLine($"Result: {result}");
        }
    }

    PythonEngine.Shutdown();
}
```

Listing 4.4: Aufrufen von Pythoncode in C#

4.2.1.4. Endpoints

Für die Benutzerinteraktion mit der Taskverwaltung wurden in der API des Applikationsservers Endpoints erstellt. Zusätzlich zu den in Abschnitt 4.1 definierten Systemoperationen wurden von der BORM-INFORMATIK AG Funktionalitäten zum Erstellen, Updaten und Löschen der Task-Konfigurationen gefordert. Tasks sollten zudem nicht nur bei Programmstart scheduled (aktiviert) oder descheduled (deaktiviert) werden können. (Siehe Kapitel 3.1.9.) Demzufolge stellen die Endpoints Funktionalitäten für das Erstellen/Updaten (Abbildung 4.9), Löschen (Abbildung 4.10), Aktivieren (Abbildung 4.11), Deaktivieren (Abbildung 4.12), Enqueuing (Abbildung 4.13) und Statusabfrage (Abbildung 4.14) von Tasks zur Verfügung. Anders als im Konzept 4.1 vorgesehen, sind sie synchron implementiert worden, wobei dies bei Bedarf mit wenig Aufwand geändert werden könnte. Die API ist als ASP.NET-Web-API (siehe Kapitel 2.5) implementiert. Entsprechend wurden die Endpoints in einer Controller-Klasse implementiert, die mit der Datenbank und dem Hangfire-Framework über einen Service kommuniziert. Das Task-Scheduling bei Applikationsstart erfolgt über einen separat implementierten Hosted Service.

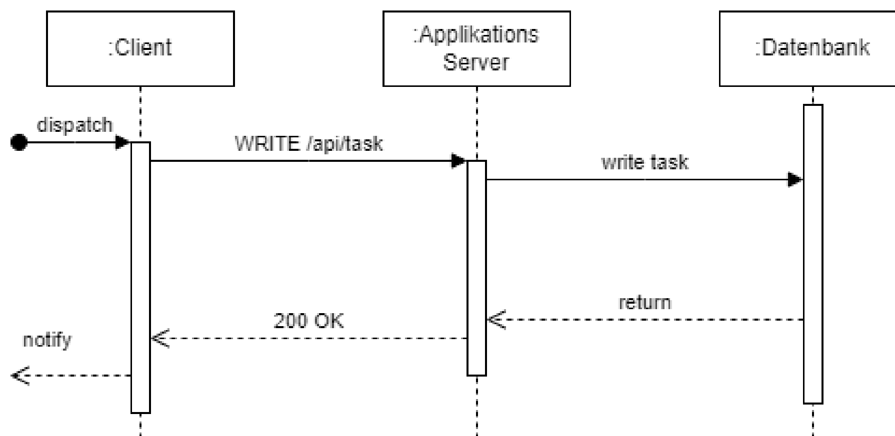

Abbildung 4.9.: Sequenzdiagramm für das Erstellen/Updaten von Tasks

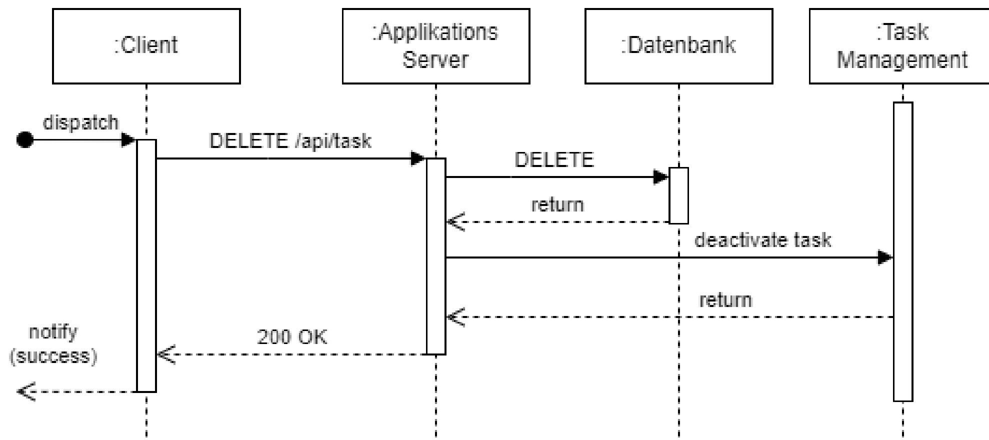

Abbildung 4.10.: Sequenzdiagramm für das Löschen von Tasks

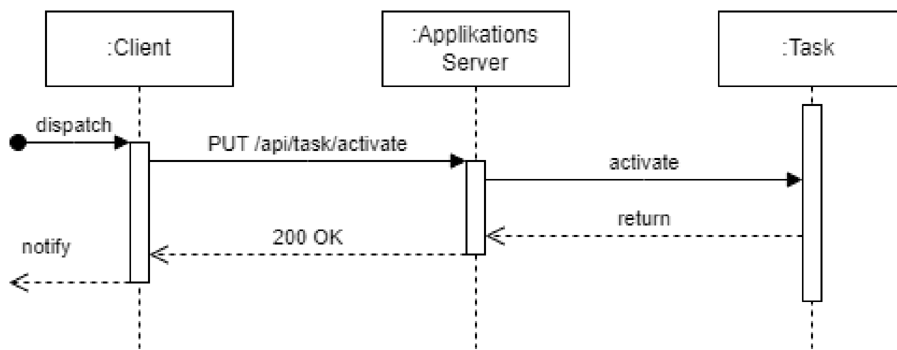

Abbildung 4.11.: Sequenzdiagramm für das Aktivieren eines Tasks

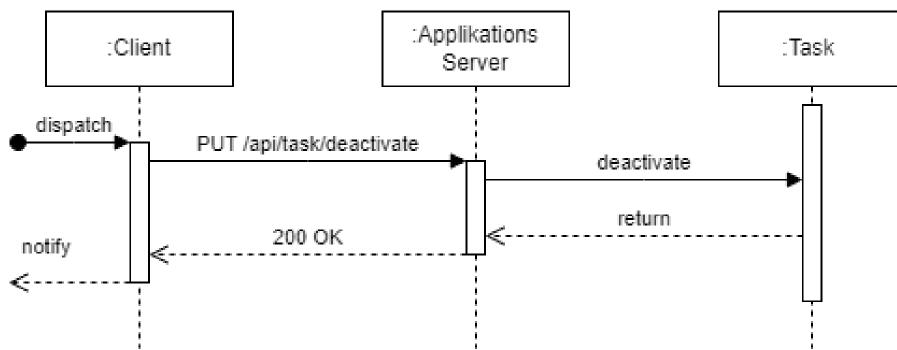

Abbildung 4.12.: Sequenzdiagramm für das Deaktivieren eines Tasks

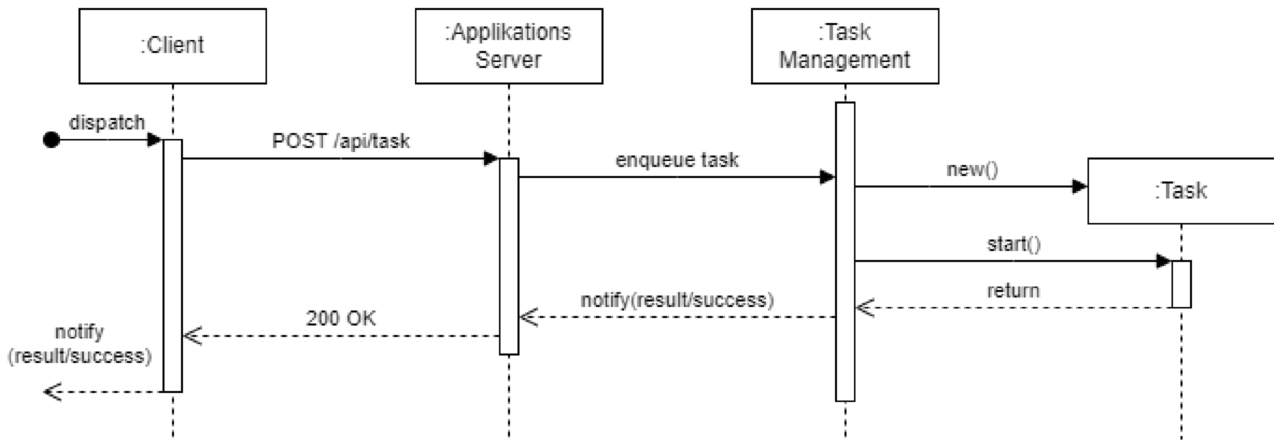

Abbildung 4.13.: Sequenzdiagramm für das Starten eines Tasks

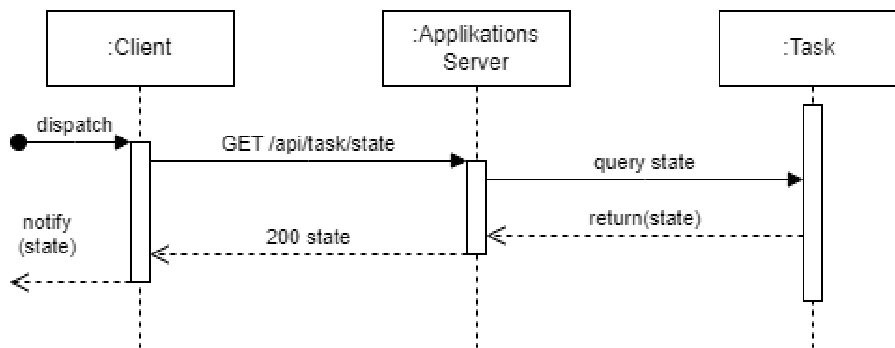

Abbildung 4.14.: Sequenzdiagramm für die Statusabfrage eines Tasks

Die Endpoints verlangen als Parameter die ID des Tasks. Während der Entwicklung wurde zuerst noch der Name verwendet, dies wurde jedoch verworfen, um zukünftige Erweiterungen zu ermöglichen. Soll ein Task bei Programmstart scheduled werden, muss das CRON-Feld einen nichtleeren String beinhalten und der Task muss aktiv sein (AKTIV-Feld hat Wert ungleich 0). Die Felder beziehen sich auf die in Abschnitt 4.2.1.1 dargestellte Datenbank-Tabelle. Ist der Task inaktiv (AKTIV-Feld hat Wert 0) und/oder ist sein CRON-String leer, kann er nicht scheduled werden. Wird ein Task aktiviert, so wird er automatisch scheduled, wenn der CRON-String nicht leer ist. Wird er deaktiviert, so wird er automatisch descheduled. Soll ein Task ausgeführt werden, so kann er enqueued werden. Dabei spielt es keine Rolle, ob dies ein Scheduled Task ist oder nicht; die Felder für CRON und AKTIV werden hierbei nicht beachtet. Wird ein Task neu erstellt oder upgedatet, so wird er automatisch deaktiviert. Für das Erhalten des Taskstatus wird in Hangfire.Job nach dem StateName über die TaskId gefragt. Um dies zu ermöglichen musste Hangfire.Job um eine Spalte TaskId sowie um einen Trigger erweitert werden. Dies wird in Abschnitt 4.2.4.1 und 4.2.1.6 genauer erläutert. Das Verhalten der Tasks bei Benutzung dieser Endpoints im laufenden System wird in Abschnitt 4.2.3.2 genauer erläutert.

Zwecks Verfikation der Funktion der einzelnen Endpoints wurde dem Applikationsserver ein Swagger-UI (siehe Kapitel 2.8) hinzugefügt. Damit konnten die Endpoints auf einfache Weise bei laufendem System getestet werden. Abbildung 4.15 zeigt das Swagger-UI für die Endpoints der Taskverwaltung.

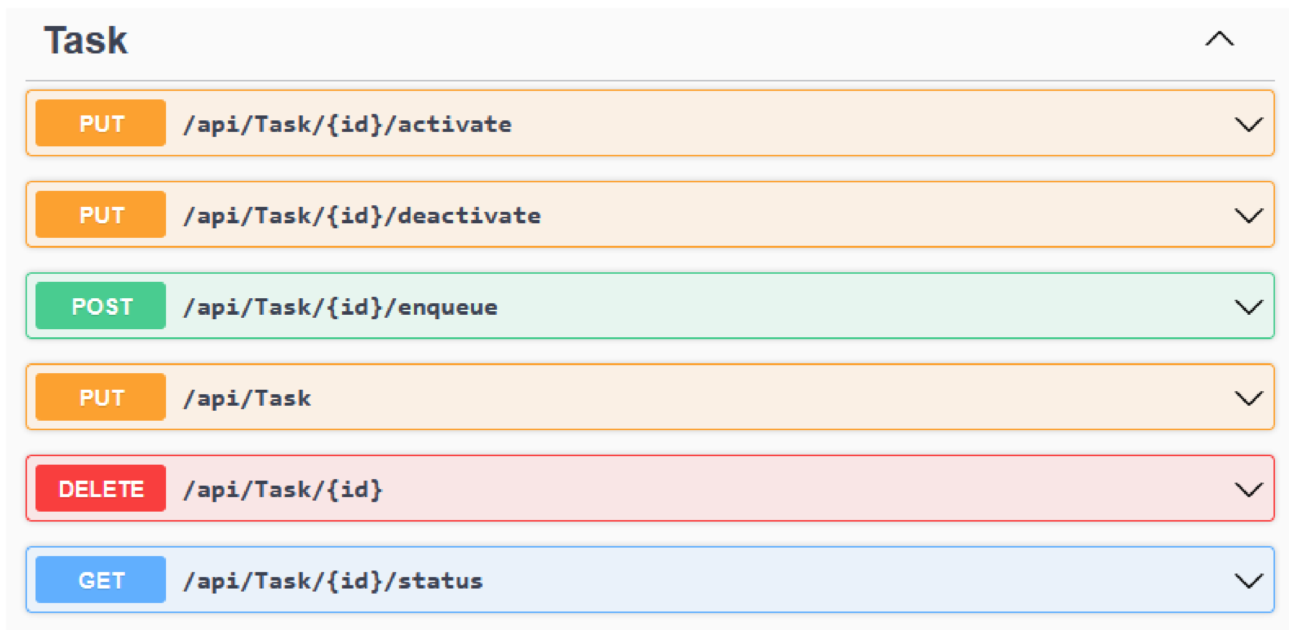

Abbildung 4.15.: Swagger-UI für die Taskverwaltungs-Endpoints

4.2.1.5. Integration von BORM-internen Python-Bibliotheken

Wie in Kapitel 3.1.11 beschrieben, wurde ein einfacher Testtask implementiert, der eine Funktion aus einer BORM-internen Python-Hilfsbibliothek aufruft. Die Bibliothek "bormapi.utils.string_utils" erwies sich hierfür als geeignet, da sie keine weiteren BORM-Python-Module importiert. Im Beispiel wird ein RTF-Dokument eingelesen und mit der Hilfsfunktion "rtf_to_plain_text" zu Plaintext konvertiert. Das Resultat wird anschliessend über Python.NET in C# ausgelesen und auf die Konsole geschrieben. Listing 4.5 zeigt den Python-Task. Die bereits in Abschnitt 4.2.1.3 erklärte Funktion "ExecutePythonFile" in der DLL QueueMethods musste um die notwendigen Pfadkonfigurationen ergänzt werden. Die komplette Funktion ist in Listing 4.6 zu sehen.

```python
from bormapi.utils import string_utils

file = "C:\\BORMGRUPPE\\BackgroundTasks\\TestTasks\\some_text.rtf"
with open(file) as rtf_file:
text = rtf_file.read()

result = string_utils.rtf_to_plain_text(text)
```

Listing 4.5: test.py

```
    private static void ExecutePythonFile(string taskFilePath)
    {
        string workspaceFolder =
            @"C:\BORMGRUPPE\BormSoftware\BormServer\Python";
        string runtimeFolder =
            @"C:\BORMGRUPPE\BormSoftware\Build\BormBusiness\BormServer\Python";

        Runtime.PythonDLL =
            $@"{runtimeFolder}\Python3111\python311.dll";
        PythonEngine.PythonPath =
            $@"{runtimeFolder}\Python3111\Lib;" +
            $@"{runtimeFolder}\Python3111\Lib\site-packages;" +
            $@"{workspaceFolder}\Root\custom;" +
            $@"{workspaceFolder}\Root\base;" +
            $@"{workspaceFolder}\Root;" +
            $@"{workspaceFolder}\flask;" +
            $@"{runtimeFolder}\..\binaries\Borm.API;";
        PythonEngine.PythonHome =
            $@"{runtimeFolder}\Python3111";
        Environment.SetEnvironmentVariable("PATH",
            $@"{runtimeFolder}\Python3111\Scripts;" +
            $@"{runtimeFolder}\Python3111;" +
            $@"{Environment.GetEnvironmentVariable("PATH")}");

        PythonEngine.Initialize();
        PythonEngine.BeginAllowThreads();

        using (Py.GIL())
        {
            using (PyModule scope = Py.CreateScope())
            {
                var code = System.IO.File.ReadAllText(taskFilePath);
                var scriptCompiled = PythonEngine.Compile(code, taskFilePath);
                scope.Execute(scriptCompiled);

                var result = scope.Get("result");
                System.Console.WriteLine($"Result: {result}");
            }
        }

        PythonEngine.Shutdown();
    }
```

Listing 4.6: ExecutePythonFile mit Konfiguration

4.2.1.6. Trigger after insert

Die von Hangfire erstellten Jobs werden in der Hangfire-Datenbank in die Tabelle Hangfire.Job geschrieben. Diese beinhaltet alle relevanten Informationen zur Ausführung eines Jobs wie die aufzurufende Methode, mitgegebene Argumente, Status etc. Für die Statusabfrage (siehe Abschnitt 4.2.1.4) musste daher eine Verknüpfung zwischen der SYS_TASK- und der Hangfire.Job-Tabelle hergestellt werden. Dazu wurde auf der Datenbank ein Trigger geschrieben. Dieser wird nach dem Einfügen eines Datensatzes in die HangfireDB, sprich beim Enqueuen eines Jobs, ausgelöst. Die TASK_ID, die, wie aus den Code-Listings in Abschnitt 4.2.1.2 ersichtlich, beim Enqueuing als Argument mitgegeben wird, wird aus den Argumenten mittels STRING_SPLIT herausgelesen und dann in der Spalte TaskId bei jedem NULL-Wert eingefügt. Da dies nach jedem Insert passiert, sollte nur ein NULL-Wert vorhanden sein. Der SQL-Trigger ist ein After-Insert-Trigger. Aus diesem Grund sollte in der definitiven Version ein Table-Lock für den Trigger gewährleistet sein, damit nicht zwischen Insert und Trigger eine Schreibtransaktion stattfinden kann und plötzlich mehrere NULL-Werte vorhanden sind und dann mit falschem Wert überschreiben werden.

4.2.2. Klassendiagramm

Im nachfolgenden Klassendiagramm 4.16 ist der aktuelle Stand der Implementation, wie in Abschnitt 4.2.1 beschrieben, dargestellt. Die Methoden wurden aus Platzgründen nur da erwähnt, wo sie für das Verständnis notwendig sind. Startup und Worker bestehen nur aus Konfiguration und/oder einer Main-Methode.

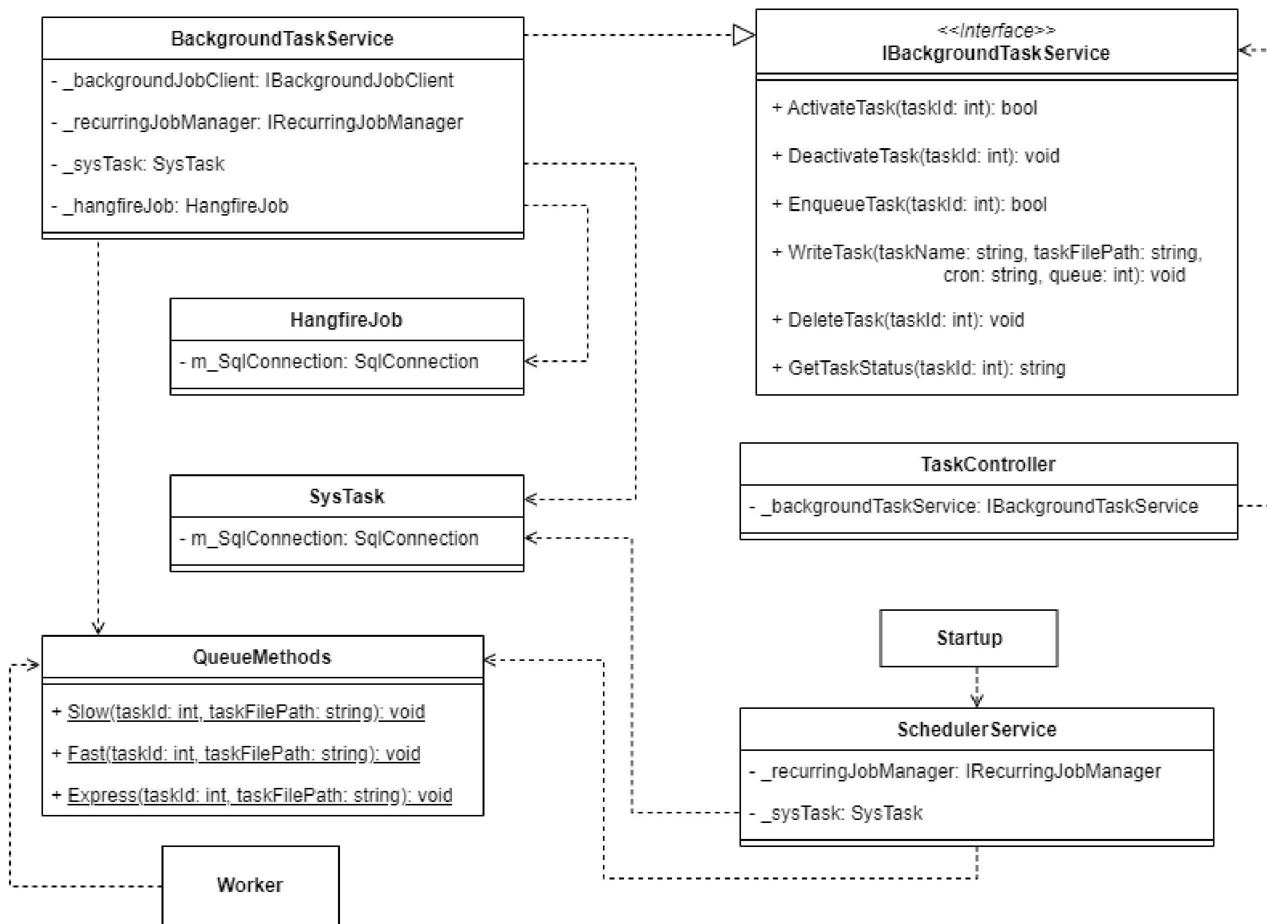

Abbildung 4.16.: Klassendiagramm des Prototyps

4.2.3. Tests

Die in Abschnitt 4.2.1 beschriebene Implementation wurde anschliessend, wie in Kapitel 3.1.10 beschrieben, mittels Unit- und Systemtests getestet. Die folgenden Abschnitte geben einen Überblick über die durchgeführten Tests und ihre Resultate.

4.2.3.1. Unit-Tests

Der Datenbankzugriff (via Borm.Core.DataAccess.SysTask) sowie die API-Controller-Klasse für die Tasks (TaskController) wurden mittels Unit-Tests getestet. Die folgende Tabelle beschreibt die implementierten Tests.

Klasse	Methode	Test
SysTask	Konstruktor	Wirft bei null als SqlConnection eine ArgumentNullException.
		Erstellt bei validen Eingaben den korrekten SysTask.
	GetAllSysTasks	Liefert alle SysTask.
		Erstellter SysTask ist im Resultat von GetAllSysTask enthalten.
	GetSysTask	Gibt bei invalidem oder falschem Input null zurück.
		Gibt den gewünschten SysTask zurück.
	WriteSysTask	Erstellt einen neuen SysTask
		Updatet vorhandenen SysTask korrekt.
		Wirft bei invalidem Namen eine Exception.
		Wirft bei invalidem Python Code Path eine Exception.
		Wirft bei invalider Queue eine Exception.
	DeactivateTask	Deaktivierter SysTask ist deaktiviert.
	ActivateTask	Aktivieren eines deaktivierten SysTasks setzt diesen aktiv.
	DeleteTask	Gelöschter Task ist nicht mehr vorhanden.
TaskController	ActivateTask	Aktivieren eines existierenden Tasks gibt "200 OK" zurück.
		Aktivieren eines nicht existierenden Tasks gibt "404 Not Found" zurück.
	EnqueueTask	Enqueuing eines existierenden Tasks gibt "200 OK" zurück.
		Enqueuing eines nicht existierenden Tasks gibt "404 Not Found" zurück.
	DeactivateTask	Deaktivieren eines Tasks gibt "200 OK" zurück.
	DeleteTask	Löschen eines Tasks gibt "204 No Content" zurück.
	WriteTask	Schreiben eines Tasks gibt "200 OK" zurück.
	GetTaskStatus	Abfragen des Zustands eines Tasks gibt "200 OK" und den Namen des Zustands zurück.

Tabelle 4.1.: Unit Tests für die Klassen SysTask.cs und TaskController.cs

4.2.3.2. Systemtests

Das Verhalten der Tasks unter verschiedenen möglichen Umständen wurde mittels Systemtests überprüft. Dazu wurden verschiedene Test-Tasks in Python implementiert. Anschliessend wurden die implementierten Operationen über die in Abschnitt 4.2.1.4 beschriebenen Endpoints unter den zu überprüfenden Konditionen ausgelöst. Im Folgenden werden die durchgeführten Tests mit ihren Resultaten sowie die gewonnenen Erkenntnisse beschrieben. Die Durchführung der ersten 6 Tests verlangte, dass dieselben Task mehrere Male hintereinander enqueued wurden. Um dies zu vereinfachen, wurde ein weiterer Endpoint zu Testzwecken implementiert, der denselben Task mehrere Male an die Queue übergab. Der Taskname sowie die Anzahl Wiederholungen konnten als Argumente übergeben werden.

10 Tasks (je 1 Min) auf dem gleichen Worker

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Alle werden sequenziell abgearbeitet	Wie erwartet

100 Tasks (je 6 Sek) auf dem gleichen Worker

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
fastTask_test.py	Alle werden sequenziell abgearbeitet	Wie erwartet

1000 Tasks (je 0.6 Sek) auf dem gleichen Worker

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
expressTask_test.py	Alle werden sequenziell abgearbeitet	Wie erwartet

Hinweis

Abbildung 4.17 zeigt die Job-Tabelle in der Hangfire-Datenbank während der Ausführung mehrerer Tasks mit einem Worker. Nur ein Job hat den Status "Processing".

8	117	474	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:46:53.933	2023-05-04 09:47:07.450
9	118	477	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:47:33.650	2023-05-04 09:49:17.653
10	119	489	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.243	2023-05-04 09:58:13.493
11	120	491	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.337	2023-05-04 09:59:13.630
12	121	493	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.340	2023-05-04 10:00:13.800
13	122	495	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.343	2023-05-04 10:01:13.920
14	123	496	Processing	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.347	NULL
15	124	483	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.350	NULL
16	125	484	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.350	NULL
17	126	485	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.353	NULL
18	127	486	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.353	NULL
19	128	487	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 09:57:01.353	NULL

Abbildung 4.17.: Auszug aus Hangfire.Job-Tabelle bei Ausführung mehrerer Tasks mit einem Worker

10 Tasks (je 1 Min) auf zwei Worker

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Alle werden gleichmässig abgearbeitet (etwa 50/50)	Wie erwartet

100 Tasks (je 6 Sek) auf zwei Worker

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
fastTask_test.py	Alle werden gleichmässig abgearbeitet (etwa 50/50)	Wie erwartet

1000 Tasks (je 0.6 Sek) auf zwei Worker

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
expressTask_test.py	Alle werden gleichmässig abgearbeitet (etwa 50/50)	Wie erwartet

Hinweis

Abbildung 4.18 zeigt die Job-Tabelle in der Hangfire-Datenbank während der Ausführung mehrerer Tasks mit zwei Worker. Zwei Jobs haben den Status "Processing".

1299	1412	5287	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.120	2023-05-04 13:06:14.237
1300	1413	5289	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.123	2023-05-04 13:06:14.643
1301	1414	5291	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.123	2023-05-04 13:06:14.997
1302	1415	5290	Processing	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.123	NULL
1303	1416	5292	Processing	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.127	NULL
1304	1417	4218	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.127	NULL
1305	1418	4219	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.130	NULL
1306	1419	4220	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.130	NULL
1307	1420	4221	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.133	NULL
1308	1421	4222	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.133	NULL
1309	1422	4223	Enqueued	{"t":"Borm.Background.Queues.QueueMethods, Borm....	["\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\TestT...	2023-05-03 13:05:36.137	NULL

Abbildung 4.18.: Auszug aus Hangfire.Job-Tabelle bei Ausführung mehrerer Tasks mit zwei Worker

Task wird während Ausführung gelöscht

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft fertig	Wie erwartet

Task wird während Ausführung deaktiviert

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft fertig und wird nicht erneut gescheduled	Wie erwartet

Non scheduledTask wird während Ausführung aktualisiert

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft mit alter Konfiguration fertig	Wie erwartet

Scheduled Task wird während Ausführung aktualisiert

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft mit alter Konfiguration fertig und wird deaktiviert	Wie erwartet

Task wird im Waiting-State gelöscht

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft fertig	Wie erwartet

Task wird im Waiting-State deaktiviert

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft fertig	Wie erwartet

Non scheduled Tasks wird im Waiting-State upgedatet

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft mit alter Konfiguration fertig	Wie erwartet

Scheduled Tasks wird im Waiting-State aktualisiert

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft mit alter Konfiguration fertig und wird deaktiviert	Wie erwartet

Worker wird gestoppt und neu gestartet -> was passiert mit scheduled Tasks?

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
fastTask_test.py	Jobs werden in die Queue geladen bis Worker zurück ist und dann sofort sequenziell von ihm abgearbeitet.	Wie erwartet

Bemerkung

Ist kein Worker vorhanden: Scheduling wird unterbrochen und wird bei einem Neustart wieder aufgenommen. Jobs werden nicht in die Queue geladen.
Annahme: Da scheduled Tasks zeitrelevant sein könnten und somit veraltet wären, wenn man sie später aufruft, werden diese ignoriert.

Worker wird gestoppt und neu gestartet -> was passiert mit laufenden Tasks?

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_writeFile_test.py	Task startet erneut von Anfang an	Wie erwartet

Worker wird gestoppt und neu gestartet -> was passiert mit waiting Tasks?

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_writeFile_test.py	Task verbleibt in der Queue	Wie erwartet

Task wird enqueued wenn kein Worker gestartet ist.

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_writeFile_test.py	Tasks landen in Queue	Wie erwartet

Bemerkung
Annahme: Da man hier selber überprüfen sollte, ob Worker vorhanden sind, reagiert dies anders als bei scheduled Tasks (siehe Bemerkung zu "Worker wird gestoppt und neu gestartet -> was passiert mit scheduled Tasks?"). Aus der State-Tabelle der Hangfire-Datenbank, in der die verschiedenen Zustände der Jobs detailliert gespeichert werden, ist ersichtlich, dass scheduled Tasks von manuell gestarteten Tasks unterschieden werden. Scheduled Tasks haben im Feld "Reason" beim Zustand "Enqueued" die Bemerkung, dass sie vom Scheduler enqueued wurden, während dieses Feld bei manuell gestarteten Tasks den Wert NULL hat (siehe Abbildung 4.19).

52	7757	2443	Enqueued	Triggered by recurring job scheduler	2023-06-09 13:10:01.030	{"EnqueuedAt":"2023-06-09T13:10:01.0298992Z","Qu...
53	7760	2443	Processing	NULL	2023-06-09 13:10:07.370	{"StartedAt":"2023-06-09T13:10:07.3692348Z","Serverl...
54	7761	2443	Succeeded	NULL	2023-06-09 13:10:13.537	{"SucceededAt":"2023-06-09T13:10:13.5359746Z","Pe...
55	7762	2444	Enqueued	NULL	2023-06-09 13:10:55.063	{"EnqueuedAt":"1686316255032","Queue":"fast"}
56	7763	2444	Processing	NULL	2023-06-09 13:10:58.593	{"StartedAt":"2023-06-09T13:10:58.5913689Z","Serverl...
57	7765	2444	Succeeded	NULL	2023-06-09 13:11:04.743	{"SucceededAt":"2023-06-09T13:11:04.7412135Z","Pe...
58	7764	2445	Enqueued	Triggered by recurring job scheduler	2023-06-09 13:11:01.087	{"EnqueuedAt":"2023-06-09T13:11:01.0859132Z","Qu...
59	7766	2445	Processing	NULL	2023-06-09 13:11:04.757	{"StartedAt":"2023-06-09T13:11:04.7558346Z","Serverl...
60	7767	2445	Succeeded	NULL	2023-06-09 13:11:10.903	{"SucceededAt":"2023-06-09T13:11:10.9011605Z","Pe...

Abbildung 4.19.: Auszug aus Hangfire.State-Tabelle bei Ausführung von scheduled Tasks

Applikationsserver wird gestoppt -> was passiert mit scheduled Tasks?

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
fastTask_test.py	Scheduling wird unterbrochen	Scheduling läuft weiter und Tasks werden abgearbeitet

Bemerkung
Gleich wie "Worker wird gestoppt und neu gestartet". Sobald die Tasks als Jobs in der HangfireDB erfasst sind, ist der Status des Applikationsservers irrelevant.

Applikationsserver wird gestoppt -> was passiert mit laufenden Tasks?

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_writeFile_test.py	Task läuft fertig	Wie erwartet

Bemerkung
Gleich wie "Worker wird gestoppt und neu gestartet". Sobald die Tasks als Jobs in der HangfireDB erfasst sind, ist der Status des Applikationsservers irrelevant.

Applikationsserver wird gestoppt -> was passiert mit waiting Tasks?

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_writeFile_test.py	Task verbleibt in der Queue	Wie erwartet

Bemerkung
Gleich wie "Worker wird gestoppt und neu gestartet". Sobald die Tasks als Jobs in der HangfireDB erfasst sind, ist der Status des Applikationsservers irrelevant.

Skript wird angepasst während Task in Queue wartet

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft mit neuer Konfiguration	Wie erwartet

Skript wird angepasst während Task läuft

Testskript	Erwartetes Ereignis	Eingetretenes Ereignis
slowTask_test.py	Task läuft mit alter Konfiguration fertig	Wie erwartet

Testskripts

Die verwendeten Testskripts sind nach der zugehörigen Queue benannt. Beim slowTask_test.py wurden zwei mögliche Resultate geschrieben, um das Ändern des Skript zu testen.

```python
from time import sleep
from datetime import datetime

lst = []
for i in range(0, 60):
    lst.append(i)
    sleep(1)

result = str(datetime.now())
# result = 100
```

Listing 4.7: slowTask_test.py

```python
from time import sleep
from datetime import datetime

for i in range(0, 60):
    print(i)
    sleep(1)

result = str(datetime.now())
```

Listing 4.8: slowTask_writeFile_test.py

```python
from time import sleep
from datetime import datetime

lst = []
for i in range(0, 60):
    lst.append(i)
    sleep(0.1)

result = str(datetime.now())
```

Listing 4.9: fastTask_test.py

```python
from time import sleep
from datetime import datetime

lst = []
for i in range(0, 60):
    lst.append(i)
    sleep(0.01)

result = str(datetime.now())
```

Listing 4.10: expressTask_test.py

4.2.4. Weitere konzeptionelle Überlegungen

Dieser Abschnitt befasst sich mit weiteren Funktionalitäten, die aus Zeit- und Komplexitätsgründen nicht im Prototyp umgesetzt, aber konzeptionell betrachtet wurden (siehe Kapitel 3.1.12). Konkret handelt es sich dabei um die Integration der Hangfire-Datenbank in die BORM-Datenbank, die Systemoperation "Task abbrechen", sowie eine korrekte Behandlung der Resultate von ausgeführten Tasks.

4.2.4.1. Integration der Hangfire-Datenbank in die BORM-Datenbank

In Zukunft wird die Hangfire-Datenbank in die BORM-Datenbank integriert. SysTask und Hangfire.Job werden wie im folgenden Diagramm 4.20 zu sehen miteinander verknüpft. Beim Enqueuen eines Jobs wird die TASK_ID, die als Argument mitgegeben wird, in die Spalte TaskId eingefügt. Diese wurde für den Prototyp bei Hangfire.Job eingefügt. Dies ermöglicht es, über die TaskId den Status des Jobs abzufragen, der jeweils von Hangfire automatisch aktualisiert wird. Sollte kein Job mit der TaskId bestehen, so wird einfach nichts zurückgegeben. Analog kann somit auch Löschen eines Jobs implementiert werden. Das Konzept hierfür ist im Abschnitt 4.2.4.2 zu finden. Der After-Insert-Trigger sollte dann, wie bereits in Abschnitt 4.2.1.6 vermerkt, um einen Table-Lock ergänzt werden, um sicherzustellen, dass nicht mehrere neu eingefügte Datensätze von demselben Trigger-Ereignis betroffen sind.

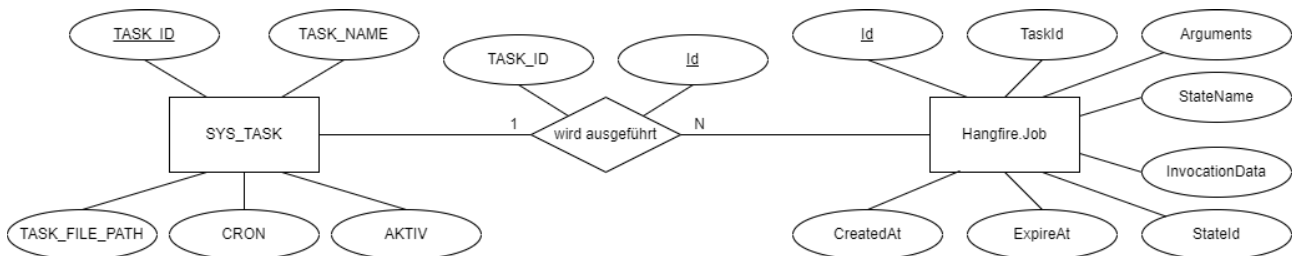

Abbildung 4.20.: Entity Relation Diagramm für SYS_TASK und zugehörige Jobs

4.2.4.2. Systemoperation Task abbrechen

Das Konzept sieht eine Möglichkeit vor, um laufende Tasks bei Bedarf abbrechen zu können (siehe Abschnitt 4.1.1). Wie in Abschnitt 4.1.2 gezeigt, wird durch die Architektur von Hangfire eine komplette Entkopplung von Enqueuing und Ausführung von Tasks gewährleistet. Dies erschwert allerdings den Abbruch laufender Tasks (in Hangfire als Job bezeichnet). Hangfire bietet zwar die Möglichkeit, den Jobs Cancellation-Tokens [70] mitzugeben. Ein Abbruch mit diesem Mechanimsus funktioniert aber nur, wenn innerhalb des Tasks regelmässig geprüft wird, ob auf dem mitgegebenen Token ein Abbruch signalisiert wurde. Dies kann jedoch im vorliegenden Fall nicht gewährleistet werden, da die Tasks in Python implementiert sind und die Abfrage demnach innerhalb des Skripts durchgeführt werden müsste. Somit besteht die einzige Möglichkeit, laufende Jobs abzubrechen, darin, den Worker, der den betreffenden Job ausführt, zu stoppen und neu zu starten. Vor dem Neustart muss ausserdem der betreffende Job-Eintrag aus der Hangfire-Datenbank gelöscht werden, da der Job sonst, wie in den Systemtests in Abschnitt 4.2.3.2 gezeigt, wieder ausgeführt wird. Die Identifikation des Jobs kann, dank der in 4.2.4.1 beschriebenen Verknüpfung der Datenbanktabellen, wie bei der Statusabfrage über die Task-Id durchgeführt werden.

Für den Löschzugriff auf die Hangfire-Datenbank muss eine entsprechende Methode in der Bibliothek Borm.Core.DataAccess implementiert werden. Für die Kontrolle der Worker-Prozesse sollte im Applikationsserver ein Service zur Prozessverwaltung implementiert werden. Dieser Service wird bei Programmstart aufgerufen und läuft im Hintergrund bis zum Programmende. Zu Beginn startet er die drei Worker-Prozesse für die drei Queues (slow, fast und express), die bisher manuell gestartet werden mussten (vergleiche Abschnitt 4.2.1.2). Ausserdem stellt er Methoden für das Beenden und Neustarten der Worker-Prozesse zur Verfügung. Der API-Controller für die Hintergrundtasks erhält einen weiteren Delete-Endpoint, um laufende Tasks abzubrechen. In diesem Endpoint werden dann die Methoden des beschriebenen Prozessmanagement-Services sowie des Datenbankzugriffs auf die Hangfire-Datenbank

aufgerufen. Der bestehende Endpoint zum Löschen von Task-Konfigurationen könnte dann bei Bedarf wie in Abbildung 4.21 gezeigt erweitert werden.

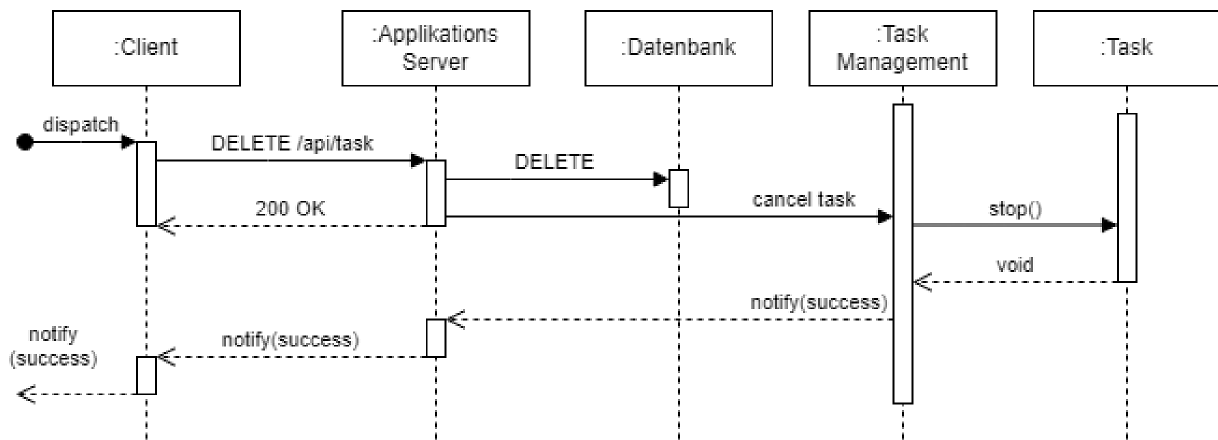

Abbildung 4.21.: Erweitertes Sequenzdiagramm für das Löschen von Tasks

4.2.4.3. Behandlung von Task-Resultaten

Die Aufgabenstellung beinhaltet die Betrachtung der Frage, wie Resultate von ausgeführten Tasks korrekt behandelt bzw. an den Endbenutzer zurückgemeldet werden sollen (siehe Kapitel 1.2). Wie in Kapitel 3.1.2 und 3.1.4 vermerkt, könnte dazu ein Message-Broker eingesetzt werden. Da Hangfire standardmässig keinen Message-Broker verwendet, wurde dieser Ansatz nach der Festlegung auf Hangfire nicht weiter verfolgt (siehe Kapitel 3.1.9). Wie in Tabelle 3.1 gezeigt, unterstützt RabbitMQ auch alle Technologien, die in der BORM-Architektur zum Einsatz kommen (vergleiche dazu auch Kapitel 1.1). In Anbetracht all dessen sollte diese Integration daher als separates Projekt weiter verfolgt werden, zumal die im Rahmen dieser Arbeit durchgeführten Tests mit RabbitMQ auf Windows (siehe Kapitel 3.1.5.2) vielversprechend sind und die Verwendung von RabbitMQ im Kontext der BORM-Software daher durchaus zu empfehlen ist.

Die folgenden Grafiken aus der RabbitMQ-Dokumentation erläutern das grundsätzliche Funktionsprinzip von RabbitMQ, wie es auch in der BORM-Software eingesetzt werden könnte. Abbildung 4.22 zeigt das einfachste Beispiel einer Nachricht, die von einer Software-Komponente (Producer) versendet und von einer anderen Software-Komponente (Consumer) empfangen wird. So könnten z.B. Task-Resultate vom Task selber an die ERP-Desktop-Applikation übermittelt werden. Der Task muss dazu einfach am Ende das Resultat über die RabbitMQ-Python-Bibliothek versenden. Falls weitere Softwarekomponenten RabbitMQ ebenfalls verwenden, könnte das in Abbildung 4.23 gezeigte Direct-Exchange Routing zum Einsatz kommen, womit Nachrichten anhand eines sogenannten Routing-Keys nur an bestimmte Empfänger übermittelt werden. [71]

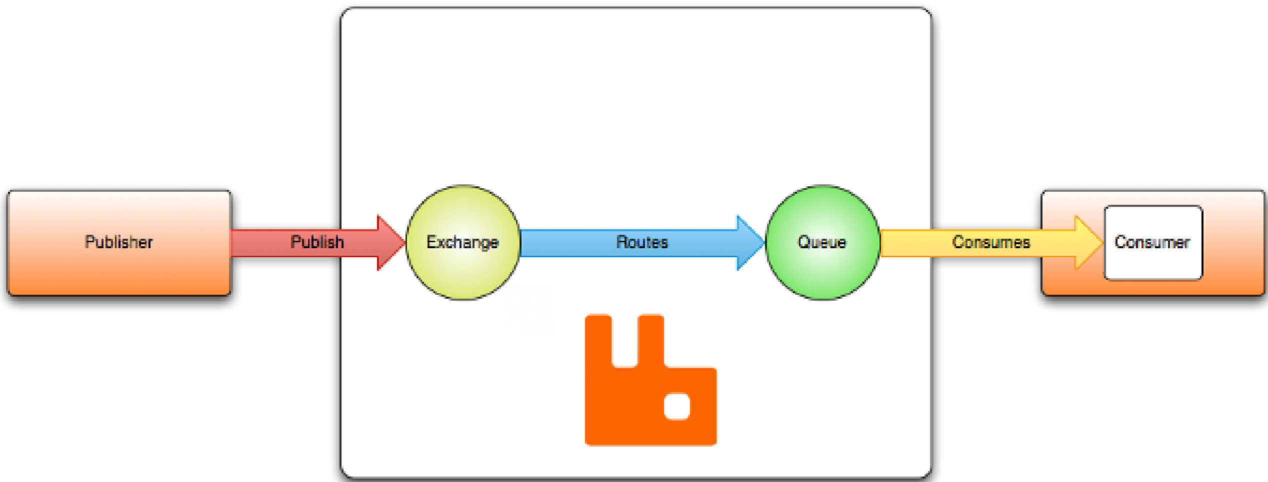

Abbildung 4.22.: Einfaches Beispiel einer Nachricht in RabbitMQ

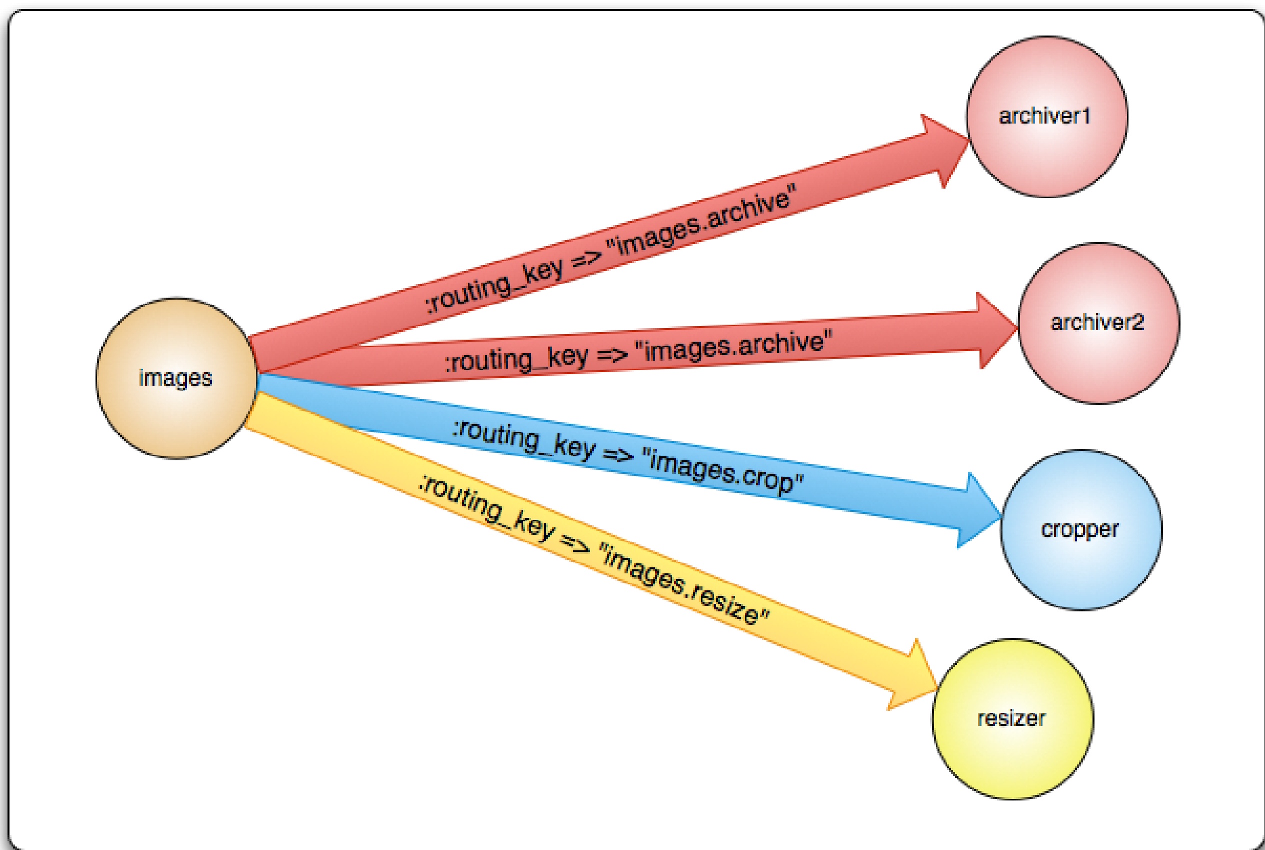

Abbildung 4.23.: Direct-Exchange Routing in RabbitMQ

Bis zur Integration von RabbitMQ können die Resultate aus Python direkt mittels des BORM-internen Moduls message_sender.py versendet werden. Dieses Modul stellt Funktionalitäten zur Verfügung, um Nachrichten an BORM-User zu versenden. Um Nachrichten zu empfangen, muss ein User im ERP-Desktop angemeldet sein. Die Nachricht wird nach Erhalt auf der Benutzeroberfläche der ERP-Software angezeigt.

4.2.4.4. Fehlerbehandlung in Tasks

Falls ein Task erfolgreich ausgeführt werden konnte, erhält er in der Hangfire.Job-Tabelle den Status "Succeeded". Falls während der Ausführung ein Fehler auftritt, führt Hangfire standardmässig bis zu 10 Versuchen in stets wachsenden Zeitabständen aus, um den Task nochmals auszuführen (Abbildung 4.24). Der erste Versuch wird nach ca. einer halben Minute durchgeführt, der letzte ca. zwei Stunden nach dem zweitletzten. Der Task erhält während dieser Zeit den Status "Scheduled" (Abbildung 4.25). Dieser Status tritt ausschliesslich nur bei fehlgeschlagenen Tasks auf. Bei normalen scheduled Tasks wird der entsprechende Job erst beim geplanten Zeitpunkt erzeugt und erhält den Status "Enqueued". Sollte der Task bei einem der Wiederholungsversuche erfolgreich ausgeführt werden können, erhält er ebenfalls den Status "Succeeded". Schlagen hingegen alle Versuche fehl, erhält er den Status "Failed" (Abbildung 4.26). Im Hinblick auf die Integration in den BormServer stellt sich hier die Frage, ob diese Versuche überhaupt durchgeführt werden sollen, oder ob ein fehlgeschlagener Task einfach direkt als "Failed" eingetragen werden soll. Hangfire bietet die Möglichkeit, die maximale Anzahl Wiederholungsversuche für die aufgerufene Task-Methode zu konfigurieren. Dies wäre eine gute Möglichkeit für allfällige Anpassungen. [72]

	Id	JobId	Name	Reason	CreatedAt	Data
1	7708	2441	Enqueued	NULL	2023-06-09 09:08:58.310	{"EnqueuedAt":"1686301738202","Queue":"fast"}
2	7709	2441	Processing	NULL	2023-06-09 09:09:13.337	{"StartedAt":"2023-06-09T09:09:13.1278401Z","Serverl...
3	7710	2441	Failed	An exception occurred during performance of the ...	2023-06-09 09:09:14.647	{"FailedAt":"2023-06-09T09:09:14.6326769Z","Excepti...
4	7711	2441	Scheduled	Retry attempt 1 of 10: Just to fail.	2023-06-09 09:09:14.830	{"EnqueueAt":"2023-06-09T09:09:37.6457437Z","Sche...
5	7712	2441	Enqueued	Triggered by DelayedJobScheduler	2023-06-09 09:09:43.137	{"EnqueuedAt":"2023-06-09T09:09:43.1347230Z","Qu...
6	7713	2441	Processing	NULL	2023-06-09 09:09:43.150	{"StartedAt":"2023-06-09T09:09:43.1492833Z","Serverl...
7	7714	2441	Failed	An exception occurred during performance of the ...	2023-06-09 09:09:43.160	{"FailedAt":"2023-06-09T09:09:43.1565048Z","Excepti...
8	7715	2441	Scheduled	Retry attempt 2 of 10: This property must be set b...	2023-06-09 09:09:43.160	{"EnqueueAt":"2023-06-09T09:10:41.1598656Z","Sche...
9	7716	2441	Enqueued	Triggered by DelayedJobScheduler	2023-06-09 09:10:43.193	{"EnqueuedAt":"2023-06-09T09:10:43.1898051Z","Qu...
10	7717	2441	Processing	NULL	2023-06-09 09:10:43.207	{"StartedAt":"2023-06-09T09:10:43.2056092Z","Serverl...
11	7718	2441	Failed	An exception occurred during performance of the ...	2023-06-09 09:10:43.213	{"FailedAt":"2023-06-09T09:10:43.2122372Z","Excepti...
12	7719	2441	Scheduled	Retry attempt 3 of 10: This property must be set b...	2023-06-09 09:10:43.217	{"EnqueueAt":"2023-06-09T09:12:41.2146245Z","Sche...

Abbildung 4.24.: Auszug aus Hangfire.State-Tabelle mit wiederholt ausgeführtem Task

	Id	StateId	StateName	InvocationData	Arguments	CreatedAt	ExpireAt	TaskId
1	2441	7747	Scheduled	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["53","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 09:08:58.237	NULL	53
2	2442	7759	Succeeded	{"Type":"Borm.Background.Queues.QueueMethods, Bor...	["111","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 13:09:06.757	2023-06-10 13:10:07.363	111
3	2443	7761	Succeeded	{"Type":"Borm.Background.Queues.QueueMethods, Bor...	["111","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 13:10:01.020	2023-06-10 13:10:13.537	111
4	2444	7765	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["111","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 13:10:55.040	2023-06-10 13:11:04.743	111
5	2445	7767	Succeeded	{"Type":"Borm.Background.Queues.QueueMethods, Bor...	["111","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 13:11:01.073	2023-06-10 13:11:10.903	111

Abbildung 4.25.: Auszug aus Hangfire.Job-Tabelle mit einem Task im Zustand "Scheduled"

	Id	StateId	StateName	InvocationData	Arguments	CreatedAt	ExpireAt	TaskId
1	2441	7854	Failed	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["53","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 09:08:58.237	NULL	53
2	2442	7759	Succeeded	{"Type":"Borm.Background.Queues.QueueMethods, Bor...	["111","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 13:09:06.757	2023-06-10 13:10:07.363	111
3	2443	7761	Succeeded	{"Type":"Borm.Background.Queues.QueueMethods, Bor...	["111","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 13:10:01.020	2023-06-10 13:10:13.537	111
4	2444	7765	Succeeded	{"t":"Borm.Background.Queues.QueueMethods, Borm.B...	["111","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 13:10:55.040	2023-06-10 13:11:04.743	111
5	2445	7767	Succeeded	{"Type":"Borm.Background.Queues.QueueMethods, Bor...	["111","\"C:\\\\BORMGRUPPE\\\\BackgroundTasks\\\\...	2023-06-09 13:11:01.073	2023-06-10 13:11:10.903	111

Abbildung 4.26.: Auszug aus Hangfire.Job-Tabelle mit einem Task im Zustand "Failed"

5. Diskussion und Ausblick

In diesem Kapitel werden die Vorgehensweisen und Methoden diskutiert, welche in Kapitel 3 beschrieben wurden. Des weiteren werden die in Kapitel 1.2 gesetzten Ziele anhand der erreichten und in Kapitel 4 beschriebenen Resultate reflektiert. Im Anschluss daran wird ein möglicher Ausblick aufgezeigt.

5.1. Diskussion

In diesem Kapitel werden die Vorgehensweise und die angewandten Methoden diskutiert.

5.1.1. Analyse und Evaluation eines geeigneten Frameworks

Die Analyse der Aufgabenstellung gestaltete sich dank den ausführlichen Erläuterungen unserer Ansprechpersonen bei der BORM-INFORMATIK AG recht einfach. Danach war eine gründliche Recherche notwendig, um eine möglichst breite Auswahl von möglichen Lösungsansätzen zu haben. Bei der anschliessenden Vorselektion fokussierten wir uns der Aufgabenstellung entsprechend auf die Taskverwaltungs-Frameworks. Die Message-Broker wurden dabei nur insofern berücksichtigt, als sie von den zur Auswahl stehenden Taskverwaltungs-Frameworks unterstützt wurden. Die Tests, die mit den einzelnen Frameworks durchgeführt wurden, waren äusserst hilfreich bei der Beurteilung derselben. Allerdings waren wir zuweilen etwas vorschnell, ein Framework für die Verwendung auszuschliessen oder zu empfehlen. So fokussierten wir uns besonders zu Anfang der Tests zu sehr auf Python-Frameworks (insbesondere Celery) und vernachlässigten die .NET-Frameworks. Dies machte den Prozess der Entscheidungsfindung z.T. unnötig kompliziert. Dennoch führten die durchgeführten Tests zu einer guten Endauswahl von möglichen Frameworks, aus denen wir in Absprache mit der BORM-INFORMATIK AG das am geeignetsten scheinende auswählen konnten.

5.1.2. Erstellen des Konzepts

Die Definition der grundlegenden Systemoperationen (siehe Kapitel 4.1.1) wurde vor der finalen Auswahl des zu verwendenden Frameworks durchgeführt, da die Systemoperationen grundsätzlich unabhängig von der Architektur sind. Dies war für die Entscheidungsfindung (siehe Kapitel 3.1.8) von Bedeutung, da insbesondere die Systemoperationen Abfragen eines Task-Zustandes und Abbrechen eines Tasks nicht bei allen Frameworks gleich gut umsetzbar sind. Bei Celery etwa erwies sich das Abbrechen, wie in Kapitel 3.1.8 beschrieben, aufgrund der mangelhaften Windows-Unterstützung als nahezu unmöglich. Die Systemoperationen hätten allerdings gerade so gut vor der Recherche definiert werden können. Dies wäre u.U. auch für die Tests der vorselektierten Frameworks von Nutzen gewesen.

Der Entwurf der Architektur und der Verteilung der Softwarekomponenten konnte im Gegensatz zu den Systemoperationen natürlich erst nach Auswahl des zu verwendenden Frameworks durchgeführt werden. Die grundsätzliche Integration von Hangfire in eine ASP.NET-Applikation war uns aus den durchgeführten Tests bereits bekannt. Durch dieses Vorwissen und die Verwendung der gut verständlichen Dokumentation von Hangfire erwies sich die Erstellung des Konzepts für die Architektur und die Verteilung als unkompliziert. Die erstellten Diagramme (siehe Kapitel 4.1.2 und 4.1.3) geben einen Überblick über die geplante Integration und erwiesen sich bei der Implementation des Prototyps als hilfreiche Übersicht.

5.1.3. Integration von Hangfire in die BormServer-Applikation

Wie bereits in 5.1.2 erwähnt, verfügten wir von den während der Auswahl des Frameworks durchgeführten Tests her über ein grundsätzliches Know-how zur Verwendung von Hangfire in einem ASP.NET-Projekt. Dies erleichterte die Integration in das bestehende BormServer-Projekt. Dennoch arbeiteten

wir meist gemeinsam an der Implementation des Prototyps, einerseits um Fehler zu vermeiden, anderseits aufgrund von z.T. mangelhaften Detailkenntnissen der BORM-Softwarearchitektur. Unter diesen Voraussetzungen konnte die Implementation des Prototyps effizient und ohne grosse Zeitverluste (z.B. aufgrund längeren Fehlersuchens) durchgeführt werden.

Die Auswahl von Hangfire für die Verwendung auf dem BormServer erwies sich grundsätzlich als richtig. Durch die insgesamt recht übersichtliche und ausführliche Dokumentation und die flüssige Integration des Frameworks vermittelt es den Eindruck einer professionell entwickelten Software. Dass es noch stets aktiv weiterentwickelt wird, wurde auch dadurch deutlich, dass während der Implementation des Prototyps ein Versionsupgrade erfolgte (Version 1.7.34 zu 1.8.0). Da wir die Implementation noch auf Version 1.7.34 begonnen hatten, führten wir sie aber auch so zu Ende. Ein Nachteil war, dass die in Python geschriebenen Tasks nun mittels eines zusätzlichen Frameworks ausgeführt werden mussten. Zudem erschwerte die Tatsache, dass Hangfire die Job-Daten völlig selbstständig verwaltet und, soweit erkenntlich, keine direkte Beziehung der Hangfire-Tabellen mit anderen Tabellen erlaubt, die Implementation der Statusabfrage sowie die Lösungsfindung für den Abbruch eines Tasks. Mittels des in Kapitel 4.2.1.6 beschriebenen Triggers wurde dies zwar ermöglicht, jedoch ist diese Lösung wohl nicht als sehr elegant zu bezeichnen.

5.1.4. Verwendung von Python.NET zur Ausführung von Python-Skripts

Die Verwendung von Python.NET zur Ausführung von Python-Modulen in C# war sehr naheliegend, da es bereits bei BORM-INFORMATIK AG zur Integration von C#-DLLs in Python eingesetzt wird und auch früher bereits zum Aufruf von Python-Modulen aus einer .NET-Umgebung verwendet worden war. Auch diese Auswahl erwies sich als richtig. Zwar erwies sich die Dokumentation, die im Wesentlichen aus einem GitHub-Wiki besteht, als nicht sonderlich übersichtlich und vollständig, genügte jedoch, um eine einfach Integration in den Prototyp umzusetzen.

Die in Kapitel 3.1.11 beschriebenen Probleme, die im Zusammenhang mit der Integration von BORM-internen Python-Bibliotheken auftraten, stellen zwar die kombinierte Verwendung von Hangfire und Python.NET in Frage. Der zusätzliche "Technologiesprung", der sich durch das Aufrufen der Python-Tasks über Python.NET ergibt, ist ohne Zweifel ein Nachteil dieser Implementation. Doch konnte durch das in Kapitel 4.2.1.5 beschriebene einfache Beispiel gezeigt werden, dass die grundsätzliche Funktionalität gewährleistet ist. Die aufgetretenen Probleme sollten sich zudem, wie in Kapitel 3.1.11 beschrieben, durch Behebung der Versionskonflikte lösen lassen. Was die Taskverwaltung betrifft, so müssten danach möglicherweise noch die in Kapitel 4.2.1.5 gezeigten Konfigurationen für Python.NET angepasst werden.

5.1.5. Implementation und Durchführung der Tests

Die Unit-Tests für den Datenbankzugriff über die SysTask-Klasse sowie für den API-Controller der Taskverwaltung wurden zeitnah zur Entwicklung der entsprechenden Klassen implementiert und erwiesen sich als hilfreich für die Überprüfung und das Refactoring der implementierten Funktionalitäten. Die Systemtests waren äusserst wichtig, um zu überprüfen, wie sich Tasks und Worker unter verschiedenen Umständen verhalten. Gesamthaft konnte mit den Unit- und Systemtests die Funktionalität des Prototyps erfolgreich verifiziert werden. Das Weglassen von Integration-Tests erwies sich in keiner Weise von Nachteil.

5.1.6. Beschränkung der implementierten Funktionalität

Wie in Kapitel 4.2.4 beschrieben, konnten einige Funktionalitäten aus Zeit- und Komplexitätsgründen nicht mehr implementiert werden. Es handelt sich dabei jedoch nicht um Funktionen, die für die Funktion der Taskverwaltung von erstrangiger Bedeutung sind. Die Tatsache, dass die Implementation dieser Funktionen zu weiteren, eher komplexen Änderungen geführt hätte (Integration der Hangfire-Datenbank in die BORM-Datenbank, Prozessverwaltung für Worker, Resultatbehandlung mittels Message-Broker), sprach dafür, keine weiteren Änderungen am funktionierenden Prototyp vorzunehmen. Gemäss Aufgabenstellung sollte der Prototyp die geforderten Funktionalitäten aufzeigen oder beinhalten. Mit den in Kapitel 4.2.4 beschriebenen Überlegungen werden die noch fehlenden

Funktionalitäten sinnvoll aufgezeigt. Der Prototyp bildet eine gute Basis, um sie später noch hinzuzufügen.

5.1.7. Anwendung der reduzierten Version von SCRUM

Die in Kapitel 3.2 beschriebene Arbeitsmethodik hat sich während des gesamten Projektverlaufs sehr gut bewährt. Die Unterteilung in zweiwöchige Entwicklungsphasen erlaubte eine einfache Terminplanung mit unseren Ansprechpersonen bei der BORM-INFORMATIK AG und bildete ein gutes Grundraster für die Arbeitsplanung. Für eine Arbeit in diesem Rahmen, die von zwei Personen durchgeführt wird, ist diese Art von Vorgehen durchaus empfehlenswert.

5.1.8. Zeitplanung

Die in Kapitel 3.5 beschriebene Zeitplanung konnte gut eingehalten werden. Weil die Sitzungen bei der BORM-INFORMATIK AG in Steinhausen stundenplanbedingt jeweils am Dienstag oder Mittwoch stattfanden, wurden diese Tage als Sprintende des vergangenen bzw. Sprintstart des folgenden Sprints definiert. Die Meilensteine sollten zwar jeweils an einem Freitag abgeschlossen werden. Bedingt durch die eben beschriebene Sprintplanung verlegten sich diese Daten dann allerdings auf den folgenden Dienstag oder Mittwoch. Dieser Unterschied war jedoch für die Durchführung der Arbeit nicht von Bedeutung. Die gesteckten Ziele konnten mit der festgesetzten Zeitplanung gut erreicht werden.

5.2. Reflexion der Ziele und Resultate

Die Ziele und die erreichten Resultate sind in etwa deckungsgleich. Es wurde ein Konzept erarbeitet wie in Kapitel 4.1 und den Unterkapiteln beschrieben. Es wurden Systemoperationen definiert 4.1.1, eine Architekturübersicht erstellt 4.1.2 und eine Verteilung der Softwaremodule entworfen. Mit dem ausgewählten Framework Hangfire können sowohl Hintergrundprozesse wie auch zeitlich gesteuerte Prozesse ausgeführt werden. Anschliessend wurde auf dieser Grundlage der Prototyp erstellt. Dieser beinhaltet folgende Funktionalitäten:
Tasks können ...

- ... erstellt werden.
- ... ausgeführt werden.
- ... geplant und zur geplanten Zeit ausgeführt werden.
- ... bearbeitet oder gelöscht werden.
- ... im Fall von geplanten Task auf aktiv oder inaktiv geschaltet werden.
- ... über ihren Status abgefragt werden.

Das Abbrechen von laufenden Tasks und die Rückmeldung des Resultats an den Benutzer konnten noch nicht implementiert werden. Diese beiden Funktionen sind jedoch wie schon in 5.1.6 erwähnt für den Prototyp nicht zwingend notwendig. Es wurde aufgezeigt, dass beide Funktionalitäten eingebaut werden können und ein Konzept dazu wurde entworfen. Hingegen war es am Schluss noch möglich, die Statusabfrage experimentell zu implementieren und nicht nur, wie in Kapitel 3.1.9 beschrieben, konzeptionell aufzuzeigen.

5.3. Ausblick

Dieser Abschnitt beschreibt mögliche Erweiterungen und Verbesserungen zum momentanen Stand.

5.3.1. Abbrechen von Tasks

Für das Abbrechen der Tasks muss noch das Abschalten und Neustarten eines Workers implementiert werden. Vor dem Neustart muss dann der Jobeintrag aus der Hangfire.Job-Liste gelöscht werden. Dies kann über die zuvor hinterlegte TaskId erfolgen. Das Starten und Beenden eines Workers wird momentan noch über die Kommandozeile ausgelöst.

5.3.2. Create- und Update-Funktionalität für Task-Konfigurationen separieren

Für das Updaten der Task-Konfigurationen wird momentan der gleiche Endpoint verwendet wie für das Erstellen. Existiert bereits ein Task mit dem gleichen Namen, wird dieser aktualisiert, statt neu erstellt. Die ID wird nicht berücksichtigt, da diese beim Erstellen noch nicht vorhanden ist. Dies kann zum ungewollten Überschreiben eines existierenden Tasks führen, falls der gleiche Name nochmals gewählt wird. Um dies zu verhindern, sollte die Funktionalität für das Update separat implementiert werden, wobei die Identifikation über die ID und nicht über den Namen erfolgen soll. Im Fall, dass kein Task mit der mitgegebenen ID existiert, sollte der Endpoint ”404 Not Found” zurückgeben.

5.3.3. Integration der Hangfire-Datenbank in die BORM-Datenbank

Für die Integration der Hangfire-Datenbank in die BORM-Datenbank müssen noch die entsprechenden Tabellen integriert werden. Dies sollte ohne eine Veränderung an der Hangfire-Datenstruktur erfolgen, da dies sonst bei einem Update zu Problemen und Wartungsarbeiten führen könnte. Ausserdem sollte auch der in Kapitel 4.2.4.1 beschriebene Table-Lock für den Trigger ergänzt werden.

5.3.4. Resultatbehandlung über Message Broker

Ein Message Broker wurde noch nicht implementiert, da dies für die Ausführung der Background Tasks nicht zwingend notwendig ist. Es würde aber ermöglichen, das Resultat an eine Schnittstelle bzw. an ein von BORM-INFORMATIK AG gewünschtes Frontend weiterzugeben. Dies würde auch das gezielte Auswählen von Tasks und Endpoints benutzerfreundlicher gestalten.

5.3.5. Ausstehende Problembehandlungen

Zu den Ausstehenden Problemen gehört unter anderem, der Wechsel von 32-Bit- zu 64-Bit-Architektur und dem einhergehenden Wechsel der zugehörigen .dll Dateien. Dies ist notwendig, um die momentane Implementation der Taskverwaltung in die BORM-Software einbinden zu können. Dies sollte viele Probleme, wie zum Beispiel das Problem mit dem Ausführen der Lagerbewertung wie in 3.1.11 beschrieben, lösen.

5.3.6. Integration in Programmverwaltung

Um verschiedene Versionen der BORM-Software für Entwickler und Projektleiter lokal verfügbar zu haben, wird BORM-intern eine sogenannte Programmverwaltung verwendet. Um die Taskverwaltung in die Programmverwaltung zu integrieren, benötigt es noch weitere Recherche. Es muss noch überprüft werden wie dies am besten gehandhabt wird, damit die Taskverwaltung im Development-Mode gestartet werden kann. Damit auch bei Kunden die Software modular eingerichtet werden kann, benötigt es ebenfalls Anpassungen. Eine Möglichkeit hierfür wäre Docker, was BORM-INFORMATIK AG in Betracht zieht, vermehrt zu nutzen.

5.3.7. Fehlerbehandlung

Wie schon in Kapitel 4.2.4.4 erwähnt, werden bei einem fehlgeschlagenen Task standardmässig mehrere Wiederholungsversuche durchgeführt, bis er den Status ”Failed” erhält. Für die Integration der Taskverwaltung auf dem BormServer bleibt noch abzuklären, ob ein fehlgeschlagener Task wiederholt werden soll und, falls ja, wie oft. Über die ebenfalls in Kapitel 4.2.4.4 erwähnte Konfigurationsmöglichkeit könnte dies eingestellt werden.

6. Verzeichnisse

6.1. Literaturverzeichnis

[1] „Evolution of Borm Architecture“. [Stand: 28.02.2023]. (2023), Adresse: https://bormgruppe.atlassian.net/wiki/spaces/ERP/pages/1901306/Evolution+of+Borm+Architecture.

[2] „Git“. [Stand: 07.03.2023. (2023), Adresse: https://git-scm.com.

[3] „GitHub“. [Stand: 07.03.2023]. (2023), Adresse: https://github.com/.

[4] „GitLab“. [Stand: 07.03.2023. (2023), Adresse: https://about.gitlab.com/de-de/.

[5] „Flask“. [Stand: 05.04.2023]. (2023), Adresse: https://palletsprojects.com/p/flask.

[6] „What is ASP.NET?“ [Stand: 30.05.2023]. (2023), Adresse: https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet.

[7] „Create web APIs with ASP.NET Core“. [Stand: 11.05.2023]. (2023), Adresse: https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-7.0.

[8] „Dependency injection in ASP.NET Core“. [Stand: 11.05.2023]. (2023), Adresse: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0.

[9] „Background tasks with hosted services in ASP.NET Core“. [Stand: 11.05.2023]. (2023), Adresse: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-7.0&tabs=visual-studio.

[10] „ADO.NET“. [Stand: 30.05.2023]. (2023), Adresse: https://learn.microsoft.com/en-us/dotnet/framework/data/adonet.

[11] „Python.NET“. [Stand: 28.03.2023]. (2023), Adresse: https://pythonnet.github.io.

[12] „Swagger“. [Stand: 03.05.2023]. (2023), Adresse: https://swagger.io/.

[13] „About Swagger“. [Stand: 03.05.2023]. (2023), Adresse: https://swagger.io/about.

[14] „Swagger UI“. [Stand: 03.05.2023]. (2023), Adresse: https://swagger.io/tools/swagger-ui.

[15] „Swashbuckle.AspNetCore.Swagger“. [Stand: 03.05.2023]. (2023), Adresse: https://www.nuget.org/packages/Swashbuckle.AspNetCore.Swagger.

[16] „Redis“. [Stand: 14.03.2023]. (2023), Adresse: https://redis.io.

[17] „RabbitMQ“. [Stand: 14.03.2023]. (2023), Adresse: https://www.rabbitmq.com.

[18] „Amazon SQS“. [Stand: 14.03.2023]. (2023), Adresse: https://aws.amazon.com/sqs.

[19] „Azure Service Bus“. [Stand: 14.03.2023]. (2023), Adresse: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview.

[20] „Apache Kafka“. [Stand: 14.03.2023]. (2023), Adresse: https://kafka.apache.org.

[21] „Apache ActiveMQ“. [Stand: 14.03.2023]. (2023), Adresse: https://activemq.apache.org.

[22] „Apache QPid“. [Stand: 14.03.2023]. (2023), Adresse: https://qpid.apache.org.

[23] „Apache RocketMQ“. [Stand: 14.03.2023]. (2023), Adresse: https://rocketmq.apache.org.

[24] „Apache Zookeeper“. [Stand: 14.03.2023]. (2023), Adresse: https://zookeeper.apache.org.

[25] „Google Cloud PubSub“. [Stand: 14.03.2023]. (2023), Adresse: https://cloud.google.com/pubsub/docs.

[26] „IronMQ“. [Stand: 14.03.2023]. (2023), Adresse: https://www.iron.io/mq.

[27] „Beanstalk“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/beanstalkd/beanstalkd.

[28] „Nats“. [Stand: 14.03.2023]. (2023), Adresse: https://nats.io.

[29] „NSQ“. [Stand: 14.03.2023]. (2023), Adresse: https://nsq.io.

[30] „ZeroMQ“. [Stand: 18.03.2023]. (2023), Adresse: https://zeromq.org.

[31] „Django background tasks“. [Stand: 14.03.2023]. (2023), Adresse: https://django-background-tasks.readthedocs.io/en/latest.

[32] „Celery“. [Stand: 14.03.2023]. (2023), Adresse: https://docs.celeryq.dev/en/stable/index.html.

[33] „Python-rq“. [Stand: 14.03.2023]. (2023), Adresse: https://python-rq.org.

[34] „Dramatiq“. [Stand: 14.03.2023]. (2023), Adresse: https://dramatiq.io/index.html.

[35] „TaskTiger“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/closeio/tasktiger.

[36] „Huey“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/coleifer/huey.

[37] „MRQ“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/pricingassistant/mrq.

[38] „KQ (Kafka Queue)“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/joowani/kq.

[39] „Hapless“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/bmwant/hapless.

[40] „Taskmaster“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/dcramer/taskmaster.

[41] „Kuyruk“. [Stand: 18.03.2023]. (2023), Adresse: https://kuyruk.readthedocs.io/en/latest.

[42] „python-task-queue“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/seung-lab/python-task-queue.

[43] „SimpleQ“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/rdegges/simpleq.

[44] „Faktory“. [Stand: 14.03.2023]. (2023), Adresse: https://contribsys.com/faktory.

[45] „Gofer.NET“. [Stand: 14.03.2023]. (2023), Adresse: https://github.com/brthor/Gofer.NET.

[46] „Coravel“. [Stand: 14.03.2023]. (2023), Adresse: https://docs.coravel.net.

[47] „Hangfire“. [Stand: 14.03.2023]. (2023), Adresse: https://www.hangfire.io.

[48] „Runly“. [Stand: 14.03.2023]. (2023), Adresse: https://www.runly.io.

[49] „JobRunr“. [Stand: 14.03.2023]. (2023), Adresse: https://www.jobrunr.io/en.

[50] „Quartz“. [Stand: 14.03.2023]. (2023), Adresse: https://www.quartz-scheduler.org.

[51] „Install Redis on Windows“. [Stand: 21.03.2023]. (2023), Adresse: https://redis.io/docs/getting-started/installation/install-redis-on-windows.

[52] „Erlang“. [Stand: 21.03.2023]. (2023), Adresse: https://www.erlang.org.

[53] „Two ways to make celery 4 run on Windows“. [Stand: 21.03.2023]. (2023), Adresse: https://distributedpython.com/posts/two-ways-to-make-celery-4-run-on-windows.

[54] „Eventlet“. [Stand: 21.03.2023]. (2023), Adresse: https://eventlet.net.

[55] „Python-rq project history“. [Stand: 21.03.2023]. (2023), Adresse: https://python-rq.org/#project-history.

[56] „Motivation for Dramatiq“. [Stand: 21.03.2023]. (2023), Adresse: https://dramatiq.io/motivation.html.

[57] „Dramatiq scheduling“. [Stand: 21.03.2023]. (2023), Adresse: https://dramatiq.io/cookbook.html#scheduling.

[58] „Is there an fcntl replacement on Windows?“ [Stand: 21.03.2023]. (2023), Adresse: https://blog.finxter.com/is-there-an-fcntl-replacement-on-windows.

[59] „Dramtiq Cookbook: Flask“. [Stand: 04.04.2023]. (2023), Adresse: https://dramatiq.io/cookbook.html#flask.

[60] „Periodiq“. [Stand: 04.04.2023]. (2023), Adresse: https://gitlab.com/bersace/periodiq.

[61] „Background Tasks with Celery“. [Stand: 04.04.2023]. (2023), Adresse: https://flask.palletsprojects.com/en/2.2.x/patterns/celery.

[62] „Celery receives task, never runs task. Celery Worker on Windows is Broken. Documentation Issues?“ [Stand: 04.04.2023]. (2023), Adresse: https://github.com/celery/celery/issues/5738.

[63] „Can't get result with rabbitmq on win7“. [Stand: 04.04.2023]. (2023), Adresse: https://github.com/celery/celery/issues/2146.

[64] „Celery 'Getting Started' not able to retrieve results; always pending“. [Stand: 04.04.2023]. (2023), Adresse: https://stackoverflow.com/questions/25495613/celery-getting-started-not-able-to-retrieve-results-always-pending.

[65] „Celery & Uplink with SIGUSR1 Signal“. [Stand: 18.04.2023]. (2023), Adresse: https://github.com/prkumar/uplink/discussions/273.

[66] „Scrum“. [Stand: 07.03.2023]. (2023), Adresse: https://www.atlassian.com/agile/scrum.

[67] „Pair Programming“. [Stand: 30.05.2023]. (2023), Adresse: https://www.agilealliance.org/glossary/pairing.

[68] „Hangfire documentation: Overview“. [Stand: 19.04.2023]. (2023), Adresse: https://docs.hangfire.io/en/latest/#overview.

[69] „Using SQL Server: Configuration“. [Stand: 19.04.2023]. (2023), Adresse: https://docs.hangfire.io/en/latest/configuration/using-sql-server.html#configuration.

[70] „CancellationToken Struct“. [Stand: 26.05.2023]. (2023), Adresse: https://learn.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken?view=net-7.0.

[71] „AMQP 0-9-1 Model Explained“. [Stand: 26.05.2023]. (2023), Adresse: https://www.rabbitmq.com/tutorials/amqp-concepts.html.

[72] „Dealing with Exceptions“. [Stand: 09.06.2023]. (2023), Adresse: https://docs.hangfire.io/en/latest/background-processing/dealing-with-exceptions.html.

6.2. Abbildungsverzeichnis

1.1. Softwarearchitektur BORM [1] 1

2.1. Mit Swagger UI generierte API-Testoberfläche 7

3.1. Gewichtete Entscheidungsmatrix 15
3.2. Ablauf der Bachelorarbeit 18

4.1. Sequenzdiagramm für das Starten eines Tasks 19
4.2. Sequenzdiagramm für das Erhalten des Tasksstatus 19
4.3. Sequenzdiagramm für das Abbrechen eines Tasks 20
4.4. Sequenzdiagramm für das Scheduling von Tasks 20
4.5. Architekturübersicht mit Hangfire 21
4.6. Übersicht über Hangfire-Workflow [68] 21
4.7. Übersicht über die Verteilung der Softwaremodule 22
4.8. Entity Relation Diagramm der SYS_TASK-Tabelle in der Datenbank 23
4.9. Sequenzdiagramm für das Erstellen/Updaten von Tasks 25
4.10. Sequenzdiagramm für das Löschen von Tasks 26
4.11. Sequenzdiagramm für das Aktivieren eines Tasks 26
4.12. Sequenzdiagramm für das Deaktivieren eines Tasks 26
4.13. Sequenzdiagramm für das Starten eines Tasks 27
4.14. Sequenzdiagramm für die Statusabfrage eines Tasks 27
4.15. Swagger-UI für die Taskverwaltungs-Endpoints 28
4.16. Klassendiagramm des Prototyps 30
4.17. Auszug aus Hangfire.Job-Tabelle bei Ausführung mehrerer Tasks mit einem Worker . . 32
4.18. Auszug aus Hangfire.Job-Tabelle bei Ausführung mehrerer Tasks mit zwei Worker . . . 32
4.19. Auszug aus Hangfire.State-Tabelle bei Ausführung von scheduled Tasks 34
4.20. Entity Relation Diagramm für SYS_TASK und zugehörige Jobs 36
4.21. Erweitertes Sequenzdiagramm für das Löschen von Tasks 37
4.22. Einfaches Beispiel einer Nachricht in RabbitMQ 38
4.23. Direct-Exchange Routing in RabbitMQ 38
4.24. Auszug aus Hangfire.State-Tabelle mit wiederholt ausgeführtem Task 39
4.25. Auszug aus Hangfire.Job-Tabelle mit einem Task im Zustand "Scheduled" 39
4.26. Auszug aus Hangfire.Job-Tabelle mit einem Task im Zustand "Failed" 39

6.3. Tabellenverzeichnis

3.1. Message-Broker 9
3.2. Taskverwaltungs-Frameworks 10

4.1. Unit Tests für die Klassen SysTask.cs und TaskController.cs 31

6.4. Verzeichnis der Code-Listings

2.1. Einfache Flask-Applikation 5
2.2. Beispiel für Pythonintegration in .NET 6
2.3. C#-Code in Python 6

4.1. Ausschnitt aus Borm.Background.Worker: Worker-Konfiguration 23
4.2. Queue-Anbindung einer Methode in Borm.Background.QueueMethods 24
4.3. Enqueuing von Tasks mittels Borm.Background.QueueMethods 24
4.4. Aufrufen von Pythoncode in C# 25
4.5. test.py 28
4.6. ExecutePythonFile mit Konfiguration 29
4.7. slowTask_test.py 35
4.8. slowTask_writeFile_test.py 35
4.9. fastTask_test.py 35
4.10. expressTask_test.py 35

A. Anhang

A.1. Aufgabenstellung gemäss Complesis

Nachstehend die Aufgabenstellung der Bachelorarbeit, welche von Complesis übernommen wurde.

Die BORM-INFORMATIK AG entwickelt und vertreibt eine flexible und durchgängige Betriebsführungslösung. Im Fokus stehen dabei die Abläufe von produzierenden und projektorientierten Betrieben im holzverarbeitenden Sektor. Das ERP-System "BormBusiness" als Herzstück wird ergänzt durch die CAD-Lösung "PointLineCAD", die das Erstellen, Visualisieren und Bearbeiten von 2D- und 3D-Daten in Echtzeit erlaubt. Zudem wird das Portfolio durch die Anbindung dieser Systeme ans Web ("BormLive") und Apps für mobile Geräte erweitert. Ein wichtiges Merkmal des Systems ist die hohe Flexibilität und Anpassbarkeit, wodurch individuelle Kundenbedürfnisse abgebildet werden können.

Neue Business-Logik wird im "BormServer" implementiert und allen Oberflächen per API zur Verfügung gestellt. Damit ein API-Aufruf aus einem Client-Prozess (Desktop- oder Weboberfläche) nicht in HTTP-Timeout läuft, muss eine technische Möglichkeit geschaffen werden, um Hintergrundprozesse zu erstellen und deren Rückmeldungen auch wieder asynchron ans Frontend zurückzumelden. Da der BormServer ein heterogenes Gebilde aus diversen Programmiersprachen und Technologien ist, bedarf es einer Analyse, welche Technologie (SW- und HW-Architektur) für die Rahmenbedingungen der Borm-Produkte und deren Anpassbarkeit am besten geeignet ist.

Des Weiteren gibt es viele Aktionen, welche nicht durch direkte Benutzereingabe, sondern zeitlich gesteuert oder regelmässig auf dem BormServer ausgeführt werden sollen. Solche "Tasks" sollen von Clientseite erstellt und verwaltet werden können. Ebenfalls müssen Ergebnis- oder Fehlermeldungen als Nachricht an bestimmte Benutzer gesendet werden können. Hier stellt sich die Frage, ob und wie Gemeinsamkeiten mit den Hintergrundprozessen genutzt werden können.

Die ERP-Desktopapplikation ist hauptsächlich in C++ geschrieben. Für das Customizing kommt die firmeneigene Sprache "BormScript" zum Einsatz. Die CAD-Applikation basiert ebenfalls auf C++ und hat ein Python-API. Das Frontend der Webanwendung ist in JavaScript implementiert. Der BormServer nutzt Java, C# und Python, wobei letzteres ebenfalls zum Customizing verwendet werden kann.

Detaillierte Aufgabenstellung: Ziel dieser Arbeit ist es, aufzuzeigen wie Hintergrundprozesse und Tasks mit dem BormServer abgebildet werden können. Folgende Punkte sind gefordert:

A) Eine Ist-Situations- und Bedarfsanalyse zu Hintergrundprozessen und Tasks im Umfeld von BormBusiness.
B) Entwicklung von Konzeptvorschlägen (SW-/HW-Architektur) für die Einbettung von Hintergrundprozessen im BormServer. Dabei ist insbesondere die Interaktion mit dem Benutzer (Direktantwort und Rückmeldung des Ergebnisses) zu berücksichtigen.
C) Entwicklung von Konzeptvorschlägen (SW-/HW-Architektur) für die Verwaltung von zeitlich gesteuerten oder regelmässigen Tasks auf dem BormServer. Hier ist ein besonderes Augenmerk auf die Administrationsoberfläche und die Notifizierung bei Erfolg oder Fehler zu richten.
D) Evaluation einer geeignet Gesamtarchitektur. Dazu werden die Varianten aus den Punkten B) und C) gemeinsam mit der BORM-INFORMATIK AG bewertet und die jeweils Beste bestimmt.
E) Ein Umsetzungsprototyp, welcher die Hauptaspekte aus den Lösungsvorschlägen aufzeigt oder beinhaltet.