

Masterthesis

**Fakultät für
Informatik**

Studienrichtung
Informatik

Peter Heinemann

Task-Management in einer Microservice Umgebung

Erstprüfer: Prof. Dr. rer. nat. Claudia Reuter

Zweitprüfer: Dominik Horn

Abgabedatum: 30.09.2024

Peter Heinemann
Mindelheimer Straße 31a
86879 Wiedergeltingen
T +49 176-77631521
peter.hnm@gmx.de
Matrikelnummer:
2141060

**Technische Hochschule
Augsburg**

An der Hochschule 1
D-86161 Augsburg
T +49 821 5586-0
F +49 821 5586-3222
www.tha.de
info@tha.de

Erstellungserklärung

Erklärung zur Abschlussarbeit Hiermit versichere ich, die eingereichte Abschlussarbeit selbstständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

Peter Heinemann

Mindelheimer Straße 31a
86879 Wiedergeltingen
T +49 176-77631521
peter.hnm@gmx.de

Matrikelnummer:
2141060

Name: Peter Heinemann

Datum: 30.09.2024

Unterschrift: P. Heinemann

**Technische Hochschule
Augsburg**

An der Hochschule 1
D-86161 Augsburg
T +49 821 5586-0
F +49 821 5586-3222
www.tha.de
info@tha.de

1 Abstract

Microservice-Architekturen haben sich in den letzten Jahren als ein beliebtes Architekturmuster für die Entwicklung von Softwareanwendungen etabliert. Unternehmen setzen zunehmend auf diesen Architekturstil, da er die Skalierbarkeit und Flexibilität von Anwendungen verbessert. Auch Workflow-Engines werden zunehmend für einen cloud-nativen Einsatz entwickelt. Mit der Zeebe-Engine hat Camunda eine Engine speziell für den Einsatz in Microservice-Umgebungen entwickelt, die den BPMN 2.0 Standard unterstützt. BPMN sieht dabei die Einbindung von Menschen in einen Geschäftsprozess durch User-Tasks vor. Unternehmen haben jedoch häufig noch weitere Quellen für Benutzeraufgaben. Daher ist ein Task-Management notwendig, dass die Aufgaben aus verschiedenen Quellen zentral verwaltet. Mit dieser Arbeit wird eine Architektur und Implementierung vorgestellt. Das zentrale Ergebnis dieser Arbeit ist ein Prototyp, der eine eigene Lösung für das Task-Management in einer Microservice-Umgebung mit der Zeebe-Engine integriert. Die Anforderungen wurden mit Consultants der Firma Miragon sowie aus Betrachtung von bestehenden Systemen erarbeitet. Der Prototyp zeigt, wie mit einer zentralen Task-Management und Domain-Driven Design eine flexible und leichtgewichtige Lösung erarbeitet wurde. Micro-Frontends eignen sich dabei besonders gut, um sowohl Low-Code als auch Pro-Code Ansätze im Frontend zu unterstützen und dem Entwickler die maximale Flexibilität zu geben.

Inhaltsverzeichnis

1	Abstract	iii
2	Einleitung	1
3	Verwandte Arbeiten	3
3.1	Microservice-Architekturen	3
3.1.1	Kommunikation	4
3.1.2	Service Discovery	6
3.1.3	Deployment	6
3.2	Human Task-Management	7
3.3	Micro-Frontend	7
4	Use Case und Anforderungen	10
5	Business Process Management	12
5.1	Was ist BPM?	12
5.2	Workflow-Engines	13
5.2.1	Temporal	15
5.2.2	Wieso BPMN 2.0?	15
5.3	BPMN 2.0 Übersicht	17
5.3.1	Aktivität	17
5.3.2	Event	18
5.3.3	Gateway	21
5.4	Vorstellung der Prozessmodelle	22
6	Task Management	24
6.1	User-Task	24
6.2	Task-Liste	25
6.2.1	Camunda	26
6.2.2	Flowable	27
6.3	Formulare	28
6.3.1	Low-Code Tools	28
6.3.2	Die Grenzen von Low-Code	33

7	Camunda Platform 7 vs. 8	34
7.1	Übersicht Camunda Platform 7	34
7.2	Übersicht Camunda Platform 8	35
7.3	Vergleich der Engines	35
7.3.1	Architektur	36
7.3.2	Auswirkungen auf das Task-Management	38
8	Architektur des Proof-of-Concept	39
9	Implementierung des Proof-of-Concept	44
9.1	Hexagonale Architektur	44
9.2	Programmiersprachen und Frameworks	46
9.3	Task-Liste	48
9.3.1	Anforderungen	49
9.3.2	Herausforderungen	49
9.4	Task-Manager	53
9.4.1	Softwaredesign	54
9.4.2	Herausforderungen	55
9.5	Prozess-Applikation	61
9.5.1	Softwaredesign	62
9.5.2	Herausforderungen	64
9.6	Anleitung zum Starten des Prototypen	67
10	Zusammenfassung und Ausblick	69
	Literaturverzeichnis	73

2 Einleitung

Die Automatisierung von Geschäftsprozessen ist ein wichtiger Bestandteil auf dem Weg zur Digitalisierung von Unternehmen. Business Process Management Plattformen (BPM) wie Camunda oder Flowable bieten dazu Lösungen an, deren zentrale Komponente eine Workflow-Engine ist.

Workflow-Engines ermöglichen die Ausführung von End-to-end Prozessen, indem sie menschliche Prozessbeteiligte und IT-Systeme miteinander verknüpfen. Dabei nutzt die Engine beider Hersteller den Business Process Model and Notation (BPMN) 2.0 Standard, der eine grafische Darstellung von Prozessen ermöglicht. Dieser Standard kennt für die Einbindung von Personen in einen Prozess den User Task. Die Engine kümmert sich jedoch nicht, wie die Aufgabe durch den Benutzer bearbeitet wird.

Das Task-Management ist dafür eine notwendige separate Komponente. BPM-Plattformen bieten diese Komponente in der Regel an, das führt jedoch zu einer Abhängigkeit vom Hersteller. Zusätzlich gibt es in Unternehmen häufig mehrere Quellen für User Tasks, wie z.B. ein Enterprise Resource Planing (ERP) System oder eine zweite Engine in einem Migrationsprojekt.

Des Weiteren gibt es in letzter Zeit interessante Entwicklungen im Bereich der Workflow-Engines, die sich mit Microservices Architekturen und cloud-native Anwendungen beschäftigen. Camunda verfolgt mit der Einführung von Camunda Platform 8 und der dafür entwickelten Zeebe-Engine einen gewissen Paradigmenwechsel. Die Engine ist nur als Remote-Engine nutzbar und daher darauf ausgelegt in einer Microservice-Umgebung betrieben zu werden. Die Aufteilung in Abteilungen bzw. Domänen mit unterschiedlichen Prozessen macht eine Microservice Architektur mit Domain-Driven Design für Unternehmen interessant.

Mit dieser Arbeit möchte ich anhand eines Proof-of-Concept (PoC) eine leichtgewichtige und flexible Architektur aufzeigen, die das Task-Management in einer Microservice-Umgebung integriert. Das Ziel ist die Entwicklung eines lauffähigen Prototyps, der beispielhaft die Zeebe-Engine von Camunda anbindet. Der Prototyp dient als erster Durchstich und wird nicht alle notwendigen Funktionalitäten eines Task-Managements bieten. Für den PoC werden drei Komponenten entwickelt:

- **Task-Liste**, die die Aufgaben anzeigt
- **Task-Manager**, der die Aufgaben verwaltet

- **Prozess-Applikationen**, die zwei Geschäftsprozesse abbilden

Es werden nur Open-Source Technologien sowie BPMN 2.0 für die Modellierung genutzt.

Bevor der PoC vorgestellt wird, wird in Kapitel 3 auf verwandte Arbeiten eingegangen. Dabei werden Arbeiten aus den Bereichen Microservice-Architekturen, Human Task-Management sowie Micro-Frontends vorgestellt.

In Kapitel 4 wird der Use Case und die Anforderungen an die Architektur erläutert. Im darauf folgenden Kapitel 5 wird auf das übergeordnete Thema Business Process Management eingegangen und wieso Prozesse ganzheitlich zu betrachten sind. Dabei steht die Workflow-Engine im Mittelpunkt. Es wird daher ein allgemeiner Überblick über Workflow-Engines gegeben und die Unterschiede in der Modellierung von Prozessen mit einem Beispiel aufgezeigt. Daraus werden die Vorteile des BPMN 2.0 Standards herausgearbeitet und erläutert, warum eine BPMN-fähige Workflow-Engine für den PoC genutzt wird. Anschließend werden die Hauptelemente des Standards erklärt, um die Basis für die darauffolgende Vorstellung der beiden Prozesse zu schaffen, die für den Prototypen implementiert werden.

Kapitel 6 liefert einen vertieften Einblick in das Task-Management und den *User-Task* als ein spezifisches Element in BPMN 2.0. Es werden zwei unterschiedliche Lösungen aus BPM-Plattformen vorgestellt und Anforderungen an die Task-Liste herausgearbeitet. Ein weiterer wichtiger Aspekt ist die Entwicklung von Task spezifischen Formularen. Dafür werden häufig Low-Code Anwendungen genutzt. Daher werden drei solcher Anwendungen vorgestellt aber auch die Grenzen von Low-Code besprochen.

In Kapitel 7 wird ein technischer Vergleich zweier BPMN-fähiger Workflow-Engines vorgenommen und erläutert, wieso die Wahl auf die Zeebe-Engine von Camunda gefallen ist und welche Herausforderungen sich daraus ergeben.

Ab Kapitel 8 wird der PoC vorgestellt. Zunächst wird ein allgemeiner Überblick über die Architektur gegeben, bevor die einzelnen Komponenten erläutert werden und deren Implementierung gezeigt wird. Es werden technische Einblicke gegeben und erläutert, wie einzelne Entscheidungen auf die zuvor erarbeiteten Anforderungen einzahlen.

Den Schluss bildet Kapitel 10 mit einer Zusammenfassung und einem Ausblick.

Die Arbeit entsteht in Zusammenarbeit mit der Firma Miragon GmbH aus Augsburg.

3 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten aus den Bereichen Microservice-Architekturen, Human Task-Management sowie Micro-Frontends vorgestellt. Diese drei Bereiche bilden die Grundlage für die Entwicklung einer eigenen Lösung.

3.1 Microservice-Architekturen

Der Wandel von lokal installierten Programmen hin zu Anwendungen, die über das Internet bereitgestellt werden und die steigende Komplexität dieser Systeme haben die Popularität von Microservice-Architekturen steigen lassen [1, S. 44] [2, Kap. 1]. Für Fowler und Lewis [3] beschreibt ein Microservice eine kleine, unabhängige und lose gekoppelte Komponente, die eine spezifische Funktionalität bereitstellt und dabei folgende Eigenschaften aufweist:

- **Componentization via Services:** Ein Service ist eine eigenständige, austauschbare und erweiterbare Komponente, mit einem explizit definierter Interface.
- **Organized around Business Capabilities:** Ein Service kümmert sich um eine spezifische Geschäftsfähigkeit („business capability“). Das beinhaltet das User Interface (UI), die Datenbank und die Logik.
- **Products not Projects:** Das Team, das für einen Service verantwortlich ist, kümmert sich um den gesamten Lebenszyklus des Services.
- **Smart endpoints and dumb pipes:** Ein Service verwendet bewährte Protokolle wie HTTP für die synchrone Kommunikation und leichtgewichtige Bussysteme (RabbitMQ oder ZeroMQ) für die asynchrone Kommunikation.
- **Decentralized Governance:** Ein Team entscheidet selbstständig über die Technologien und Frameworks, die für den Service verwendet werden.
- **Decentralized Data Management:** Jeder Service kümmert sich um seine eigene Datenbank.
- **Infrastructure Automation:** Teams nutzen Automatisierungstechniken für das Testen und Bereitstellen von Services, sodass der Aufwand und Komplexität für das Bereitstellen mehrerer Services insgesamt geringer ist als bei einem Monolithen.

- ***Design for failure:*** Ein Service kann zu jeder Zeit ausfallen. Diese Ausfälle müssen bemerkt (z.B. durch Monitoring oder Logging) und im besten Falle automatisch behandelt werden.
- ***Evolutionary Design:*** Bevor ein Service entwickelt wird, muss sich der Entwickler darüber Gedanken machen, wie er diesen „schneiden“ möchte. Der Service wird sich jedoch im Laufe der Zeit verändern. Wird der Service größer, kann er in kleinere Services aufgeteilt werden. Merkt der Entwickler, dass bei Änderungen in einem Service, auch ein anderer angepasst werden muss, ist das vielleicht ein Zeichen dafür, dass die beiden Services zusammengehören.

Alshuqayran u. a. [1] haben die bis dahin veröffentlichten Arbeiten zu Microservice-Architekturen untersucht. Dabei wurden die Vorteile mit Flexibilität, Skalierbarkeit, Wartbarkeit, Ausfallsicherheit, Zuverlässigkeit sowie die Trennung von Belangen beschrieben. Zu den am häufig genannten Herausforderungen zählen die Kommunikation zwischen den Services, das Bereitstellen der Services (Deployment) sowie das Finden der Services über das Netzwerk (Service Discovery). Im weiteren Verlauf dieser Arbeit werden die englischen Begriffe Deployment und Service Discovery verwendet.

3.1.1 Kommunikation

Bei einer Service-orientierten Architektur (SOA) wie der Microservice Architektur, kommunizieren die Services über das Netzwerk miteinander anstelle von direkten Methodenaufrufen. Dabei gibt es verschiedene Möglichkeiten der Kommunikation, die sich in der Komplexität und Performance unterscheiden. Newman [2, Kap. 4] unterscheidet zwischen synchronen (blocking) und asynchronen (non-blocking) Kommunikationsarten. Im Normalfall werden Microservices beide Arten nutzen, um miteinander zu kommunizieren. Eine asynchrone Kommunikation ist vorteilhaft, wenn die Antwortzeit des Services nicht bekannt ist oder der Service lange benötigt, um die Anfrage zu bearbeiten.

Beliebte Technologien zur Kommunikation umfassen HTTP/REST, gRPC und die Verwendung von Message-Brokern [2, Kap. 5].

Representational State Transfer (REST) ist ein Architekturstil, der die Kommunikation zwischen Services über das Hypertext Transfer Protocol (HTTP) beschreibt. Dabei wird ein Service über eine URL angesprochen und die Anfrage wird über die HTTP-Methoden GET, POST, PUT oder DELETE gesendet.

gRPC ist ein von Google entwickeltes Remote Procedure Call (RPC) Framework, das auf HTTP/2 basiert. Im Quellcode sieht ein gRPC-Aufruf aus wie ein lokaler Methodenaufruf auf der Seite des Clients. Tatsächlich wird diese im Remote-Service ausgeführt. HTTP/2 ist die weiterentwickelte Version von HTTP/1.1 und bietet unter anderem Multiplexing, das bedeutet, dass mehrere Anfragen gleichzeitig über eine Verbindung gesendet werden

können.

Message-Broker sind spezielle Software, die Nachrichten entgegennehmen und in eine Warteschlange (Queue) einreihen. Ein Service kann Nachrichten aus der Warteschlange lesen und darauf reagieren.

Viennot u. a. [4] haben hierzu das System *Synapse* entwickelt, dass die Kommunikation von data-driven Web-Applikationen optimiert. Im Speziellen betonen die Autoren die wichtige Rolle des Application Programming Interface (API). Das im Falle von Synapse auf publish/subscribe Systemen basiert.

Wenn Microservices verwendet werden, um ganze Prozesse abzubilden, muss die Kommunikation zwischen ihnen entsprechend gesteuert werden. Die Reihenfolge der Services ist dabei entscheidend. Ruecker [5, Kap. 8] veranschaulicht das an einem Beispielprozess über die Auftragserfüllung in einem Unternehmen. Dabei sollen Waren erst ausgeliefert werden, wenn die Bezahlung erfolgt ist. Diese Anordnung der Services und das Steuern der Kommunikation kann durch Choreografie oder Orchestrierung erfolgen.

Bei der **Orchestrierung** von Services gibt es eine zentrale Instanz (im Folgenden Orchestrator genannt), die die Services steuert. Newman [2, Kap. 6] schreibt dabei von einem „command-and-control-Ansatz“. Ruecker [5, Kap. 8] spricht von einer Command-Driven Architektur. Der Orchestrator kennt die Reihenfolge und bestimmt, wann welcher Service aufgerufen wird. Dadurch ergibt sich eine klare Struktur, die einfach zu verfolgen und zu überwachen ist. Workflow-Engines werden häufig für die Orchestrierung von Services verwendet. Sie übernehmen dabei die Rolle des Orchestrators und steuern die Reihenfolge der Services. Das hat den Vorteil, dass es einen zentralen Punkt gibt, der für den Prozessfluss verantwortlich ist und dadurch auch nur eine Stelle, an der der Prozess angepasst werden muss [5, Kap. 8]. Als Nachteil der Orchestrierung wird die Enge Kopplung der Services mit dem Orchestrator bemängelt [2, Kap. 6].

Choreographie ist ein dezentraler Ansatz, bei dem die Services selbst wissen, wie sie miteinander kommunizieren. Dadurch fällt eine zentrale Instanz weg, die die Services steuert. Newman [2, Kap. 6] beschreibt diesen Ansatz im Vergleich zur Orchestrierung als eine „trust-but-verify“ Architektur. Die Services kommunizieren über Events miteinander. Ein Event ist etwas, das passiert ist. Aus technischer Sicht ist es ein Objekt mit einem *Header*, die das Event beschreiben und einem *Body*, der die Informationen enthält, die übertragen werden sollen [6, S. 2]. Eine Komponente kann dabei für sich selbst entscheiden, ob sie auf ein Event reagiert oder nicht. Anstelle eines Orchestrators wird hierfür ein Message-Broker verwendet, der die Events verteilt. Dieser auch als Event-Driven Architecture (EDA) bekannte Ansatz setzt primär auf eine asynchrone Kommunikation [6, S. 1] [5, Kap. 8], die bei einer hohen Anzahl von Services, dazu führen

kann, dass es nur noch schwer nachzuvollziehen ist, wann welcher Service aufgerufen wird [5, Kap. 8]. Der Vorteil liegt in einer loseren Kopplung der Services als bei einer Orchestrierung [2, Kap. 6].

3.1.2 Service Discovery

Um die Kommunikation zwischen den Services zu ermöglichen, müssen die Services sich untereinander finden. Newman [2, Kap. 5] zeigt zwei verbreitete Lösungen auf.

1. Mit dem **Domain Name System (DNS)** kann für den Service eine Domäne registriert werden, die auf die IP-Adresse des Services zeigt. Der Service ist somit über die Domäne erreichbar. Probleme mit DNS kommen dann auf, wenn der Service in einer dynamischen Umgebung läuft.
2. Über die Implementierung eines **Service Registry** können sich die Services bei einer zentralen Instanz registrieren. Andere Services können dann über die Registry den Service finden. Bekannte Lösungen, wie z.B. Consul, Zookeeper oder Kubernetes, bieten ein solches Registry an.

3.1.3 Deployment

Das Deployment beschreibt den Vorgang, wie eine Anwendung auf einem Server bereitgestellt wird. Newman [2, Kap. 8] betont dabei die Rolle von Container, die mittlerweile andere Formen der Bereitstellung über physische oder virtuelle Maschinen abgelöst haben. Container sind ressourceneffizienter gegenüber physischen Maschinen und haben weniger Overhead als virtuelle Maschinen. Docker ist dabei eine der bekanntesten Technologien für Container und erleichtert das Deployment von Anwendungen [7]. Die Beliebtheit von Containern in Microservice Architekturen hat wiederum die Notwendigkeit von Container-Orchestrierungssystemen hervorgebracht. Dabei ist Kubernetes ein bekanntes Open-Source-System, das die Verwaltung von Containern in sogenannten Pods übernimmt [8, Kap. 5]. Ein Pod ist dabei die kleinste Einheit in Kubernetes und kann aus einem oder mehreren Containern bestehen. Diese Einheiten werden wiederum auf Nodes bereitgestellt [2, Kap. 8]. Ein Node kann dabei eine physische oder virtuelle Maschine sein. Durch Kubernetes ist es möglich neue Anwendungen ohne Downtime auszurollen, sowie diese zu skalieren [8, Kap. 10].

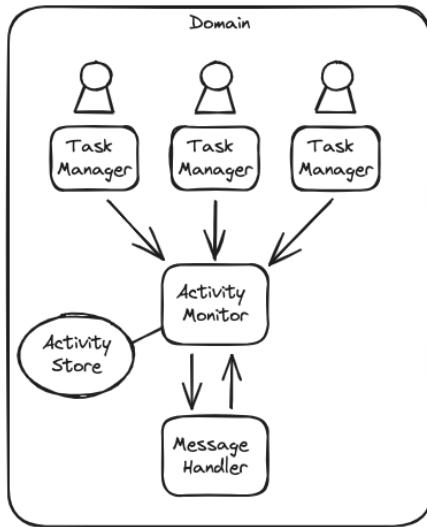


Abbildung 3.1: Architektur des von Kreifelts u. a. [13, Abb. 4] entworfenen Task-Manager

3.2 Human Task-Management

Dieser Abschnitt behandelt den technischen Aspekt in Arbeiten aus dem Bereich des Human Task-Management. Das Ziel von Task-Managern ist es, die Arbeit von Menschen zu organisieren. Eine zentrale Anlaufstelle für alle Aufgaben, die einen Wissensarbeiter betreffen, trägt zu seiner erhöhten Effektivität und Effizienz bei. Dies geschieht durch eine bessere Priorisierung und Übersicht [9]–[11]. Die Aufgaben können aus verschiedenen Quellen stammen, wie beispielsweise aus E-Mail-, Workflow- oder ERP-Systemen [12].

Bellotti u. a. [9] haben dafür die Desktop-Anwendung **TaskVista** entwickelt, das ein leichtgewichtiges Frontend zum Verwalten von To-do-Listen bietet. Ein User kann unter anderem seine eigenen Aufgaben erstellen, priorisieren, Aufwand schätzen und delegieren. Die UI ist einfach und intuitiv gehalten, um den Fokus auf die Aufgaben zu legen. Jedoch unterstützt die Anwendung keine Kollaboration zwischen den Wissensarbeitern. Jeder Nutzer hat seine eigene Instanz von TaskVista und kann nur seine eigenen Aufgaben verwalten.

Kreifelts u. a. [13] nutzten hingegen eine Client-Server-Architektur für ihr System zur Verwaltung von Aufgaben in verteilten Teams. Die Architektur ist in Abbildung 3.1 dargestellt. Dabei kontrolliert der **Task Manager** die benutzerspezifische Darstellung der **Task List**. Daher gibt es auch für jeden Benutzer eine eigene Instanz des Task Managers. Der **Activity Monitor** kümmert sich um alle Änderungen an einem Task und speichert diese im **Activity Store**. Da Änderungen an einem Task auch aus anderen Domänen kommen kann, kommuniziert der Activity Monitor mit dem **Message Handler**.

Anders als bei TaskVista liegt hier der Fokus auf Kollaboration und Kommunikation zwischen den Wissensarbeitern, auch in unterschiedlichen Domänen. Das Task-Management ist domänenweit implementiert und muss daher mit den anderen Activity Stores kommunizieren. Dies erhöht die Komplexität des Systems. Da die Arbeit aus dem Jahr 1993 stammt, verwendet sie veraltete Technologien wie X.400 für die Kommunikation. Auch werden Tasks von den Benutzern eingestellt und sind nicht Teil eines automatisierten Prozesses.

Während die ersten beiden Systeme eine relativ einfache Darstellung der Aufgaben in sogenannten Task-Listen bieten, haben Abdellaoui u. a. [10] einen Task-Manager spezifisch für Fluglotsen entworfen. Anstelle einer Task-Liste müssen Radarbilder und Flugpläne dargestellt werden. Das System **BATMAN** erleichtert die Einarbeitung von Fluglotsen und trägt zu einer besseren Ressourcenauslastung bei. Dabei gehen die Autoren wenig auf die technische Umsetzung ein, sondern beschreiben die Anforderungen und die Funktionalitäten des Systems, die sehr spezifisch für Fluglotsen sind.

3.3 Micro-Frontend

Micro-Frontends verfolgen ähnlich zu den Microservices einen großen Monolithen in kleinere, unabhängige Teile zu zerlegen. Dabei ähneln die Vorteile denen von Microservices [14]–[16]. Damit gehen auch ähnliche Herausforderungen einher, wie die Kommunikation zwischen den Micro-Frontends, Performance und Sicherheit [14] [16, Kap. 3]. Newman [2, Kap. 14] und Mezzalira [16, Kap. 3] beschreiben zwei unterschiedliche Ansätze, wie Micro-Frontends in eine Webseite eingebettet werden können.

Page-Based Decomposition[2] bzw. **Vertical Split**[16] beschreibt einen Ansatz, in dem die Webseite in mehrere Seiten aufgeteilt wird. Jede Seite stellt eine Domäne dar und kommuniziert mit einem eigenen Microservice. Abbildung 3.2 zeigt den vertikalen Aufbau eines Micro-Frontends. Newman [2, Kap. 14] sieht den Vorteil bei diesem Ansatz durch eine einfachere Implementierung. Der Benutzer navigiert über Links zwischen den Seiten und es benötigt keine JavaScript-Frameworks, um Änderungen am DOM vorzunehmen.

Widget-Based Decomposition[2] bzw. **Horizontal Split**[16] beschreibt einen Ansatz, bei dem die Webseite in mehrere Bereiche unterteilt wird. Jeder dieser Bereiche wird durch ein eigenes Team entwickelt und kommuniziert mit einem Microservice. Ein Beispiel für einen horizontalen Aufbau eines Micro-Frontends, ist in Abbildung 3.3 dargestellt. Dabei enthält der umschließende Container die Micro-Frontends. Dieser Ansatz eignet sich besonders für Single-Page-Applications (SPA), da die Navigation zwischen den Bereichen ohne Seiten-Wechsel erfolgt. Auch können dadurch wiederverwendbare Widgets erstellt

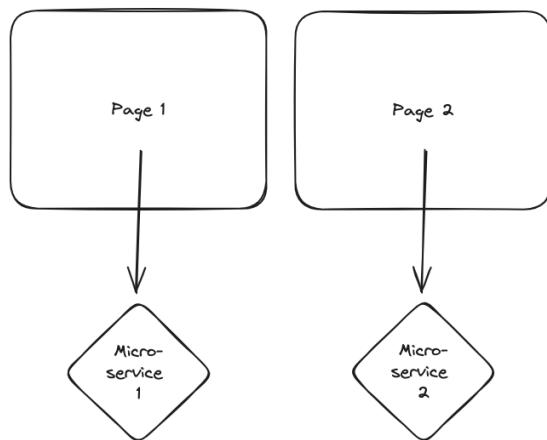


Abbildung 3.2: Vertikaler Aufbau eines Micro-Frontends

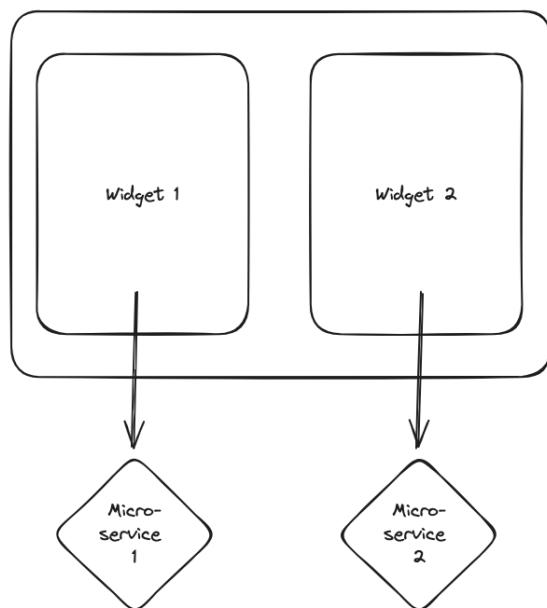


Abbildung 3.3: Horizontaler Aufbau eines Micro-Frontends

werden [2, Kap. 14]. Allerdings kommen die Vorteile auch mit einer komplexeren Implementierung. Es gibt hierfür zwei Ansätze, diese zu realisieren:

Der **iFrame** ist ein einfacher Weg, um ein Micro-Frontend in eine Webseite einzubetten. Dabei handelt es sich um keine neue Technologie, sondern um ein HTML-Element, das eine andere Webseite in sich lädt. Dadurch wird der iFrame auch von älteren Browsern unterstützt. Durch *MessageEvents* kann die Kommunikation zwischen dem Container und dem Micro-Frontend erfolgen. Aufgrund der starken Isolierung ist die Kommunikation

unter iFrames erschwert. Die Vorteile bei der Nutzung von iFrames sind die einfache Integration und die starke Isolierung der Micro-Frontends. Jedoch sind iFrames sehr CPU-intensiv, vor allem wenn mehrere iFrames auf einer Seite eingebettet sind. Neben der Performance ist das dynamische Styling in responsiven Webseiten problematisch, da die Größe des iFrames nicht von seinem Inhalt abhängig ist und somit nicht dynamisch angepasst werden kann. [2], [14]–[16]

Eine Alternative zu iFrames sind **Web Components**. Web Components sind eine Sammlung von Web-Technologien, die es ermöglichen, eigene HTML-Elemente zu erstellen. Die Idee hinter Web Components ist es, wiederverwendbare Komponenten zu erstellen, die in jedem Framework verwendet werden können. Web Components teilen sich das gleiche *window Object*, was die Kommunikation zwischen den Komponenten vereinfacht. Auch sind sie weniger Performance intensiv als iFrames, da kein neuer Kontext erstellt wird. Dafür handelt es sich um eine neuere Technologie, die eventuell von älteren Browsern nicht unterstützt wird. Auch ist der Einstieg etwas erschwert, da sich erst mit den unterschiedlichen Technologien, wie *Shadow DOM* oder *Custom Elements* auseinandergesetzt werden muss. [14]–[16]

4 Use Case und Anforderungen

Frühere Workflow-Engines wie die von Leung und Chung [17] oder Camunda [18] haben einen monolithischen Ansatz. Dabei ist das Human Task-Management ein zentraler Bestandteil der Engine. Im folgenden wird von Task-Management gesprochen, wenn es um die Verwaltung von Aufgaben geht, die durch Menschen bearbeitet werden. Durch die Vorteile von Cloud-Computing, wie Skalierbarkeit und Flexibilität, haben Unternehmen zunehmend damit begonnen ihre Anwendungen in Cloud-Plattformen zu betreiben [19, Kap. 1]. Dadurch gibt es nun auch Entwicklungen von Workflow-Engines für den Einsatz in cloud-native Anwendungen [20]. Cloud-native Workflow-Engines, wie *Conductor* [21], *Cadence* [22] oder *Temporal* [23] haben sich auf die Orchestrierung von Microservices spezialisiert. Das Task-Management spielt dabei keine zentrale Rolle.

Im Gegensatz dazu setzt Camunda [24] mit ihrer Zeebe-Engine auf den BPMN 2.0 Standard, der eine Einbindung von Menschen in einen Geschäftsprozess durch User-Tasks vorsieht. Allerdings wurde bei der Umstrukturierung der Engine wichtige Funktionen für das Task-Management in die proprietäre *Camunda Tasklist* ausgelagert. Abbildung 4.1 zeigt die Operationen, die nur durch die Tasklist API abgedeckt sind.¹ Im speziellen die wegfallende Möglichkeit User-Tasks abzufragen erschwert die Integration von einer eigenen Lösung für das Task-Management. Unternehmen, die sich aufgrund des endenden Supports von C7 mit der Migration auf C8 beschäftigen, oder C8 neu einführen wollen, müssen sich mit der Frage auseinandersetzen, wie sie das Task-Management in ihre Architektur integrieren. Zusätzlich kommt es in Unternehmen häufig vor, dass es mehrere Quellen für User-Tasks gibt. Diese Quelle kann eine zweite Engine sein, speziell in einem Migrationsszenario ist das keine Seltenheit aber auch aus einem ERP-System können User-Tasks entstehen.

Mit dieser Arbeit möchte ich eine leichtgewichtige und flexible Architektur aufzeigen, die eine eigene Lösung für das Task-Management mit der Zeebe-Engine in einer Microservice-Umgebung integriert. Die Anforderungen an die Architektur wurden zusammen mit Consultants der Firma Miragon erarbeitet. Daher liegt der Fokus auf technischen Anforderungen, die im folgenden erläutert werden:

¹<https://docs.camunda.io/docs/apis-tools/tasklist-api-rest/migrate-to-zeebe-user-tasks/>

Operation	Tasklist API	Zeebe Task API (8.5)
Query tasks	✓ All types	← Use Tasklist API
Get task	✓ All types	← Use Tasklist API
Retrieve task variables	✓ All types	← Use Tasklist API
Get task form	✓ All types	← Use Tasklist API
Change task assignment	✓ Job worker-based tasks	✓ Zeebe tasks
Complete task	✓ Job worker-based tasks	✓ Zeebe tasks
Update task	-	✓ Zeebe tasks
Safe and retrieve draft variables	✓ All types	← Use Tasklist API

Abbildung 4.1: Auflistung an Funktionalitäten die durch die Tasklist API bzw. Zeebe Task API abgedeckt sind [24, "Migrate to Zeebe user tasks"].

- **KISS-Prinzip:** Die Architektur soll einfach und verständlich sein. Dadurch soll die Entwicklungszeiten für Prozess-Applikationen verkürzt werden.
- **Flexibilität:** Die Architektur soll flexibel sein, um unterschiedliche Technologien und Frameworks zu unterstützen. Der Entwickler soll dabei nicht in ein zu enges Korsett gezwungen werden.
- **Engine-Kompatibilität:** Die Architektur soll es ermöglichen verschiedene Engines anzubinden. Der PoC zeigt exemplarisch die Anbindung der Zeebe-Engine.
- **Unterstützung im Frontend:** Einbindung von Technologien, die Entwickler mit wenig oder keiner Erfahrung in der Frontend-Entwicklung unterstützen.
- **Einsatz von Open-Source Technologien:** Bei der Implementierung der Architektur sollen Open-Source Technologien eingesetzt werden.
- **Docker:** Die Komponenten der Architektur laufen in Docker-Containern. Dadurch soll das Deployment in Produktivumgebungen, die z.B. Kubernetes nutzen, vereinfacht werden.

5 Business Process Management

Wie bereits erwähnt, wird für den PoC eine Workflow-Engine verwendet, die den BPMN-Standard unterstützt. BPMN ist ein beliebtes Werkzeug im übergeordneten Bereich des BPM. Daher wird in diesem Kapitel eine Übersicht über BPM gegeben sowie detaillierter auf Workflow-Engines eingegangen und wieso sich für eine BPMN-fähige Engine entschieden wurde. Als dritter Punkt wird eine Einführung in den BPMN 2.0-Standard gegeben, bevor die beiden Prozesse vorgestellt werden, die im PoC genutzt werden.

5.1 Was ist BPM?

In der Literatur finden sich verschiedene Definitionen von BPM.

Elzinga u. a. [25, S. 119] definiert BPM als „*einen systematischen und strukturierten Ansatz, um Prozesse zu analysieren, verbessern, kontrollieren und zu verwalten, mit dem Ziel die Qualität von Produkten und Dienstleistungen zu verbessern*“ [Übers. d. Verf.]. Für die European Association of BPM [26] ist BPM „*... ein systematischer Ansatz, um sowohl automatisierte als auch nicht automatisierte Prozesse zu erfassen, zu gestalten, auszuführen, zu dokumentieren, zu messen, zu überwachen und zu steuern und damit nachhaltig die mit der Unternehmensstrategie abgestimmten Ziele zu erreichen.*“

Beide Definitionen betonen den ganzheitlichen Ansatz von BPM. Dabei steht der Geschäftsprozess im Mittelpunkt. Ein Geschäftsprozess ist eine Abfolge von Aktivitäten, die ein bestimmtes Ziel verfolgen. Dabei ist es unerheblich, ob die Aktivitäten automatisiert oder manuell durchgeführt werden. BPM umfasst daher eine organisatorische als auch technische Sicht auf Prozesse und ist somit funktionsübergreifend [27, S. 216]. Der ganzheitliche Ansatz von BPM wird auch als BPM-Lifecycle bezeichnet. Macedo de Morais u. a. [28] haben mehrere Modelle für den BPM-Lifecycle betrachtet und anschließend ein eigenes Modell entwickelt, das in Abbildung 5.1 in einer vereinfachten Form dargestellt ist. Der Startpunkt ist meist die Erhebung eines Prozesses. Laut Macedo de Morais u. a. [28, Abb. 10] spielen in die Prozessanalyse auch Faktoren wie Unternehmensstrategie oder beteiligte Stakeholder eine Rolle. Das wurde in der Abbildung 5.1 ausgelassen. Nach der Analyse folgt die Modellierung des Prozesses. Dafür können unterschiedliche Modellierungssprachen genutzt werden. Ereignisgesteuerte Prozessketten (EPK) oder BPMN sind dabei bekannte Modellierungssprachen speziell

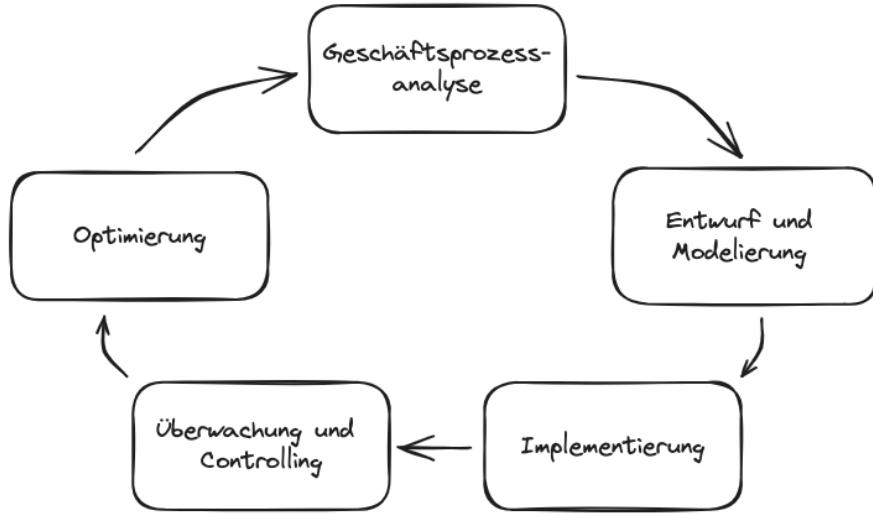


Abbildung 5.1: Vereinfachte Darstellung des BPM-Lifecycle nach Macedo de Morais u. a. [28, Abb. 10]

für die grafische Darstellung von Geschäftsprozessen. Dieser Prozess wird anschließend mithilfe einer Workflow-Engine implementiert. Nach erfolgreicher Implementierung wird der Prozess durch das Prozesscontrolling überwacht. Bei der Überwachung werden Prozesskennzahlen (KPIs) genutzt, anhand derer Optimierungspotenziale erkannt werden können.

Entlang des Lebenszyklus helfen verschiedene Tools die entsprechenden Aufgaben zu erfüllen. Im Bereich der Analyse und Optimierung hat sich der Begriff des Process Mining etabliert.

Firmen wie Celonis oder Noreja entwickeln hierfür Anwendungen, die auf den Daten von Prozessausführungen basieren und daraus optimierte Prozessmodelle ableiten. Für die Modellierung, Ausführung und Überwachung von Prozessen, kann z.B. die Camunda Platform genutzt werden. Dabei nutzt Camunda ihre eigene Workflow-Engine, die den BPMN-Standard unterstützt.

Die nächsten Abschnitte geben eine Übersicht über diese Themenbereiche.

5.2 Workflow-Engines

Eine Workflow-Engine ist eine Anwendung, die die Ausführung von Prozessen (Workflows) steuert. Sie ist somit die zentrale Komponente in einem BPM-System [29, S. 6]. Dabei besteht ein Prozess aus einer Abfolge von zusammenhängenden Aufgaben (Tasks), die durch Menschen oder Systeme in einer bestimmten Reihenfolge ausgeführt werden.

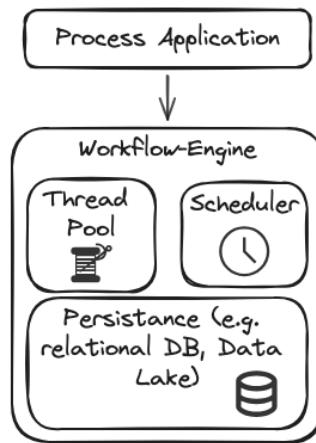


Abbildung 5.2: Vereinfachte Darstellung einer Workflow-Engine [5, Kap. 2]

Laut Ruecker [5, Kap. 3] umfassen die Kernfunktionen einer Workflow-Engine das Managen des Prozesszustands, die korrekte Ausführung von Tasks (*Scheduling*) und die Verwaltung von Prozessdefinitionen und -instanzen (*Versioning*) zu den Kernfunktionen einer Workflow-Engine. Diese drei Kernfunktionen ermöglichen das Ausführen von lang laufenden Prozessen.

Eine Workflow-Engine kann, wie in Abbildung 5.2 zu sehen, als ein separater Service in einer Microservice-Architektur betrieben werden. Dadurch wird eine klare Trennung zwischen der eigenen Applikation und der Workflow-Engine erreicht [5, Kap. 2]. Alternativ gibt es auch Engines die direkt in die eigene Applikation eingebettet werden können [18, “Architecture Overview”].

Neben der Architektur der Workflow-Engine gibt es auch Unterschiede in der Modellierung von Prozessen. Workflows können textbasiert durch JSON oder YAML Dateien definiert werden [20], [21], als Code in einer Programmiersprache [23] oder durch Petri-Netze [30]. Mit dem Standard BPMN 2.0 gibt es auch eine speziell für die Modellierung von Geschäftsprozessen entwickelte Sprache [31]. BPMN beschreibt eine grafische Darstellung von Prozesselementen wie z.B. Tasks oder Events und wie diese in XML dargestellt werden. Die XML-Datei wiederum kann von Workflow-Engines, die den Standard unterstützen, eingelesen und ausgeführt werden. Zwei beliebte Open-Source-Workflow-Engines, die BPMN unterstützen, werden von den Unternehmen Camunda und Flowable entwickelt. Beide Unternehmen bieten neben der Workflow-Engine auch eine Plattform an, die auch die Modellierung von BPMN-Modellen und Formularen sowie das Task-Management und Monitoring von Prozessen ermöglicht.

5.2.1 Temporal

Temporal ist eine Open-Source Workflow-Engine, die entwickelt wurde, um verteilte, fehlertolerante und skalierbare Anwendungen zu ermöglichen. In diesem Abschnitt soll ein kurzer Überblick über die Funktionalitäten von Temporal gegeben werden. Dabei liegt der Fokus auf der Modellierung von Prozessen. Wie bereits erwähnt werden Prozesse durch das Schreiben von Code definiert. Temporal bietet mehrere SDKs für unterschiedliche Programmiersprachen an. In diesem Abschnitt wird, wenn Codebeispiele gezeigt werden, die Java SDK genutzt. Folgend sind die drei Hauptkonzepte dargestellt [23]:

- **Worker:**¹ Worker sind für die Ausführung von Workflows und Activities verantwortlich. Ein Worker ist ein Service, der kontinuierlich auf eine Task-Queue lauscht und die darin enthaltenen Workflows und Activities ausführt.
- **Workflow:**² Ein Workflow beschreibt den Ablauf eines Prozesses. Er kennt die Aktivitäten und entscheidet, welche Aktivität als nächstes ausgeführt wird. Dafür werden in der Programmierung bekannte Konzepte wie Schleifen oder Verzweigungen genutzt.
- **Activity:**³ Eine Activity wird durch einen Workflow aufgerufen. Sie wird allerdings nicht direkt ausgeführt, sondern in die Task-Queue geschrieben. Der Worker hört auf die Queue und führt die Activity aus.

In Abbildung 5.3 ist ein einfaches Beispiel für einen Workflow in Temporal dargestellt. Im Workflow wird das Prozessmodell definiert und implementiert. Es ist daher erforderlich, die verwendete Programmiersprache zu beherrschen, um den Prozess anzupassen und generelle Kenntnisse über die Programmierung um den Prozess zu verstehen.

5.2.2 Wieso BPMN 2.0?

Im vorangegangenen Abschnitt wurde eine Variante vorgestellt, die Prozesse durch Code definiert. BPMN 2.0 hingegen bietet eine grafische Darstellung von Prozessen die auch von Nicht-Programmierern verstanden werden kann. Abbildung 5.4 zeigt den gleichen Prozess wie in Abbildung 5.3 in BPMN 2.0. Dabei ist der Personenkreis von Nicht-Programmierern, nicht nur auf die Fachabteilungen begrenzt, auch Personen aus dem Prozesscontrolling können so erkannte Probleme in einem Modell direkt markieren und kommentieren [5, Kap. 5]. Zusätzlich kann BPMN 2.0 auch als Ersatz für das UML-Aktivitätsdiagramm genutzt werden. Somit ist es auch bei Diskussionen zwischen Entwicklern ein nützliches Werkzeug und es führt dazu, dass die Dokumentation aktuell

¹<https://docs.temporal.io/workers>

²<https://docs.temporal.io/workflows>

³<https://docs.temporal.io/activities>

```

● ● ●
1  @WorkflowInterface
2  public interface OrderWorkflow {
3      @WorkflowMethod
4      String processOrder(String itemId);
5  }

● ● ●
1  public class OrderWorkflowImpl implements OrderWorkflow{
2      /**
3      * Configuration for retry Strategy, Timeout, etc.
4      */
5      ActivityOptions options = ActivityOptions.newBuilder()
6          .setStartToCloseTimeout(Duration.ofSeconds(5))
7          .build();
8
9      /**
10     * Activities registered with @ActivityInterface
11    */
12    private final OrderActivities activities =
13        Workflow.newActivityStub(OrderActivities.class, options);
14
15    @Override
16    public String processOrder(String itemId) {
17        Boolean itemIsAvailable = activities.checkIfResourceIsAvailable(itemId);
18
19        if(itemIsAvailable) {
20            return activities.sendResource(itemId);
21        } else {
22            return activities.sendOrderDeniedMail();
23        }
24    }
25 }

```

Abbildung 5.3: Beispiel für einen Workflow in Temporal

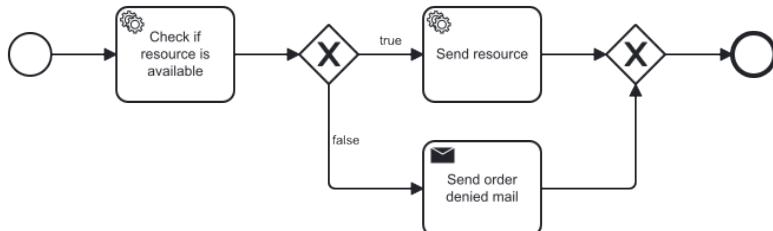


Abbildung 5.4: Beispiel für einen Workflow in BPMN 2.0

bleibt. Jedoch ist nur die grafische Darstellung nicht ausreichend. Die Prozesse müssen auch ausgeführt werden können. Daher baut der BPMN 2.0 Standard auf eine XML-Struktur auf. Hersteller von Workflow-Engines können diesen Standard implementieren. Dadurch wird gewährleistet, dass der Prozess in unterschiedlichen Workflow-Engines ausgeführt werden kann.

5.3 BPMN 2.0 Übersicht

BPMN wurde ursprünglich von White [32] entwickelt. Dabei lag der Fokus vor allem auf der grafischen Darstellung von Geschäftsprozessen. Die erste Version des Standards wurde 2004 von der Business Process Management Initiative (BPMI) veröffentlicht. Das Ziel „... war es, eine Notation bereitzustellen, die für alle Prozessbeteiligten leicht verständlich ist ...“ [32]. Dazu gehören nicht nur die Entwickler, sondern auch die Fachabteilungen und das Management. Wie White u. a. [33] aufzeigt, gibt es eine Ähnlichkeit zwischen BPMN und dem durch die Object Management Group (OMG) entwickelten UML 2.0 Aktivitätsdiagramm. Aktivitätsdiagramme werden in der Softwareentwicklung genutzt, um die Abläufe von Programmen zu modellieren. Somit sollte es für Entwickler leicht sein, BPMN zu erlernen. Der damalige Standard war jedoch nicht ausreichend ausgereift, sodass Workflow-Engines die Prozesse automatisiert ausführen konnten. *Business Process Execution Language for Web Services* (BPEL4WS) war zu dieser Zeit der Standard für die automatische Ausführung von Prozessen. White [32, S. 10] beschreibt zwar wie eine Konvertierung von BPMN zu BPEL4WS erfolgen kann, jedoch fehlten auch in der Version 1.2 noch einige Konstrukte, die für die Automatisierung von Prozessen notwendig sind [29, S. 8]. Das wurde erst nach der Übernahme der BPMI durch die OMG und der Veröffentlichung von BPMN 2.0 erreicht. Im Folgenden werden die Elemente von BPMN 2.0 vorgestellt, so wie sie auch von der OMG in der Spezifikation [31] definiert wurden. Anschließend werden die beiden Prozesse gezeigt, die auch im PoC genutzt werden.

5.3.1 Aktivität

Eine Aktivität ist ein Prozessschritt, der ausgeführt wird. Es gibt drei verschiedene Arten von Aktivitäten: **Task**, **Sub-Prozess** und **Call Activity**.

Der **Task** ist eine atomare Aktivität und kann nicht weiter unterteilt werden. Erreicht ein Prozess einen Task, wird dieser entweder durch eine Person oder durch ein System ausgeführt. Hier zeigt sich, dass es verschiedenen Typen von Tasks gibt. Abbildung 5.5 zeigt diese mit ihren jeweiligen Icons.

- **Service-Task:** Ein Service-Task ist ein Task, der von einem System ausgeführt wird. Der Task wird beendet, wenn das aufgerufene System ihn als abgeschlossen zurückmeldet.
- **Send-Task:** Ein Send-Task sendet eine Nachricht an einen externen Prozessbeteiligten. Der Task wird mit dem Versenden der Nachricht beendet.
- **Receive Task:** Ein Receive-Task wartet auf eine Nachricht von einem externen Prozessbeteiligten. Der Task wird beendet, wenn die Nachricht empfangen wurde.

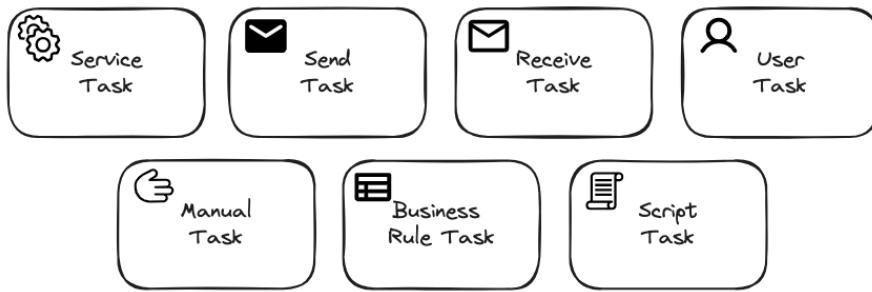


Abbildung 5.5: Die verschiedenen Typen von Tasks in BPMN 2.0

- **User-Task**: Ein User-Task ist ein Task, der von einer Person ausgeführt wird. Der Task wartet so lange, bis die Person ihn als abgeschlossen meldet.
- **Manual-Task**: Wie beim User-Task wird mit dem Manual-Task ausgedrückt, dass eine Person den Task ausführt. Der Task wird sofort abgeschlossen, es wird nicht auf eine Eingabe des Users gewartet.
- **Business-Rule-Task**: Ein Business-Rule-Task führt eine Geschäftsregel aus. Hierfür wird eine *Business Rule Engine* benötigt, die eine Eingabe erwartet und eine Ausgabe zurückgibt, entsprechend der vorher definierten Regeln. Der Task stellt somit eine Schnittstelle zwischen dem Prozess und der Business Rule Engine dar.
- **Script-Task**: Mit dem Script-Task, kann ein Script definiert werden, dass durch die Process Engine direkt ausgeführt wird. Somit muss das Script in einer programmiersprache geschrieben werden, die von der Process Engine unterstützt wird.

Ein **Sub-Prozess** ist ein Prozess innerhalb eines Prozesses und daher eine nicht-atomare Aktivität. In einem Sub-Prozess können dieselben Elemente wie in einem „normalen“ Prozess vorkommen. Für den übergeordneten Prozess wird der Sub-Process wie ein Task betrachtet. Abbildung 5.6 zeigt die zwei Arten der Darstellung eines Sub-Process. Sub-Prozesse eignen sich um ein Abbruchkriterium für mehrere Tasks zu definieren (Kapitel 5.3.2) oder wenn mithilfe des *Ad hoc* Sub-Prozesses innerhalb der BPMN Konventionen ein unstrukturierter Ablauf dargestellt werden soll. Zwei besondere Arten des Sub-Prozesses ist die *Transaktion* und der *Event-Sub-Prozess*.

Die **Call Activity** ist eine Aktivität, die einen anderen Prozess aufruft. Dadurch können wiederverwendbare Prozesse definiert und in anderen Prozessen aufgerufen werden.

5.3.2 Event

Ein Event ist ein Ereignis, das den Prozess beeinflusst. Es gibt drei Arten von Events:

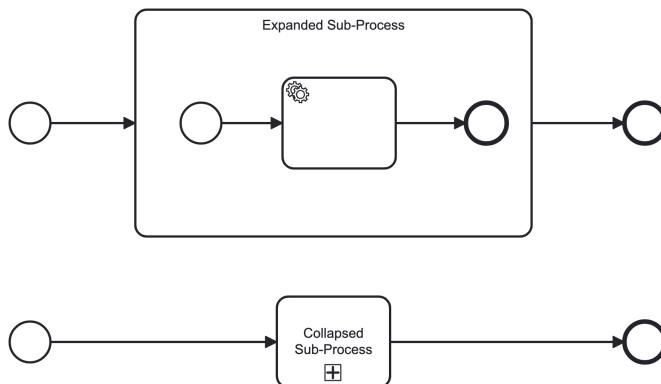
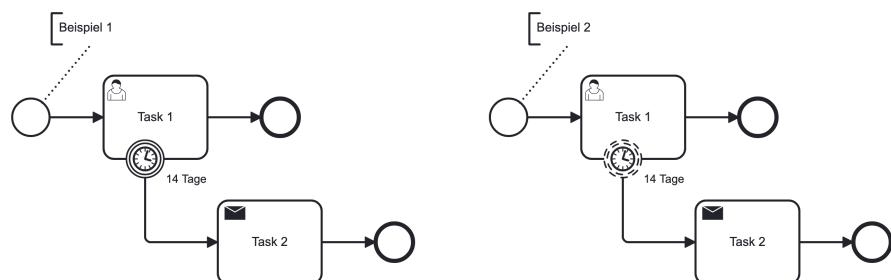


Abbildung 5.6: Die verschiedene Darstellungen eines Sub-Prozesses in BPMN 2.0

Abbildung 5.7: Beispiel für *interrupting* und *non-interrupting* Boundary-Events

- **Start-Event:** Der Startpunkt eines Prozesses.
- **Intermediate-Event:** Ein Event, das während der Ausführung eines Prozesses auftritt.
- **End-Event:** Der Endpunkt eines Prozesses.

Um Events besser beschreiben zu können, muss kurz auf das Tokenprinzip eingegangen werden. Ein Token ist eine Markierung, die den aktuellen Prozessschritt anzeigt. Ein Token wird beim **Start-Event** erzeugt und fließt durch den Prozess, bis es durch ein **End-Event** verbraucht wird. Stößt ein Token auf ein **Intermediate-Event**, wird es dort entweder angehalten, bis das Event „gefangen“ wird, oder es „schmeißt“ ein Event, das von einem anderen Prozessschritt konsumiert werden kann. Zusätzlich können Intermediate-Events an eine Aktivität angeheftet werden. Diese sogenannten Boundary-Events können nur ausgelöst werden, solange das Token sich auf der Aktivität befindet. Start Events und Boundary-Events können Token konsumieren (*interrupting*) oder nicht (*non-interrupting*). Das ist am besten anhand eines Beispiels zu erklären.

Abbildung 5.7 zeigt zwei Beispiele eines Boundary-Timer-Events. Das erste Boundary Timer-Event ist *interrupting*. Dadurch wird das Token, das sich auf der Aktivität *Task 1* befindet konsumiert. Das hat zur Folge, dass *Task 2* nur einmal ausgeführt werden kann.

Types	Start			Intermediate				End
	Top-Level	Event Sub-Process Interrupting	Event Sub-Process Non-Interrupting	Catching	Boundary Interrupting	Boundary Non-Interrupting	Throwing	
None	○						○	○
Message	✉	✉	✉	✉	✉	✉	✉	✉
Timer	⌚	⌚	⌚	⌚	⌚	⌚		
Error		⚠			⚠			⚠
Escalation		Ⓐ	Ⓐ		Ⓐ	Ⓐ	Ⓐ	Ⓐ
Cancel					☒			☒
Compensation		⟳			⟳		⟳	⟳
Conditional	☰	☰	☰	☰	☰	☰		
Link				☲			☲	
Signal	△	△	△	△	△	△	△	△
Terminate								●
Multiple	pentagon	pentagon	pentagon	pentagon	pentagon	pentagon	pentagon	pentagon

Abbildung 5.8: Die verschiedenen Typen von Events in BPMN 2.0 [31, Tab. 10.93]

Im zweiten Beispiel wird das Token auf *Task 1* nicht konsumiert. Das *non-interrupting Boundary-Event* erzeugt alle 14 Tage ein neues Token und führt *Task 2* aus.

Ähnlich zu den Tasks gibt es auch verschiedene Typen von Events. Dabei kann aber nicht jeder Typ als Start-Event, Intermediate-Event oder End-Event genutzt werden. Abbildung 5.8 gibt eine komplette Übersicht über die Event-Typen. Eine genauere Beschreibung der einzelnen Typen kann der Spezifikation [31] entnommen werden.



Abbildung 5.9: Die verschiedenen Typen von Gateways in BPMN 2.0

5.3.3 Gateway

Das Gateway kontrolliert den Sequenzfluss eines Prozesses. Trifft ein Token auf ein Gateway, kann es in Abhängigkeit von den Bedingungen des Gateways unterschiedliche Wege nehmen. Verschiedene Arten von Gateway erlauben es auch das eingehende Token zu duplizieren (*split*) oder zu kombinieren (*merge*). Abbildung 5.9 zeigt die verschiedenen Typen von Gateways.

- **Exclusive Gateway:** Hat ein Exclusive Gateway mehrere ausgehende Sequenzflüsse, wird nur einer davon ausgewählt. Es symbolisiert somit das logische *XOR*. Hat das Gateway mehrere eingehende Sequenzflüsse, wird das ankommende Token ohne weiteres an den ausgehenden Sequenzfluss weitergeleitet (*simple merge*).
- **Parallel Gateway:** Wie der Name schon sagt, werden bei einem Parallel Gateway alle ausgehenden Sequenzflüsse parallel ausgeführt. Das bedeutet dieses Gateway kann aus einem eingehenden Token mehrere ausgehende Tokens erzeugen. Dieses Gateway kann auch dafür genutzt werden, um mehrere eingehende Tokens zu einem Token zu kombinieren (*merge*). Dafür muss sich die Process Engine merken, wie viele Tokens erstellt wurden, da erst beim Eintreffen aller Tokens der Prozess weitergeführt werden kann.
- **Inclusive Gateway:** Das Inclusive Gateway stellt das logische *OR* dar. Die ausgehenden Sequenzflüsse werden basierend auf den Bedingungen des Gateways ausgewählt. Anders zum Exclusive Gateway müssen daher alle Bedingungen geprüft werden.
- **Complex Gateway:** Das Complex Gateway wird genutzt, wenn sich komplexe Kontrollstrukturen nicht durch eine einfache Bedingung oder eine Kombination von Bedingungen darstellen lassen.
- **Event-Based Gateway:** Das Event-Based Gateway wird genutzt, wenn der weitere Sequenzfluss vom eintreten eines Events abhängig ist und nicht von einer Bedingung.

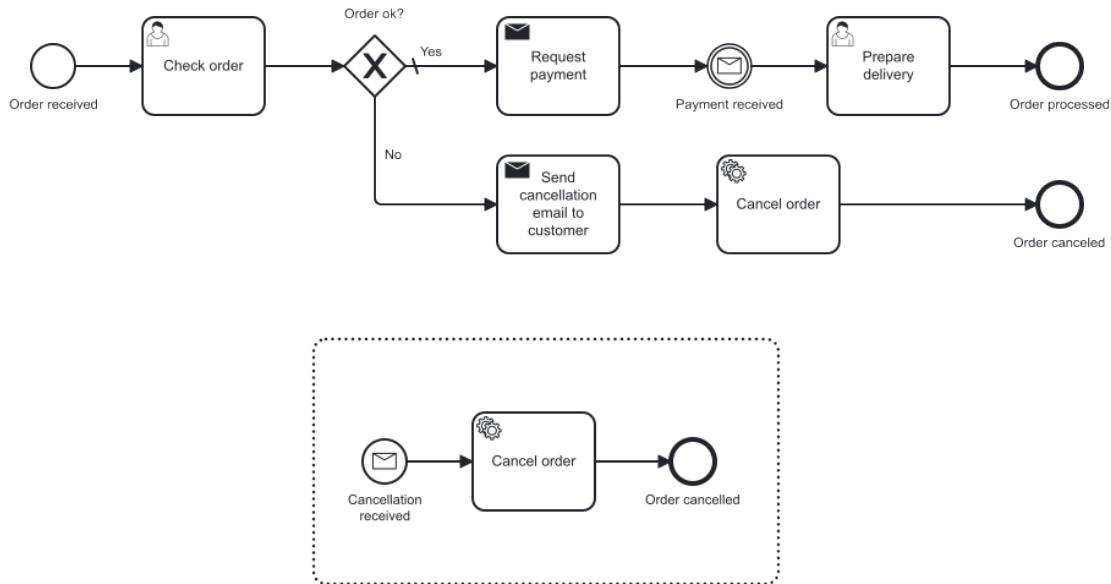


Abbildung 5.10: Vereinfachter Bestellprozess als BPMN Diagramm

5.4 Vorstellung der Prozessmodelle

In den vorherigen Abschnitten wurden die wichtigsten Elemente von BPMN 2.0 vorgestellt sowie eine Übersicht über Process Engines gegeben. OMG [31] spezifiziert noch weitere Elemente, die jedoch für den PoC nicht relevant sind. Im PoC wird ein stark vereinfachter Order-to-Cash-Prozess verwendet. Der Fokus liegt auf den technischen Aspekten und nicht auf den fachlichen. Der Prozess wird in zwei Teile aufgeteilt⁴.

Abbildung 5.10 zeigt den **Bestellprozess**. Der Prozess beginnt mit dem Eintreffen einer Bestellung. Mit dem Start-Event wird eine Prozessvariable *orderId* initialisiert. Dadurch ist die Prozessinstanz mit der Bestellung verknüpft. Daraufhin wird die Bestellung durch eine Person geprüft. Der User-Task *Check order* bekommt dafür die *orderId* übergeben, um die Bestellung aus der Datenbank zu laden und erzeugt eine Output-Variable *isOrderValid*. Anhand dieser neuen Prozessvariable prüft das Exclusive Gateway, welcher nachfolgende Pfad eingeschlagen wird. Ist die Bestellung gültig, wird der Send-

⁴Die Prozesse können auch als BPMN Diagramme im Repository gefunden werden.

Bestellprozess: <https://github.com/Miragon/ma-zeebe-taskmanagement/blob/b2c19c0550352006cdf9b495de7963fd64f03281/processes/order/src/main/resources/bpmn/order.bpmn>

Bezahlprozess: <https://github.com/Miragon/ma-zeebe-taskmanagement/blob/b2c19c0550352006cdf9b495de7963fd64f03281/processes/payment/src/main/resources/bpmn/payment.bpmn>

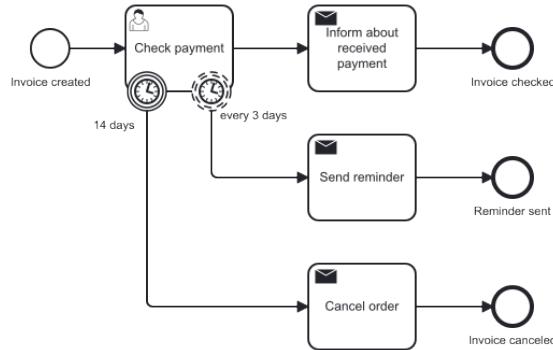


Abbildung 5.11: Vereinfachter Bezahlprozess als BPMN Diagramm

Task **Request payment** ausgeführt. Nach dem Senden der Nachricht wandert der Token zu einem Intermediate Receive-Event weiter und wartet auf den Zahlungseingang.

Die gesendete Nachricht enthält die *orderId* sowie den fälligen Zahlungsbetrag und startet den **Bezahlprozess**, der in Abbildung 5.11 dargestellt ist. Es wäre auch möglich statt einem Send-Task eine Call Activity oder auch ein Sub-Prozess aufzurufen. Durch die Nutzung des Send-Tasks wird eine klare Trennung zwischen den Domänen geschaffen und die Koppelung zwischen den Prozessen reduziert. Mit dem Start-Event wird eine Rechnung erstellt und die Prozessvariable *invoiceId* initialisiert. Der User-Task **Check payment** prüft den Zahlungseingang. An dem User-Task hängen zwei Boundary-Timer-Events. Eines ist *interrupting* und wird nach 14 Tagen ausgelöst. Das andere ist *non-interrupting* und wird alle drei Tage ausgelöst. Während das *non-interrupting* Event lediglich eine E-Mail an den Kunden sendet, wird bei dem *interrupting* Event die Bestellung storniert.

Die Stornierung wird auch über einen Send-Task realisiert, der den Sub-Prozess im Bestellprozess aufruft. Der Sub-Prozess startet mit einem *interrupting* Message-Event, das zur Folge hat, dass das Token, das auf den Zahlungseingang wartet verworfen und der Prozess mit der Stornierung beendet wird. Ist die Zahlung eingetroffen, wird der Bestellprozess benachrichtigt und der Bezahlprozess beendet. Das Token im Bestellprozess wandert einen Schritt weiter zum nächsten User-Task **Prepare delivery**. Wird dieser vom Benutzer abgeschlossen, wird der Prozess beendet.

6 Task Management

Im Kontext von BPM wird häufig von Prozessautomatisierung gesprochen. Dadurch kann der Eindruck entstehen, dass der Mensch in diesem Prozess keine Rolle spielt. Das ist nur selten der Fall. BPM-Plattformen wie Camunda oder Flowable bieten daher eine Komponente für das Task-Management an. Diese Komponente besteht aus einer UI, die die Aufgaben anzeigt, die von einem Wissensarbeiter bearbeitet werden müssen. Im Folgenden wird diese UI als *Task-Liste* bezeichnet. Erfordert die Bearbeitung einer Aufgabe die Eingabe von Daten, werden häufig Formulare dafür verwendet. Hersteller von BPM-Plattformen bieten häufig zusätzlich auch Low-Code Lösungen an, um die Entwicklung von Formularen zu vereinfachen. Formulare werden mit dem User-Task verknüpft und entsprechend angezeigt, wenn der Task bearbeitet wird. Sie dienen somit als eine UI für den User-Task.

6.1 User-Task

In Abschnitt 5.3.1 wurde der User-Task bereits im Kontext einer Aktivität vorgestellt. Im technischen Kontext kann der User-Task durch die Definition von Parametern wie Formularen, Benutzereingaben, und Ausführungsregeln weiter spezifiziert werden. Es wird festgelegt, welche Daten dem Benutzer präsentiert werden und welche Informationen er bereitstellen muss, um die Aufgabe abzuschließen. Diese Aufgaben sind eng mit den technischen Spezifikationen des Workflow-Management-Systems und den Schnittstellen zur Benutzerinteraktion verknüpft. Sobald eine Workflow-Engine auf einen User-Task im Prozessablauf trifft, wird eine Reihe von Schritten eingeleitet, um die Interaktion mit einem menschlichen Benutzer zu ermöglichen. Technisch betrachtet übernimmt die Engine dabei die Aufgabe, den Workflow zu „pausieren“ und auf eine menschliche Aktion zu warten. Der Prozess wird so lange angehalten, bis der User-Task abgeschlossen ist. Die folgenden Schritte beschreiben den Ablauf:

1. **Identifizierung des User-Tasks:** Wenn die Engine auf einen User-Task trifft, erkennt sie, dass es sich um eine menschliche Aufgabe handelt, die nicht automatisch durch Maschinen durchgeführt werden kann. An diesem Punkt pausiert sie die automatische Ausführung des Prozesses.

2. **Task-Zuweisung und Bereitstellung:** Die Engine delegiert den User-Task an ein Task-Management-System oder eine Schnittstelle, die für die Benutzerinteraktion verantwortlich ist (z.B. eine UI-Komponente oder ein externes Task-Handling-System). Workflow-Engines von BPM-Systemen wie z.B. Zeebe bieten keine direkte UI an, daher geschieht dies oft über Systeme wie *Camunda Tasklist* oder benutzerdefinierte Frontends.
3. **Datenbereitstellung:** Die Engine übermittelt alle notwendigen Prozessdaten (Input-Parameter) an das System, das den User-Task bereitstellt. Das System stellt diese Daten dem Benutzer in einer geeigneten Form (z.B. in einem Formular) dar, damit der Benutzer die Aufgabe bearbeiten kann.
4. **Warten auf Benutzereingabe:** Der Prozess bleibt in einem Wartezustand, bis der Benutzer die erforderlichen Aktionen abgeschlossen hat. Während dieser Zeit werden oft weitere Prozessschritte blockiert, da sie von der Benutzereingabe abhängig sind.
5. **Task-Abschluss:** Sobald der Benutzer die Aufgabe abgeschlossen hat, gibt das Task-Management-System eine Nachricht an die Engine zurück. Das geschieht in der Regel über einen API-Aufruf oder einen Trigger-Mechanismus, der der Engine mitteilt, dass der User-Task abgeschlossen wurde.
6. **Fortsetzen des Prozesses:** Die Engine nimmt den Prozessablauf nach dem Abschluss des User-Tasks wieder auf. Die Ergebnisse oder Ausgaben, die vom Benutzer geliefert wurden (z.B. durch ein Formular), können dabei als Output-Parameter verarbeitet und in den nächsten Prozessschritten verwendet werden. Wird z.B. die *Camunda Tasklist* mit *Camunda Forms* genutzt, stehen alle Input-Parameter des User-Tasks als Variablen zur Verfügung und die im Formular definierten Variablen werden als Prozessvariablen an den Prozess gegeben.
7. **Task-Handling-Logik:** In komplexeren Fällen kann die Engine zusätzliche Funktionen bereitstellen, wie z.B. Eskalationslogik, Wiederholungen bei Fehlern oder Zeitüberschreitungen, falls ein Benutzer die Aufgabe nicht rechtzeitig bearbeitet. Diese Logik wird oft in Kombination mit BPMN-Ereignissen wie Timern oder Fehler-Handling-Mechanismen implementiert.

6.2 Task-Liste

Die Task-Liste ist die zentrale Anlaufstelle für Aufgaben, die einen Wissensarbeiter betreffen. In den nächsten beiden Abschnitten werden die Task-Listen von Camunda [24] und Flowable [34] vorgestellt.

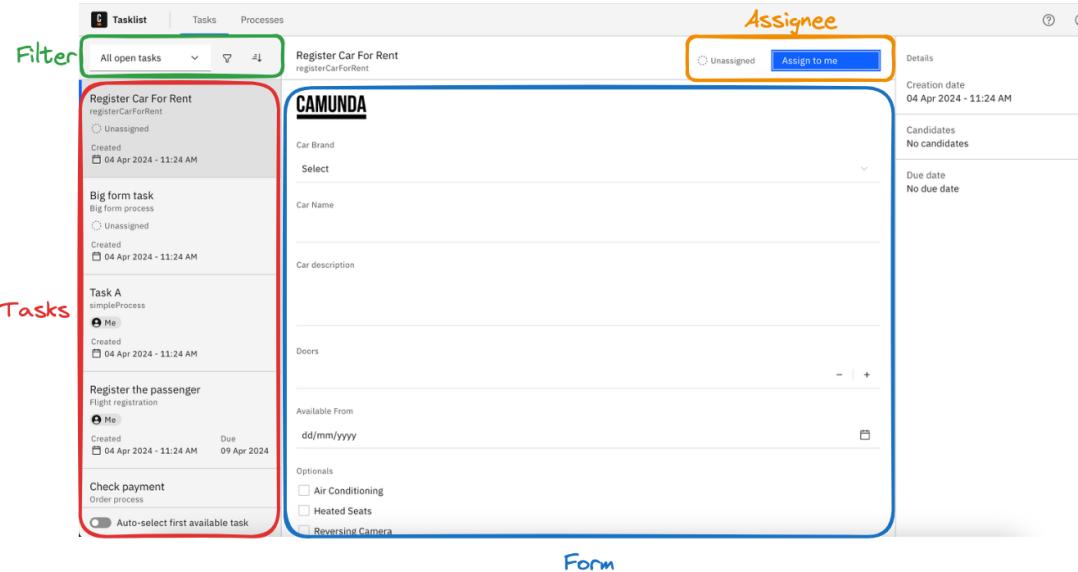


Abbildung 6.1: Screenshot der Camunda Tasklist [24, “Overview and example use case”]

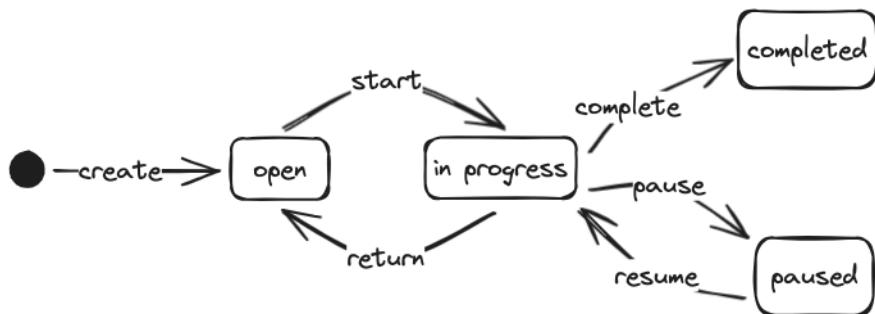


Abbildung 6.2: Mögliche Zustände und Zustandsübergänge eines User-Task

6.2.1 Camunda

Camunda bietet mit der *Tasklist* eine UI an, die in Abbildung 6.1 zu sehen ist. Dabei wurde ein einfaches und übersichtliches Design gewählt, das sowohl Tasks als auch Prozesse anzeigt. Auf der linken Seite werden die Aufgaben aufgelistet. Diese können nach verschiedenen Kriterien gefiltert werden. So können z.B. nur Aufgaben angezeigt werden, die einem zugeordnet sind oder die ein bestimmtes Fälligkeitsdatum haben. Formulare werden rechts daneben angezeigt, wenn ein Task ausgewählt wird. Ist das Formular geöffnet, kann der Wissensarbeiter sich die Aufgabe zuweisen und bearbeiten. Daran ist bereits zu erkennen, dass eine Aufgabe verschiedene Zustände haben kann. In Abbildung 6.2 ist zu sehen, welche Zustände und Zustandsübergänge möglich sind.

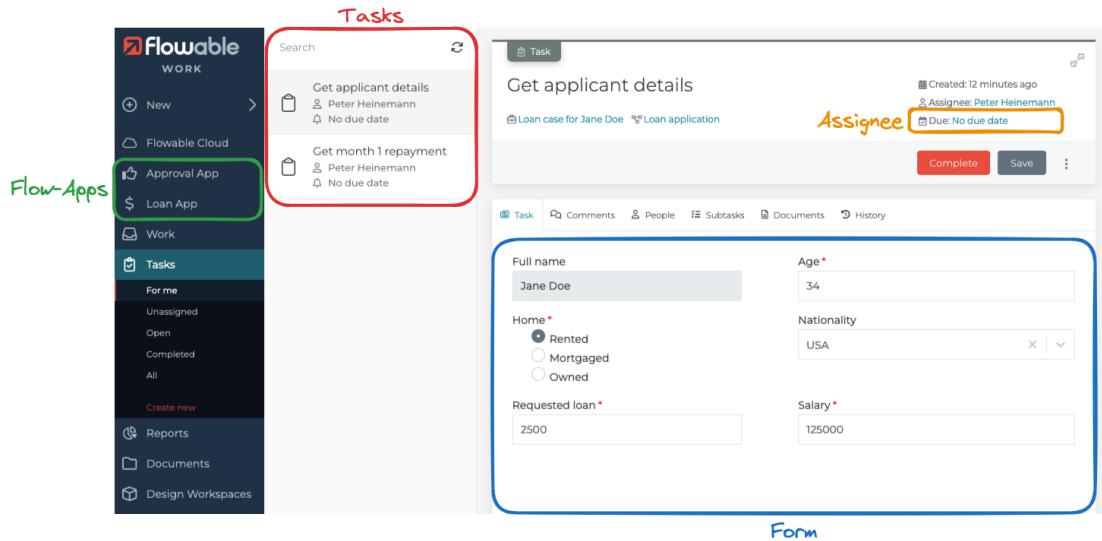


Abbildung 6.3: Screenshot von Flowable-Work

6.2.2 Flowable

Flowable sieht sich als ein „*low-code automation*“ Plattform. Das verspricht, dass auch Personen ohne Programmierkenntnisse ganze Applikationen erstellen können. Dafür gibt es das Konzept der *Flow-App*¹. Unter einer Flow-App werden alle Diagramme und Formulare zusammengefasst, die für eine Applikation notwendig sind. Eine App wird entsprechend gesondert im Arbeitsbereich „*Flowable Work*“ angezeigt. Dieser Arbeitsbereich ist in Abbildung 6.3 zu sehen. Anders als bei Camunda handelt es sich bei Flowable-Work nicht nur um eine reine Task-Liste, die Aufgaben anzeigt. Es ist vielmehr ein Arbeitsbereich, in dem auch Dokumente und Berichte erstellt werden können. Daher ist es auch möglich nicht nur einfache Formulare anzuzeigen, sondern auch komplexe UIs wie Dashboards oder Diagramme. Diese benötigen unter Umständen nicht einmal einen Bezug zu einem spezifischen Task. Einen Filter wie bei Camunda gibt es nicht. Das Filtern erfolgt über das Menü auf der linken Seite sowie über die Suchfunktion. Wie im Screenshot zu sehen kann nach „*For me*“, „*Created by me*“ usw. gefiltert werden. Formulare wiederum werden rechts daneben angezeigt. Auch das zuweisen und bearbeiten von Aufgaben ist darüber möglich.

¹<https://documentation.flowable.com/latest/develop/fe/flow-app>

6.3 Formulare

Formulare sind ein wichtiger Bestandteil des Task-Managements. Sie dienen dazu, die Eingabe von Daten zu ermöglichen. Dabei können Formulare auf zwei Arten erstellt werden. Die Nutzung von Low-Code Tools versprechen eine schnelle und einfache Erstellung von Formularen ohne Programmierkenntnisse. Pro-Code beschreibt den entgegengesetzten Ansatz, bei dem die Formulare durch Programmierer erstellt werden. Dabei sollen Frameworks wie React oder Angular helfen, um komplexe Formulare zu erstellen. In den nächsten beiden Abschnitten werden der Low-Code Ansatz vorgestellt sowie seine Grenzen besprochen. Dabei werden die Formular-Builders von Camunda, Flowable und ein Open-Source-Projekt vorgestellt, das die Technologie JSON Forms zur Formularerstellung nutzt.

6.3.1 Low-Code Tools

Low-Code Plattformen werben mit dem Versprechen, Anwendungen oder Artefakte ohne oder mit nur geringen Programmierkenntnissen erstellen zu können. Die Zielgruppe sind sogenannte **Citizen Developer** [35]. Diese sind in der Regel keine professionellen Entwickler, sondern Mitarbeiter aus Fachabteilungen mit technischem Verständnis. Sie nutzen häufig Low-Code Plattformen, die technische Details abstrahieren und die Erstellung von Anwendungen durch Drag-and-Drop oder visueller Programmierung ermöglichen.

Camunda bietet zusammen mit ihrem BPMN Modeler ein Form-Builder an. Dafür wird die eigenen Form-Library² genutzt. Die Formulare werden als JSON-Datei gespeichert und können in der Task-Liste angezeigt werden. Abbildung 6.4 zeigt den Form-Builder. Auf der linken Seite des Form-Builders können die verschiedenen Formularelemente ausgewählt und per Drag-and-Drop in den Editor gezogen werden. Der Benutzer kann zwischen vielen verschiedenen Elementen wählen, wie z.B. Textfelder, Dropdown-Menüs oder Checkboxen. Aber auch komplexere Elemente wie Tabellen oder HTML-Views sind möglich. Über HTML-Views kann eigener HTML-Code eingefügt werden, um die Darstellung weiter zu individualisieren. Zusätzlich ist auch eine iframe-Integration möglich. Hierfür muss entweder eine statische URL angegeben oder dynamisch ein Wert über eine Variable übergeben werden. Eine Konfiguration der Elemente ist ebenfalls möglich. Dadurch können z.B. Validierungsregeln oder Bedingungen hinzugefügt werden, die ein Feld z.B. dynamisch anzeigen oder verstecken. Im Preview-Bereich wird das Formular angezeigt, und es können Testdaten eingegeben werden. Das daraus entstehende

²<https://github.com/bpmn-io/form-js>

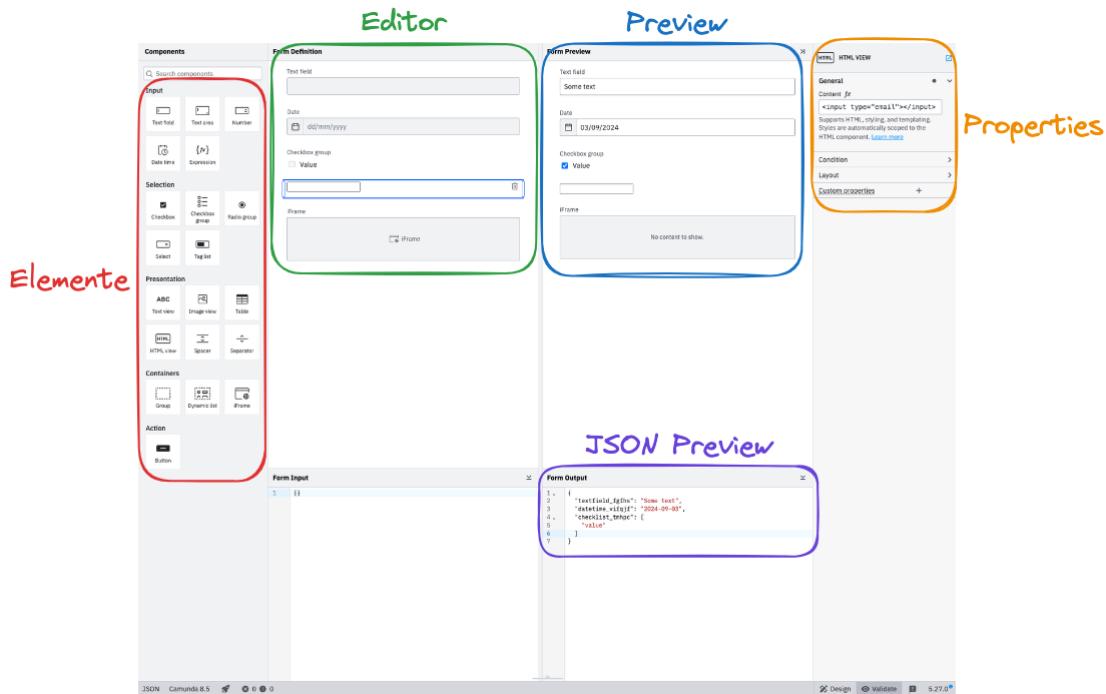


Abbildung 6.4: Screenshot des Camunda Form-Builders

JSON-Objekt wird ebenfalls angezeigt.

Der Form-Builder ist als Desktop-Anwendung verfügbar.

Flowable bietet ebenfalls einen Form-Builder an, der allerdings nur als Web-Anwendung verfügbar ist. Die Komponente wird als *Flowable-Design* bezeichnet und ermöglicht unter anderem die Erstellung von Formularen. Der Form-Builder ist in Abbildung 6.5 zu sehen. Der Form-Builder hat einen ähnlichen Aufbau wie der von Camunda. Allerdings gibt es weitaus mehr Elemente, die zur Verfügung stehen. Dabei ist es schwer den Überblick zu behalten. Die Zielgruppe sind Citizen Developer, die Kenntnisse in HTTP/REST haben müssen um alle Funktionen nutzen zu können. Über den *REST-Button* oder Tabellen können Daten aus der Datenbank geholt werden. Dadurch ist es möglich komplexere Dashboards zu erstellen, die z.B. alle Tasks eines Mitarbeiters anzeigen. Benutzer müssen sich in der Regel in die API-Dokumentation einlesen, um die relevanten Endpoints zu finden. Der Form-Builder ist weniger intuitiv Nutzbar und benötigt mehr Einarbeitungszeit als der von Camunda. Eine Preview kann über einen *Preview-Button* angezeigt werden.

Miranum IDE ist eine Open-Source-Entwicklungsumgebung (IDE), die von der Firma Miragon entwickelt wurde. Sie besteht aus einer Sammlung von Plugins für Visual Studio Code (VS Code) und wurde entworfen, um die Verwaltung und Bearbeitung von Artefakten im Zusammenhang mit Prozessanwendungen zu vereinfachen.

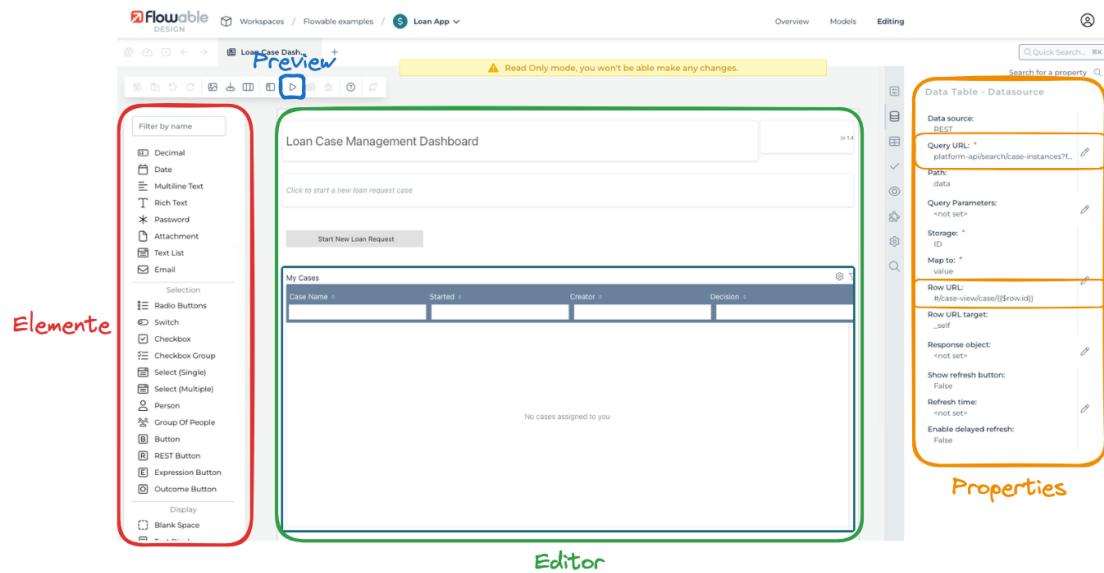


Abbildung 6.5: Screenshot des Flowable Form-Builders

Für den PoC wurden im Speziellen zwei Plugins genutzt. Zum einen der *Miranum Modeler*. Der Modeler ermöglicht es in VS Code BPMN-Modelle, speziell für die Workflow-Engines von Camunda, zu erstellen und zu bearbeiten. Das zweite Plugin *Miranum JSON Forms* wird verwendet, um Formulare basierend auf JSON Forms³ per Drag-and-Drop zu erstellen. In Abbildung 6.6 ist der Aufbau des Editors zu sehen. Die Formularelemente sind in *Controls* und *Layouts* unterteilt. Die *Controls* beinhalten die verschiedenen Formularelemente, wie z.B. Textfelder, Dropdown-Menüs oder Checkboxen. Jedes Feld kann weiter konfiguriert werden, indem auf ein Zahnradsymbol geklickt wird, das angezeigt wird wenn über das Feld „gehovert“ wird. Die Formularelemente können ausgewählt werden und per Drag-and-Drop in den Editor gezogen werden. Zusätzlich gibt es eine Vorschau, die die Änderungen in Echtzeit anzeigt und das Testen des Formulars ermöglicht. Dabei wird das JSON-Objekt sowie Validierungsfehler angezeigt, die beim Ausfüllen des Formulars entstehen.

JSON Forms ist ein Open-Source Framework, das es Entwicklern ermöglicht, Formulare dynamisch auf Basis von JSON-Datenstrukturen zu generieren. Es nutzt JSON Schema, um die Struktur und Validierung von Formulareingaben zu definieren, und UI Schemas, um das Layout und die Anordnung der Formularelemente zu bestimmen. Das Ziel von JSON Forms ist es, die Erstellung und Pflege von Formularen zu vereinfachen, indem die Logik zur Darstellung von der Datenstruktur und Validierung getrennt wird.

³<https://jsonforms.io/>

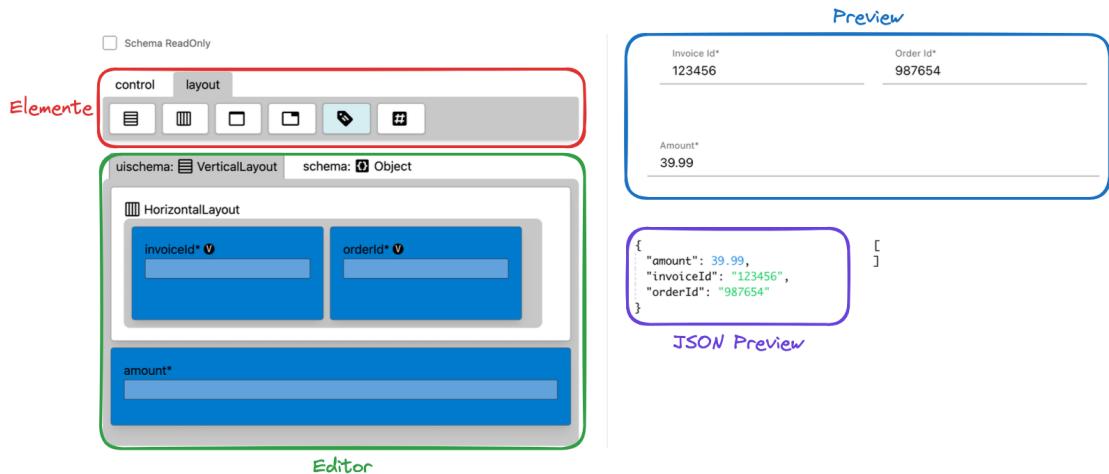


Abbildung 6.6: Miranum JSON Forms Editor in der Miranum IDE.

JSON Schema ist eine standardisierte Spezifikation, die dazu dient, die Struktur eines JSON-Dokuments zu beschreiben. Es wird verwendet, um die Datentypen, Wertebereiche und Validierungsregeln eines JSON-Objekts festzulegen. In JSON Forms dient das JSON Schema dazu, die Struktur und Validierung der Daten zu bestimmen, die durch das Formular eingegeben werden sollen.

Das **UI Schema** bestimmt wie die Formularelemente auf der Benutzeroberfläche dargestellt werden. Das UI Schema beeinflusst die Reihenfolge der Felder, Gruppierung oder Tabs und spezifische Darstellung von bestimmten Feldern, wie z.B. Dropdown-Menüs oder Checkboxen. Das UI Schema ist flexibel und kann unabhängig vom JSON Schema angepasst werden, um das gewünschte Layout oder die Benutzererfahrung zu ermöglichen.

Abbildung 6.7 zeigt ein Beispiel einer JSON Schema Datei und dem dazugehörigen UI Schema. Das UI Schema arbeitet Hand in Hand mit dem JSON Schema, um die Darstellung zu kontrollieren. In diesem Beispiel beinhaltet die Datenstruktur die Attribute *invoiceId*, *orderId*, *amount* und *isAccepted*. Diese Felder werden entsprechend ihres Typs im Formular dargestellt. Eine *number* wird als numerisches Eingabefeld dargestellt, der *string* als Textfeld, während ein *boolean* als Checkbox angezeigt wird. Mit zusätzlichen Optionen wie z.B. *minLength* kann die Validierung der Eingaben weiter spezifiziert werden. Das UI Schema definiert die Anordnung der Felder. *VBoxLayout* sorgt dafür, dass die Felder untereinander angeordnet werden und *HorizontalLayout* platziert die Felder nebeneinander. Das Attribut *scope* verweist auf die entsprechenden Felder im JSON Schema und mit *options* kann die Darstellung weiter angepasst werden. Im Beispiel sind alle Felder als *readonly* markiert, was bedeutet, dass sie nicht bearbeitet werden können. Zusätzlich kann für jedes Feld ein *label* definiert werden, das als Beschriftung

The diagram illustrates the relationship between JSON Schema and UI Schema. On the left, a box labeled "JSON Schema" contains the following code:

```

1 {
2   "$schema": "http://json-schema.org/draft-07/schema#",
3   "type": "object",
4   "properties": {
5     "amount": {
6       "type": "number"
7     },
8     "invoiceId": {
9       "type": "string",
10    "minLength": 1
11  },
12  "isAccepted": {
13    "type": "boolean"
14  },
15  "orderId": {
16    "type": "string",
17    "minLength": 1
18  }
19 },
20 "required": [ "amount", "invoiceId", "orderId" ]
21 }

```

On the right, a box labeled "UI Schema" contains the following code:

```

1 {
2   "type": "VerticalLayout",
3   "elements": [
4     {
5       "type": "Control",
6       "scope": "#/properties/invoiceId",
7       "options": {
8         "readonly": true
9       }
10      },
11      {
12        "type": "Control",
13        "scope": "#/properties/orderId",
14        "options": {
15          "readonly": true
16        }
17      },
18      {
19        "type": "HorizontalLayout",
20        "elements": [
21          {
22            "type": "Control",
23            "scope": "#/properties/amount",
24            "options": {
25              "readonly": true
26            }
27          },
28          {
29            "type": "Control",
30            "scope": "#/properties/isAccepted",
31            "label": "Valid"
32          }
33        ]
34      }
35    ]
36  }

```

Abbildung 6.7: Ein Beispiel für eine JSON Schema und UI Schema Datei.

The screenshot shows a rendered JSON Form with three fields:

- Invoice Id ***: A text input field containing the value "7d6ba0e5-0b43-4594-8d5d-5ee2141dbec".
- Order Id ***: A text input field containing the value "02d8b5d1-fdde-4d31-b4c5-c785612c9bd".
- Amount ***: A numeric input field containing the value "11", with a dropdown arrow next to it. To its right is a checkbox labeled "Valid" which is unchecked.

Abbildung 6.8: Das gerenderte Formular als Ergebnis der beiden JSON Dateien aus Abbildung 6.7.

für das Feld dient. Das Ergebnis ist ein Formular, wie es in Abbildung 6.8 dargestellt ist.

JSON Forms bietet somit eine elegante Lösung zur Generierung von Formularen auf Basis von JSON Schemas und UI Schemas. Während JSON Schemas die Struktur und Validierung der Formulareingaben definieren, bietet das UI Schema eine flexible Möglichkeit, das Layout und die Darstellung der Formularelemente zu steuern. Zusammen ermöglichen sie es Entwicklern, dynamische, validierte Formulare zu erstellen, die sich leicht anpassen und erweitern lassen. Die Trennung von Datenstruktur (JSON Schema)

und Darstellung (UI Schema) sorgt für eine klare und wartbare Formulare.

6.3.2 Die Grenzen von Low-Code

Im vorherigen Abschnitt wurden Low-Code Tools vorgestellt, die es ermöglichen, Formulare ohne Programmierkenntnisse zu erstellen. Flowable bietet den Editor mit den meisten Funktionen an. Dadurch benötigt der Benutzer mehr Einarbeitungszeit und muss sich auch mit technischen Details vertraut machen.

Für die Miranum-IDE als auch den Camunda Form-Builder ist weniger technisches Verständnis notwendig. Allerdings muss der Benutzer auch hier zumindest die Grundlagen von JSON kennen, und im Fall von Camunda die FEEL-Notation beherrschen. Während sich Flowable als Low-Code Plattform versteht, möchte Camunda eher Entwickler ansprechen [5, Kap. 1]. Für Ruecker [5, Kap. 1] war das Versprechen von Low-Code BPM-Plattformen, Prozessautomatisierung ohne Entwicklungserfahrung direkt durch Fachabteilungen zu ermöglichen, ein Trugschluss.

Vor allem, wenn komplexe Prozesse implementiert werden sollen stoßen Low-Code Tools an ihre Grenzen. Das führt dazu, dass Unternehmen, die Low-Code Tools eingeführt haben auf Entwickler zurückgreifen, die dann sich mit den Tools auseinandersetzen müssen. Dies erweist sich oft als langwieriger und frustrierender Prozess.

Aufgrund dieser Grenzen wurde im Prototyp darauf geachtet, dass auch ein Pro-Code Ansatz unterstützt wird. Formulare sollen dadurch durch Entwickler mit Kenntnissen in der Programmierung von Frontend-Anwendungen und User Experience (Ux) erstellt werden.

7 Camunda Platform 7 vs. 8

Camunda Platform 7 (C7) ist eine beliebte BPM-Platform und wird von vielen Unternehmen genutzt. Mittlerweile gibt es mit Camunda Platform 8 (C8) einen Nachfolger, der jedoch einige Änderungen mit sich bringt. Im Folgenden wird ein Vergleich zwischen den beiden Versionen gezogen. Dabei wird auf die Unterschiede der beiden Engines eingegangen und dadurch die Zeebe-Engine genauer vorgestellt. Die Informationen stammen aus den offiziellen Dokumentationen von Camunda [18], [24].

7.1 Übersicht Camunda Platform 7

C7 ist eine Open-Source-Plattform für das Workflow- und Geschäftsprozessmanagement. Sie unterstützt die Modellierung und Automatisierung von Geschäftsprozessen und ist besonders in Java-basierten Unternehmensanwendungen verbreitet. Die Plattform bietet eine leistungsstarke Engine zur Ausführung von BPMN 2.0-Prozessen. C7 wurde ursprünglich 2013 veröffentlicht. Seitdem wurden kontinuierlich Verbesserungen und neue Versionen eingeführt. Die Plattform entsprang aus dem Open-Source-Projekt Activiti¹. Jedoch hat Camunda angekündigt, den Support im Jahr 2027 einzustellen². Die Architektur von C7 weist die folgenden Hauptkomponenten auf:

1. **Workflow Engine:** Das Herzstück von Camunda. Die Engine führt BPMN-Prozesse aus und unterstützt Decision Management (DMN).
2. **Cockpit:** Eine webbasierte Administrations- und Monitoring-Oberfläche, um Prozesse zu überwachen, Instanzen zu verwalten und Fehler zu beheben.
3. **Tasklist:** Eine Benutzerschnittstelle für menschliche Interaktionen im Prozess, in der Benutzer ihre zugeordneten Aufgaben einsehen und bearbeiten können.

¹<https://github.com/Activiti/Activiti>

²<https://docs.camunda.org/enterprise/announcement/>

7.2 Übersicht Camunda Platform 8

C8 ist die Nachfolgeversion, die C7 ablöst. Im Gegensatz zu ihrem Vorgänger konzentriert sich C8 stärker auf Cloud-native Architekturen und bietet eine verbesserte Skalierbarkeit. Die Plattform ermöglicht weiterhin die Modellierung und Automatisierung von Geschäftsprozessen, bietet jedoch eine modernisierte Architektur, die besonders auf verteilte Systeme und microservice-basierte Umgebungen ausgerichtet ist.

Ein zentrales Element von C8 ist die Zeebe-Engine, die für eine hochgradig skalierbare und fehlertolerante Ausführung von BPMN-Prozessen konzipiert wurde. Im Vergleich zur vorherigen Workflow Engine aus C7 ist Zeebe darauf ausgelegt, eine große Anzahl von Prozessinstanzen parallel zu verarbeiten, ohne dabei die Performance zu beeinträchtigen. Dadurch ergibt sich eine native Unterstützung von Event-getriebenen Architekturen, die insbesondere in modernen Cloud-Anwendungen eine große Rolle spielen.

Die wichtigsten Komponenten von C8 umfassen:

1. **Zeebe (Workflow Engine):** Eine verteilte und fehlertolerante BPMN-Engine, die für den Einsatz in Microservices-Architekturen optimiert wurde. Sie unterstützt die Modellierung und Ausführung von BPMN 2.0-Prozessen.
2. **Tasklist:** Wie bei C7 bietet auch C8 eine Benutzerschnittstelle, die menschliche Interaktionen in Prozessen ermöglicht. Benutzer können ihre zugeordneten Aufgaben einsehen und bearbeiten.
3. **Operate:** Ein mächtiges Monitoring-Tool, das den Betriebsstatus der laufenden Prozesse visualisiert. Es bietet Einblicke in den Zustand von Prozessinstanzen und ermöglicht die Analyse und Fehlerbehebung bei Fehlfunktionen.
4. **Optimize:** Eine Komponente, die Reporting und Analyse von Prozessen ermöglicht. Sie bietet eine grafische Oberfläche zur Erstellung von Berichten und Dashboards, um die Performance und Effizienz von Geschäftsprozessen zu überwachen.
5. **Connectors:** In C8 gibt es vorgefertigte Konnektoren, die es ermöglichen, Prozesse einfach mit externen Services und APIs zu integrieren. Dies vereinfacht die Anbindung an verschiedene Systeme innerhalb einer verteilten Architektur.

7.3 Vergleich der Engines

In diesem Abschnitt wird noch etwas detaillierter auf die Unterschiede der beiden Engines eingegangen. Dabei werden die Änderungen an der Architektur der Engines genauer betrachtet und wie diese die Skalierbarkeit und Performance beeinflusst. Anschließend wird noch speziell auf die Auswirkungen auf das Task-Management eingegangen.

7.3.1 Architektur

Die Architektur von **C7** basiert auf einer BPMN-Engine, die in einer Java-basierten Umgebung ausgeführt wird [18, "Architecture Overview"]. Die Engine ist synchron und wird normalerweise wie bereits erwähnt in monolithischen oder containerisierten Anwendungen integriert. Die REST API ermöglicht den Zugriff auf die Workflow Engine über HTTP, um von externen Systemen aus, auf Prozessinstanzen zuzugreifen. C7 verwendet eine relationale Datenbank für das Prozessmanagement und die Speicherung der Prozessinstanzen. Jede Prozessausführung ist dabei synchron, was bedeutet, dass der Prozess und die Datenbank ständig kommunizieren, um den aktuellen Status zu speichern und zu aktualisieren. Das führt dazu, dass die Performance stark von der Leistung der Datenbank abhängt und das System bei hohen Lasten an Skalierungsgrenzen stößt. Um die Performance zu optimieren, müssen oft Maßnahmen wie Datenbank-Sharding oder Lastverteilung implementiert werden.

Die Engine kann direkt in einer Java-Anwendung eingebettet werden oder als eigenständiger Service betrieben werden. Somit liegt das Einsatzgebiet in monolithischen Anwendungen, die auf Java basieren als auch in Microservice-Architekturen. Am besten wird das anhand der beiden Möglichkeiten Service-Tasks zu implementieren deutlich. C7 kennt zwei verschiedene Ansätze, um Geschäftslogik in einem Geschäftsprozess auszuführen. Beide dienen dem Zweck, einen Service-Task in einem BPMN-Prozess mit spezifischer Logik zu versehen, haben jedoch grundlegende Unterschiede in ihrer Funktionsweise, Architektur und Anwendungsbereichen. Der erste Ansatz ist die Verwendung eines **JavaDelegate** [18, "Delegation Code"].

Ein JavaDelegate ist eine Klasse, die in Java geschrieben wird und die Logik direkt innerhalb der Camunda Process Engine ausführt. Es handelt sich um eine synchron ausgeführte Logik, die eng in die Engine eingebunden ist. Der zweite Ansatz ist die Verwendung eines **External Task**[18, "External Tasks"].

Ein External Task ist eine verteilte Architektur, bei der die eigentliche Ausführung der Logik außerhalb der Camunda Process Engine stattfindet. Ein externer Worker übernimmt die Aufgabe und führt die Logik unabhängig von der Engine aus.

JavaDelegate sind durch ihre enge Integration in die Engine typischer für monolithische Anwendungen, während External Tasks für verteilte Systeme und Microservices besser geeignet sind.

C8 führt eine vollständig neue Architektur ein, die Event-Streaming Mechanismen nutzt. Die Zeebe-Engine, ist asynchron und speziell für Cloud-native Umgebungen entwickelt worden. Die Engine ist als Remote-Engine konzipiert und wird über gRPC oder REST angesprochen. Diese Anfragen werden durch ein oder mehrere Gateways an die Zeebe-Broker verteilt. Die Broker stellen verteilte Workflow-Engines dar. Jedoch wird keine Geschäftslogik in den Brokern ausgeführt. *JobWorker* fragen bei den Brokern an, ob Aufgaben für sie vorhanden sind und führen die Geschäftslogik aus. Jede Statusänderung

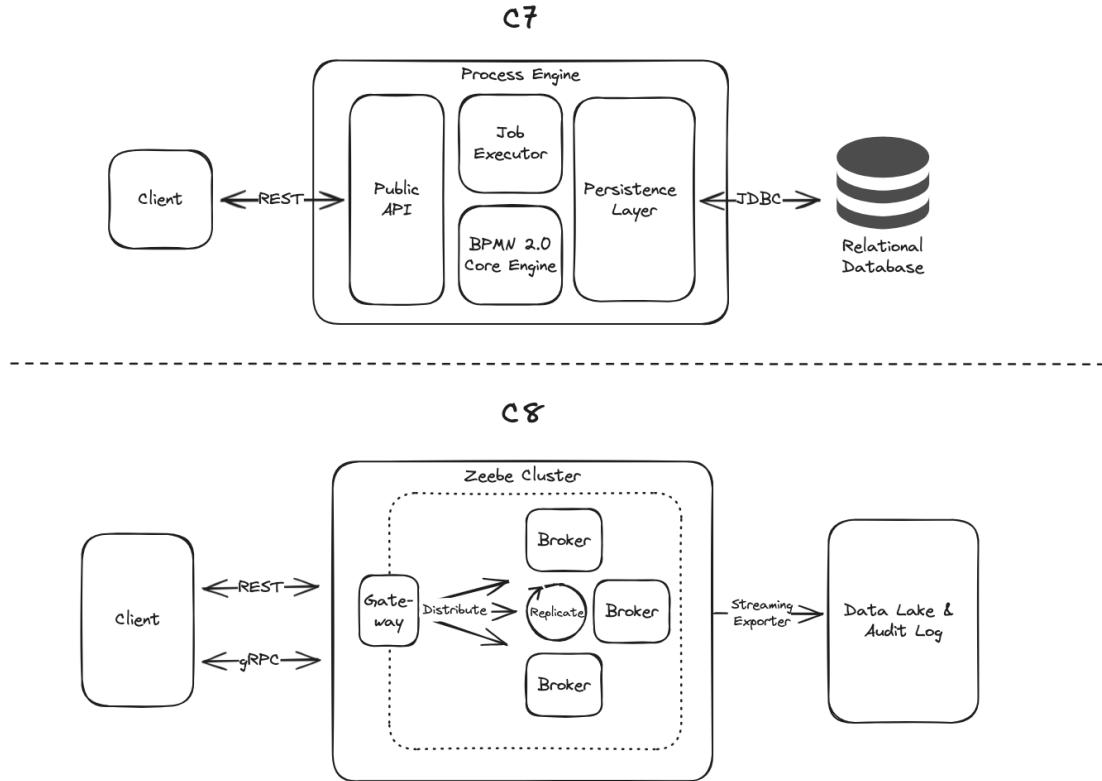


Abbildung 7.1: Architekturvergleich von C7 [18, “Architecture Overview”] und C8 [24, “Architecture”]

wird als *Record* von einem Broker in einen Event-Stream geschrieben. Dabei handelt sich um ein Append-Only Log, dessen Records unveränderlich (*immutable*) sind. Auf das Konzept der Record wird in Abschnitt 9.4.2 noch genauer eingegangen. Alle Records zu einem Workflow geben den aktuellen Zustand des Prozesses wieder. Dadurch kann jeder Broker in einem Cluster den aktuellen Zustand ermitteln. Durch den Einsatz von mehreren Brokern pro Cluster erreicht C8 seine horizontale Skalierbarkeit und Fehlertoleranz. Es ist möglich sich in diesen Event-Stream einzuhacken, um z.B. spezifische Records in eine eigene Datenbank zu schreiben. Um die ineffiziente dauerhafte Speicherung von Zustandsänderungen im Log zu umgehen werden regelmäßig Snapshots erstellt. Abbildung 7.1 zeigt die Architektur der beiden Engines im Vergleich, wenn C7 auch als eine Remote-Engine verwendet wird.

7.3.2 Auswirkungen auf das Task-Management

Diese Änderungen haben Einfluss auf das Task-Management, insbesondere auf eigene Implementierungen. Während die zentrale API und Datenbank von C7 eine einfache und direkte Abfrage von User-Tasks ermöglicht, wurde das mit C8 bzw. Zeebe komplexer. Eine API speziell nur für User-Tasks wurde in die proprietäre Tasklist-Komponente ausgelagert. Wird diese Komponente nicht benutzt, müssen Events, die User-Tasks betreffen, über Job Worker oder Exporter abgefangen und in einer eigenen Datenbank gespeichert werden. Der Entwickler muss dabei auf alle relevanten Events reagieren. Eine Task-Liste soll nur aktive Aufgaben anzeigen, somit muss grundsätzlich auf das erstellen aber auch abbrechen von Aufgaben reagiert werden. Ein Abbruch kann z.B. durch ein Boundary-Event ausgelöst werden (siehe Abbildung 5.7). Weitere Events, die für ein vollumfängliches Task-Management relevant sind, ist das delegieren und zuweisen von Aufgaben.

Wie eine Implementierung mit JobWorker als auch einen Exporter aussehen kann, wird mit dem PoC gezeigt.

8 Architektur des Proof-of-Concept

Eine komplette Übersicht der Architektur des PoC ist in Abbildung 8.1 dargestellt. Das Ziel ist es, eine leichtgewichtige und flexible Architektur zu entwickeln, die einfach zu überblicken ist. Anhand des Schaubildes wird auch die Kommunikation zwischen den Komponenten deutlich. Im Folgenden werden die einzelnen Komponenten der Architektur betrachtet. Eine detailliertere Beschreibung der Komponenten wird im nächsten Kapitel gegeben.

- **Task-Liste:** Die Task-Liste ist die UI über die Tasks angezeigt und bearbeitet werden. Sie kennt standardmäßig nur den Task-Manager, von dem sie die Tasks und Metadaten über die Prozess-Applikationen erhält. Die Task-Liste hat somit kein direktes Wissen über die Prozess-Applikationen. Dennoch muss sie in der Lage sein mit diesen zu kommunizieren. Sie kann Prozesse starten, Formulare laden, Tasks aktualisieren und abschließen.
- **Nginx:** Der Nginx-Server dient als Reverse-Proxy für die Task-Liste. Er empfängt die Anfragen der Task-Liste, leitet sie an den entsprechenden Service weiter und liefert die Antworten zurück. Dadurch können Backend-Server entlastet und Lastverteilung (Load Balancing) implementiert werden. Nginx bietet zudem Sicherheitsvorteile, wie etwa das Verbergen von Backend-Details und den Schutz vor direkten Zugriffen. Somit ist es notwendig, wenn ein neuer Service hinzugefügt wird, die Konfiguration des Nginx anzupassen. In einer Produktivumgebung könnte Nginx durch Kubernetes ersetzt werden.
- **Task-Manager:** Der Task-Manager verwaltet alle User-Tasks. Dafür muss er mit der Zeebe-Engine kommunizieren, um die Tasks zu erhalten und zu aktualisieren, falls ein Task abgebrochen wurde. Er dient auch als zentrale Einheit, über die der Task abgeschlossen wird. Schließt ein User einen Task ab, wird zunächst die Prozess-Applikation darüber informiert, die das an den Task-Manager weiterleitet. Der Task-Manager aktualisiert seine Datenbank entsprechend und schickt einen Command an die Zeebe-Engine, um den Task auch dort abzuschließen, damit der Prozess weiterlaufen kann.
- **Prozess-Applikation:** Es kann beliebig viele Prozess-Applikationen geben. Um mit der Task-Liste zu kommunizieren, muss sich eine Prozess-Applikation beim Task-Manager registrieren. Gleichzeitig muss sie REST Endpoints bereitstellen, über die die Task-Liste die Formulare laden und die Tasks aktualisiert und abschließen

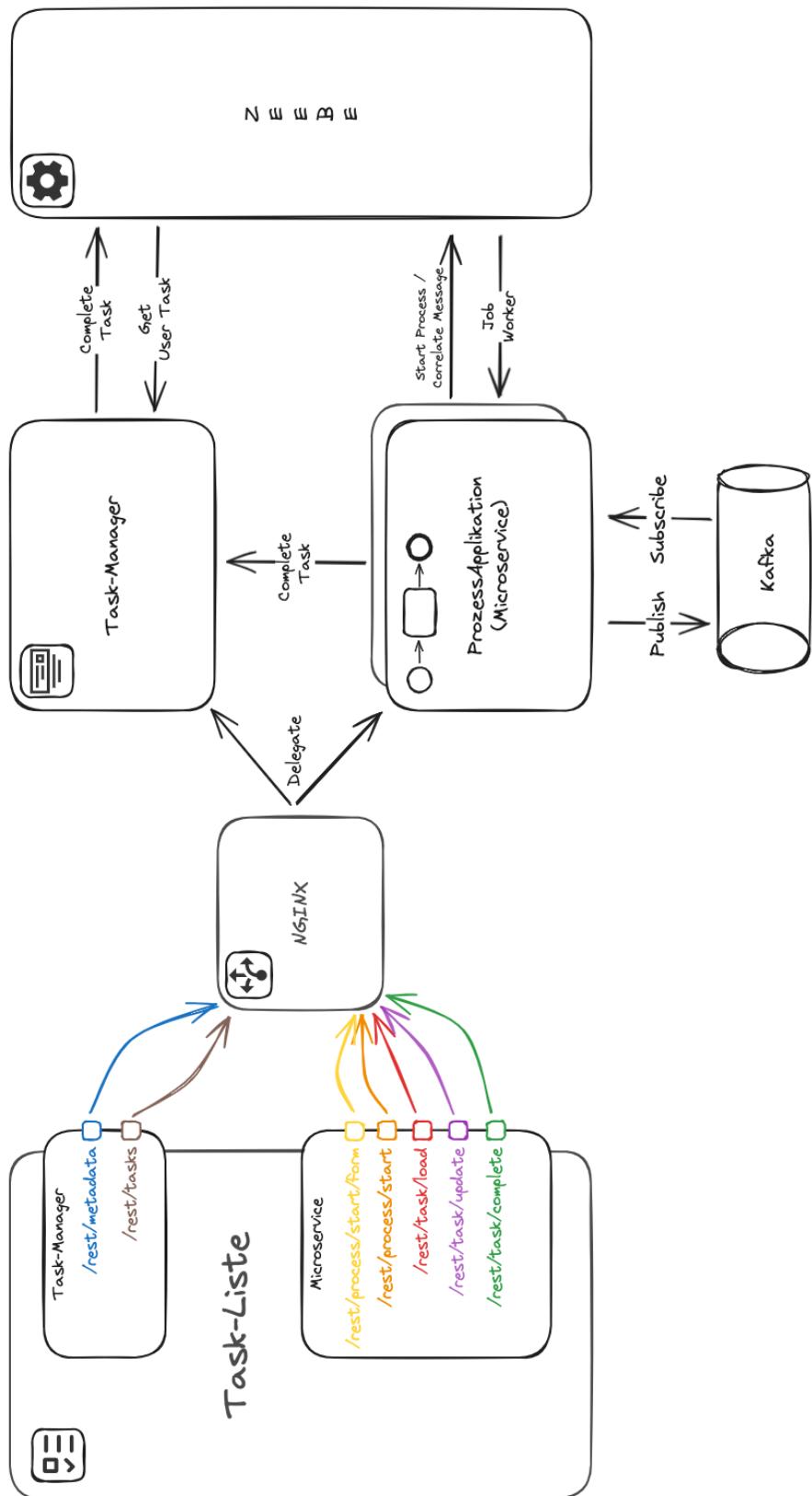


Abbildung 8.1: Architektur des PoC

kann. Das Starten von Prozessen ist optional, da nicht jeder Prozess über die Task-Liste gestartet werden muss. Muss die Anwendungen mit einer anderen Prozess-Applikation kommunizieren passiert das über einen Message-Broker. In diesem Beispiel handelt es sich dabei um Kafka. Eine Prozess-Applikation muss auch mit der Zeebe-Engine kommunizieren, um Prozesse zu starten oder auch Nachrichten zu korrelieren.

- **Kafka** (Message-Broker): Kafka ist eine Event-Streaming-Plattform, die als Kommunikationsplattform zwischen den Prozess-Applikationen dient. In Kafka werden Daten in Topics organisiert, die von Produzenten geschrieben und von Konsumenten gelesen werden. Es speichert die Nachrichten (Records) auf Festplatten und bietet somit eine persistente Speicherung und Wiederabruflbarkeit. Diese Speicherung der Records, unterscheidet Kafka von anderen traditionellen Message-Brokern wie RabbitMQ oder ActiveMQ .
- **Zeebe-Engine**: Die Zeebe-Engine ist die Workflow-Engine, die die Prozesse ausführt. Auf die Details der Zeebe-Engine wurde bereits im Kapitel 7.3 eingegangen.

In der Abbildung 8.2 wird deutlich wie der komplette Order-to-Cash Prozess auf technischer Ebene abläuft. Dabei ist nur der Happy-Path dargestellt. Das bedeutet, dass immer, wenn die Engine auf ein Gateway trifft, dem ersten Sequenzpfad gefolgt wird. Im Diagramm wurden die Nachrichtenflüsse entsprechend der Aktivitäten aus den Prozessen gruppiert. Die Prozesse wurden mit dem Abschnitt 5.4 vorgestellt.

Der Startpunkt bei einem **User-Task** ist die Zeebe-Engine, die diesen an den Task-Manager schickt. Dieses Vorgehen ist asynchron und kann unterschiedlich lange dauern. In der Task-Liste ist ein Mechanismus eingebaut, der nach einer kurzen Zeit einen neuen fetch-Request an den Task-Manager schickt, um sicherzustellen, dass neue Tasks auch angezeigt werden. Allerdings kann es auch vorkommen, dass der Benutzer manuell die Seite aktualisieren muss, um die Tasks zu sehen. Als nächstes selektiert er einen Task, in dem er auf diesen klickt. Das führt einen POST-Request an die jeweilige Prozess-Applikation aus, um das Formular und dessen Daten zu laden. Der User füllt das Formular aus und schickt es ab. Die Daten gehen zunächst an die Prozess-Applikation wo sie gespeichert werden. Anschließend sendet die Prozess-Applikation über HTTP/REST einen Command an den Task-Manager, um den Task abzuschließen. Der Task-Manager aktualisiert seine Datenbank und leitet den Command an die Zeebe-Engine weiter um den Task auch dort zu beenden.

Send-Tasks werden auch von der Zeebe-Engine durch JobWorker gestartet, die durch die Prozess-Applikationen implementiert sind. Die Prozess-Applikation sendet eine Nachricht an Kafka, die von der anderen Prozess-Applikation gelesen wird. Darauf wird entweder ein Prozess gestartet oder die Nachricht korreliert. Beides hat zur Folge, dass die konsumierende Prozess-Applikation einen Command an Zeebe sendet.

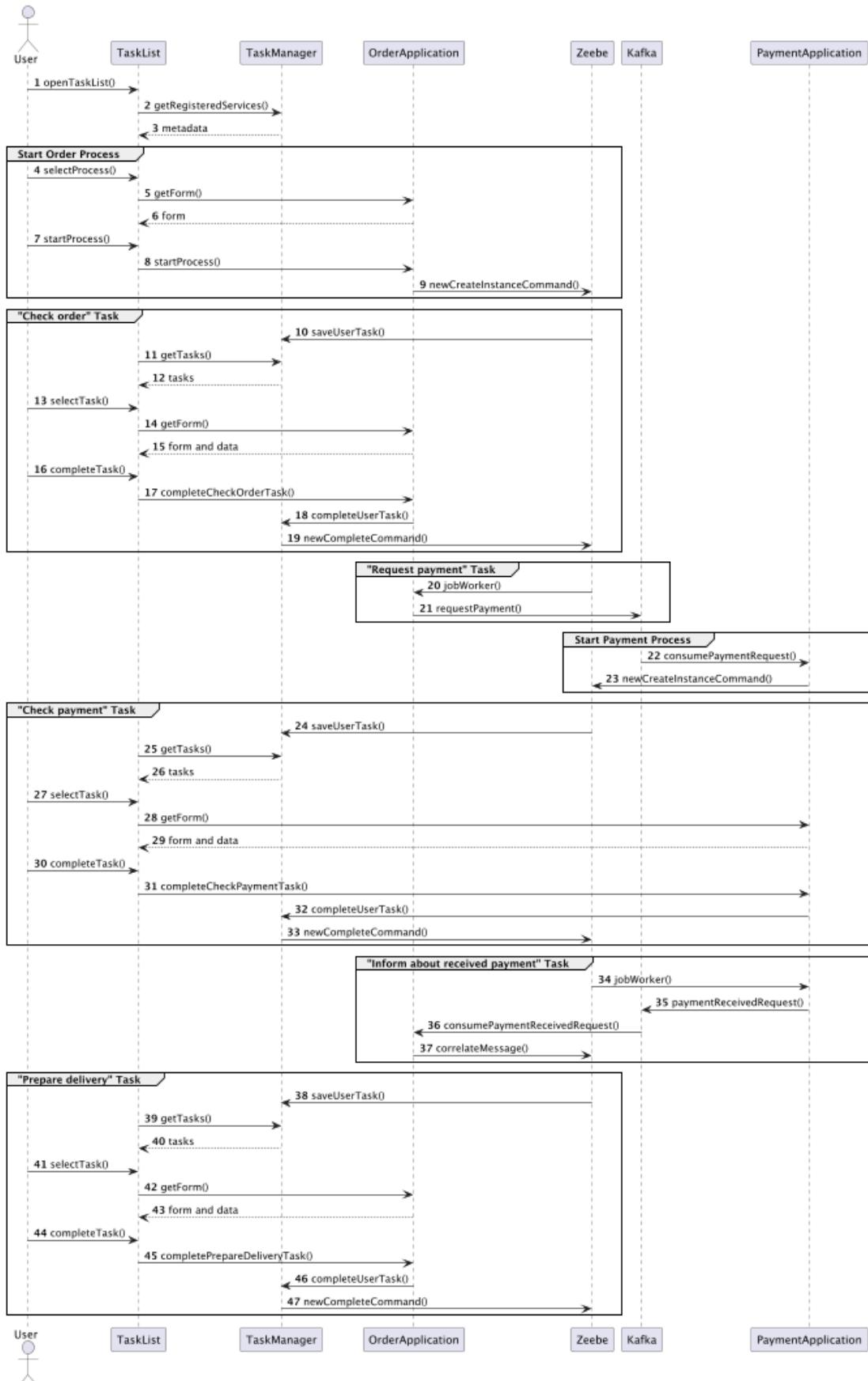


Abbildung 8.2: Sequenzdiagramm des Order-to-Cash Prozesses

Bisher war der Startpunkt immer die Zeebe-Engine. Eine Ausnahme bildet das **Start-Event** des Bestellprozesses. Hier wird der Prozess durch den User gestartet. Beim erstmaligen Öffnen der Task-Liste wird initial ein fetch-Request an den Task-Manager geschickt, um Informationen über die registrierten Prozess-Applikationen zu erhalten. Dadurch wird der Task-Liste bekannt gemacht welche der registrierten Prozesse über die Task-Liste gestartet werden können und wie diese erreicht werden. Der User klickt anschließend auf den Prozess. Im Falle der Bestell-Applikation hat das zunächst zur Folge das ein Formular geladen wird. Der User füllt das Formular aus und schickt es ab. Die Prozess-Applikation nimmt die Daten entgegen, speichert diese und sendet einen Command an die Zeebe-Engine, um den Prozess zu starten.

9 Implementierung des Proof-of-Concept

In diesem Kapitel wird detailliert auf die einzelnen Komponenten der Architektur eingegangen und wie diese implementiert wurden. Hierfür werden die verwendeten Tools und Technologien vorgestellt, sowie UML-Diagramme und Code-Beispiele gezeigt. Der gesamte Quellcode ist auf GitHub¹ verfügbar. An den entsprechenden Stellen wird jedoch noch einmal gesondert auf den Quellcode verwiesen. Der PoC kann mit einem Docker-Compose File gestartet werden. Die Docker-Compose Datei zusammen mit allen benötigten Konfigurationsdateien befindet sich im Repository².

Für den Task-Manager als auch den Prozess-Applikationen wurde die hexagonale Architektur genutzt, wie sie durch Hombergs [36] beschrieben wird. Daher, wird im nächsten Kapitel auf diesen Ansatz eingegangen, bevor jede Komponente der Architektur genauer betrachtet wird.

9.1 Hexagonale Architektur

Die hexagonale Architektur, auch „Ports and Adapters“ genannt, ist ein Architekturmuster, das von Cockburn [37] beschrieben wurde. Das Ziel ist es, die Domänenlogik von technischen Details zu trennen und hat daher Gemeinsamkeiten mit dem Domain-Driven Design (DDD) von Evans [38] und der Clean Architecture von Martin [39]. Die hexagonale Architektur und Clean Architecture beschreiben, wie DDD in der Praxis umgesetzt werden kann.

DDD ist ein Ansatz zur Softwareentwicklung, der sich stark auf die Modellierung komplexer Geschäftsdomänen konzentriert. Die Domäne ist das zentrale Objekt der Anwendung. In einem Bestellprozess wäre die Domäne die Bestellung, die von technischen Details wie der Datenbank, in der sie gespeichert wird oder der UI in der bestimmte Daten angezeigt werden, getrennt sein soll. Dadurch soll klar strukturierte, verständliche und wartbare Softwarearchitekturen entstehen. Eine beliebte Methode um DDD umzusetzen ist das Schichtenmodell mit einer eigenen Schicht für die Domänenlogik. Es hat sich

¹<https://github.com/miragon/ma-zeebe-taskmanagement>

²<https://github.com/Miragon/ma-zeebe-taskmanagement/tree/b2c19c0550352006cdf9b495de7963fd64f03281/stack>

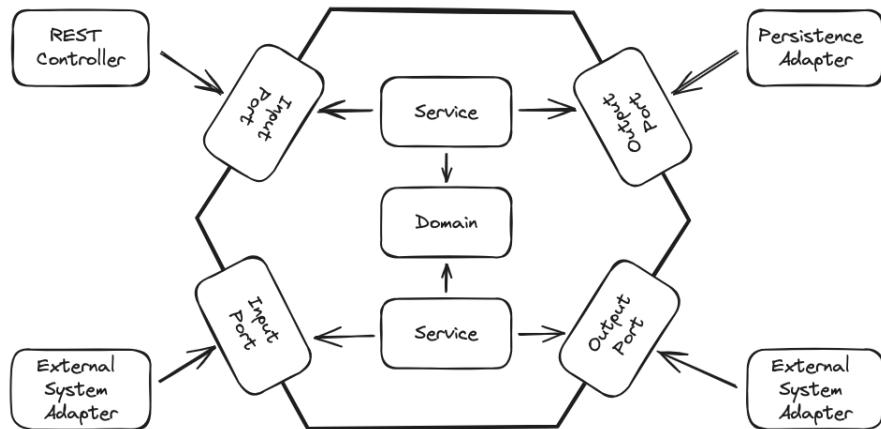


Abbildung 9.1: Abstrakte Darstellung der hexagonalen Architektur [36, Abb. 3.4]

jedoch gezeigt, dass sich dabei Teams entlang der Schichten organisieren. Gleichzeitig gibt es Abhängigkeiten zwischen den Schichten. Bei einem Drei-Schichten-Modell mit den Schichten *Web*, *Domain* und *Persistance* gibt es immer eine Abhängigkeit von oben nach unten (*Web* → *Domain* → *Persistance*). Muss eine Änderung implementiert werden, dann gehen die Teams häufig den Weg des geringsten Widerstandes und ändern die Schicht, die keine Abhängigkeiten hat. Dadurch wird sehr viel Logik in die *Persistance*-Schicht geschrieben [36, Kap. 2].

Die **Hexagonale Architektur** versucht dieses Problem zu lösen, indem es klare Regeln bestimmt. Es gibt keine Schichten mehr, sondern Adapter, die über Ports mit der Domäne kommunizieren. Abbildung 9.1 zeigt eine abstrakte Darstellung der hexagonalen Architektur. Die Pfeile stellen die Abhängigkeiten dar. Daran ist zu sehen, dass alle Pfeile nach innen zu dem Domänen-Objekt zeigen. Um dies zu erreichen wird *Dependency Inversion* und *Dependency Injection* genutzt. Beide Konzepte sind aus der objektorientierten Programmierung bekannt und in dem Spring Framework vorhanden, dass für die Implementierung des PoC genutzt wurde. In der Abbildung sind auch die einzelnen Komponenten der Architektur zu sehen. Der **Adapter** ist ein externer Service, der die Domäne nutzt (*In-Adapter*) oder von der Domäne genutzt wird (*Out-Adapter*). Ein Adapter spricht nie direkt mit der Applikation, sondern immer über einen **Port**. Entsprechend der Aufteilung bei den Adapters gibt es auch *In-Ports* und *Out-Ports*. *In-Ports* werden auch als *Use-Cases* bezeichnet. Dabei ist ein Port immer als Interface definiert. *In-Ports* werden von **Services** implementiert und *Out-Ports* von den *Out-Adaptoren*. Einzig in einem Service wird ein Domänen-Objekt genutzt. Diese Aufteilung kann wie in Abbildung 9.2 durch *Packages* in der Projektstruktur abgebildet werden.

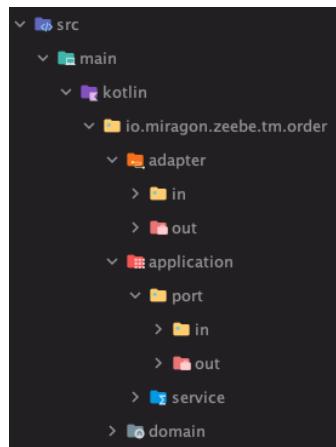


Abbildung 9.2: Mögliche Projektstruktur der hexagonalen Architektur

9.2 Programmiersprachen und Frameworks

Für die Implementierung der Komponenten des Prototypen wurden verschiedene Programmiersprachen und Frameworks genutzt. Abbildung 9.3 zeigt eine Übersicht der verwendeten Technologien getrennt nach Frontend und Backend.

Für die Microservices wurde die Programmiersprache Kotlin sowie das Spring Framework genutzt.

Kotlin³ ist eine moderne, statisch typisierte Programmiersprache, die von JetBrains entwickelt wurde und seit 2017 offiziell von Google für die Android-Entwicklung unterstützt wird. Sie ist vollständig interoperabel mit Java, was bedeutet, dass Kotlin-Code nahtlos in Java-Projekte integriert werden kann. Kotlin zeichnet sich durch Prägnanz, Sicherheit und Ausdrucksstärke aus, da viele typische Fehlerquellen wie Null-Zeiger-Referenzen und überflüssiger Boilerplate-Code vermieden werden. Dank dieser Eigenschaften wird Kotlin nicht nur für mobile Apps, sondern auch für Backend-Entwicklung, Webanwendungen und sogar Data Science immer beliebter.

Das **Spring Framework**⁴ ist ein weit verbreitetes Open-Source-Framework für die Entwicklung von Java-Anwendungen, insbesondere für Enterprise- und Webanwendungen. Es bietet eine umfassende Infrastruktur für verschiedene Anwendungsbereiche, darunter Dependency Injection (DI), Aspektorientierte Programmierung (AOP) und Transaktionsmanagement. Spring vereinfacht die Entwicklung durch eine modulare Architektur und ermöglicht die nahtlose Integration mit anderen Frameworks und Technologien wie Hibernate oder JPA. Zudem erleichtert es die Entwicklung von Microservices und cloudbasierten Anwendungen durch das Spring Boot-Framework, das schnelle und

³<https://kotlinlang.org/>

⁴<https://spring.io/projects/spring-framework>

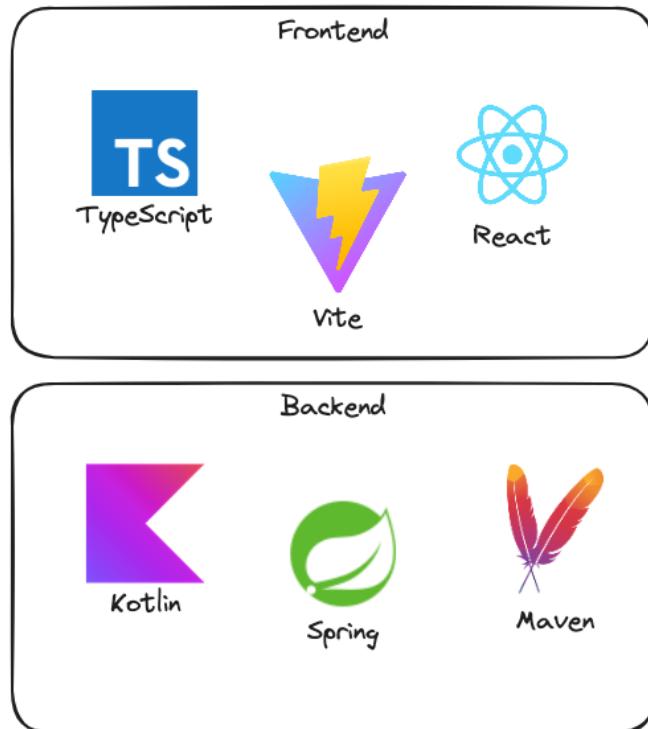


Abbildung 9.3: Verwendete Programmiersprachen und Frameworks

konfigurationsarme Setups ermöglicht.

Als Build-Tool wurde **Maven**⁵ genutzt. Maven ist ein weit verbreitetes Build-Management-Tool in der Java-Entwicklung, das hauptsächlich zur Verwaltung von Projekten und deren Abhängigkeiten sowie zur Automatisierung von Build-Prozessen verwendet wird. Es basiert auf dem Konzept eines Projekt Object Model (POM), einer XML-Datei, in der alle Informationen über das Projekt, die Abhängigkeiten, Plugins und die Build-Konfigurationen gespeichert werden. Maven ermöglicht es Entwicklern, Projekte konsistent zu bauen, testen und zu deployen, während es automatisch externe Bibliotheken herunterlädt und verwaltet. Zudem erleichtert es die Zusammenarbeit im Team, indem es eine standardisierte Struktur und wiederholbare Builds bietet. Die Entscheidung für Maven wurde unter anderem auch durch vorhandene Plugins, die in Abschnitt 9.5.2 beschrieben werden, beeinflusst.

Für die Frontend-Entwicklung wurde das React-Framework zusammen mit der Programmiersprache TypeScript genutzt.

TypeScript⁶ ist eine Erweiterung von JavaScript, die statische Typisierung einführt. Es

⁵<https://maven.apache.org/>

⁶<https://www.typescriptlang.org/>

ermöglicht Entwicklern Typen explizit zu definieren, was zu einer besseren Codequalität und Fehlererkennung bereits während der Entwicklungszeit führt. Da TypeScript auf JavaScript basiert, kann bestehender JavaScript-Code problemlos integriert werden. Es wird oft in großen Projekten verwendet, um die Wartbarkeit und Skalierbarkeit des Codes zu verbessern. Nach der Kompilierung wird der TypeScript-Code in reguläres JavaScript umgewandelt, sodass er in jedem Browser oder Node.js-Umgebung lauffähig ist.

React⁷ ist eine JavaScript-Bibliothek, die von Facebook entwickelt wurde, um interaktive Benutzeroberflächen zu erstellen. Es basiert auf einer komponentenbasierten Architektur, bei der die Benutzeroberfläche in wiederverwendbare, unabhängige Komponenten unterteilt wird. Diese Komponenten verwalten ihren eigenen Zustand und können einfach miteinander kombiniert werden, um komplexe Anwendungen zu entwickeln. React verwendet ein virtuelles DOM, das nur die tatsächlich geänderten Teile der Benutzeroberfläche aktualisiert, was zu einer verbesserten Performance führt. Die Bibliothek ist besonders beliebt für Single-Page-Anwendungen und bietet eine einfache Integration mit anderen Tools und Frameworks. Die Entscheidung hinsichtlich React wurde auch durch die Verwendung von Material-UI⁸ beeinflusst, einer React-Komponentenbibliothek, sowie dem `@jsonforms/material-renderers`-Paket⁹, das die Integration von JSON Forms in React-Anwendungen ermöglicht.

Als Build-Tool für die Frontend-Entwicklung wurde **Vite**¹⁰ verwendet. Die Entscheidung wurde getroffen, da bereits Erfahrung mit Vite vorliegt und ein Plugin, das in Abschnitt 9.3.2 beschrieben wird, den Build vereinfacht.

9.3 Task-Liste

Die Task-Liste ist das Frontend, über das die Aufgaben angezeigt und bearbeitet werden. Im Folgenden wird die Task-Liste vorgestellt, die im Rahmen des PoC entwickelt wurde. Dabei wurde sich stärker an der Camunda Tasklist als an Flowable-Work orientiert, da die Camunda Tasklist ein einfaches und intuitives Design hat. Der Quellcode ist auf GitHub¹¹ verfügbar.

⁷<https://react.dev/>

⁸<https://mui.com/material-ui/>

⁹<https://www.npmjs.com/package/@jsonforms/material-renderers>

¹⁰<https://vitejs.dev/>

¹¹<https://github.com/Miragon/ma-zeebe-taskmanagement/tree/main/apps/tasklist>

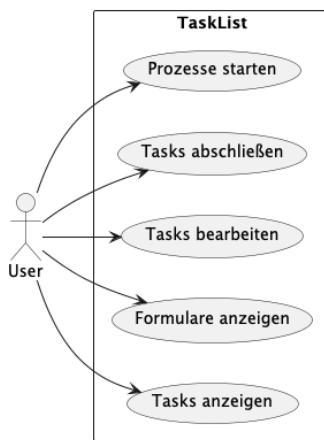


Abbildung 9.4: Use-Case Diagramm der Task-Liste

9.3.1 Anforderungen

Zunächst wurden die minimalen Anforderungen an die Task-Liste aus Sicht eines Benutzers definiert. Dazu wurde das UML Use-Case Diagramm in Abbildung 9.4 erstellt. Die Kernfunktion der Task-Liste ist es die Aufgaben anzuzeigen und abzuschließen. Um einen Task anzuzeigen, muss ein Formular geladen werden. Über das Formular kann der Benutzer Eingaben bzw. Informationen zum Task einsehen. Diese drei Funktionen wurden als erstes implementiert. Später kamen noch Anforderungen hinzu, um Prozesse starten zu können und Daten zu einer Aufgabe zu aktualisieren, ohne diese abzuschließen. Um die Benutzerinteraktion mit der Task-Liste darzustellen wurde anschließend die beiden Aktivitätsdiagramme in Abbildung 9.5 und 9.6 erstellt. Das Erstellen des Aktivitätsdiagramms machte deutlich, dass nicht jeder Prozess über die Task-Liste gestartet wird. Auch muss beachtet werden, dass bereits beim Prozessstart ein Formular notwendig sein kann.

Zusätzlich ging aus den Anforderungen an die Architektur (Kapitel 4) hervor, dass bei der Entwicklung von Formularen ein Low-Code als auch ein Pro-Code Ansatz unterstützt werden soll und das dafür Open-Source Technologien genutzt werden sollen.

9.3.2 Herausforderungen

Als erste Herausforderung wurde die **Service Discovery** identifiziert. Die Task-Liste kennt nur den Task-Manager, jedoch nicht die verfügbaren Prozesse. Daher wurden zwei Ansätze evaluiert.

Der erste Ansatz war die Implementierung eines **Broadcasting**.

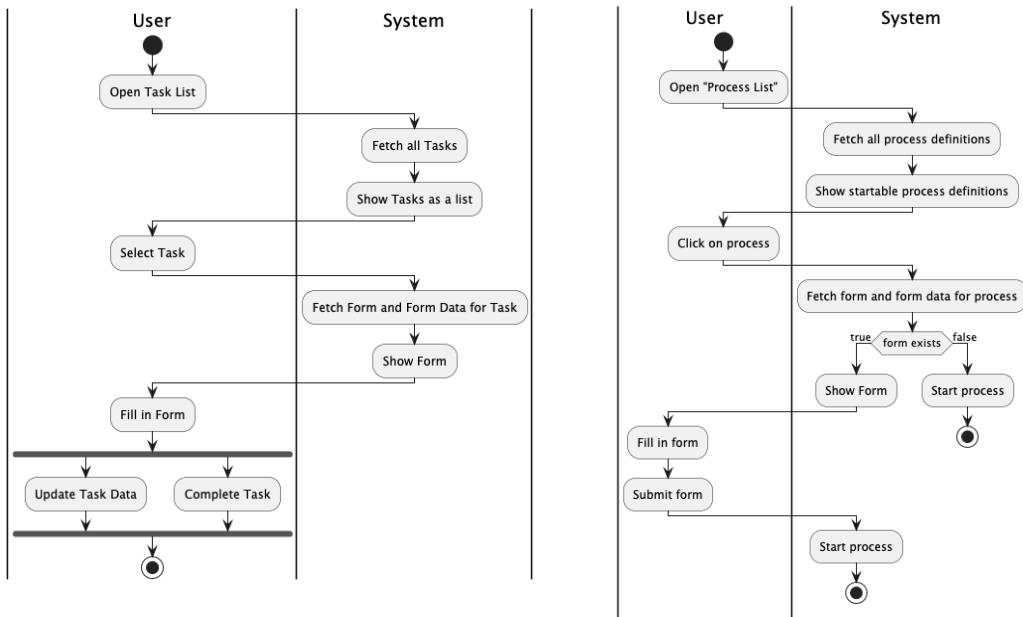


Abbildung 9.5: Aktualisieren und abschließen eines Tasks

Abbildung 9.6: Starten eines Prozesses

Broadcasting ist ein Begriff aus der Netzwerktechnik und beschreibt das Senden von Nachrichten an alle Teilnehmer eines Netzwerks. Hierfür wurde auch ein Script¹² für den Nginx Service geschrieben, der eine Anfrage an eine in der Konfiguration hinterlegte URL an alle Services weiterleitet und deren Antworten gesammelt an die Task-Liste zurückgibt.

Der zweite Ansatz war die Implementierung über ein leichtgewichtiges **Service-Registry**. Wobei sich jede Prozess-Applikation bei dem Task-Manager registriert und dieser die Informationen an die Task-Liste weitergibt. Eine Prozess-Applikation wird dabei über eine Konfigurationsdatei im Task-Manager eingelesen.

Schlussendlich wurde der zweite Ansatz gewählt, da er einfacher zu implementieren war und weniger fehleranfällig ist. Auch ist die Implementierung eines Broadcastings abhängig von der Infrastruktur auf der die Services laufen.

Das Objekt, das die Informationen von registrierten Applikationen enthält, ist als Klassendiagramm in Abbildung 9.7 dargestellt. An den beiden Klassen *UserTask* und *Process* ist zu sehen, dass der Entwickler die Endpoints seiner Prozess-Applikationen in der Konfigurationsdatei eintragen kann. Wenn er keine spezifischen Endpoints angibt, werden die Standard-Endpoints wie in Abbildung 8.1 zu sehen genutzt. Das soll dem Entwickler wie in den Anforderungen (Kapitel 4) gefordert mehr Flexibilität geben,

¹²<https://github.com/Miragon/ma-zeebe-taskmanagement/blob/b2c19c0550352006cdf9b495de7963fd64f03281/stack/nginx/njs/broadcast.js>

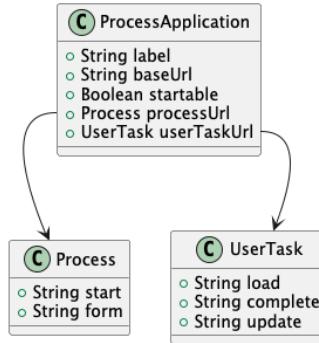


Abbildung 9.7: Klassendiagramm der Service-Registry

seine Anwendung so zu gestalten, wie er es für richtig hält. Die restlichen Attribute sind nicht optional und vor allem die *baseUrl* ist wichtig, damit die Task-Liste die Prozess-Applikation erreichen kann. Zusätzlich wird an dem Boolean *startable* erkannt, ob der Prozess über die Task-Liste gestartet werden kann.

Die nächste Herausforderung war das Implementieren der **Formulare**. Wie bereits erwähnt, sollten ein Low-Code und Pro-Code Ansatz unterstützt werden.

Für den Low-Code Ansatz wurde das Open-Source Framework *JSON Forms* genutzt, das bereits im Abschnitt 6.3.1 vorgestellt wurde.

Um den Pro-Code Ansatz zu implementieren wurde in die Task-Liste ein Micro-Frontend integriert, dass über einen *iframe* selbst entwickelte Formulare anzeigt.

Die Entscheidung für *JSON Forms* fiel, da *JSON Schema* dem Entwickler viele Möglichkeiten bietet den Entwicklungsprozess zu vereinfachen. Dazu wird noch genauer im Abschnitt 9.5.2 eingegangen. Der Pro-Code Ansatz wurde berücksichtigt, da viele Unternehmen bereits eine eigene Komponentenbibliothek haben und diese nutzen wollen. Die Nutzung eines *iframes* wurde der *Web Components* vorgezogen, da die Performance-Nachteile in der Task-Liste nur gering wiegen, da nur ein *iframe* geladen wird und die Implementierung einfacher ist. Der Entwickler hat die komplette Kontrolle über die Technologien, die er nutzen will und kann auch auf externe Bibliotheken zurückgreifen. Es ist somit auch möglich eigene HTTP-Requests zu implementieren, die die Daten aus der Prozess-Applikation oder anderen Services holen. Allerdings ist auch eine Schnittstelle zu der Task-Liste vorhanden. Dadurch kann aus dem *iframe* die bereits vorhandenen Methoden in der Task-Liste genutzt werden. Die Schnittstelle wird mit der *postMessage*-API und dem *Message-Event* realisiert. Die einzige Konvention, die eingehalten werden muss, ist, dass das entwickelte Formular als eine einzige HTML-Datei vorliegt. Das kann z.B. durch die Nutzung eines Bundlers wie Vite zusammen mit dem Plugin *vite-plugin-singlefile*¹³ erreicht werden. Eine Beispiel-Implementierung mit React

¹³<https://www.npmjs.com/package/vite-plugin-singlefile>

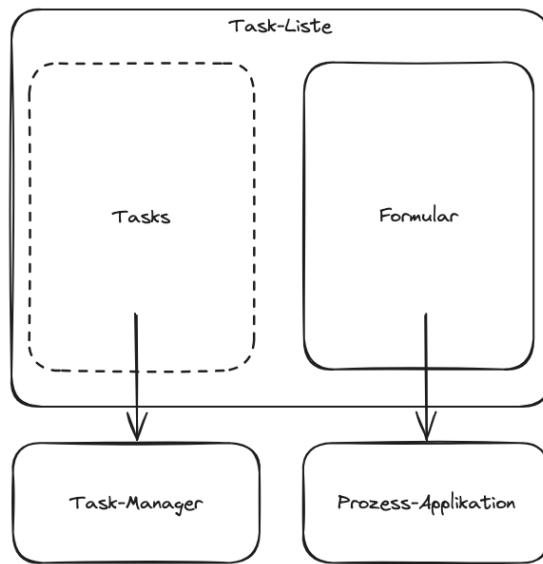


Abbildung 9.8: Horizontaler Aufbau der Task-Liste

und TypeScript ist auf GitHub¹⁴ verfügbar. Es handelt sich dabei um ein horizontales Micro-Frontend, dass wie in Abbildung 9.8 aufgebaut ist. An der Abbildung ist zu erkennen, dass der Container die Task-Liste ist und das Micro-Frontend die Formulare steuert. Dabei kommuniziert die Task-Liste mit dem Task-Manager und das Formular mit der entsprechenden Prozess-Applikation.

Das dritte und noch teilweise bestehende Problem ist die **einheitliche Darstellung** in der gesamten Task-Liste. Die Task-Liste als auch die Formulare sollten ein einheitliches Design haben. Im PoC wurde das, durch die Nutzung von *Material-UI*, einer React-Komponentenbibliothek, die das Material Design von Google umsetzt, erreicht. Die Task-List ist in Abbildung 9.9 zu sehen. Die Bibliothek gibt es auch für Angular (Angular Material UI) und Vue.js (Vuetify). Allerdings stellt das eine ungewünschte Abhängigkeit dar, da der Entwickler gezwungen ist, diese Bibliothek zu nutzen, wenn er ein eigenes Formular entwickelt.

Ein weiteres Problem, das die UI betrifft, liegt an der Entscheidung einen *iframe* zu nutzen. Dieser macht es schwierig, die Größe des Formulars dynamisch anzupassen, da die Höhe des *iframes* fix festgelegt werden muss. Diese beiden Probleme wurde nicht weiter verfolgt, da der Fokus auf einem lauffähigen Prototyp lag.

¹⁴<https://github.com/Miragon/ma-zeebe-taskmanagement/tree/b2c19c0550352006cdf9b495de7963fd64f03281/apps/react-forms>

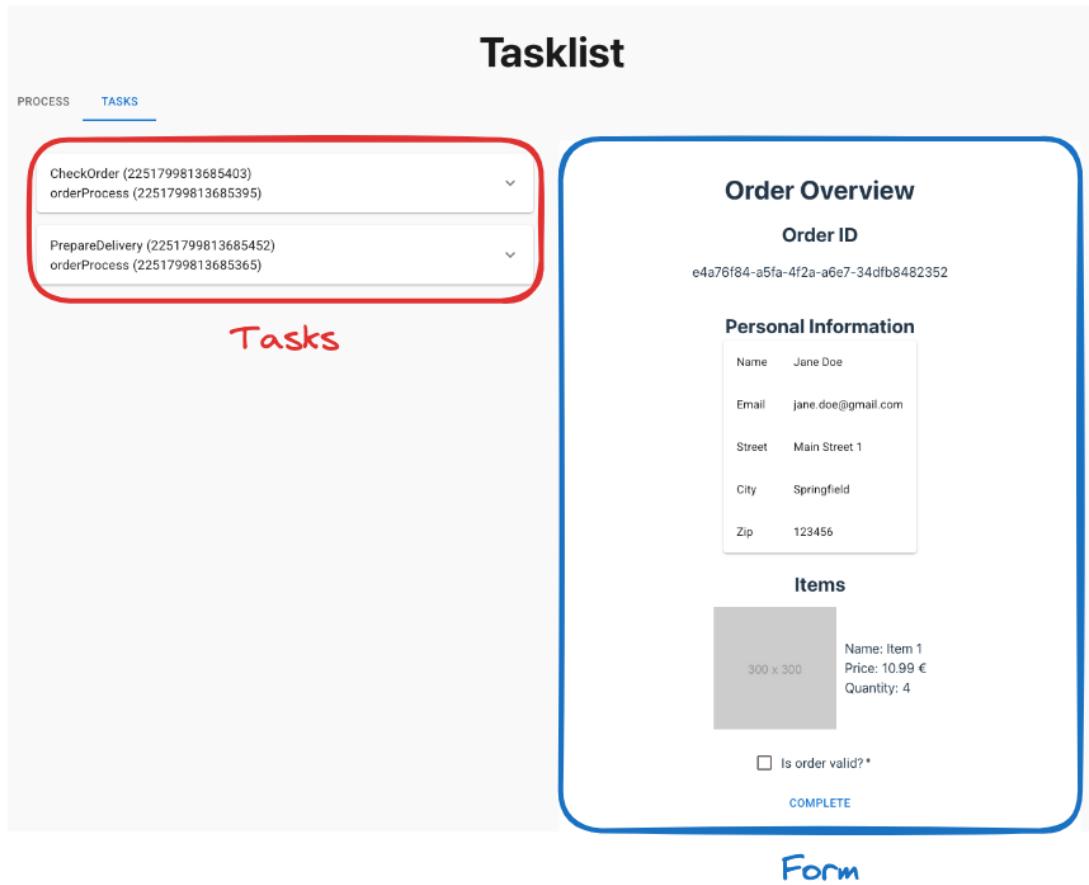


Abbildung 9.9: Screenshot der Task-Liste

9.4 Task-Manager

Der Task-Manager wurde als eine zentralisierte Komponente entwickelt, die Tasks aus verschiedenen Quellen entgegennimmt, verwaltet und an die Task-Liste übermittelt. Der Quellcode ist auf GitHub¹⁵ verfügbar. Für den PoC wurde die Zeebe-Engine als einzige Quelle genutzt, um Tasks zu erhalten. Im Folgenden wird die hexagonale Architektur des Task-Managers beschrieben, die gewählt wurde, um eine einfache Erweiterung um weitere Quellen zu ermöglichen. Im Weiteren wird auf die Herausforderungen eingegangen, die vor allem mit der Zeebe-Engine zusammenhängen, und die Frage, wie die User-Tasks von der Engine an den Task-Manager übermittelt werden.

¹⁵<https://github.com/Miragon/ma-zeebe-taskmanagement/tree/2e5fe3631a48fc9ca22f2a0c0901735f0a35b2ac/taskmanager>

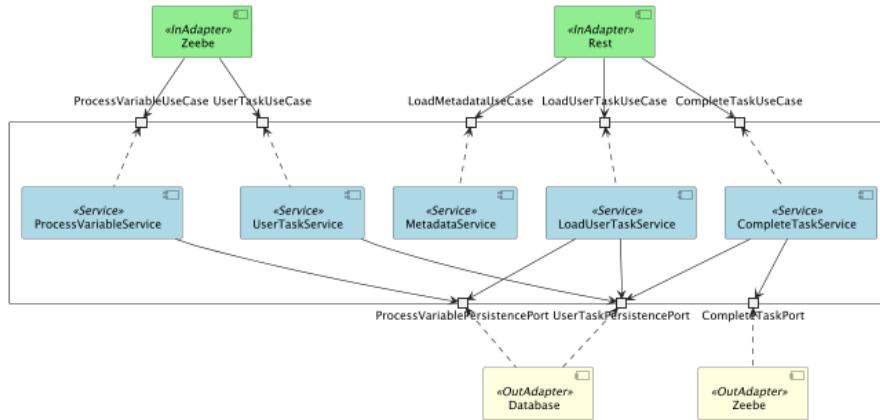


Abbildung 9.10: Komponentendiagramm der hexagonalen Architektur des Task-Managers

9.4.1 Softwaredesign

Die hexagonale Architektur ist in Abbildung 9.10 als ein UML-Komponentendiagramm dargestellt. Der Task-Manager implementiert fünf Use-Cases, die als In-Ports definiert sind:

1. **LoadUserTaskUsecase**: Lädt alle aktiven User-Tasks aus der Datenbank. Wird durch die Task-Liste aufgerufen.
2. **CompleteTaskUseCase**: Schließt einen User-Task ab. Wird durch die Prozess-Applikationen aufgerufen.
3. **LoadMetadataUseCase**: Lädt die Metadaten einer Prozess-Applikation. Wird durch die Task-Liste aufgerufen. Hierbei handelt es sich um das Service-Registry, dass bereits in Abschnitt 9.3.2 beschrieben wurde. Prozess-Applikationen müssen in einer Konfigurationsdatei angeben, wie sie erreicht werden können.
4. **UserTaskUseCase**: Speichert und aktualisiert User-Tasks in der Datenbank. Wird durch Systeme aufgerufen, die als eine Quelle für User-Tasks dienen. Im PoC ist das die Zeebe-Engine.
5. **ProcessVariableUseCase**: Speichert und aktualisiert Prozessvariablen in der Datenbank. Das ist ein Sonderfall, der mit der Zeebe-Engine zusammenhängt, wenn ein Exporter verwendet wird um User-Tasks an den Task-Manager zu senden. Dieser Fall wird im nachfolgenden Abschnitt 9.4.2 genauer beschrieben.

Die Implementierung der Use-Cases erfolgt in den Services, die wiederum Out-Ports verwenden, um auf externe Systeme zuzugreifen. Um weitere Quellen, wie ein ERP-System oder eine zweite Workflow-Engine, zu integrieren, müssen neue Adapter implementiert werden, die lediglich die In-Ports nutzen müssen um User-Tasks zu speicher. Dadurch

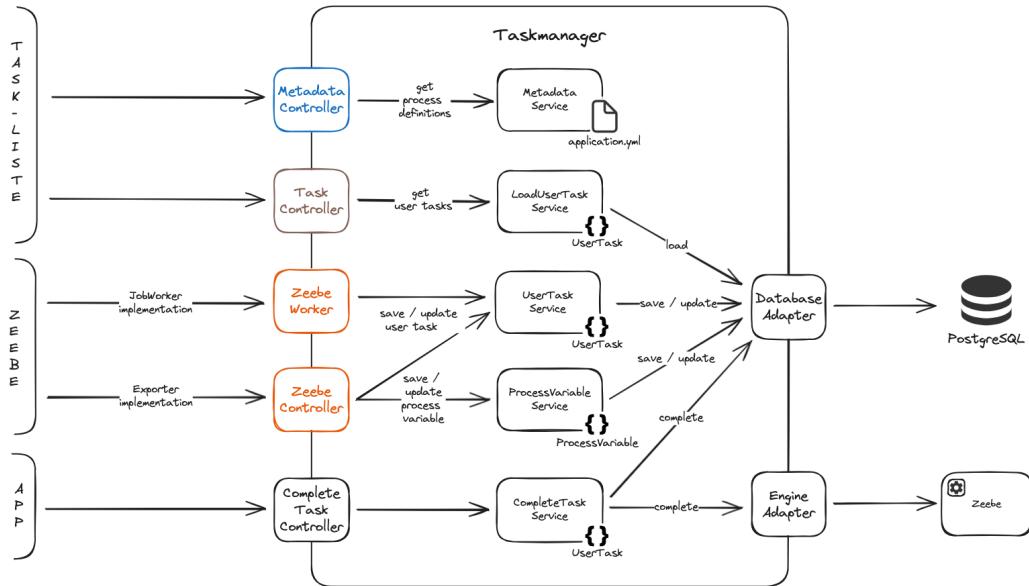


Abbildung 9.11: Der Task-Manager und alle angeschlossenen Systeme

wird die Erweiterung des Task-Managers vereinfacht und die Domänenlogik bleibt unberührt. Abbildung 9.11 zeigt nochmal detailliert den Aufbau des Task-Managers, und welche externen Systeme für den PoC angeschlossen wurden. Die Verbindungen wurden möglichst gering gehalten, um die Abhängigkeiten zu minimieren und ein leichtgewichtiges System zu schaffen. Lediglich eine Datenbank und die Zeebe-Engine werden vom Task-Manager genutzt, während die Task-Liste und die Prozess-Applikationen auf den Task-Manager zugreifen.

9.4.2 Herausforderungen

Der Task-Manager hat eine einfache und klare Struktur, weshalb die Implementierung der Use-Cases und Services ohne größere Probleme erfolgen konnte. Jedoch war es eine Herausforderung, die Zeebe-Engine als Quelle für User-Tasks zu nutzen. Bereits in Abschnitt 7.3 wurde beschrieben, dass durch das Fehlen einer API es erschwert wurde User-Tasks zu erhalten. Camunda [24] beschreibt zwei Ansätze, die dieses Problem lösen, die im Folgenden detailliert untersucht werden.

Der erste Lösungsansatz ist die Nutzung eines **JobWorker**¹⁶ [24, “Job workers”]. Ein JobWorker ist ein autonomer Prozess oder Service, der mit der Zeebe-Engine interagiert, um spezifische Aufgaben oder Jobs innerhalb eines Workflows zu übernehmen und

¹⁶<https://docs.camunda.io/docs/components/concepts/job-workers/>

auszuführen. Ein JobWorker wird mit einem bestimmten Job Type registriert und ist verantwortlich dafür, Jobs zu übernehmen, die diesem Typ zugeordnet sind.

Wenn ein Prozess in der Zeebe-Engine einen Task erreicht, der mit einem *JobType* versehen ist, erstellt die Engine einen entsprechenden Job. Ein JobWorker fordert solche Jobs in einem regelmäßigen Intervall an („*polling*“), führt die definierte Geschäftslogik aus und sendet eine Antwort zurück an die Engine. Die Antwort kann entweder den Erfolg oder das Scheitern des Jobs signalisieren. Die Schritte eines JobWorkers in der Interaktion mit der Zeebe-Engine sind wie folgt:

1. **Job anfordern:** Der JobWorker fragt die Engine nach Jobs eines bestimmten Typs.
2. **Job verarbeiten:** Sobald ein Job zugewiesen wurde, führt der JobWorker die definierte Geschäftslogik aus.
3. **Job abschließen:** Nach Abschluss meldet der JobWorker das Ergebnis zurück an die Zeebe-Engine, die den Fortschritt im Prozess verwaltet.

Für jeden JobWorker kann ein Zeitlimit (*Timeout*) definiert werden, nach dem die Engine den Job einem anderen Worker zuweist. Die Architektur ist in Abbildung 9.12 zu sehen. Zu den wichtigsten Merkmalen zählen:

- **Entkopplung:** JobWorker und die Zeebe-Engine sind unabhängig voneinander und kommunizieren über gRPC oder REST/HTTP. Diese Entkopplung erlaubt es, dass JobWorker in beliebigen Programmiersprachen implementiert und auf verschiedenen Maschinen oder Cloud-Diensten gehostet werden können.
- **Asynchrone Verarbeitung:** Jobs werden asynchron ausgeführt, was bedeutet, dass die Workflow-Engine nicht auf das Ergebnis des JobWorkers wartet, bevor sie andere Teile des Prozesses weiter ausführt. Das ermöglicht hohe Parallelität und Skalierbarkeit.
- **Polling:** Ein JobWorker „*pollt*“ die Zeebe-Engine regelmäßig, um zu überprüfen, ob neue Jobs des jeweiligen Typs verfügbar sind. Sobald ein Job gefunden wurde, übernimmt der Worker diesen und führt ihn aus.

User-Tasks haben einen besonderen JobTyp (`io.camunda.zeebe:userTask`). Worker mit diesem Typ bekommen den User-Task als Parameter übergeben. Diese Implementierung ist etwas älter und wird nur noch bedingt empfohlen. Der große Nachteil ist, dass der Zustand nicht von Zeebe verwaltet wird [24, „User tasks“]. Dadurch fehlen Informationen, wie z.B. der eingetragene Bearbeiter („*Assignee*“), oder die Deadline.

In Abbildung 9.13 ist der JobWorker, wie er im PoC implementiert wurde dargestellt¹⁷. In dem JobWorker wurde auch ein Timeout definiert. Das hat den Grund, da es nicht möglich ist, auf eventuelle „*interrupting*“ Events zu reagieren. Ist an einem User-Task

¹⁷<https://github.com/Miragon/ma-zeebe-taskmanagement/blob/2e5fe3631a48fc9ca22f2a0c0901735f0a35b2ac/taskmanager/src/main/kotlin/io/miragon/zeebe/taskmanager/adapter/in/zeebe/UserTaskListener.kt>

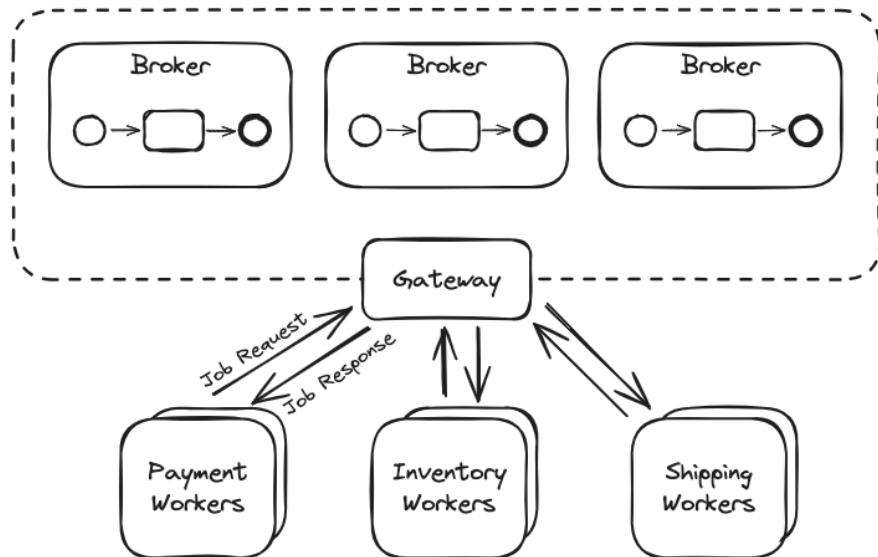


Abbildung 9.12: Architektur eines JobWorkers in der Zeebe-Engine [24, “Job workers”]

```

● ● ●
1 @Component
2 class UserTaskListener(
3     private val useCase: UserTaskUseCase,
4 )
5 {
6     @JobWorker(type = "io.camunda.zeebe:userTask", timeout = 10000, autoComplete = false)
7     fun receiveTask(job: ActivatedJob)
8     {
9         useCase.sync(
10             SyncCommand(
11                 key = job.key,
12                 elementId = job.elementId,
13                 processInstanceKey = job.processInstanceKey,
14                 bpmnProcessId = job.bpmnProcessId,
15                 processDefinitionKey = job.processDefinitionKey,
16                 expiresAt = Instant.ofEpochMilli(job.deadline),
17                 variables = job.variablesAsMap,
18             )
19         )
20     }
21 }

```

Abbildung 9.13: JobWorker für User-Tasks in der Zeebe-Engine

z.B. ein Boundary-Timer-Event angebracht und wird ausgelöst, bleibt der User-Task in der Datenbank des Task-Managers bestehen. Die Task-Liste würde weiterhin den User-Task anzeigen, obwohl dieser eigentlich abgebrochen wurde. Der Timeout führt dazu, dass nach dem Ablauf der Zeit ein neuer Broker den Job übernimmt. Damit wird der JobWorker auf ein neues ausgeführt. Dadurch ist es möglich ein „*Ablaufdatum*“ für den User-Tasks zu definieren. Jedes Mal, wenn der JobWorker erneut ausgeführt wird, wird das Ablaufdatum um die definierte Zeit im Timeout verlängert. Wurde der User-Task abgebrochen nimmt ihn auch kein Broker mehr auf und das Ablaufdatum wird nicht mehr verlängert. Um nur aktuelle User-Tasks anzuzeigen, werden in der Task-Liste nur Tasks angezeigt, die noch nicht abgelaufen sind. Ein Nachteil bei diesem Vorgehen ist, dass jeder Timeout auch zu einem Datenbankzugriff führt.

Eine zweite Möglichkeit ist es, die User-Tasks über einen **Exporter**¹⁸ an den Task-Manager zu senden [24, „Exporters“]. Zeebe schreibt mittels Event-Streaming sogenannte Records in ein Log. Die Broker können auf diesen Stream zugreifen und wissen anhand der Records, den aktuellen Zustand des Prozesses. Neben den Brokern können auch Exporter auf diesen Stream zugreifen. Camunda hat selbst einen eigenen Exporter geschrieben, um Records nach Elasticsearch zu senden [24, „Elasticsearch exporter“]. Die Komponenten *Tasklist*, *Operate* und *Optimize* greifen auf Elasticsearch zu, um die für ihren jeweiligen Zweck notwendigen Informationen zu erhalten. On-Premise Installationen können auch eigene Exporter implementieren, um damit Records an andere Systeme zu senden.

Ruecker [40] beschreibt in seinem Blog-Beitrag, detailliert wie die Records in ein Append-Only Log geschrieben werden. Zunächst ist jedoch zu beachten, das ein Record entweder ein *Command* oder *Event* darstellt.

1. Ein Command kann z.B. durch einen Client entstehen, wenn dieser eine Aktivität abschließt.
2. Zeebe fügt diesen Command zum Log hinzu.
3. Der Prozess geht einen Schritt weiter und trifft auf eine neue Aktivität.
4. Das löst zunächst ein *CREATING*-Event aus, gefolgt auf ein *CREATED*-Event, die beide dem Log hinzugefügt werden.

Dieses Vorgehen ist nochmal in Abbildung 9.14 bildlich dargestellt. Dabei wird der Record auch etwas detaillierter beschrieben. Denn aus technischer Sicht handelt es sich um ein Objekt, das unter anderem die Attribute *intent* und *value* besitzt. Aus dem *intent* (dt. Absicht) kann der Status abgelesen werden. Welchen *intent* ein Record haben kann, hängt von dem *value*-Objekt ab. Es gibt viele verschiedene Value-Objekte. Das wohl wichtigste ist der **Job**. Der Job kann z.B. eine Aktivität darstellen und enthält Informationen, wie z.B. die Prozessinstanz-ID oder die ID der Aktivität, die im BPMN Diagramm hinterlegt wurde.

¹⁸<https://docs.camunda.io/docs/self-managed/concepts/exporters/>

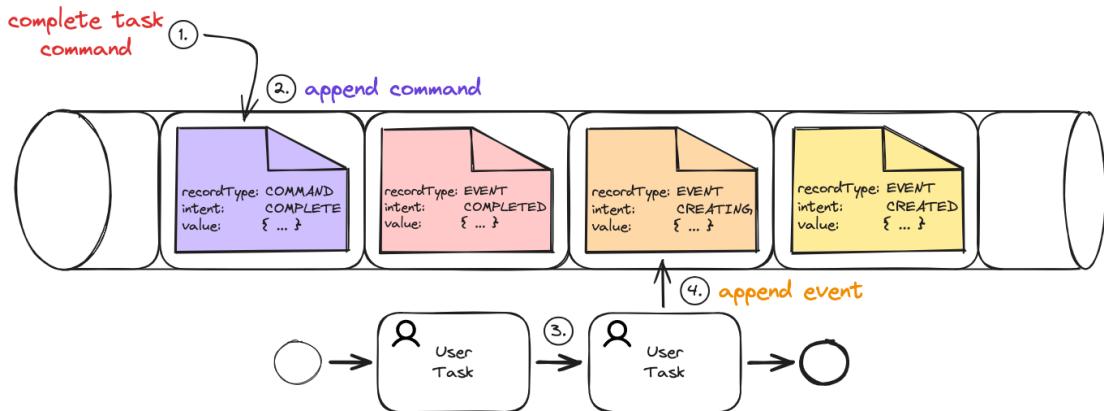


Abbildung 9.14: Event-Stream mit verschiedenen Records [40, “The event handling loop”]

Bis Zeebe Version 8.4 wurde ein User-Task auch als ein Job betrachtet. Das hatte zur Folge, dass der Lebenszyklus eines User-Tasks nicht in der Engine verwaltet wurde, sondern nur in der *Tasklist*-Komponente. Eigenentwicklungen hatten somit keine Möglichkeit über die Engine herauszufinden, ob ein User-Task bereits einem Bearbeiter zugewiesen wurde oder nicht. Mit der Zeebe Version 8.5 wurde die neue Zeebe Task API eingeführt [24, “Migrate to Zeebe user tasks”]. Dadurch wird der User-Task nun von der Engine verwaltet und es ist möglich über Clients verschiedene Operationen auszuführen. Diese Operationen wurden bereits im Kapitel 4 in der Abbildung 4.1 aufgelistet. Mit der Zeebe Task API wurde auch ein neues Value-Objekt eingeführt, das den User-Task repräsentiert. In einem Exporter kann dadurch nach diesem Value-Objekt gefiltert werden, um nur User-Tasks zu erhalten. Für den PoC wurde ein Exporter geschrieben, der die User-Tasks über HTTP an den Task-Manager sendet. Der Quellcode ist in Abbildung 9.15 dargestellt. Ein Custom Exporter muss das Interface **Exporter** implementieren.

Im Speziellen muss die Methode `export()` vorhanden sein. Diese Methode bekommt den Record übergeben und kann damit arbeiten.

Durch die Methode `configure()` kann der Exporter konfiguriert werden. Dabei kann ein Filter definiert werden oder aber auch Werte aus einer Konfigurationsdatei gelesen werden. Die Konfigurationsdatei ist in Abbildung 9.17 auf der linken Seite dargestellt. In der Datei wird die `baseUrl` des Task-Managers definiert. Dieser Wert wird in der Methode `configure()` ausgelesen und der Klassen-Variable `baseUrl` zugewiesen.

Wie an dem implementierten *RecordFilter* aus Abbildung 9.15 zu sehen ist, werden nicht nur User-Tasks an den Task-Manager gesendet. Sondern es ist auch notwendig die Prozessvariablen zu speichern. Diese erzeugen einen eigenen Record und werden in einer eigenen Tabelle im Task-Manager gespeichert. Die Zuordnung zwischen User-Task und Prozessvariablen erfolgt über die Prozessinstanz-ID, wenn die Task-Liste die User-Tasks

```

1 class RecordFilter : Context.RecordFilter
2 {
3     override fun acceptType(recordType: RecordType?): Boolean
4     {
5         return recordType == RecordType.EVENT
6     }
7
8     override fun acceptValue(valueType: ValueType?): Boolean
9     {
10        return valueType == ValueType.USER_TASK || valueType == ValueType.VARIABLE
11    }
12 }

```

```

1 class UserTaskExporter : Exporter
2 {
3
4     override fun configure(context: Context)
5     {
6         val filter = RecordFilter()
7         context.setFilter(filter)
8         this.log = context.logger
9
10        this.config = context
11            .configuration
12            .instantiate(ExporterConfiguration::class.java)
13
14        this.baseUrl = config.baseUrl
15
16        if (baseUrl.isEmpty())
17        {
18            log.warn("No URL provided for user task exporter")
19        } else
20        {
21            log.info("User Task Exporter configured with URL: $baseUrl")
22        }
23
24        this.client = OkHttpClient.Builder()
25            .retryOnConnectionFailure(true)
26            .build()
27    }
28
29    override fun export(record: Record<*>)
30    {
31        if (record.intent.name() == "CREATING")
32        {
33            return
34        }
35
36        val recordString = mapper.writeValueAsString(record)
37
38        log.info("User Task Exporter Record: $recordString")
39
40        val url = if (record.intent.name() == "CREATED")
41        {
42            "$baseUrl/rest/${record.valueType.name.lowercase()}/save"
43        } else
44        {
45            "$baseUrl/rest/${record.valueType.name.lowercase()}/update"
46        }
47
48        postUserTaskRecord(url, recordString)
49
50        this.controller.updateLastExportedRecordPosition(record.position)
51    }
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102 }

```

Abbildung 9.15: Exporter, der User-Tasks an den Task-Manager sendet

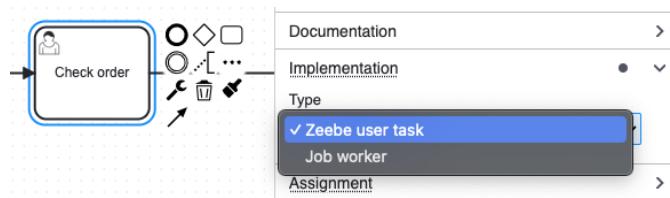


Abbildung 9.16: Konfiguration des BPMN Diagramms für die Zeebe Task API

anfordert.

Soll der Exporter auch für Zeebe Versionen vor 8.5 genutzt werden, muss der RecordFilter entsprechend angepasst werden, sodass statt des Value-Types USER_TASK der Typ JOB verwendet wird. Es ist auch wichtig im BPMN Diagramm zu definieren, ob die neue Zeebe Task API verwendet werden soll oder die bisherige Implementierung über JobWorker. Abbildung 9.16 zeigt, an welcher Stelle diese Einstellung getroffen werden kann.

Exporter werden während der Startphase der Broker geladen. Er muss daher zusammen mit allen Abhängigkeiten zu diesem Zeitpunkt als .jar Datei vorliegen. Wenn z.B. das Docker Image von Zeebe verwendet wird, kann der Exporter als Volume gemountet werden. Abbildung 9.17 zeigt, die Konfiguration des Zeebe-Brokers in der *docker-compose.yml* Datei und welche Zeilen hinzugefügt werden müssen, um den Exporter zu laden.

9.5 Prozess-Applikation

Für den PoC wurden zwei Prozess-Applikationen entwickelt, die den Order-to-Cash Prozess aus Abschnitt 5.4 implementieren. In diesem Abschnitt wird nur auf die Bestell-Applikationen eingegangen. Beide Applikationen sind auf GitHub¹⁹ verfügbar. Das Ziel hierbei ist es die Zusammenhänge aus dem Prozessmodell und der Softwarearchitektur zu zeigen. Dies ist anhand der Vorstellung einer Applikation ausreichend. Außerdem sind beide Applikationen ähnlich aufgebaut, und die Struktur der Bezahl-Applikation kann aus dem gezeigten abgeleitet werden. Anhand der Prozess-Applikationen wird auch gezeigt, wie die Entscheidung hinsichtlich JSON Forms in der Task-Liste die Entwicklungszeit verkürzen kann.

¹⁹<https://github.com/Miragon/ma-zeebe-taskmanagement/tree/2e5fe3631a48fc9ca22f2a0c0901735f0a35b2ac/processes>

```

docker-compose.yml
1 zeebe:
2   image: camunda/zeebe:8.5.6
3   container_name: zeebe
4   ports:
5     - "26500:26500" # gRPC
6     - "9600:9600"
7     - "8088:8088" # REST
8   env_file:
9     - zeebe.env
10  environment:
11    - SPRING_CONFIG_ADDITIONAL_LOCATION=/usr/local/zeebe/config/exporter.yml # TODO: Add this line if using Zeebe exporter
12    - JAVA_TOOL_OPTIONS=-Xms512m -Xmx512m
13  restart: always
14  volumes:
15    - ./zeebe/exporter.yml:/usr/local/zeebe/config/exporter.yml # TODO: Add this line if using Zeebe exporter
16    - ./zeebe/zeebe-user-task-exporter.jar:/usr/local/zeebe/lib/zeebe-user-task-exporter.jar # TODO: Add this line if using Zeebe exporter
17  networks:
18    - internal
19  depends_on:
20    - elasticsearch:
21      condition: service_healthy

Exporter Konfigurationsdatei (exporter.yml)
1 zeebe:
2   broker:
3   exporters:
4     user-task-exporter:
5       className: io.miragon.zeebe.tm.exporter.UserTaskExporter
6       jarPath: /usr/local/zeebe/lib/zeebe-user-task-exporter.jar
7       args:
8         maxBatchSize: 100
9         maxBlockingTimeoutMs: 1000
10        flushIntervalMs: 1000
11
12        baseUrl: http://host.docker.internal:8090

```

Abbildung 9.17: Konfiguration des Zeebe-Brokers und des Exporters

9.5.1 Softwaredesign

Wie bereits erwähnt wurde für alle Microservices die hexagonale Architektur genutzt. Die Prozess-Applikationen sind dabei keine Ausnahme. Auch wenn hier nicht wie im Task-Manager die Anforderung besteht weitere Engines anzubinden, so ist es dennoch sinnvoll die Software möglichst flexibel zu gestalten, um zukünftige Anforderungen zu erfüllen.

Der Aufbau der Bestell-Applikation ist in Abbildung 9.18 als UML-Komponentendiagramm dargestellt. Dieses musste jedoch vereinfacht werden, damit das Diagramm nicht zu unübersichtlich wird. Wird das Diagramm mit dem Prozess aus Abbildung 5.10 verglichen, so werden folgende Zusammenhänge deutlich.

Für einen **User-Task** im Prozessmodell müssen REST-Controller implementiert werden. Diese werden von der Task-Liste aufgerufen. Die verschiedenen Controller wurden im Diagramm zusammengefasst. Für das Laden des Formulars, der Bestelldaten und für das Abschließen des Tasks wurde jeweils ein eigener Controller implementiert. Jeder dieser

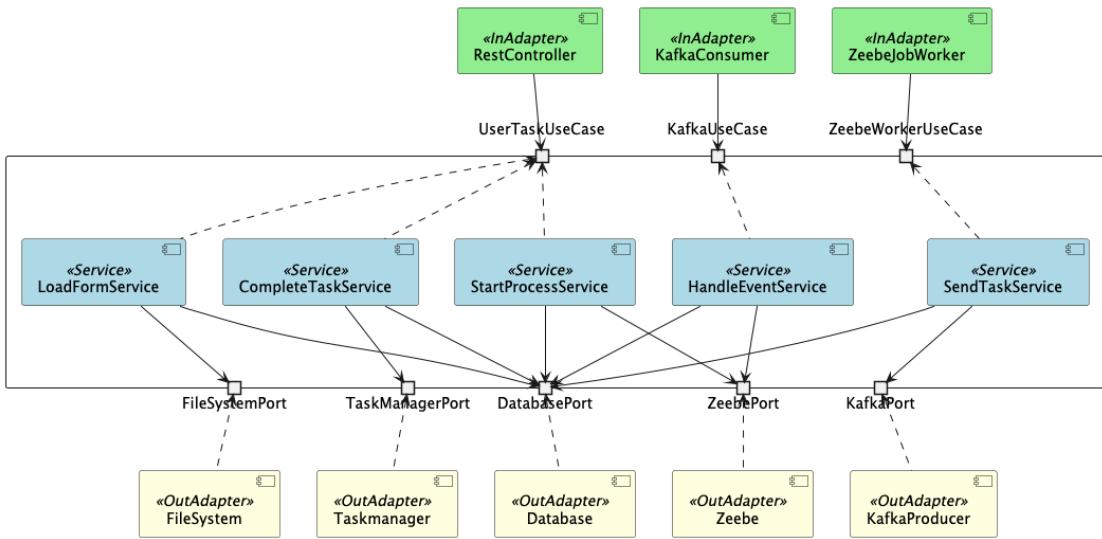


Abbildung 9.18: Vereinfachtes UML-Komponentendiagramm der Bestell-Applikation

Controller nutzt einen Use-Case (In-Port) um damit mit einem Service zu kommunizieren. Die Use-Cases werden durch Services implementiert, die wiederum die Domänen-Objekte nutzen. Für den Bestellprozess ist die Bestellung und das Produkt ein Domänen-Objekt. Es ist auch zu erkennen, dass für einen User-Task in der Applikation kein Aufruf an Zeebe erfolgt. Bestelldaten werden aus der Datenbank geladen, das Formular wird aus einer Datei im Filesystem gelesen und der Task wird über den Task-Manager abgeschlossen.

Für einen **Send-Task** im Prozessmodell wird ein Zeebe-JobWorker implementiert. Der JobWorker wird von Zeebe aufgerufen, wenn der Token den Send-Task erreicht. Ein Send-Task stellt eine Nachricht an ein anderes System dar. In Fall des Bestellprozesses wird eine Nachricht an die Bezahl-Applikation gesendet oder an einen Service, der das Versenden einer E-Mail übernimmt. Die Applikation folgt den Empfehlungen von Ruecker [5, Kap. 8] und vermischt Choreografie und Orchestrierung. Abbildung 9.19 zeigt genau diesen Teil des Prozesses. Ruecker [5] unterscheidet zwischen einen Command (Orchestrierung) und einem Event (Choreografie). Ein Command drückt eine Absicht aus, während ein Event eine Tatsache beschreibt. Im Falle des Bestellprozesses muss sichergestellt werden, dass die Bezahlung erfolgt ist, bevor die Bestellung abgeschlossen werden kann. Der Bestellprozess kontrolliert somit den Ablauf und orchestriert die Bezahlung. Für das Versenden der E-Mail ist es nicht notwendig, dass der Bestellprozess den Ablauf kontrolliert. Im Prozess wird ein Event dafür ausgelöst und das Token wandert weiter. Somit schreiben Send-Tasks einen *Record* nach Kafka. Dabei handelt es sich entweder um einen Command oder ein Event. Das ist aus dem Diagramm daran zu erkennen, das der *SendTaskService* neben der Datenbank nur auf den *KafkaPort* zugreift.

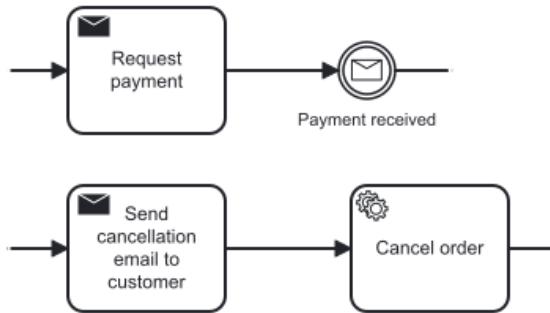


Abbildung 9.19: Ausschnitt aus dem Bestellprozess

Für jedes **Event**, außer Boundary-Events, wird im Prozessmodell ein *Consumer* implementiert. Im Bestellprozess gibt es davon zwei, das Start-Event des Sub-Prozesses und das Intermediate-Event, das auf den Zahlungseingang wartet. Das Start-Event, das den Bestellprozess startet, wird von der Task-Liste ausgelöst und ist somit ein Sonderfall, der im nachfolgendem Absatz genauer betrachtet wird. Ein *Consumer* liest die Nachrichten aus Kafka und ruft anschließend die Applikation durch einen *In-Port* auf. Der Service aktualisiert den Status der Bestellung in der Datenbank und ruft anschließend Zeebe auf, damit der Prozess weiterlaufen kann. Camunda spricht hierbei von *Message Correlation* [24, „Message correlation“]

Das **Start-Event** ist wie bereits erwähnt ein Sonderfall. Der Bestellprozess wird durch die Task-Liste gestartet. Daher hat dieses Event mehr Ähnlichkeiten mit einem User-Task als mit einem Event. Es wird ein eigener REST-Controller implementiert, der zunächst ein Formular einliest und dieses an die Task-Liste zurückgibt. Erst nachdem der Benutzer das Formular ausgefüllt hat, wird die Bestellung in der Datenbank anlegt und anschließend Zeebe aufgerufen, um den Prozess zu starten. Im Bezahlprozess ist das Start-Event ein „echtes“ Event, das von der Bestell-Applikation ausgelöst wird.

9.5.2 Herausforderungen

Die größte Herausforderung bei der Entwicklung der Prozess-Applikationen war die Kommunikation mit der Task-Liste. Hierbei stellten sich folgende Fragen:

1. Wie werden Formulare identifiziert?
2. Welche Daten müssen mit der Task-Liste ausgetauscht werden?

Zu der ersten Frage gibt es verschiedene Ansätze.

Wird z.B. die *Camunda Taklist* genutzt, dann muss das Formular einen Form-Key besitzen, der im BPMN Modeler mit einem User-Task verknüpft wird. Das Formular

muss anschließend zusammen mit dem Prozess ausgerollt werden. Dabei wird es in einer Datenbank gespeichert und über den Form-Key identifiziert. Zum Erstellen der Formulare muss der Camunda Modeler genutzt werden, der im Abschnitt 6.3.1 vorgestellt wurde. Allerdings kann im BPMN Modeler auch ein *Custom form key* hinterlegt werden, um ein eigenes Formular zu nutzen. Wird die Camunda *Tasklist* jedoch nicht verwendet, dann ist es kompliziert den im BPMN Diagramm hinterlegten Form-Key zu erhalten. Daher ist ein zweiter möglicher Ansatz, einen Form-Key als Input-Variablen im User-Task zu definieren. Der Form-Key wird dadurch dem User-Task als Prozessvariable übergeben und im Task-Manger gespeichert. Die Task-Liste erhält den User-Task mit allen Variablen und gibt diesen an die Prozess-Applikation weiter. In der Applikation wäre es nun möglich den Form-Key einer Datei im Filesystem zuzuordnen und diese einzulesen oder den Key für eine Datenbankabfrage zu nutzen. Hier stellt sich die Frage, ob es Notwendig ist, den Form-Key im BPMN Diagramm zu hinterlegen. Der Vorteil hierbei ist es, dass die Verbindung zwischen Formular und User-Task einfacher geändert werden kann, vor allem wenn die Formulare in einer Datenbank gespeichert werden. Der Nachteil besteht in einer zusätzlichen Prozessvariable pro User-Task.

Für den PoC wurde ein dritter Ansatz gewählt. Dieser sieht es vor, dass der Form-Key in der Prozess-Applikation hinterlegt wird. Das kann direkt im Quellcode passieren oder in einer Konfigurationsdatei. Dieser Ansatz wurde gewählt, da die Formulare nicht in einer Datenbank abgespeichert werden, sondern als Datei im Filesystem vorliegen. Dadurch wird die Architektur einfacher gehalten und es ist auch nicht notwendig einen zusätzlichen Mechanismus zu implementieren, der das Bereitstellen („*deployen*“) von Formularen ermöglicht. Welches Formular eingelesen wird, wird anhand des User-Tasks entschieden, der durch die Task-Liste übergeben wird.

Bezüglich der zweiten Frage wurde zunächst, die in Abbildung 9.20 zu sehende grafische Übersicht über die Kommunikation zwischen der Task-Liste und der Prozess-Applikation erstellt. Anschließend wurde das Klassendiagramm aus Abbildung 9.21 erstellt, das die ausgetauschten Daten detailliert darstellt. Die gezeigten Klassen wurden in eine Modul *tasklistApi*²⁰ ausgelagert, das von den Prozess-Applikationen als Abhängigkeit eingebunden wird. Die Klasse *UserTaskDto* wird an die Prozess-Applikation übergeben, wenn das Formular und dessen Daten geladen werden sollen. Als Antwort erhält die Task-Liste das *FormDto*. Dabei wird zwischen dem *JsonForm* und der *HtmlForm* unterschieden. Das Form-Objekt enthält die Daten zur Präsentation als auch die Daten, die präsentiert werden sollen. Außerdem muss der Entwickler angeben, ob die Daten aktualisiert werden können. Das ist nicht immer gewünscht, vor allem bei User-Task, deren Kernfunktion es ist Daten anzuzeigen. Wird ein Task abgeschlossen, dann wird die Klasse *CompleteTaskDto* übergeben, die den *UserTaskDto* beinhaltet, damit der richtige Task abgeschlossen wird.

²⁰<https://github.com/Miragon/ma-zeebe-taskmanagement/tree/b2c19c0550352006cdf9b495de7963fd64f03281/tasklistApi>

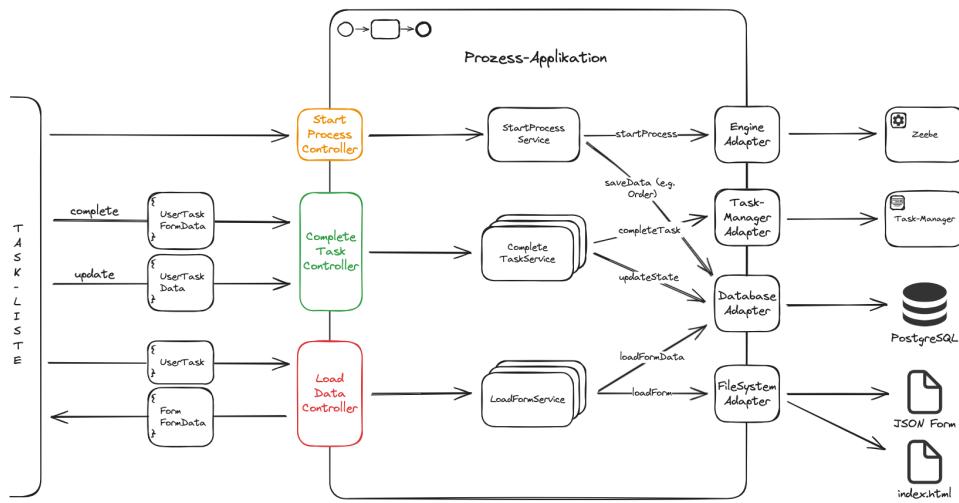


Abbildung 9.20: Kommunikation zwischen Task-Liste und Prozess-Applikation

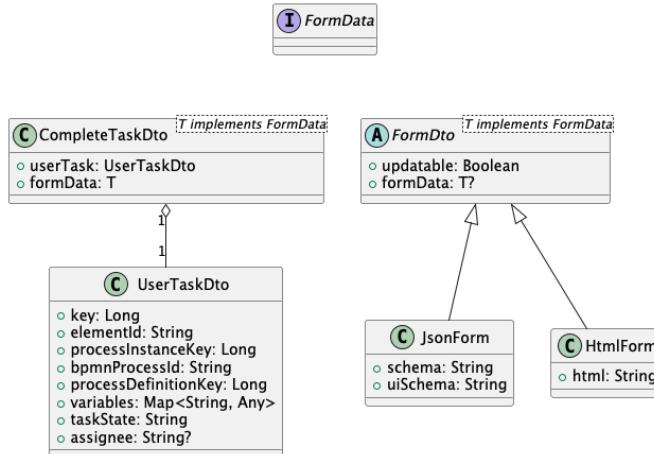


Abbildung 9.21: Klassendiagramm der Data Transfer Objects (DTO)

Die zweite Herausforderung hat mit der Formular-Generierung zu tun. Wie bereits erwähnt sollte, durch die Verwendung eines Low-Code-Ansatzes die Entwicklungszeit verkürzt werden. Daher wurde in die Task-Liste eine Component zur Anzeige von JSON Forms integriert. Diese Entscheidung wurde getroffen, da JSON Forms auf JSON Schema basiert. Für JSON Schema gibt es bereits Generatoren, die aus Plain-Old-Java-Objects (POJOs) JSON Schemas generieren und umgekehrt. Das Maven-Plugin *jsonschema-generator*²¹ kann aus einem POJO ein entsprechendes JSON Schemas generieren. Dabei

²¹<https://github.com/victools/jsonschema-generator/tree/cdf5aff71c6f2035fcdfa90c539f0465d055e10/jsonschema-maven-plugin>

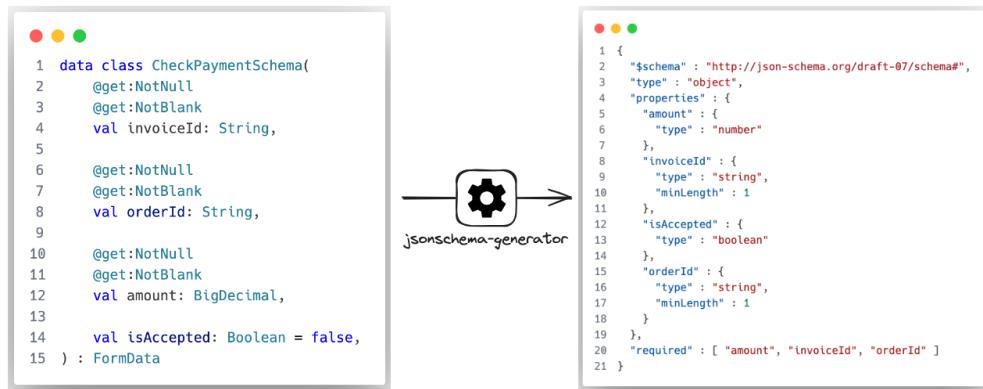


Abbildung 9.22: Generierung eines JSON Schemas aus einem POJO

werden die Annotationen aus der Bibliothek *jackarta-validation-api* berücksichtigt. Somit muss der Entwickler keine Kenntnisse über die Frontend-Entwicklung haben, um Formulare zu erstellen, die bereits die Validierung der Eingaben unterstützen. Mit dem Plugin *jsonschema2pojo*²² kann der Vorgang auch umgekehrt erfolgen. Aus dem JSON Schema wird ein POJO generiert, in der Prozess-Applikation verwendet werden kann. In Abbildung 9.22 ist ein Beispiel zu sehen, wie aus einem POJO mit Annotationen für die Validierung ein JSON Schema generiert wird. Daran ist auch zu erkennen, wie die Annotationen in das JSON Schema übernommen werden. Ist ein Attribut mit *@NotNull* annotiert, dann wird dieses im JSON Schema als *required* markiert. Ein *@NotBlank* führt zu einem *minLength* von 1. Das JSON Schema kann im Miranum JSON Forms Editor geöffnet werden und die im Schema definierten Felder sind als fertige Bausteine verfügbar und müssen nur noch per Drag-and-Drop in den Editor gezogen werden. Anschließend kann der Entwickler noch weitere Anpassungen vornehmen. Abbildung 9.23 zeigt ein Beispiel in dem unter *unscoped* die Felder als vorkonfigurierte Bausteine verfügbar sind.

9.6 Anleitung zum Starten des Prototypen

Dieser Abschnitt dient als Anleitung und Hilfestellung, um den PoC zu starten. Hierfür muss *Docker* installiert sein²³, da alle Komponenten in Containern laufen. Es wurde darauf geachtet, dass das Starten des PoC so einfach wie möglich ist. Daher wird *Docker Compose*²⁴ verwendet. Dadurch kann die gesamte Umgebung mit einem einzigen Befehl gestartet

²²<https://github.com/joelittlejohn/jsonschema2pojo>

²³<https://docs.docker.com/engine/install/>

²⁴<https://docs.docker.com/compose/>

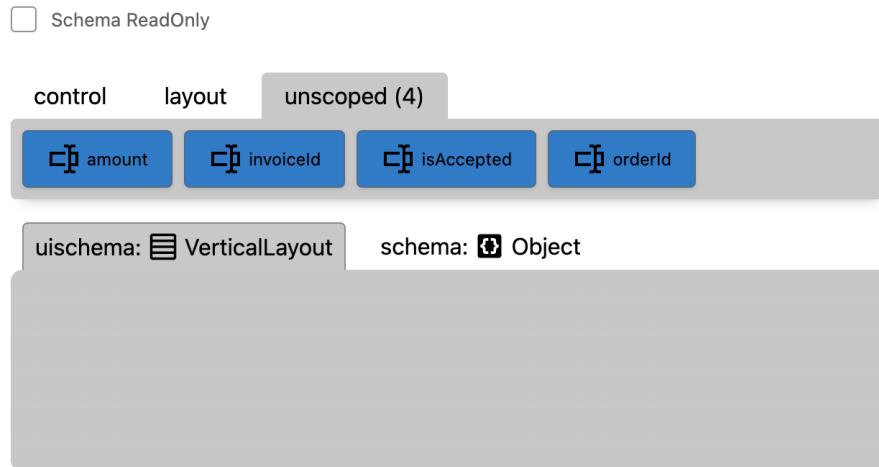


Abbildung 9.23: Vorkonfigurierte Bausteine im Miranum JSON Forms Editor

und konfiguriert werden. Die Konfiguration wird in einer `docker-compose.yml`²⁵ Datei definiert.

In der Standardkonfiguration werden *JobWorker* verwendet, um die User-Tasks von der Zeebe-Engine zu erhalten. Die Konfiguration kann jedoch angepasst werden, um einen *Exporter* zu nutzen. Dafür wurde in der `docker-compose.yml` Datei Kommentare gesetzt mit dem prefix *TODO*. Viele Editoren wie *IntelliJ* können diese Kommentare hervorheben, um sie leichter zu finden. Zusätzlich muss in den einzelnen BPMN Diagrammen als Option der „Zeebe user task“ ausgewählt werden. Das wurde bereits in Abbildung 9.16 exemplarisch aufgezeigt. Die beiden Prozess-Applikationen müssen im Anschluss entweder neu gebaut und gestartet werden oder die Prozesse neu werden neu deployed.

Docker wird die notwendigen Images herunterladen sowie die Komponenten, **Task-Liste**, **Task-Manager**, **Bestell-Applikation** und **Bezahl-Applikation** bauen und starten. Es wird zusätzlich die Camunda Komponente *Operate* verwendet, die für den PoC nicht notwendig ist. Dadurch ist es jedoch möglich den Prozess zu überwachen und zu analysieren. Im Browser kann Operate unter `http://localhost:8081` aufgerufen werden. Die Anmeldung erfolgt mit den Standard-Credentials *demo/demo*. Die Task-Liste ist unter `http://localhost:8080` erreichbar.

²⁵<https://github.com/Miragon/ma-zeebe-taskmanagement/blob/2e5fe3631a48fc9ca22f2a0c0901735f0a35b2ac/stack/docker-compose.yml>

10 Zusammenfassung und Ausblick

Mit dieser Arbeit wurde sich mit der Frage beschäftigt, wie eine leichtgewichtige Architektur für das Task-Management in einer Microservice-Umgebung aussehen kann. Unternehmen mit einer Microservice-Architektur, die verschiedene Quellen für Benutzaufgaben aufweist als auch Unternehmen, die sich mit der Migration auf die Zeebe-Engine beschäftigen, können von dieser Arbeit profitieren. Das zentrale Ergebnis ist ein Prototyp, der die Machbarkeit der Architektur zeigt.

Dabei wurde der BPMN 2.0 Standard sowie eine BPMN-fähige Engine verwendet. Diese Entscheidung wurde getroffen, da der BPMN 2.0 Standard eine Einbindung von Menschen in einen Geschäftsprozess bereits vorsieht. Engines die diesen Standard unterstützen besitzen somit auch die notwendigen technischen Voraussetzungen. Zusätzlich bietet BPMN 2.0 eine Notation, die auch von Fachabteilungen verstanden wird und für Entwickler eine grafische Darstellung des Prozesses bietet, der sich eventuell über mehrere Microservices erstreckt.

Im Hinblick auf die Task-Liste wurden zwei bestehende Lösungen betrachtet. Daraus ergaben sich Anforderungen. Davon wurden die wichtigsten mit dem Prototyp umgesetzt. Diese umfassen die Anzeige, das Bearbeiten, Abschließen und Aktualisieren von User Tasks. Zusätzlich können Prozessinstanzen über die Task-Liste gestartet werden.

Des Weiteren wurde mit dieser Arbeit die Zeebe-Engine mit ihrem Vorgänger verglichen. Dabei wurde festgestellt, dass die Zeebe-Engine eine vollständig neue Architektur einführt, die auf Event-Streaming Mechanismen basiert. Durch die Aufteilung der Engine in mehrere Broker erreicht Zeebe horizontale Skalierbarkeit und eine hohe Fehlertoleranz und ist speziell für Microservice-Umgebungen geeignet.

Das ausgegebene Ziel dieser Arbeit war einen ersten Durchstich zu erreichen, um die Machbarkeit der Architektur zu zeigen. Die Anforderungen wurden vor allem aus technischer Sicht definiert. Im Folgenden werden die Anforderungen noch einmal wiederholt und aufgezeigt, wie die vorgeschlagene Architektur diese Anforderungen erfüllt.

- **KISS-Prinzip:** Durch die Wahl einer hexagonalen Architektur und dem zentralen Task-Manager wird die Komplexität der Architektur reduziert. Es gibt nur eine zentrale Anlaufstelle für das Task-Management und neue Quellen für User Tasks

können einfach durch Adapter angebunden werden. Des Weiteren wurde eine einfache Service-Registry implementiert, die das Problem der Service-Discovery löst und über eine Konfigurationsdatei konfiguriert werden kann.

- **Flexibilität:** Es wurde darauf geachtet möglichst wenig Vorgaben zu machen, bzw. alternative Lösungswege anzubinden. Natürlich besteht im Backend durch die Microservice-Umgebung bereits eine gewisse Flexibilität. Die Task-Liste legt jedoch fest, dass für Prozessanwendungen REST als Kommunikationsschnittstelle angeboten werden muss. Durch die Implementierung des *iframes* kann diese Vorgabe umgangen werden. Da in einem *iframe* der Entwickler die volle Kontrolle hat, kann er auch seine eigenen HTTP-Requests implementieren und somit auch GraphQL verwenden.
- **Engine-Kompatibilität:** Hierfür wurde beispielhaft die Zeebe-Engine angebunden.
- **Unterstützung im Frontend:** JSON Forms wurde als Technologie ausgewählt, um Entwickler mit wenig Erfahrung in der Frontend-Entwicklung zu unterstützen. Diese Technologie ermöglicht es Formulare nahezu automatisch aus POJOs zu generieren. Sodass zum einen die Entwicklungszeit verkürzt wird und zum anderen die Fehleranfälligkeit bezüglich Dateninkonsistenzen reduziert wird.
- **Einsatz von Open-Source Technologien:** Es wurden ausschließlich Open-Source Technologien eingesetzt. Dazu zählt die Zeebe-Engine selbst aber auch JSON Forms und die Miranum IDE.
- **Docker:** Der komplette Prototyp läuft in Docker-Containern.

Allerdings ist wie bereits erwähnt der Prototyp nur ein erster Durchstich und vor allem auf die technischen Anforderungen ausgerichtet. Hierfür wäre eine Evaluation in einem realen Szenario notwendig um auch fachliche Anforderungen zu berücksichtigen. Auch wurden die Themen Sicherheit und Performance ausgeklammert. Es wäre z.B. noch interessant zu untersuchen, wie ein dezentraler Task-Manager umgesetzt werden kann und welche Vorteile aber auch Herausforderungen dabei entstehen. Des Weiteren könnte die Anbindung weiterer Quellen, wie ein ERP-System oder auch einer nicht BPMN-fähigen Workflow-Engine wie Temporal, interessant sein. Zuletzt ist noch die Task-Liste zu erwähnen. Dieser fehlen noch Funktionen bezüglich Filtern, Sortieren, Delegieren, Priorisieren und Suchen. Auch hinsichtlich des Designs gibt es noch Optimierungsbedarf. Hier wäre es interessant Web-Components als Alternative zu den verwendeten *iframe* gegenüberzustellen.

An sich zeigt der Prototyp jedoch, dass die Architektur funktioniert und die Anforderungen erfüllt.

Abbildungsverzeichnis

3.1	Architektur des von Kreifelts u. a. [13, Abb. 4] entworfenen Task-Manager	7
3.2	Vertikaler Aufbau eines Micro-Frontends	8
3.3	Horizontaler Aufbau eines Micro-Frontends	9
4.1	Auflistung an Funktionalitäten die durch die Tasklist API bzw. Zeebe Task API abgedeckt sind [24, "Migrate to Zeebe user tasks"].	11
5.1	Vereinfachte Darstellung des BPM-Lifecycle nach Macedo de Moraes u. a. [28, Abb. 10]	13
5.2	Vereinfachte Darstellung einer Workflow-Engine [5, Kap. 2]	14
5.3	Beispiel für einen Workflow in Temporal	16
5.4	Beispiel für einen Workflow in BPMN 2.0	16
5.5	Die verschiedenen Typen von Tasks in BPMN 2.0	18
5.6	Die verschiedene Darstellungen eines Sub-Prozesses in BPMN 2.0	19
5.7	Beispiel für <i>interrupting</i> und <i>non-interrupting</i> Boundary-Events	19
5.8	Die verschiedenen Typen von Events in BPMN 2.0 [31, Tab. 10.93]	20
5.9	Die verschiedenen Typen von Gateways in BPMN 2.0	21
5.10	Vereinfachter Bestellprozess als BPMN Diagramm	22
5.11	Vereinfachter Bezahlprozess als BPMN Diagramm	23
6.1	Screenshot der Camunda Tasklist [24, "Overview and example use case"]	26
6.2	Mögliche Zustände und Zustandsübergänge eines User-Task	26
6.3	Screenshot von Flowable-Work	27
6.4	Screenshot des Camunda Form-Builders	29
6.5	Screenshot des Flowable Form-Builders	30
6.6	Miranum JSON Forms Editor in der Miranum IDE.	31
6.7	Ein Beispiel für eine JSON Schema und UI Schema Datei.	32
6.8	Das gerenderte Formular als Ergebnis der beiden JSON Dateien aus Abbildung 6.7.	32
7.1	Architekturvergleich von C7 [18, "Architecture Overview"] und C8 [24, "Architecture"]	37
8.1	Architektur des PoC	40

8.2	Sequenzdiagramm des Order-to-Cash Prozesses	42
9.1	Abstrakte Darstellung der hexagonalen Architektur [36, Abb. 3.4]	45
9.2	Mögliche Projektstruktur der hexagonalen Architektur	46
9.3	Verwendete Programmiersprachen und Frameworks	47
9.4	Use-Case Diagramm der Task-Liste	49
9.5	Aktualisieren und abschließen eines Tasks	50
9.6	Starten eines Prozesses	50
9.7	Klassendiagramm der Service-Registry	51
9.8	Horizontaler Aufbau der Task-Liste	52
9.9	Screenshot der Task-Liste	53
9.10	Komponentendiagramm der hexagonalen Architektur des Task-Managers	54
9.11	Der Task-Manager und alle angeschlossenen Systeme	55
9.12	Architektur eines JobWorkers in der Zeebe-Engine [24, “Job workers”]	57
9.13	JobWorker für User-Tasks in der Zeebe-Engine	57
9.14	Event-Stream mit verschiedenen Records [40, “The event handling loop”]	59
9.15	Exporter, der User-Tasks an den Task-Manager sendet	60
9.16	Konfiguration des BPMN Diagramms für die Zeebe Task API	61
9.17	Konfiguration des Zeebe-Brokers und des Exporters	62
9.18	Vereinfachtes UML-Komponentendiagramm der Bestell-Applikation . .	63
9.19	Ausschnitt aus dem Bestellprozess	64
9.20	Kommunikation zwischen Task-Liste und Prozess-Applikation	66
9.21	Klassendiagramm der Data Transfer Objects (DTO)	66
9.22	Generierung eines JSON Schemas aus einem POJO	67
9.23	Vorkonfigurierte Bausteine im Miranum JSON Forms Editor	68

Literaturverzeichnis

- [1] N. Alshuqayran, N. Ali und R. Evans, „A Systematic Mapping Study in Microservice Architecture“, in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 2016, S. 44–51. doi: 10.1109/SOCA.2016.15 (siehe S. 3, 4).
- [2] S. Newman, *Building Microservices*. Ö'Reilly Media, Inc.", 2021 (siehe S. 3–9).
- [3] M. Fowler und J. Lewis, *Microservices*, <https://martinfowler.com/articles/microservices.html>, [Online; accessed 01. September 2024], 2014 (siehe S. 3).
- [4] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu und J. Nieh, „Synapse: a microservices architecture for heterogeneous-database web applications“, in *Proceedings of the tenth european conference on computer systems*, 2015, S. 1–16 (siehe S. 5).
- [5] B. Ruecker, *Practical Process Automation*. Ö'Reilly Media, Inc.", 2021 (siehe S. 5, 6, 14, 15, 33, 63, 71).
- [6] B. M. Michelson, „Event-driven architecture overview“, *Patricia Seybold Group*, Jg. 2, Nr. 12, S. 10–1571, 2006 (siehe S. 5).
- [7] D. Jaramillo, D. V. Nguyen und R. Smart, „Leveraging microservices architecture by using Docker technology“, in *SoutheastCon 2016*, IEEE, 2016, S. 1–5 (siehe S. 6).
- [8] B. Burns, J. Beda, K. Hightower und L. Evenson, *Kubernetes: up and running*. Ö'Reilly Media, Inc.", 2022 (siehe S. 6).
- [9] V. Bellotti, B. Dalal, N. Good, P. Flynn, D. G. Bobrow und N. Ducheneaut, „What a to-do: studies of task management towards the design of a personal task list manager“, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Ser. CHI '04, Vienna, Austria: Association for Computing Machinery, 2004, S. 735–742. doi: 10.1145/985692.985785 (siehe S. 7).
- [10] R. Abdellaoui, I. Stasicka, M. Finke, A. Pick und N. Ahrenhold, „Supporting Air Traffic Controllers in handling sector specific tasks, enabled by the use of the Boundary Arrival Task Manager“, in *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, 2023, S. 1–10. doi: 10.1109/DASC58513.2023.10311241 (siehe S. 7).
- [11] R. Tommy, C. Kurniawan, Niccosan und B. A. Makalew, „Improving Employee Performance Through Digitalization: Designing a Web Based Human Resource Management“, in *2022 International Conference on Information Management and Technology (ICIMTech)*, 2022, S. 655–660. doi: 10.1109/ICIMTech55957.2022.9915155 (siehe S. 7).
- [12] D. Schulte, „Towards a Human Task Management Reference Model.“, in *ZEUS*, Citeseer, 2012, S. 104–111 (siehe S. 7).
- [13] T. Kreifelts, E. Hinrichs und G. Woetzel, „Sharing To-Do Lists with a Distributed Task Manager“, in *Proceedings of the Third European Conference on Computer-Supported Cooperative Work 13–17 September 1993, Milan, Italy ECSCW '93*, G. de Michelis, C. Simone und K. Schmidt, Hrsg.

Dordrecht: Springer Netherlands, 1993, S. 31–46. doi: 10.1007/978-94-011-2094-4_3 (siehe S. 7, 71).

- [14] C. Jackson, *Micro Frontends*, <https://martinfowler.com/articles/micro-frontends.html>, [Online; accessed 10. September 2024], 2019 (siehe S. 7, 9).
- [15] M. Geers, *Micro Frontends in Action*. "Manning Publications", 2020 (siehe S. 7, 9).
- [16] L. Mezzalira, *Building Micro-Frontends*. Ö'Reilly Media, Inc.", 2021 (siehe S. 7–9).
- [17] K. Leung und J. Chung, „The Liaison Workflow engine architecture“, in *Proceedings of the 32nd Annual Hawaii International Conference on Systems Sciences. 1999. HICSS-32. Abstracts and CD-ROM of Full Papers*, Bd. Track5, 1999, 10 pp.-. doi: 10.1109/HICSS.1999.772960 (siehe S. 10).
- [18] Camunda, *The Camunda 7 Manual*, <https://docs.camunda.org/manual/7.21/>, [Online; accessed 22. May 2024], 2024 (siehe S. 10, 14, 34, 36, 37, 71).
- [19] G. Yu, P. Chen und Z. Zheng, „Microscaler : Cost-Effective Scaling for Microservice Applications in the Cloud With an Online Learning Approach“, *IEEE Transactions on Cloud Computing*, Jg. PP, S. 1–1, Apr. 2020. doi: 10.1109/TCC.2020.2985352 (siehe S. 10).
- [20] Z. Ding, Y. Zhou, S. Wang und C. Jiang, „SCAFE: A Service-Centered Cloud-Native Workflow Engine Architecture“, *IEEE Transactions on Services Computing*, Jg. 16, Nr. 5, S. 3682–3695, 2023. doi: 10.1109/TSC.2023.3259989 (siehe S. 10, 14).
- [21] *Conductor*, <https://github.com/Netflix/conductor>, [Online; accessed 12. September 2024], 2024 (siehe S. 10, 14).
- [22] *Cadence*, <https://github.com/uber/cadence>, [Online; accessed 12. September 2024], 2024 (siehe S. 10).
- [23] Temporal, *Documentation*, <https://docs.temporal.io/>, [Online; accessed 12. September 2024], 2024 (siehe S. 10, 14, 15).
- [24] Camunda, *Camunda 8 Docs*, <https://docs.camunda.io/>, [Online; accessed 22. May 2024], 2024 (siehe S. 10, 11, 25, 26, 34, 37, 55–59, 64, 71, 72).
- [25] D. Elzinga, T. Horak, C.-Y. Lee und C. Bruner, „Business process management: survey and methodology“, *IEEE Transactions on Engineering Management*, Jg. 42, Nr. 2, S. 119–128, 1995. doi: 10.1109/17.387274 (siehe S. 12).
- [26] European Association of BPM, *Business Process Management BPM Common Body of Knowledge*. Verlag Dr. Götz Schmidt GmbH, 2009 (siehe S. 12).
- [27] R. G. Lee und B. G. Dale, „Business process management: a review and evaluation“, *Business process management journal*, Jg. 4, Nr. 3, S. 214–225, 1998 (siehe S. 12).
- [28] R. Macedo de Moraes, S. Kazan, S. Inês Dallavalle de Pádua und A. Lucirton Costa, „An analysis of BPM lifecycles: from a literature review to a framework proposal“, *Business Process Management Journal*, Jg. 20, Nr. 3, S. 412–432, 2014 (siehe S. 12, 13, 71).
- [29] J. Freund und B. Rücker, *Praxisbuch BPMN*, 5. Aufl. Carl Hanser Verlag München, 2017 (siehe S. 13, 17).
- [30] S. Alpers, C. Becker, A. Oberweis und T. Schuster, „Microservice based tool support for business process modelling“, in *2015 IEEE 19th International Enterprise Distributed Object Computing Workshop*, IEEE, 2015, S. 71–78 (siehe S. 14).

- [31] O. M. G. OMG, *Business Process Model and Notation (BPMN)*, <http://www.omg.org/spec/BPMN/2.0/>, 2011 (siehe S. 14, 17, 20, 22, 71).
- [32] S. A. White, „Introduction to BPMN“, *Ibm Cooperation*, Jg. 2, Nr. 0, S. 0, 2004 (siehe S. 17).
- [33] S. A. White u. a., „Process modeling notations and workflow patterns“, *Workflow handbook*, Jg. 2004, Nr. 265-294, S. 12, 2004 (siehe S. 17).
- [34] Flowable, *Enterprise Documentation*, <https://documentation.flowable.com/latest/>, [Online; accessed 22. May 2024], 2024 (siehe S. 25).
- [35] Gartner, *Citizen Developer*, <https://www.gartner.com/en/information-technology/glossary/citizen-developer>, [Online; accessed 20. September 2024], 2024 (siehe S. 28).
- [36] T. Hombergs, *Get Your Hands Dirty on Clean Architecture: A hands-on guide to creating clean web applications with code examples in Java*. Packt Publishing Ltd, 2019 (siehe S. 44, 45, 72).
- [37] A. Cockburn, *Hexagonal architecture*, <https://alistair.cockburn.us/hexagonal-architecture/>, [Online; accessed 14. September 2024], 2005 (siehe S. 44).
- [38] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2003 (siehe S. 44).
- [39] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, 2017 (siehe S. 44).
- [40] B. Ruecker, *How we built a highly scalable distributed state machine*, <https://blog.bernd-ruecker.com/how-we-built-a-highly-scalable-distributed-state-machine-f2595e3c0422>, [Online; accessed 20. September 2024], 2019 (siehe S. 58, 59, 72).