MAS Software Engineering 2025

Hydra

Multi-Incident Management - Simplified

Reto Kunz

Mario Leonini

Fabian Ott

OST – Ostschweizer Fachhochschule

Referent: Andres Koch

Co-Referent: Thomas Höltschi

15.09.2025

Hinweis zum Dokument

Bei dem vorliegenden Dokument handelt es sich um eine für Lehrveranstaltungen aufbereitete Version der Masterarbeit. Dabei wurden einzelne Inhalte aus Gründen des Datenschutzes und/oder des Urheberrechts entfernt, geschwärzt oder unkenntlich gemacht. Die Autoren erlauben der OST ausdrücklich die Verwendung dieser Version für Lehrveranstaltungen sowie die Publikation auf ePrints.

„Man soll die Dinge so einfach wie möglich machen, aber nicht einfacher.“

Albert Einstein

Danksagung

In erster Linie möchten wir uns bei unserem Betreuer Herrn Andres Koch bedanken. Er hat uns während der Arbeit in regelmässigen Meetings unterstützt und mit wertvollen Inputs sowie hilfreichen Vorschlägen wesentlich zum Gelingen dieser Arbeit beigetragen.

Ebenfalls danken wir unserem Co-Betreuer Herrn Thomas Höltschi, der uns bei der Zwischenpräsentation mit konstruktiven Rückmeldungen und guten Ideen unterstützt hat.

Weiter möchten wir der Feuerwehr [redacted] unseren Dank aussprechen. Kommandant [redacted] sowie sein Stellvertreter [redacted] haben uns in der Anfangsphase des Projekts einen umfassenden Einblick in ihre aktuelle Arbeitsweise gegeben und mit ihren Erfahrungen und Erwartungen an eine mögliche Softwarelösung entscheidend dazu beigetragen, die Anforderungen praxisnah zu gestalten.

Verwendungsrechte

Mit der OST - Ostschweizer Fachhochschule wurde folgende Vereinbarung getroffen:

- Die OST - Ostschweizer Fachhochschule darf die Masterarbeit zur internen Beurteilung und Prüfung verwenden, sich jedoch nicht an einer kommerziellen oder anderweitig weitergehenden Nutzung beteiligen.
- Nach Abschluss der Masterarbeit verbleiben die uneingeschränkten Nutzungs- und Verwendungsrechte, einschliesslich des Rechts auf Weiterentwicklung und kommerzielle Nutzung des Systems, bei den Verfassern.
- Die Veröffentlichung der Masterarbeit oder Teile davon - insbesondere des Source Codes - gegenüber Dritten ausserhalb des Prüfungskreises (z. B. Referent, Co-Referent, Studienleitung) ist ohne ausdrückliche Zustimmung der Verfasser nicht gestattet. Ausgenommen davon ist ein zur Veröffentlichung freigegebenes Abstract mit zugehörigem Titel der Masterarbeit, welches der OST zur Verfügung gestellt wird.

Die Vereinbarung zu den Verwendungsrechten wurde von der Studiengangleitung des MAS-SE akzeptiert und unterzeichnet. Das Original der Vereinbarung ist als beiliegendes Dokument *Antrag-Verwendungsrechte.pdf* zu finden.

Hinweis: Bei dem vorliegenden Dokument handelt es sich um eine gekürzte, geschwärzte Version der Masterarbeit, nicht um die in der Vereinbarung erwähnte originale Arbeit. Die Autoren erlauben der OST ausdrücklich die Verwendung dieser Version für Lehrveranstaltungen sowie die Publikation auf ePrints. Abgesehen davon bleibt die obige Vereinbarung unverändert bestehen.

Abstract

Hydra - Multi-Incident Management - Simplified

Die vorliegende Masterarbeit beschäftigt sich mit der Entwicklung eines Softwaresystems zur Unterstützung der freiwilligen Feuerwehr bei der Bewältigung von Mehrfachereignissen, wie z. B. Hochwasser. Das System *Hydra* zielt darauf ab, die administrative und kommunikative Belastung während solcher Ereignisse zu reduzieren. Die webbasierte Plattform ermöglicht die Erfassung und Priorisierung von Meldungen sowie das Zuweisen von Einsatzkräften zu Ereignissen. Die Visualisierung laufender Einsätze auf einer interaktiven Karte sowie ein Kanban-Board bieten eine gute Übersicht über die aktuelle Lage. Diese Visualisierung ist in zwei Varianten verfügbar, einerseits für den Einsatzleiter und andererseits für die Truppchefs.

Die Arbeit basiert auf einer umfassenden Analyse der Anforderungen und Prozesse der themengebenden Feuerwehr. Diese identifizierte die Notwendigkeit eines einfachen und erweiterbaren Systems, das ohne grossen Schulungs- und Wartungsaufwand eingesetzt werden kann.

Erste Rückmeldungen der Anwender zeigen, dass eine einfache übersichtliche Darstellung der Ereignisse die administrative Belastung reduziert und dazu beitragen kann, die Effizienz der Einsatzkräfte zu steigern.

Verfasser/innen:	Reto Kunz, Mario Leonini, Fabian Ott
Referent/in:	Andres Koch
Co-Referent/in:	Thomas Höltschi
Veröffentlichung (Jahr):	2025
Zitation:	Reto Kunz, Mario Leonini, Fabian Ott, Hydra - Multi-Incident Management - Simplified. OST - Ostschweizer Fachhochschule: Masterarbeit
Schlagworte:	Software Engineering, Softwarearchitektur, Webanwendung, API, Feuerwehr, Hydra

Management Summary

Ausgangslage

Mehrfachereignisse entstehen häufig im Zusammenhang mit Unwettern wie Hochwasser oder Stürmen und führen in kurzer Zeit zu vielen, räumlich verteilten Einsätzen. Dabei wird die Einsatzleitung durch die Komplexität, welche die Koordination von Einsatzkräften und Material bei einer Vielzahl von Ereignissen mit sich bringt, stark gefordert. Gerade freiwillige Feuerwehren verfügen im Gegensatz zu Berufsfeuerwehren oft nicht über umfangreiche Einsatzleitsysteme und arbeiten deshalb mit einfachen Hilfsmitteln wie Flipcharts oder Whiteboards. Dies erschwert zusätzlich die strukturierte Erfassung von Meldungen, die Priorisierung von Massnahmen sowie die Koordination von Einsatzkräften.

Vor diesem Hintergrund wurde *Hydra* konzipiert: Eine einfach zu verwendende, erweiterbare Weblösung, die dabei hilft, eingehende Meldungen systematisch zu erfassen, Ereignisse zu priorisieren sowie Trupps und Material zu disponieren. Zudem bietet das System auf einer Karte eine übersichtliche Visualisierung der aktuellen Lage.

Vorgehen

Die Arbeit folgte einem agilen, mehrstufigen Vorgehen mit einer initialen Analyse- und Architekturphase sowie kurzen Iterationen in der Umsetzung unter Einbezug der themengebenden Feuerwehr. Zu Beginn wurden Problemstellung und Prozesse mittels Interviews, Dokumentenanalyse und Beobachtung erhoben. Daraus entstanden klar definierte User Stories und ein priorisierter Backlog für das Erstellen eines *Minimum Viable Products (MVP)*. Ein früher Softwareprototyp sorgte dafür, dass Ende-zu-Ende-Funktionalität schnell sichtbar wurde und Feedback unmittelbar in die nächsten Iterationen einfloss. Zur Erarbeitung einer gemeinsamen fachlichen Sprache wurden in der Analysephase Methoden aus dem *Domain-Driven Design (DDD)* eingesetzt.

Für die Umsetzung kamen bewährte Engineering-Praktiken zum Einsatz: versioniertes Arbeiten mit Git, Code-Reviews, Unit- und Integrationstests sowie automatisierte Builds und Qualitätssicherung in CI/CD-Pipelines. Eine einfache Observability (Logging, Metriken) aller Systemkomponenten wurde von Beginn an berücksichtigt.

Technologisch basiert das Backend auf Java 21 mit Spring Boot, das Frontend auf React mit TypeScript und Vite. Einsatzdaten werden in MongoDB verwaltet, als Event Streaming Platform wird Kafka eingesetzt. Die Kartenfunktionalität wurde mit OpenLayers realisiert. Containerisierung mit Docker und automatisierte Pipelines auf GitLab ermöglichen automatisierte Builds, Tests sowie halbautomatisierte Deployments. Die synchrone Kommunikation zwischen den Sytemkomponenten wurde ausschliesslich mit REST-Schnittstellen realisiert, für asynchrone Kommunikation kommen Kafka und Websockets zum Einsatz. Diese Schnittstellen wurden mit OpenAPI respektive AsyncAPI definiert.

Erreichte Ziele und Erkenntnisse

Als Ergebnis dieser Arbeit liegt ein voll funktionsfähiges *MVP* der *Hydra*-Applikation vor. Diese unterstützt den gesamten Ablauf von der Erfassung von Meldungen über die Kategorisierung und Priorisierung bis zur Abarbeitung von Ereignissen. Das System visualisiert die aktuelle Lage auf einer Übersichtskarte, auf welcher alle Meldungen, Einsätze und Truppstandorte ersichtlich sind. Zudem bietet ein Kanban-Board eine gute Übersicht über den aktuellen Zustand der zu bearbeitenden Ereignisse.

Die Bedienung erfolgt zielgruppengerecht über zwei Oberflächen: Im umfangreicheren Operator-UI steuern Einsatzleitung und Führungsunterstützung das Geschehen in der Zentrale, während das kompaktere Field-UI für Einsatzkräfte im Ausseneinsatz auf schnelle Eingaben und mobile Nutzung ausgelegt ist.

Ausserdem erlaubt es die Architektur des Systems, einfach zusätzliche Komponenten wie Archivierung, Dokumentation oder eine Stammdatenverwaltung einzubinden.

Die entwickelte Lösung zeigt, dass auch kleinere Feuerwehren mit begrenzten Mitteln von einer digitalen Unterstützung profitieren können. In einer Vorführung bei der themengebenden Feuerwehr wurde die Applikation sehr positiv aufgenommen. Das Feedback bestätigte die grundsätzliche Tauglichkeit im Einsatzalltag und lieferte zugleich weitere Featurewünsche und Verbesserungsvorschläge.

Literaturquellen

Für Domänenanalyse und architektonische Leitentscheide waren insbesondere *Domain-Driven Design* (Evans, 2003) sowie *Clean Architecture* (Martin, 2017) richtungsweisend. Zusätzlich wurden öffentlich zugängliche Ausbildungsunterlagen des Feuerwehrverbands Thurgau wie z. B. das Dokument „Einsatzablauf Mehrfachereignis“ (Thurgaufire, n. d.) herangezogen. In der Design- und Implementationsphasen wurden unter anderem Konzepte aus *Clean Code* (Martin, 2008) und *Design Patterns* (Gamma et al., 1994) angewandt. Technische Diagramme orientieren sich an *UML 2.5* (Kecher & Salvanos, 2015). Eine vollständige Auflistung der verwendeten Literatur ist im Quellenverzeichnis zu finden.

Inhaltsverzeichnis

1. Einleitung

Ereignisse wie Stürme, Überschwemmungen oder grossflächige Brände stellen die Feuerwehren regelmässig vor besondere Herausforderungen. Werden durch ein solches Ereignis mehrere Schadensmeldungen an verschiedenen Standorten ausgelöst, spricht man von einem Mehrfachereignis. In diesen Situationen muss die Einsatzleitung gleichzeitig verschiedene Einsatzorte im Blick behalten, Prioritäten festlegen, Einsatztrupps koordinieren und den Überblick über verfügbares Material bewahren. Die Gleichzeitigkeit der Einsätze führt dabei zu einer hohen Komplexität und einem erheblichen organisatorischen Aufwand.

Besonders für die freiwilligen Feuerwehren stellt dies ein grosses Problem dar. Sie verfügen in der Regel über begrenzte Mittel, wenig Erfahrung mit derartigen Extremereignissen und müssen trotzdem in kurzer Zeit viele wichtige Entscheidungen treffen. Während professionelle Berufsfeuerwehren oft auf umfangreiche Einsatzführungssysteme zurückgreifen können, stehen den freiwilligen Feuerwehren meist nur einfache Hilfsmittel zur Verfügung. Dies erschwert die strukturierte Erfassung von Meldungen, die Koordination der Einsatzkräfte sowie die Priorisierung von Massnahmen.

Die Kombination aus hohem Koordinationsaufwand und begrenzten Ressourcen macht eine schlanke, praxistaugliche Unterstützungslösung erforderlich. Das hier erarbeitete System *Hydra* verfolgt das Ziel, Meldungen strukturiert zu erfassen, Einsätze sachgerecht zu priorisieren und Trupps sowie Material koordiniert einzusetzen. Dies bei geringem Schulungs- und Betriebsaufwand sowie einer Ausrichtung auf die Bedürfnisse freiwilliger Feuerwehren.

Der Name *Hydra* verweist auf das mehrköpfige Wesen der griechischen Mythologie. Die Köpfe dienen als Metapher für die Vielzahl parallel zu bearbeitenden Aufträge während eines Mehrfachereignisses. Zugleich steht *Hydra* für die enge Verbindung der Feuerwehr mit dem Element Wasser, sei es beim Löschen von Bränden oder beim Bekämpfen von Hochwasser.

2. Ausgangslage

Mehrfachereignisse entstehen häufig im Zuge von Grossereignissen wie Hochwasser oder Stürmen. Zu Beginn ist nicht immer erkennbar, dass es sich um mehr als ein einzelnes Ereignis handelt. Bei der ersten Alarmierung wird zunächst ein normaler Einsatz ausgelöst. Erst mit den nachfolgenden Meldungen wird sichtbar, dass ein Mehrfachereignis vorliegt. Oft ist nicht abschätzbar, wie viele Meldungen in welchem zeitlichen Abstand noch folgen. Es kann durchaus vorkommen, dass fünfzig Ereignisse oder mehr innerhalb von ein bis zwei Stunden gemeldet werden.

Freiwilligen Feuerwehren steht für die Koordination solcher Lagen häufig keine unterstützende Software zur Verfügung. Zwar existieren Lösungen auf dem Markt, diese sind jedoch meist Teil umfangreicher Einsatzleitsysteme mit hohen Anschaffungs- und Lizenzkosten, erheblichem Schulungsbedarf und einigem Wartungsaufwand. Einfache und intuitive Systeme, die gezielt auf die Unterstützung bei Mehrfachereignissen ausgerichtet sind, fehlen. In der Praxis kommen deshalb nur punktuell frei verfügbare Hilfsmittel für einzelne Anwendungsfälle zum Einsatz.

Vor diesem Hintergrund hat die themengebende Feuerwehr [redacted] den Wunsch nach einer einfachen, erweiterbaren und praxistauglichen Lösung für die Koordination von Mehrfachereignissen geäussert.

Die genauen Abläufe zur Bewältigung von Ereignissen können sich kantonal oder gar von Feuerwehr zu Feuerwehr unterscheiden. Die nachfolgende Beschreibung stützt sich exemplarisch auf die Abläufe der themengebenden Feuerwehr sowie auf Ausbildungsunterlagen des Feuerwehrverbands Thurgau. Das Dokument „Einsatzablauf Mehrfachereignis“ (Thurgaufire, n. d.) und weitere Unterlagen sind online frei verfügbar und geben Einblick in typische Abläufe und Herausforderungen bei der Bewältigung von Mehrfachereignissen.

2.1. Referenzprozess Mehrfachereignis

Das nachfolgende UML-Aktivitätsdiagramm in Abbildung 2.1 zeigt einen groben Prozessablauf bei einem Mehrfachereignis. Ein solches beginnt in der Regel wie ein Einzelereignis: Ein Geschädigter meldet einen Schaden telefonisch bei der *kantonalen Notrufzentrale (KNZ)*. Die *KNZ* nimmt die Meldung entgegen, triagiert und leitet sie an die zuständige örtliche Feuerwehr weiter. Diese Erstalarmierung erfolgt per Telefon und SMS. Daraufhin begeben sich die Einsatzkräfte in das Feuerwehrmagazin und beginnen mit der Bearbeitung des Ereignisses. Treffen im Anschluss weitere Meldungen bei der *KNZ* ein, wechselt diese in den „Mehrfachereignis-Modus“. Die bereits im Einsatz stehende Feuerwehr erhält die neu eintreffenden Meldungen von der *KNZ* per E-Mail. Diese werden in der Einsatzzentrale erfasst und durch die Einsatzkräfte im Feld bearbeitet. Nach Abschluss eines Ereignisses erfolgt, sofern sinnvoll, eine Rückmeldung an den Melder bzw. die Melderin. Sind alle Ereignisse abgearbeitet und treffen keine neuen Meldungen mehr ein, wird der Gesamteinsatz beendet.

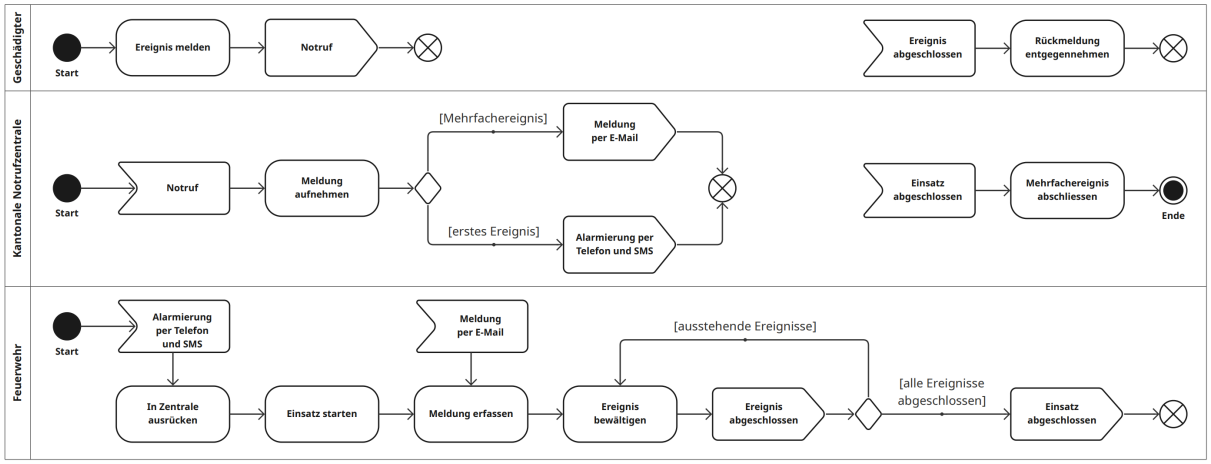


Abbildung 2.1.: UML-Aktivitätsdiagramm Mehrfachereignis

2.2. Bewältigung Mehrfachereignis

Sobald ein Mehrfachereignis erkannt ist, stellt auch die Einsatzzentrale der örtlichen Feuerwehr auf den entsprechenden Modus um. Heute ist die Koordination von mehreren Ereignissen typischerweise papierbasiert organisiert: Ein Whiteboard mit *Lagekarte* und ein Kanban-ähnliches Board – im Fachjargon *„Journal“* genannt – bilden die zentralen Führungsinstrumente.

Neue Meldungen werden auf einem speziellen Formular erfasst und anschliessend aufgeteilt: Der Hauptteil verbleibt bei der Einsatzleitung, ein Abschnitt mit Informationen für die Rekognoszierung geht an den Erkundungstrupp, ein weiterer Teil wird im Journal auf dem Whiteboard angeheftet. Diese pragmatische Vorgehensweise ist einfach und setzt wenig Schulung voraus. Mit zunehmender Ereigniszahl und einer sich dynamisch verändernden Lage stösst sie jedoch schnell an Grenzen. Die Lagekarte muss kontinuierlich aktualisiert werden. Das ist auf Papier unhandlich, da einmal eingetragene Notizen sich nicht ohne Weiteres korrigieren lassen.

Auch die Informationsdichte auf Laufzetteln und im Journal ist begrenzt. Häufig wird beim Einsatztrupp nur der Name des Truppchefs notiert. Dadurch ist es schwierig, alle im Einsatz stehenden Kräfte sowie deren Aufenthaltsorte verlässlich im Blick zu behalten. Beim Material werden meist lediglich die Fahrzeuge erfasst, obwohl in vielen Situationen detailliertere Angaben zweckmässig wären. Hinzu kommt, dass örtliche Feuerwehren bei Grossereignissen oft Unterstützung durch Nachbarwehren erhalten. Diese temporären personellen und materiellen Ressourcen müssen ebenfalls verwaltet werden, was vom vorhandenen Prozess nur unzureichend unterstützt wird. Um den Überblick zu wahren, sind häufig zusätzliche, nicht standardisierte Notizen nötig. Trotz des minimalistisch gehaltenen Journals wird es bei vielen Ereignissen schnell schwierig, den Gesamtüberblick zu behalten und sowohl Journal als auch Lagekarte auf dem aktuellen Stand zu halten.

Auch die Kommunikation mit den Einsatzkräften im Feld ist herausfordernd. Neben den etablierten Funkkanälen werden für die Bestandsaufnahme in der Praxis teilweise Bilder und Kurznachrichten über Messenger-Dienste wie WhatsApp genutzt. Gemäss Ausbildungsunterlagen gehören solche Informationen zur Einsatzdokumentation. In der Nachbearbeitung kann das zu erheblichem Mehraufwand führen, da verstreute Chatverläufe manuell zusammengetragen werden müssen.

2.3. Zielsetzung

Die Herausforderung bei Mehrfachereignissen besteht darin, viele sich dynamisch entwickelnde Schadenereignisse gleichzeitig zu führen. Meldungen treffen in unbekannter Anzahl und in unregelmässigen Abständen ein, Prioritäten ändern laufend, und der aktuelle Status von Trupps sowie die Verfügbarkeit von Material sind unübersichtlich. Die heute üblichen, überwiegend manuellen Führungsinstrumente (Whiteboard/Lagekarte, Journal, Laufzettel) sind in dieser Situation nur schwer aktuell zu halten, fehleranfällig und skalieren schlecht. Rückmeldungen aus dem Feld erreichen die Zentrale über unterschiedliche Kanäle (Funk, E-Mail, Messenger mit Fotos) und sind während des Einsatzes und in der Nachbearbeitung nur mit hohem Aufwand zusammenzuführen. Ein konsistentes, jederzeit aktuelles Lagebild ist damit nur eingeschränkt verfügbar.

Benötigt wird eine leichtgewichtige, praxistaugliche Lösung, die Meldungen strukturiert erfasst, die Priorisierung und Disposition unterstützt, den Status von Trupps und Material transparent macht, ein laufend aktualisiertes Lagebild bereitstellt sowie den Informationsfluss zwischen Zentrale und Feld unterstützt. Hier soll *Hydra* Abhilfe schaffen. Zudem soll das System den Aufwand für die Erstellung der Einsatzdokumentation erheblich reduzieren und somit auch Debriefings vereinfachen.

Bestimmte Funktionen liegen bewusst ausserhalb des angestrebten Systemumfangs. Die Einsatzleitung von Einzelereignissen sowie die Routenführung zum Einsatzort werden nicht abgedeckt, da hierfür bereits geeignete Systeme vorhanden sind. Wegen Datenschutzbedenken und aufgrund des sehr beschränkten Nutzens wird zudem auf die GPS-Lokalisierung einzelner Einsatzkräfte verzichtet. Gemäss der themengebenden Feuerwehr existieren derzeit keine verbindlichen rechtlichen Vorgaben zur Dokumentation und Nachvollziehbarkeit des Einsatzablaufs.

Aufbauend auf dieser Ausgangslage wurde in der vorliegenden Arbeit eine auf die Bedürfnisse freiwilliger Feuerwehren zugeschnittene Softwarelösung erarbeitet. Das folgende Kapitel *Vorgehensweise und Methodik* erläutert die dafür angewandte Vorgehensweise im Detail.

`Begriffliche Klarstellung:` Die in diesem Dokument häufig vorkommenden Domänenbegriffe *Meldung* und *Ereignis* bezeichnen fachliche Objekte (z. B. Schadensmeldung bzw. konkreter Einsatzfall) und sind nicht mit den technischen Begriffen *Meldung* (*Message*, z. B. Nachricht in asynchroner Kommunikation) oder *Ereignis* (*Event*, technisches Ereignis) zu verwechseln. Wo erforderlich, wird im Folgenden explizit zwischen fachlichen und technischen Artefakten unterschieden. Ansonsten ist in der Regel von den fachlichen Objekten die Rede. Eine detaillierte Auflistung aller Domänenbegriffe findet sich im Kapitel Glossar.

3. Vorgehensweise und Methodik

Ausgehend von der beschriebenen Ausgangslage wurde ein passendes Vorgehensmodell für die vorliegende Arbeit festgelegt. Mehrere Faktoren prägten die Wahl: Die anfänglich unklaren Anforderungen, die zeitliche sowie räumliche Verteilung des Teams und die ausschliesslich virtuelle Zusammenarbeit bei gleichzeitigem Vollzeitpensum aller Teammitglieder.

3.1. Wahl des Vorgehensmodells

Zu Projektbeginn lagen zentrale Anforderungen und Domänenkenntnisse nur ansatzweise vor. Ein agiles Vorgehen bot sich daher an, um in kurzen Iterationen Feedback vom betreuenden Referenten und vom Themengeber einzuholen und Erkenntnisse unmittelbar in Analyse und Design einfliessen zu lassen. Klassische Frameworks wie *Scrum* erwiesen sich für die gegebene Konstellation als zu schwergewichtig: Regelmässige *Daily-Meetings* waren weder zeitlich noch organisatorisch sinnvoll durchführbar, und der Overhead für *Planning*, *Reviews* und *Retrospektiven* hätte einen unverhältnismässig grossen Teil des Zeitbudgets gebunden. Es erschien deshalb nicht zielführend, Scrum oder andere etablierte Frameworks derart hinzubiegen, dass es „irgendwie passt“, bloss um sie referenzieren zu können. Stattdessen wurde der agile Grundgedanke bewusst hochgehalten, ohne ein konkretes Framework strikt zu implementieren.

> „Agility is principally about mindset, not practices.“
>
> *Jim Highsmith, Agile Project Management*

3.2. Initiale Analyse- und Architekturphase

Die Arbeit startete mit einer Phase des Verstehens: Anforderungsanalyse, Domänenanalyse und ein erster Architekturentwurf. Ohne tragfähiges Fundament kann auch in agilen Modellen keine gute Software entstehen. In dieser Phase fanden enge Abstimmungen mit der themengebenden Feuerwehr statt. Zusätzlich wurde *apprenticing* betrieben, indem ein Teammitglied an einer Feuerwehrübung teilnahm. Dadurch konnten wichtige Erkenntnisse gewonnen werden. Einzelne Ideen erwiesen sich als nicht praxistauglich und wurden verworfen, andere Bedürfnisse wurden bestätigt.
Für die Domänenmodellierung wurden Ansätze des *Domain-Driven Design (DDD)* verwendet, insbesondere die Modellierung von Haupt- und Subdomänen sowie *Bounded Contexts* (Evans, 2003). Auf konsequente taktische *DDD*-Patterns in Design und Implementierung wurde später bewusst verzichtet. Dennoch waren die gewonnenen Erkenntnisse und Konzepte weiterhin prägend für das Systemdesign.

3.3. Iterative Phase

Im Anschluss erfolgte der Übergang zu einem iterativen Vorgehen. Erkenntnisse aus der Implementierung flossen fortlaufend in Anforderungen und Domänenmodell zurück. Die Arbeitsorganisation orientierte sich lose an *Lean-Agile* Prinzipien: Ein Kanban-Board für das Task-Management, regelmässiger, aber nicht starr taktierter Austausch innerhalb des Teams sowie wiederkehrende Feedbackmeetings mit dem betreuenden Referenten.

Aus den initial erarbeiteten Anforderungen und User Stories wurde eine Grobplanung mit Meilensteinen erstellt. Das Kanban-Board diente jedoch als zentrales Planungsinstrument, mit den User Stories als Grundlage für die einzelnen Arbeitspakete. Bereits früh wurde ein *Walking Skeleton* realisiert, um Projektrisiken zu reduzieren und die Tauglichkeit der Architektur zu validieren (siehe Abschnitt 8.1 und Anhang E).

3.4. Priorisierung und MVP

Da das Zeitbudget der Masterarbeit absehbar nicht für die vollständige Realisierung aller Funktionen ausreichte, wurde der Projektumfang gezielt begrenzt. Ziel war es, eine Architektur zu entwerfen, die sämtliche bekannten Anforderungen grundsätzlich trägt. In der Umsetzung sollte aber zunächst nur ein *Minimum Viable Product (MVP)* realisiert werden, welches der themengebenden Feuerwehr bereits konkreten Nutzen bringen kann. Die Priorisierung war von der Frage getrieben, welche Funktionen für einen ersten produktiven Einsatz unverzichtbar sind und welche ohne Verlust der Zielerreichung zeitlich nachgelagert werden können. Nicht-kritische Funktionen wurden zurückgestellt, sofern ihre spätere Integration durch die gewählte Architektur abgesichert war. Die daraus resultierende Systemabgrenzung wird in Abschnitt 6.7 erläutert.

Zusammengefasst lässt sich das gewählte Vorgehen wie folgt beschreiben:

- **Vorgehensmodell:** Kombination aus initialer Analyse-/Architekturphase und iterativer Umsetzung mit kurzen Feedbackzyklen sowie frühe Validierung durch ein Walking Skeleton.
- **Projektplanung und -abwicklung:** Grobe Meilensteinplanung und operative Steuerung über ein Kanban-Board mit klar geschnittenen Tasks pro User Story.
- **Priorisierung:** Systematische Fokussierung auf MVP-relevante Funktionen und explizite Scope-Abgrenzung (siehe Abschnitt 6.7).
- **Risikomanagement:** Frühe Architekturerprobung und laufende Neubewertung im Team, dokumentiert in einer Risikomatrix (Abbildung E.1). Kontinuierliche Anpassung des Backlogs bei neuen Erkenntnissen.

Das gewählte Vorgehen verbindet eine tragfähige initiale Analyse- und Architekturarbeit mit einer leichtgewichtigen, iterativen Umsetzung. Die bewusste Fokussierung auf ein MVP, das frühe Walking Skeleton sowie der kontinuierliche Austausch mit Stakeholdern reduzierten Risiken und erhöhten die Zielklarheit. DDD diente als Methodengerüst in der frühen Phase, prägte Begriffe und Abgrenzungen und beeinflusste so das spätere Design, ohne als strenges Vorgehensframework weitergeführt zu werden.

4. Anforderungen

Das Ziel der Anforderungsanalyse war es, die Anforderungen an ein System zu erheben, das freiwillige Feuerwehren bei der effizienten Verwaltung und Koordination von Mehrfachereignissen unterstützt. Dabei soll *Hydra* als flexibles Werkzeug für Feuerwehren in der Schweiz einsetzbar sein, unabhängig von deren Grösse oder Struktur.

Zur Erhebung der Anforderungen wurden Interviews und Workshops mit der Feuerwehr [redacted] durchgeführt und auch an Übungen teilgenommen. Diese enge Zusammenarbeit ermöglichte eine detaillierte Analyse realer Abläufe und Herausforderungen und bildete die Grundlage für die im Folgenden dokumentierten Rollen, User Stories und Qualitätsanforderungen.

4.1. Rollenübersicht

An der Bewältigung eines Einsatzes sind verschiedene Rollen mit unterschiedlichen Aufgaben und Verantwortlichkeiten beteiligt. Jede dieser Rollen stellt spezifische Anforderungen an das System *Hydra*. Nachfolgend sind die identifizierten Rollen aufgeführt. Sie definieren die jeweiligen Perspektiven und Zuständigkeiten während eines Einsatzes. Um sich als Entwickler besser mit den vorhandenen Rollen identifizieren zu können, wurde mit Personas gearbeitet. Eine detaillierte Auflistung dieser Personas befindet sich im Anhang A.

- **Einsatzleiter/in (Hauptquartier):** Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
- **Operator/in (Hauptquartier):** Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
- **Rekognoszierende/r (Feldeinheit):** Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation.
- **Truppchef/in (Feldeinheit):** Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Duis aute irure dolor in reprehenderit in voluptate.
- **Einsatzkraft (Feldeinheit):** Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

- **Administrator/in:** Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

4.2. Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben, welche fachlichen Leistungen das System bereitstellen muss, um die Einsatzleitung und die Einsatzkräfte im Feld wirksam zu unterstützen. Im Vordergrund stehen dabei die strukturierte Erfassung und Bearbeitung von Meldungen und Ereignissen, die Disposition von Ressourcen sowie eine übersichtliche Darstellung der Lage.

Die wichtigsten funktionalen Anforderungen lassen sich in folgenden Kernthemen zusammenfassen:

- **Meldungen erfassen und verarbeiten:** Das System soll neue Meldungen aufnehmen, kategorisieren und bei Bedarf anpassen oder löschen können.
- **Ereignisse ableiten und verwalten:** Aus Meldungen können Ereignisse entstehen, deren aktueller Status im System verfolgt werden kann.
- **Ressourcen disponieren:** Das System soll Einsatzkräfte und Material sowie deren Verfügbarkeit verwalten.
- **Lage visualisieren:** Meldungen, Ereignisse und Ressourcen sollen auf einer Karte und auf einem Kanban-Board, dem sogenannten „Journal“, sichtbar sein.
- **Nachvollziehbarkeit:** Die Ereignisse und Benutzeraktionen sollen geloggt werden, so dass der Einsatzverlauf dokumentiert ist.

Die detaillierten funktionalen Anforderungen wurden in Form von *User Stories* (Beck, 2000) erarbeitet und im klassischen Template-Format nach Cohn (2004) formuliert. In dieser frühen Projektphase wurde bewusst auf die Ausarbeitung formeller Use Cases verzichtet, da sich User Stories besser für die gewählte Agile vorgehensweise eignen. Die Bearbeitung einer User Story im agilen Modell beinhaltet nicht nur die Implementierung, sondern durchläuft auch alle anderen Phasen (Architektur, Design etc.) des *SDLC (Software Development Lifecycle)*. Da zum Bearbeitungszeitpunkt einer User Story mehr über das System und die Anforderungen bekannt sind, lassen sich die Details einer User Story zu diesem Zeitpunkt genauer spezifizieren, falls nötig auch bis zum Detaillierungsgrad eines *Use Cases*.

Ungeachtet dessen werden die User Stories in diesem Dokument mithilfe von *Use-Case-Diagrammen* visualisiert, da sich diese Form der Dokumentation hervorragend eignet, um die Abhängigkeiten zwischen den Rollen und den verschiedenen User Stories zu veranschaulichen. Die folgende Auflistung gibt einen Überblick über die User Stories.

Allgemein (A)

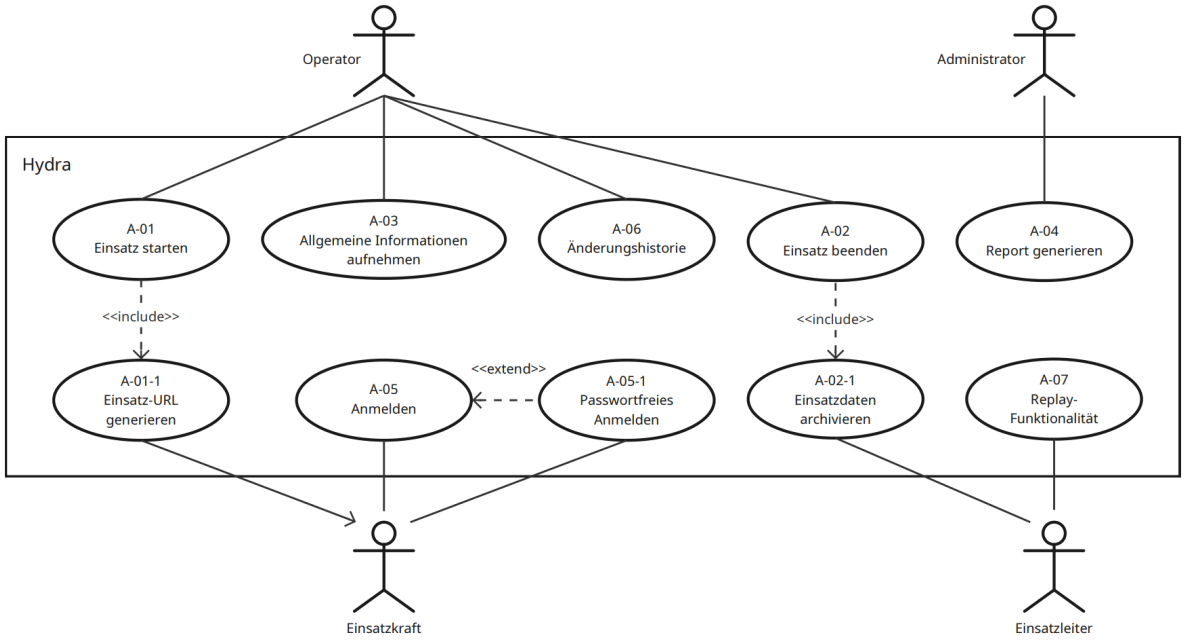


Abbildung 4.1.: User Stories Allgemein

ID	User Story
A-01	**Einsatz starten:** Als Operator möchte ich einen Einsatz starten, damit er bearbeitet und koordiniert werden kann.
A-01-1	**Einsatz-URL generieren:** Als Einsatzkraft möchte ich mich mittels Einsatz-URL direkt am aktuellen Einsatz anmelden können. Die URL wird automatisch beim Starten eines Einsatzes generiert.
A-02	**Einsatz beenden:** Als Operator möchte ich einen Einsatz beenden können, damit er als abgeschlossen markiert wird und nicht mehr bearbeitet werden kann.
A-02-1	**Einsatzdaten archivieren:** Als Einsatzleiter möchte ich, dass ein Einsatz beim Abschliessen archiviert wird, damit der Einsatzablauf auch zu einem späteren Zeitpunkt nachvollziehbar bleibt.
A-03	**Allgemeine Informationen aufnehmen:** Als Operator möchte ich allgemeine Informationen zu einem Einsatz erfassen, damit diese im Journal sichtbar sind.
A-04	**Report generieren:** Als Administrator oder Operator möchte ich einen Einsatzbericht generieren können, damit der Einsatzablauf sichtbar wird.
A-05	**Anmelden:** Als Einsatzkraft möchte ich mich im System anmelden können, damit ich alle relevanten Informationen für meinen Trupp erhalte.

A-05-1	**Passwortfreies Anmelden:** Als Einsatzkraft möchte ich mich einfach (z. B. ohne Passwort, via QR-Code oder Link) anmelden können, damit das System im Ernstfall schnell einsatzbereit ist.
A-06	**Änderungshistorie:** Als Operator möchte ich die Änderungshistorie der verschiedenen Objekte anschauen können, damit ich bei Unklarheiten die richtige Person fragen kann.
A-07	**Replay-Funktionalität:** Als Einsatzleiter möchte ich die Gesamtsituation auf der Lagekarte zurückspulen können, damit ich den bisherigen Einsatzverlauf visuell dargestellt bekomme.

Tabelle 4.1.: User Stories Allgemein

Meldungsverwaltung (MV)

Abbildung 4.2.: User Stories Meldungsverwaltung

ID	User Story
MV-01	[redacted]
MV-01-1	[redacted]
MV-02	[redacted]

Dieser Inhalt wurde aus Vertraulichkeitsgründen aus der publizierten Version entfernt.

4.3. Qualitätsanforderungen

Die *Qualitätsanforderungen* (auch *nichtfunktionale Anforderungen*) legen fest, welche Eigenschaften *Hydra* neben seinen funktionalen Leistungen erfüllen muss. Sie beschreiben zentrale Qualitätskriterien wie Zuverlässigkeit, Benutzerfreundlichkeit, Leistungsfähigkeit und Flexibilität und stellen sicher, dass das System den besonderen Anforderungen im Einsatzgeschehen gerecht wird.

Die folgende Auflistung dient als Grundlage für Architektur- und Designentscheidungen und stellt sicher, dass *Hydra* unter Einsatzbedingungen zuverlässig und robust einsetzbar ist.

ID	Beschreibung
QA-01	Das System soll auch für ungeübte Anwender eine einfache Bedienung ermöglichen, damit die Nutzung ohne lange Einarbeitung möglich ist. Operatoren sollen das System nach 5 Minuten Einführung bedienen können, für Feldeinheiten soll eine Minute ausreichen.
QA-02	Das System soll die Erfassung von Meldungen und Ereignissen mit möglichst geringem Aufwand ermöglichen, um Eingaben effizient und fehlerarm durchführen zu können. Mehrfache Erfassungen derselben Daten sollen nicht notwendig sein.
QA-03	Das System muss Änderungen innerhalb weniger Sekunden für alle relevanten Nutzer sichtbar machen, um eine Zusammenarbeit in nahezu Echtzeit zu gewährleisten.
QA-04	Das System muss bei Mehrfachereignissen mindestens 100 Meldungen innerhalb von zwei Stunden erfassen können und die Zustandsänderungen dieser Objekte über mehrere Stunden verwalten können.
QA-05	Das System muss personenbezogene Daten im Einklang mit der DSGVO verarbeiten, insbesondere Daten zu Geschädigten.
QA-06	Das System muss alle Änderungen protokollieren, um Debriefings und nachträgliche Analysen durch Reports zu ermöglichen.
QA-07	Das System soll rollenbasierte Sichten bereitstellen, sodass Anwender nur die für sie relevanten Informationen und Aktionen angezeigt bekommen.
QA-08	Das System muss für Feuerwehren in der ganzen Schweiz einsetzbar sein und muss mit schweizerischen Gegebenheiten (Adressformate, Kartendaten, Sprachen etc.) zurechtkommen.
QA-09	Das System soll eine Sprachumschaltung bieten.
QA-10	Das System soll später auf weitere Regionen erweitert werden können.

QA-11	Das System soll bestehende Alarmierungs- und Fremdsysteme nicht ersetzen, sondern nur optional und klar definiert (z. B. per E-Mail) integrieren.
QA-12	Das System soll sowohl in der Cloud als auch bei Bedarf lokal in der Einsatzzentrale gehostet werden können.
QA-13	Das System muss nicht redundant oder hochverfügbar ausgeführt werden. Die Feuerwehr ist den Umgang mit technischen Störungen und unerwarteten Ereignissen im Einsatz gewohnt und kann stets auf einfache Fallback-Systeme wie Whiteboards zurückgreifen.
QA-14	Das System muss unvermeidbaren Störfaktoren im Feld wie schlechter Netzabdeckung oder leeren Akkus nicht Rechnung tragen. Es soll nicht andere, hochverfügbare Kommunikationskanäle ersetzen. Der Operator soll jedoch die Möglichkeit haben, Eingaben für Feldeinheiten vorzunehmen, falls diese das System nicht selbst bedienen können.

Tabelle 4.6.: Qualitätsanforderungen

Damit sind die zentralen Anforderungen an das System *Hydra* festgelegt. Sie bilden die Grundlage für Architekturentscheidungen, die Priorisierung im Backlog und ein gemeinsames Verständnis der Systemanforderungen. Offene Punkte werden im Projektverlauf schrittweise konkretisiert. Im folgenden Kapitel werden die fachlichen Grundlagen vertieft, auf denen die Architektur und das Lösungsdesign aufbauen.

5. Domänenanalyse

Die Domänenanalyse bildet den Ausgangspunkt für die Entwicklung einer fachlich fundierten Softwarearchitektur. Ziel ist es, die Problemstellung in klar abgegrenzte Domänen zu zerlegen, Verantwortlichkeiten zu definieren und eine gemeinsame Sprache *(Ubiquitous Language)* zu schaffen. Auf diese Weise wird sichergestellt, dass spätere Architekturentscheidungen nicht nur technischen, sondern vor allem fachlichen Kriterien folgen.

Angelehnt an die Methodik des *Domain-Driven Design (DDD)* (Evans, 2003), wird die Architektur des vorliegenden Systems massgeblich durch eine tiefgründige Analyse der Fachdomäne getrieben. Grundlage hierfür sind Gespräche mit der Feuerwehr [redacted] sowie Schulungsunterlagen, Prozessbeschreibungen und die daraus abgeleiteten User Stories (siehe Abschnitt 4.2). Aus diesen Quellen wurde ein Domänenmodell entwickelt, das die wesentlichen fachlichen Aufgabenbereiche und deren Zusammenhänge abbildet.

5.1. Haupt- und Subdomänen

Nach Vaughn Vernon ist die Modellierung von Domänen und Subdomänen dem strategischen DDD zuzuordnen. Sie dient dazu, die grossen fachlichen Zusammenhänge sichtbar zu machen, Systemgrenzen zu erkennen und klare Schwerpunkte für die Architektur zu setzen (Vernon, 2013).

Abbildung 5.1 zeigt die Aufteilung der Problemdomäne in Hauptdomänen und deren jeweilige Subdomänen. Jede Domäne beschreibt einen zentralen Verantwortungsbereich der Feuerwehr und macht deutlich, welche fachlichen Aufgaben durch das System unterstützt werden sollen.

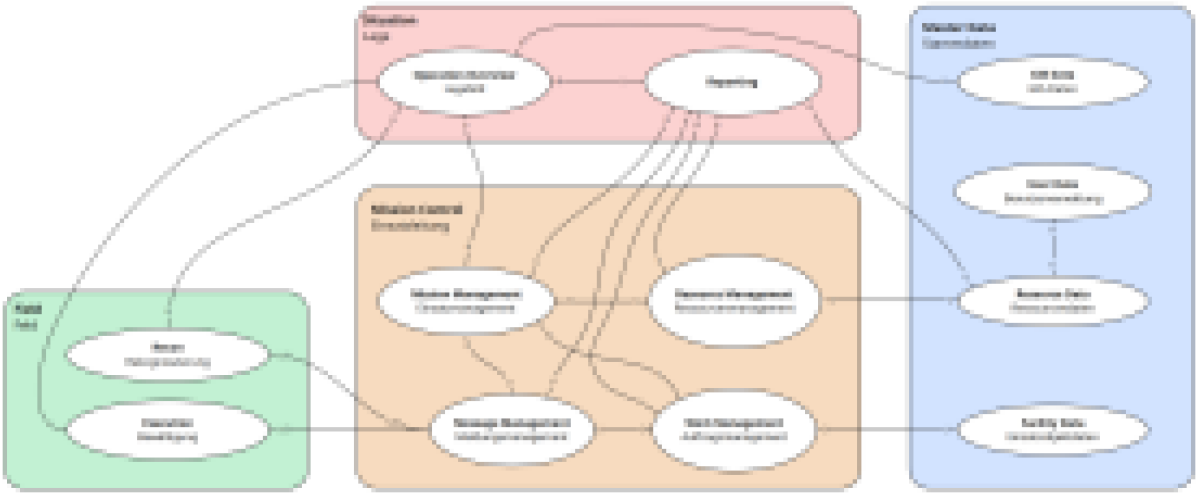


Abbildung 5.1.: Haupt- und Subdomänen

Dieser Inhalt wurde aus Vertraulichkeitsgründen aus der publizierten Version entfernt.

ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident.

Lorem Ipsum (Geschwärzt): Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.

- Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
- Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua (z. B. tempor incididunt).
- Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua, ut enim ad minim veniam. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
- Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Durch die Abbildung der Haupt- und Subdomänen werden fachliche Zusammenhänge und Abhängigkeiten sichtbar, die in der Architekturphase gezielt berücksichtigt werden können. Sie bildet eine wichtige Grundlage für die Definition von Systemgrenzen und Schnittstellen.

5.2. Bounded Contexts

Aufbauend auf den zuvor beschriebenen Haupt- und Subdomänen zeigt Abbildung 5.2 die Einteilung in Bounded Contexts. Ziel dieser Abstraktion ist es, jene Bereiche voneinander abzugrenzen, die zwar fachlich eng zusammenhängen, jedoch unterschiedliche Bedeutungen und Modelle verwenden. Jeder Context bildet ein eigenes konsistentes Modell mit klar definierter Ubiquitous Language und sorgt damit für eindeutige Begriffe innerhalb seiner Grenzen.

Im *Headquarters Context* befinden sich die Domänen *Situation* und *Mission Control*. Beide arbeiten eng zusammen und teilen ein gemeinsames Vokabular. Begriffe wie „Einsatz“, „Ereignis“ oder „Meldung“ haben hier dieselbe Bedeutung und werden durch identische Entitäten repräsentiert.

Der *Field Context* bildet hingegen seinen eigenen abgegrenzten Bereich. Zwar überschneidet sich das verwendete Fachvokabular teilweise mit dem des *Headquarters Context*, doch unterscheiden sich die Bedeutungen im Detail. Während im Hauptquartier komplexe Modelle mit zahlreichen Attributen und Beziehungen erforderlich sind, genügen im Feld

oft vereinfachte Repräsentationen. Diese Unterschiede rechtfertigen eine saubere Trennung, um unnötige Kopplung und Komplexität zu vermeiden.

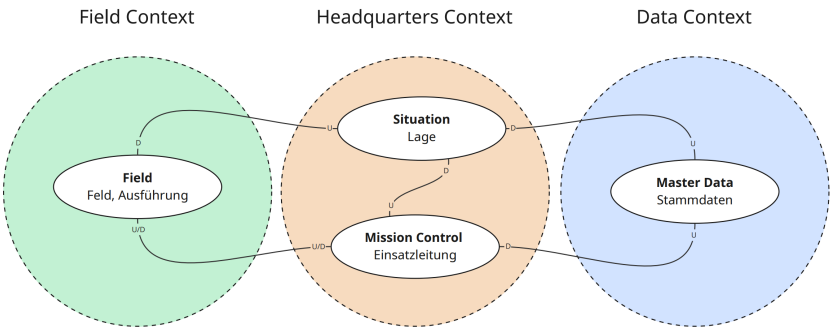


Abbildung 5.2.: Bounded Contexts um Hauptdomänen

Auch die Domäne *Master Data* bildet einen eigenständigen *Data Context*. Sie stellt zwar Informationen wie Ressourcen-, Benutzer- oder Objektdaten bereit, jedoch müssen diese nicht deckungsgleich mit den Entitäten anderer Kontexte sein. So entspricht ein Benutzer in den Stammdaten nicht zwingend einer Einsatzkraft im *Headquarters Context*. Die Eigenständigkeit des *Data Contexts* erlaubt es, Stammdaten unabhängig zu pflegen und flexibel in andere Kontexte einzubinden.

Durch diese Aufteilung in drei Bounded Contexts wird den unterschiedlichen Bedeutungen und Detailgraden innerhalb der Domänen Rechnung getragen. Gleichzeitig entsteht eine klare Entkopplung, welche es ermöglicht, die Applikationsteile für verschiedene Domänen so unabhängig wie möglich zu entwickeln.

5.3. Context Map

Während die Bounded Contexts die fachliche Abgrenzung einzelner Modelle sicherstellen, entsteht in der Realität dennoch die Notwendigkeit, Informationen zwischen ihnen auszutauschen. Unterschiedliche Modelle müssen integriert, Daten synchronisiert und Zuständigkeiten geklärt werden. Um diese Beziehungen klar und explizit zu beschreiben, wird im Domain-Driven Design das Konzept der *Context Map* eingeführt (Evans, 2003). Sie macht sichtbar, welche Kontexte in welcher Form miteinander verbunden sind.

Die erarbeitete Context Map in Abbildung 5.3 verdeutlicht die Beziehungen zwischen den drei zentralen Kontexten *Headquarters*, *Field* und *Data* mittels Schnittstellenmuster, wie sie von Evans (2003) vorgeschlagen und von Vernon (2013) weiter detailliert wurden.

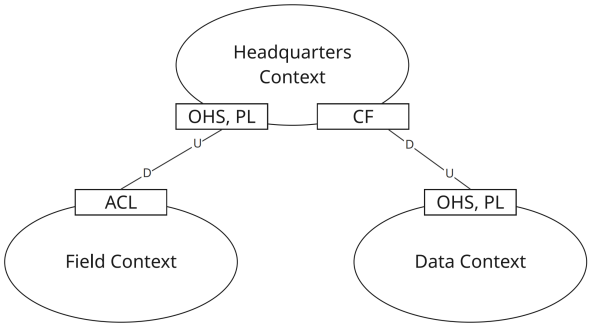


Abbildung 5.3.: Context Map

Headquarters ↔ Field Context

Zwischen *Headquarters* und *Field* besteht eine bidirektionale Beziehung.

Der *Headquarters Context* ist hier als *Upstream* dargestellt, da er Aufträge und Befehle an den *Field Context* übermittelt.

Der *Field Context* ist *Downstream*, liefert aber Rückmeldungen zurück an den *Headquarters Context*.

Um semantische Unterschiede zu überbrücken, etwa dass Entitäten im Feld eine andere Detailtiefe haben als im Hauptquartier, wird ein `Anti-Corruption Layer (ACL)` eingesetzt. Dieses Muster schützt das Domänenmodell des Hauptquartiers vor einer „Kontamination“ durch vereinfachte oder abweichende Strukturen des Feldes. Entwickler im *Field Context* können ihr Modell damit pragmatisch einfach und feldnah halten, während Entwickler im *Headquarters Context* mit einem detaillierteren Modell arbeiten. Die Verantwortung für die Übersetzung liegt beim ACL, was die Kopplung reduziert, aber auch zusätzlichen Implementierungsaufwand mit sich bringt.

Gleichzeitig wird der Zugriff über `Open Host Service (OHS)` und `Published Language (PL)` standardisiert. Der *Headquarters Context* stellt damit klar definierte APIs und ein gemeinsames Sprachmodell bereit. Entwickler im *Headquarters Context* müssen diese Schnittstellen konsistent halten, während Entwickler im *Field Context* auf die Stabilität und Lesbarkeit der Published Language vertrauen können.

Headquarters ↔ Data Context

Zwischen *Headquarters* und *Data Context* wird das Muster `Conformist (CF)` angewandt. Für gemeinsame Modelle übernimmt der *Headquarters Context* die Modellierung des *Data Contexts* weitgehend so, wie sie vorliegt. Dies reduziert den Integrationsaufwand, macht aber gleichzeitig das Headquarters-Modell in diesem Bereich abhängig von Entscheidungen im *Data Context*. Dieser Tradeoff wurde bei der Modellierung in Kauf genommen, da Modelle für Stammdaten in der Regel eher stabil sind und fast immer auch Anpassungen an der Business-Logik zur Folge haben. Die eher selten zu erwartenden Anpassungen im *Headquarters Context* müssen also mit grosser Wahrscheinlichkeit sowieso gemacht werden.

Darüber hinaus stellt der *Data Context* seine Funktionalitäten über ein `Open Host Service (OHS)` und eine `Published Language (PL)` bereit. Dadurch entstehen klare,

dokumentierte Integrationspunkte, die es dem *Headquarters Context* erleichtern, mit dem *Data Context* zu interagieren.

5.4. Domänenmodell Headquarters-Kontext

Nach der Analyse der Haupt- und Subdomänen sowie der Einteilung in Bounded Contexts hat sich gezeigt, dass der *Headquarters Context* die zentrale Rolle im System einnimmt. Der *Data Context* ist für ein Minimum Viable Product (MVP) nicht zwingend nötig, und der *Field Context* weist im Vergleich zum *Headquarters Context* eine wesentlich geringere Komplexität auf. Um den zentralen *Headquarters Context* besser zu verstehen, wurde für diesen ein *Domänenmodell (Domain Model)* erstellt.

Abbildung 5.4 zeigt das Domänenmodell des *Headquarters Contexts* als Entity-Relation-Model in Crow's-Foot-Notation (Everest, 1976). Diese etwas unübliche Form für ein Domänenmodell wurde bewusst gewählt, da sie die Beziehungen zwischen den Entitäten der Domäne klar sichtbar macht und sich auf das Wesentliche fokussiert. Sie eignet sich daher besonders gut, um die Abhängigkeiten innerhalb der Domäne bereits in einer frühen Phase transparent zu machen.

Abbildung 5.4.: Domain Model des Headquarters-Kontexts

Zu Beginn der Domänenmodellierung wurde bewusst auf eine zu starke Detaillierung verzichtet. Viele der im Domain-Driven Design bereitgestellten Konzepte wie Entities, Aggregates oder Repositories dienen vor allem der späteren Verfeinerung des Modells bis hin zu einem Detaillierungsgrad, der direkt in eine Implementierung überführt werden kann. Das ist jedoch nicht das Ziel des hier dargestellten Modells. Es soll lediglich zu einem besseren Verständnis der Domäne beitragen. Zu diesem Zweck wurden die zentralen Entitäten des *Headquarters Contexts* sowie einige naheliegende, für die Domäne wesentlichen

Value Objects und deren Beziehungen modelliert.

Eine Entität *(Entity)* stellt ein Domänen-Objekt oder ein „Ding“ dar, welches in der fachlichen Domäne der Feuerwehr verwendet wird. Jedes dieser Domain-Entites bildet dabei ein zentrales fachliches Objekt ab, die im System gespeichert, verarbeitet und über APIs zugänglich gemacht wird.

Aus den in Abschnitt 4.2 beschriebenen funktionalen Anforderungen und dem Domänenmodell aus Abbildung 5.4 lassen sich bereits Entitäten erkennen, auf welche für ein MVP nicht verzichtet werden kann.

In Tabelle 5.1 sind die für die Applikation zentralen Entities aufgelistet und beschrieben.

Entity	**Beschreibung**
Mission	Repräsentiert einen Einsatz (Mehrfachereignis) der Feuerwehr. Eine Mission bildet den Rahmen für alle weiteren Vorgänge, wie etwa die Erfassung von Meldungen (Message), die Bearbeitung von Ereignissen (Incident) sowie den Einsatz von Trupps (Squad). Eine Mission enthält ausserdem Metadaten wie Bezeichnung, Start- und Endzeitpunkt, Status und allgemeine Informationen.
Message	Stellt eine Meldung dar, die während eines Einsatzes erfasst wird. Sie enthält eine Beschreibung des gemeldeten Sachverhalts (z. B. „Baum auf Strasse“), den Namen oder die Kennung des Meldenden sowie allfällige Positionsangaben oder Anhänge. Eine Message kann optional zur weiteren Bearbeitung in einen Incident überführt werden.
Incident	Ein konkret zu bearbeitendes Ereignis, das aus einer Meldung entstanden ist oder manuell erstellt wurde. Ein Incident hat eine Priorität, einen Bearbeitungsstatus (z. B. Offen, Rekognoszierung, Bereit, in Bearbeitung, Warten, Erledigt), eine Beschreibung sowie einen zugewiesenen Trupp (Squad).
Squad	Ein Einsatztrupp der Feuerwehr, bestehend aus mehreren Personen. Trupps werden von einem Truppchef geführt und können Incidents zugewiesen bekommen. Ein Squad enthält den Namen des Truppchefs, optional einen Truppnamen und zugewiesenes Material. Ein Squad ist entweder für die Rekognoszierung oder die Bearbeitung eines Ereignisses zuständig.

Tabelle 5.1.: Übersicht über zentrale Entities im System

Der Einsatz *(Mission)* steht im Zentrum der identifizierten Entitäten. Alle übrigen Entities sind immer mit einem Einsatz verbunden. So wird z. B. eine Meldung oder ein Incident stets im Kontext einer Mission erfasst und bearbeitet. Dies ist eine wichtige Tatsache, welche vor allem für das Design in Kapitel 7 von zentraler Bedeutung sein wird.

Mit einer gemeinsamen Fachsprache und klaren Kontextgrenzen schafft die Domänenanalyse die Grundlage für eine an der Fachdomäne ausgerichteten Softwarelösung. Anhand dieser Basis sowie der ermittelten Anforderungen wird im nächsten Kapitel der Entwurf der Architektur hergeleitet.

6. Architektur

Die Domänenanalyse hat die fachlichen Grenzen und Verantwortlichkeiten in Form von Hauptdomänen, Subdomänen und Bounded Contexts sichtbar gemacht. Aus der Einteilung der Hauptdomänen in Subdomänen lassen sich bereits erste potenzielle Kandidaten für Systemkomponenten erahnen. Für eine sinnvolle Aufteilung des Gesamtsystems in einzelne Komponenten ist es jedoch ebenso notwendig, die wesentlichen Datenflüsse im System zu verstehen.

6.1. Flow Analysis

Um eine grobe Übersicht über die Datenflüsse durch das System zu gewinnen, wurde die Methodik der Flow Analysis (Qvortrup, n. d.) angewandt. Ziel war es, die dominanten Datenflüsse auf einer hohen Abstraktionsebene zu identifizieren und visuell darzustellen, bewusst ohne Anspruch auf Vollständigkeit oder Detailtiefe. Der Fokus lag dabei auf den wichtigsten Datenflüssen des Systems, da diese massgeblich die Architektur beeinflussen.

Wie bereits in der Domänenanalyse in Abschnitt 5.2 deutlich wurde, stellen Feld und Hauptquartier jeweils eigene Kontexte dar. Deshalb wurde beim Identifizieren der dominanten Datenflüsse davon ausgegangen, dass es im Feld und im Hauptquartier verschiedene User Interfaces geben wird. Die in Abbildung 6.1 dargestellten Komponenten sind rein konzeptionell und keinesfalls als finale Systemkomponenten zu verstehen.

Abbildung 6.1.: Diagramm Flow Analysis

Dieser Inhalt wurde aus Vertraulichkeitsgründen aus der publizierten Version entfernt.

6.2. Systemkomponenten

Die User Stories, die Qualitätsanforderungen sowie das vertiefte Verständnis der Domäne und der dominanten Datenflüsse ergeben ein konsistentes Bild des Gesamtsystems. Auf dieser Basis lassen sich die benötigten Systemkomponenten zielgerichtet identifizieren. Die für *Hydra* definierten Komponenten orientieren sich eng am Domänenmodell, wobei User-Interface-Komponenten bewusst klar von fachlichen Kernkomponenten getrennt sind. Für die Domänen *Mission* und *Field* wurden jeweils eigene UI-Komponenten vorgesehen, da sich Aufgaben, Informationsdichte und Bedienanforderungen deutlich unterscheiden. Die fachlichen Kernkomponenten befinden sich alle in der Domäne *Mission*. Unterstützende Komponenten für Lagebild, Archivierung und Dokumentation liegen in der Domäne *Situation*. Die Verwaltung der Stammdaten erfolgt in der Domäne *Master Data*. Abbildung 6.2 gibt einen Überblick über die identifizierten Systemkomponenten.

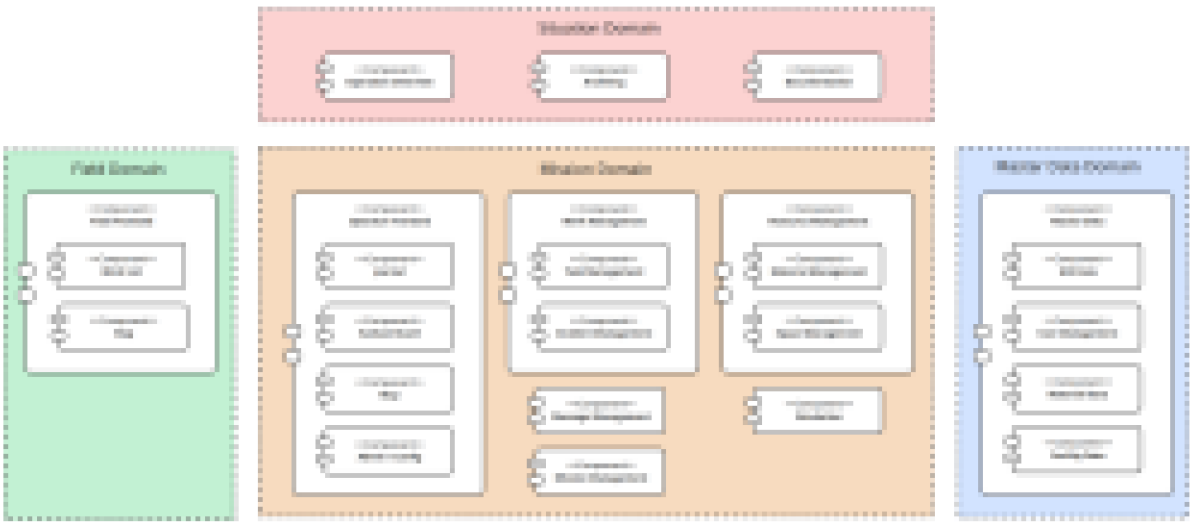


Abbildung 6.2.: Systemkomponenten

Lorem Ipsum: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat, sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Dolor Sit Amet: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Consectetur Adipiscing: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Dieser Inhalt wurde aus Vertraulichkeitsgründen aus der publizierten Version entfernt.

Umsetzung.

Für das Deployment des Systems wird konsequent auf Container gesetzt. Alle implementierten *Hydra*-Services werden als Docker-Images gebaut und als Docker-Container betrieben. Ebenso werden alle Drittanbieter-Services wie MongoDB und Kafka in Containern betrieben. Die Konfiguration erfolgt deklarativ über *Docker Compose* (Services, Umgebungsvariablen, Netzwerke, Volumes), wodurch reproduzierbare Deployments in Entwicklungs- und Demo-Umgebungen gewährleistet werden.

Das *Deployment Model* der zentralen *Hydra*-Services wird in Abbildung 6.9 gezeigt. Unterstützende Infrastruktur-Services wie z. B. Reverse Proxy oder Schema Registry sind der Einfachheit halber nicht dargestellt.

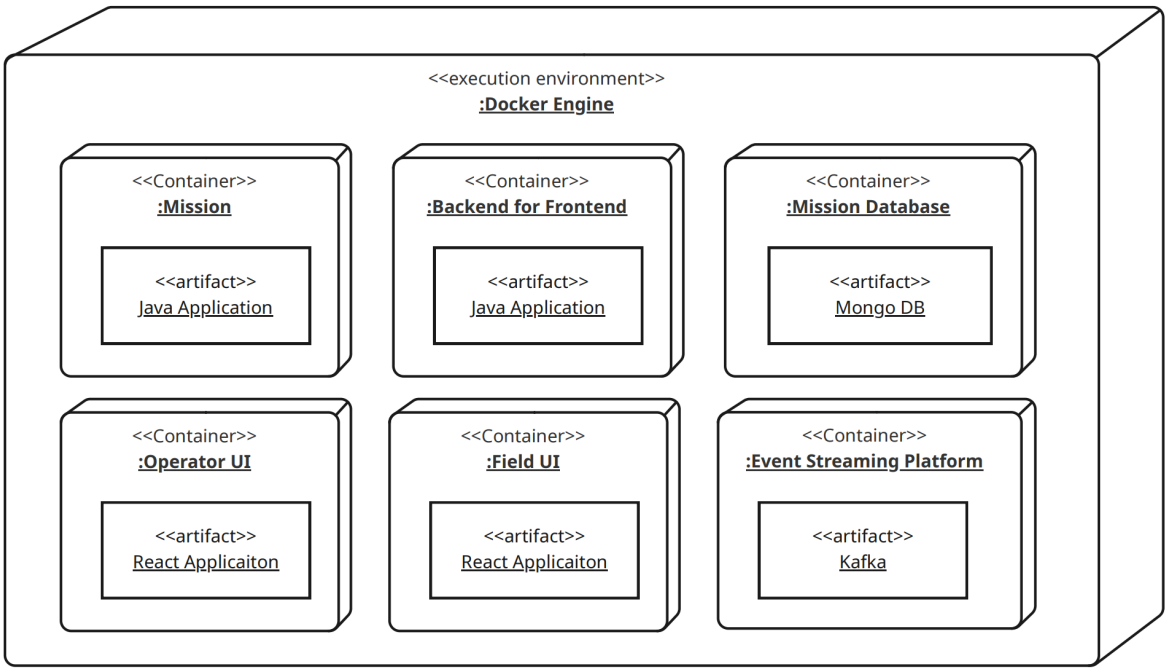


Abbildung 6.9.: UML-Verteilungsdiagramm (Deployment Diagram)

Die dargestellte Architektur orientiert sich konsequent am Domänenmodell und hält die Software damit eng an der Fachdomäne. Das skizzierte Deploymentmodell, die Technologiewahl und die High-Level-Spezifikation der Schnittstellen bilden die Grundlage für ein konsistentes Detaildesign und die anschliessende Implementation der Applikation. Das nächste Kapitel baut darauf auf und dokumentiert das konkrete Design. Einige der hier gezeigten Architekturentscheidungen sind in späteren Iterationen als Korrekturen eingeflossen, nachdem neue Erkenntnisse gewonnen wurden. Wie sich die Architektur im Verlauf der Arbeit entwickelt hat, kann dem der Arbeit beigelegten *Entscheidungslog.pdf* entnommen werden (separates Dokument im Abgabeverzeichis).

7. Design

Das Design der Backend-Services wurde zu Beginn im Rahmen des Walking Skeletons entworfen und im weiteren Verlauf des Projekts iterativ überarbeitet und erweitert. Mit jeder Iteration wurden neue User Stories integriert und bestehende Strukturen kontinuierlich verbessert. Im Mittelpunkt stand dabei, eine gute Testbarkeit und Wartbarkeit aufrechtzuerhalten sowie eine saubere Entkopplung der Businesslogik von den umgebenden Architekturschichten sicherzustellen. Dies wurde durch den Einsatz von Dependency Injection im Spring-Framework erheblich erleichtert.

Im Folgenden werden die wichtigsten Design-Entscheidungen erläutert und ihre Rolle in der Gesamtarchitektur aufgezeigt.

7.1. Layering und Package-Struktur

Hinsichtlich des Layerings orientiert sich das Design an den Prinzipien der Clean Architecture (Martin, 2017). In Abbildung 7.1 ist zu sehen, wie die Schichten aufgebaut sind. Im Mittelpunkt stehen dabei Entities (gelb), welche die fachlichen Konzepte der Domäne repräsentieren. Sie sind bewusst frei von Abhängigkeiten zu anderen Teilen der Applikation gehalten, um die Kernlogik möglichst stabil und unabhängig zu gestalten. Die Applikations-Logik (rot) verwendet diese Entities, implementiert die Use Cases und bildet zusammen mit den Entities die Geschäftslogik. Darauf aufbauend stellt die Infrastrukturschicht (grün) technische Funktionen wie Persistenz oder Kommunikation bereit. Die konkreten Technologien, wie beispielsweise die Datenbank (MongoDB), REST-Schnittstellen oder das Spring-Framework, befinden sich im äussersten Layer (blau). Ein wesentliches Prinzip dabei ist, dass jede Schicht ausschliesslich auf innere Schichten zugreifen darf. Diese Anordnung ermöglicht es, Technologien bei Bedarf auszutauschen, ohne dass die Business-Logik in den Kernschichten angepasst werden muss.

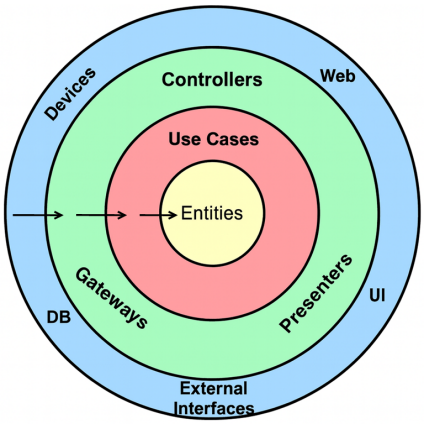


Abbildung 7.1.: Schalenmodell der Clean Architecture.
Quelle: Martin (2012)

Im Unterschied zur klassischen Clean Architecture wurde jedoch nicht ein Use-Case-zentrierter Ansatz verfolgt. Stattdessen lag der Schwerpunkt auf einer klaren Entkopplung der Schichten. Ziel war es, eine flexible und testbare Architektur zu schaffen, die sowohl Erweiterungen als auch spätere technologische Anpassungen unterstützt.

Diese Trennung spiegelt sich auch in der Package-Struktur der einzelnen Komponenten wider:

- `*.domain.model` enthält die Entities,
- `*.domain.application` die Applikations-Logik,
- `*.infrastructure` die Infrastrukturklassen und technischen Implementierungen.

Darüber hinaus existieren separate Packages für die Spring-Konfigurationsklassen (`*.config`) sowie für die generierten APIs (`*.api`), die der Infrastrukturschicht zugeordnet sind. Diese klare Trennung erleichtert die Orientierung im Code und führt zu einer technologieunabhängigen Business-Logik.

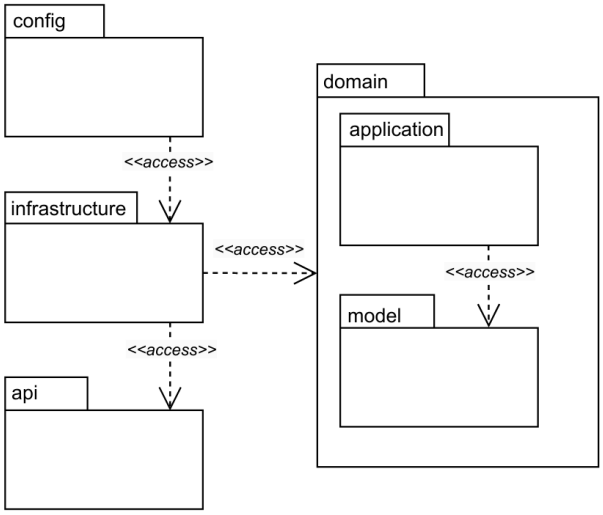


Abbildung 7.2.: Package-Struktur

7.2. Common Package

Im Kapitel Abschnitt 6.7 wurde aufgezeigt, welche Systemkomponenten im Rahmen der Masterarbeit implementiert wurden. Für jede der in Abbildung 6.8 dargestellten Komponente existiert ein eigener Package-Namespace, sodass eine saubere Entkopplung zwischen den Systemteilen gewährleistet ist. Jede dieser Komponenten bzw. Subkomponenten folgt für sich der in Abschnitt 7.1 beschriebenen Package-Struktur.

Trotz dieser klaren Trennung gibt es gemeinsame fachliche und technische Elemente, die in mehreren Komponenten verwendet werden. Dazu gehören *Value-Objects* wie etwa *Location*, *Address*, *IncidentState*, sowie *Category* und *Priority* für Meldungen und Ereignisse (siehe Abbildung 5.4). Auf technischer Ebene werden z. B. gemeinsame Interfaces und Basisklassen für Service- und Exception-Klassen, einheitliche Error-Codes oder Typen für Change-Events mehrfach verwendet.

Um diese *Shared Types* zentral zu bündeln, wurde die Bibliothekskomponente `ch.hydra.service.common` eingeführt. Sie kann von allen Systemkomponenten genutzt werden, besitzt selbst jedoch keine Abhängigkeiten zu anderen Teilen des Systems.

Die *Common*-Komponente beinhaltet einerseits Elemente, die ausschliesslich von den Backend-Services verwendet werden und daher direkt in Java implementiert sind. Andererseits enthält sie technologieübergreifende Typdefinitionen, die sowohl im Backend als auch im Frontend benötigt werden. Diese werden in einer technologieneutralen *OpenAPI*-Definitionsdatei gepflegt und mit Hilfe von entsprechenden Generatoren als Java- und TypeScript-Artefakte generiert. Mit der gewählten Monorepository-Projektstruktur für das Backend (siehe Abschnitt 8.2) können die generierten Java-Klassen direkt referenziert werden, eine Package-Registry ist nicht erforderlich. Die generierten TypeScript-Artefakte werden in einer Package-Registry in Github verwaltet.

Auf diese Weise verfügen sämtliche Backend-Services, sowie das Frontend über eine gemeinsame Basis. Änderungen an den gemeinsamen Typen müssen nur an einer zentralen Stelle vorgenommen werden, was den Wartungsaufwand reduziert und gleichzeitig Konsistenz im gesamten System sicherstellt. Damit leistet die *Common*-Bibliothekskomponente einen wichtigen Beitrag zur Standardisierung und langfristigen Wartbarkeit der gesamten Applikation.

7.3. Entities

Im Zentrum des in Abschnitt 7.1 beschriebenen Schalenmodells stehen die Entities. Sie bilden das Kernmodell der Domäne und stellen die fachlichen Konzepte dar, auf denen die Anwendungslogik aufbaut. Die Entities werden im Backend innerhalb des Mission-Service verwaltet und sind als Java-Klassen implementiert.

Obwohl Domain-Entities in der Clean Architecture grundsätzlich frei von technischen Abhängigkeiten modelliert werden sollten, enthalten die Entities im Projekt *Hydra* bewusst einige *Spring-Data-Annotationen* wie `@Document`, `@Id`, `@CreatedDate`, `@LastModifiedDate`. Dies ist ein pragmatischer Ansatz, der die Integration mit der Datenbank stark vereinfacht. Die Annotationen wirken hier lediglich als Metadaten und beeinflussen die fachliche Modellierung nicht. Somit sind die Entities weitestgehend technologieunabhängig und enthalten ausschliesslich fachliche Attribute.

In Abschnitt 5.4 wurden die zentralen *Domain Entities* bereits identifiziert und beschrieben. Bei der Domänenanalyse in Kapitel 5 wurde festgestellt, dass Entities wie *Message*, *Incident* und *Squad* immer im Zusammenhang mit einem Einsatz *(Mission)* stehen. Ein Einsatz bildet somit die Grundlage, auf welche sich die anderen Entities beziehen.

Message, Incident und Squad werden daher im Folgenden als *einsatzbezogen* oder *einsatzabhängig*, Mission als *nicht-einsatzbezogen* oder *einsatzunabhängig* bezeichnet.

Ob ein Entity einsatzbezogen ist oder nicht ist sehr essentiell, denn die CRUD-Operationen für einsatzbezogene Entities hängen immer vom übergeordneten Einsatz ab (siehe Abschnitt 7.5.1). Das bedeutet, dass Create-, Read-, Update- und Delete-Operationen für diese Entities nur im Kontext eines konkreten Einsatzes ausgeführt werden können. So muss zum Beispiel beim Erstellen eines neuen Incidents angegeben werden, zu welcher Mission er gehört. Diese Verbindung wird über die eindeutige ID des entsprechenden

Dieser Inhalt wurde aus Vertraulichkeitsgründen aus der publizierten Version entfernt.

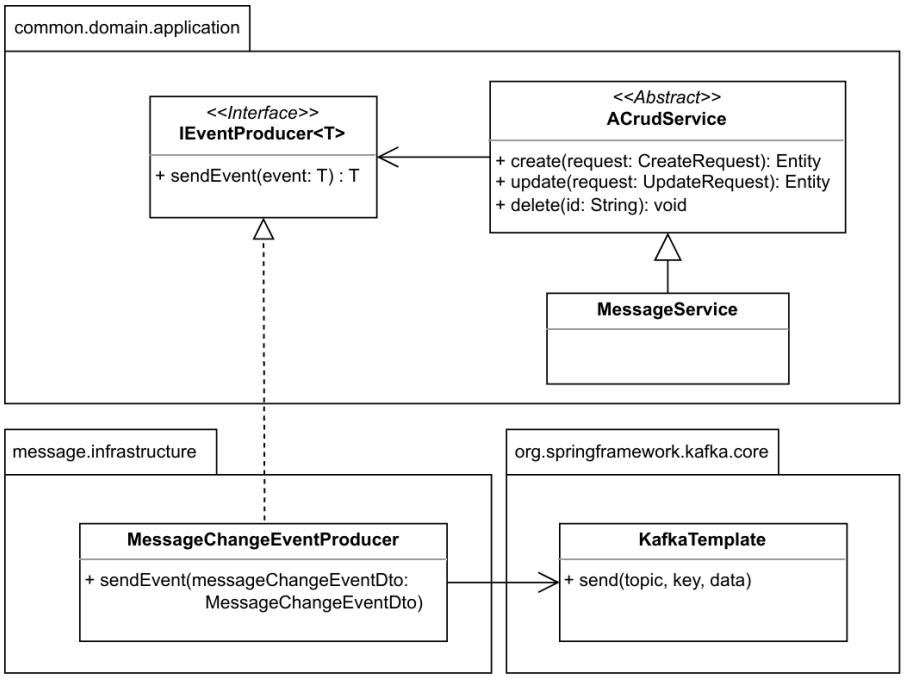


Abbildung 7.16.: Klassendiagramm Event Producer

Mit diesem Ansatz kann die eingesetzte Technologie im Infrastrukturlayer jederzeit ausgetauscht werden, ohne dass die zentrale Business-Logik davon betroffen ist.

7.6.3.2. Notifications

Benutzer sollen zeitnah über Änderungen im System informiert werden. Um ein ständiges Polling im Frontend zu vermeiden, wurde dies über eine *WebSocket*-Verbindung umgesetzt.

Die Benachrichtigungen werden vom *Backend-for-Frontend (BFF)* an die Frontends gesendet. Das *BFF* abonniert hierzu alle relevanten *Kafka*-Topics. Sobald ein `ChangeEvent` in einem Topic eintrifft, benachrichtigt das *BFF* die verbundenen Frontends über eine *WebSocket*-Verbindung. Das vereinfachte Sequenzdiagramm in Abbildung 7.17 illustriert den Nachrichtenfluss zwischen den beteiligten Komponenten.

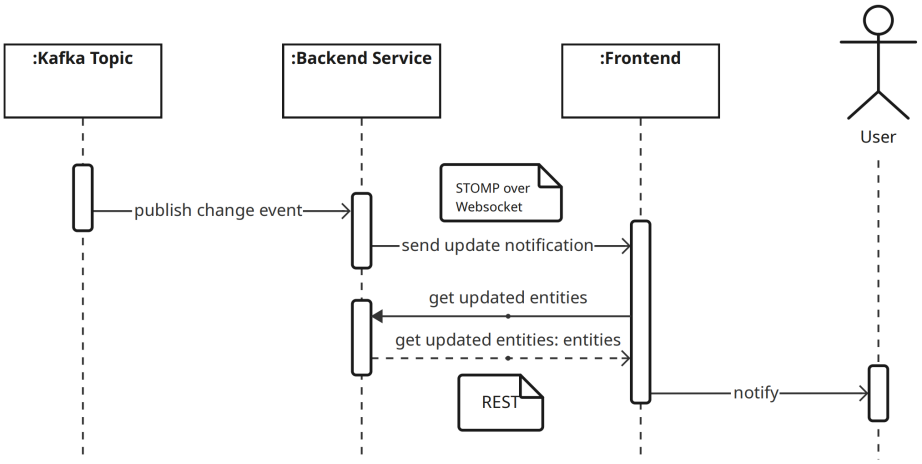


Abbildung 7.17.: Vereinfachtes Sequenzdiagramm Benachrichtigungen

Für die Kommunikation mit den Frontends wird das *STOMP*-Subprotokoll über *WebSockets* eingesetzt. Die Frontend-Clients abonnieren die für sie relevanten *STOMP*-Destinationen und erhalten dadurch gezielt die Benachrichtigungen, die ihren Zuständigkeitsbereich betreffen. Jede Benachrichtigung enthält die Information, welcher Entitätstyp verändert wurde. So wissen die Frontends, dass der lokal vorliegende Zustand nicht mehr aktuell ist. Die Frontends fordern daraufhin den aktuellen Stand der betroffenen Entitäten über die synchrone *REST*-Schnittstelle nach. Anschliessend wird der Benutzer benachrichtigt, dass eine Entität aktualisiert wurde.

Ein detailliertes Sequenzdiagramm mit allen involvierten Objekten sowohl im Backend als auch im Frontend findet sich im Anhang C.

Für die Entkopplung der Domänenlogik von den verwendeten Technologien ergibt sich eine ähnliche Problemstellung wie für die Event Producer in Abbildung 7.16. Das Klassendiagramm in Abbildung 7.18 verdeutlicht, wie diese Entkopplung für die Benachrichtigungen im *BFF* umgesetzt wurde.

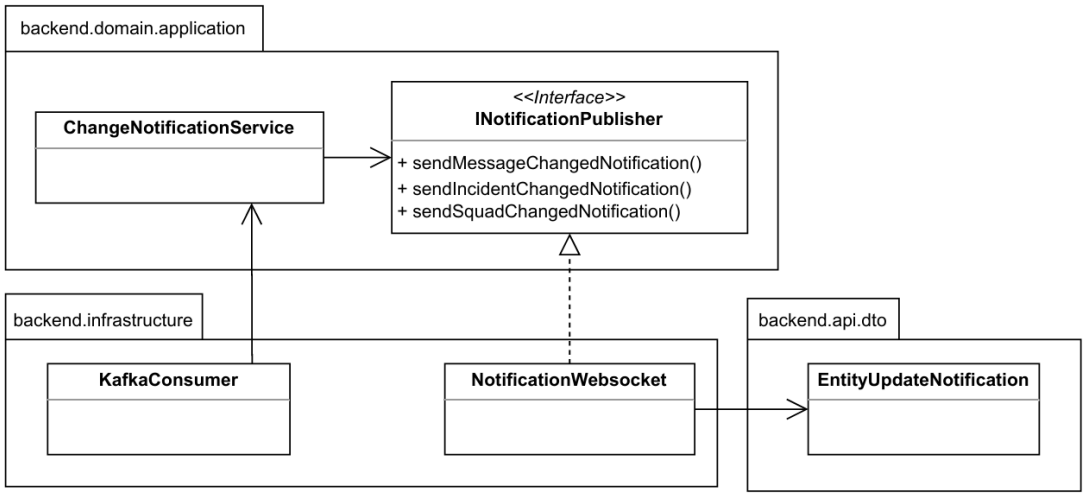


Abbildung 7.18.: Klassendiagramm Notification-Mechanismus

Wann immer eine `ChangeNotification` in einem *Kafka*-Topic eintrifft, wird diese vom `KafkaConsumer` dem `ChangeNotificationService` zur Verarbeitung weitergegeben. Der `ChangeNotificationService` in der Domänenschicht delegiert alle Benachrichtigungsaufgaben an das Interface `INotificationPublisher`. Dieses Interface definiert Methoden zum Versenden von Benachrichtigungen für unterschiedliche Entitätstypen (z. B. Meldungen, Ereignisse oder Trupps). Die konkrete Implementierung des Interfaces erfolgt wiederum im Infrastrukturlayer: Der `NotificationWebsocket` liefert die Benachrichtigungen über *WebSocket / STOMP* an die angeschlossenen Clients aus. Für die Datenübertragung wird das DTO `EntityUpdateNotification` verwendet, das die notwendigen Informationen für die Frontends beinhaltet.

Durch diese Struktur bleibt auch hier die zentrale Business-Logik vollständig unabhängig von der verwendeten Benachrichtigungstechnologie, wodurch eine hohe Austauschbarkeit und Testbarkeit erreicht wird.

7.7. Fehlerbehandlung

Die Backend-Services verfügen über eine zentrale Fehlerbehandlung. Diese fängt alle während der Request-Verarbeitung auftretenden Exceptions ab und konvertiert diese in definierte HTTP-Antworten mit einem einheitlichen Fehlerformat. Hierfür wurden verschiedene spezifische Exceptions implementiert, um unterschiedliche Fehlerfälle in der Businesslogik abzubilden.

7.7.1. Exceptions

Die Exceptions erweitern alle die gemeinsame Basisklasse `ServiceException` aus dem Package `ch.hydra.service.common.exceptions`. Diese kapselt zusätzliche Informationen wie einen Fehlercode, den passenden HTTP-Status und einen Zeitstempel. Dadurch wird eine konsistente und erweiterbare Fehlerkommunikation zwischen Backend und Frontend ermöglicht.

Die Verwendung spezifischer Exception-Typen erleichtert zudem die Fehlersuche im Log, da bereits anhand des Typs der Exception erkennbar ist, um welchen Fehler es sich handelt.

Jede Exception enthält alle relevanten Informationen zur Fehlersituation und kann über einen zentralen Exception-Handler in einen standardisierten Response-Body serialisiert werden (siehe Abschnitt 7.7.3). Dies ermöglicht eine saubere Trennung von Business-Logik und technischer Fehlerverarbeitung.

Eine detaillierte Übersicht aller implementierten Exceptions ist in Anhang G zu finden.

7.7.2. Exception-Factories

Zur Vereinheitlichung der Fehlerbehandlung in den generischen CRUD-Services (siehe Abschnitt 7.5.1) werden Factories eingesetzt, die spezifische Exceptions für jede Komponente erzeugen. Diese Factories implementieren das Interface `IEntityExceptionFactory` und dienen als zentrale Erzeugungsstelle für `NotFound`- und `Deleted`-Exceptions eines bestimmten Entitytyps. Die konkreten Factories werden bei der Erzeugung der Services als Beans injiziert.

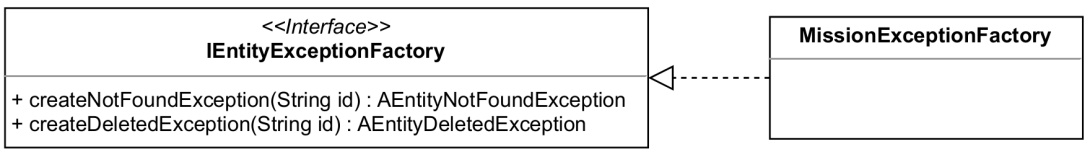


Abbildung 7.19.: MissionExceptionFactory

Das Interface `IEntityExceptionFactory` definiert zwei Methoden zur Erzeugung typisierter Exceptions:

- `createNotFoundException(String id)`
  Erzeugt eine spezifische `AEntityNotFoundException` für das gegebene Entity.
- `createDeletedException(String id)`
  Erzeugt eine spezifische `AEntityDeletedException` für das gegebene Entity.

7.7.3. Exception-Handler

Für die zentrale Fehlerbehandlung im Kontext von Web-Requests bietet Spring eine integrierte Lösung, die im Projekt durch die Klasse `ApiExceptionHandler` im Package `ch.hydra.service.common.error` umgesetzt wurde.

Das Konzept basiert auf den beiden Spring-Annotationen `@RestControllerAdvice` und `@ExceptionHandler`. Mit `@RestControllerAdvice` wird eine Klasse als globaler Error-Handler für alle REST-Controller angegeben. Innerhalb dieser Klasse können pro Exception-Typ Methoden definiert werden, die mit `@ExceptionHandler` annotiert sind. Spring sorgt dafür, dass bei einer während eines Web-Requests auftretenden Exception automatisch diejenige Methode aufgerufen wird, die dem Typ der Exception entspricht.

Auf diese Weise können sowohl erwartbare fachliche Fehler (z. B. eine `ServiceException`) als auch unerwartete technische Fehler abgefangen und in eine standardisierte Antwort vom Typ `ErrorResponseDto` transformiert werden. Diese wird vom REST-Controller an den Client zurückgegeben, typischerweise als JSON-Body der HTTP-Response mit einem passenden HTTP-Status. Dadurch erhält der Client konsistente und verständliche Fehlermeldungen.

7.7.3.1. Fehlerformat

Alle Fehlermeldungen folgen dem gemeinsamen Schema `ErrorResponseDto` aus `ch.hydra.service.common`. Das Fehlerobjekt enthält die folgenden Attribute:

- `errorCode`: Ein semantischer Fehlercode (`ApiErrorCode`-Enum), der den Fehler typisiert
- `message`: Eine Fehlermeldung zu Logging-Zwecken
- `timestamp`: Der Zeitpunkt des Fehlers im RFC-3339-Format
- `path`: Die URI der betroffenen Ressource

Sowohl das `ErrorResponseDto` als auch der darin verwendete Enum-Typ `ApiErrorCode` werden automatisch aus der OpenAPI-Definition der *Common*-Komponente generiert. Dies sowohl für das Java-Backend als auch für das TypeScript-Frontend. Damit stehen die gleichen Typen in beiden Systemteilen zur Verfügung, was die Implementierung von Fehleranzeigen im Frontend vereinfacht und die Konsistenz erhöht.

Das `ErrorResponseDto`-Objekt wird im Exception-Handler in einen JSON-Body serialisiert und als Teil der HTTP-Response zurückgegeben. Auf diese Weise ist eine robuste und nachvollziehbare Fehlerkommunikation zwischen Backend und Frontend gewährleistet.

7.7.3.2. Integration mit FeignException und ErrorParser

Bei der Kommunikation zwischen den Backend-Services können Fehler auch in einem aufgerufenen Service entstehen. In diesem Fall löst der für die Datenübertragung aufgerufene REST-Client eine `FeignException` aus.

Um diese Fehler konsistent zu behandeln, wurde die Klasse `ErrorParser` implementiert. Diese übernimmt die Aufgabe, Fehlerantworten von Feign-Clients zu parsen und in `ServiceException`-Instanzen zu überführen. Hierzu wird der Body der `FeignException`

zu einem `ErrorResponseDto` deserialisiert, aus dem eine neue `ServiceException` erzeugt wird. Diese kann im aufrufenden Service wie jede andere Ausnahme behandelt und in der zentralen Fehlerbehandlung erneut in ein `ErrorResponseDto` übersetzt werden.

7.8. Data-Provider

Beim Start der Applikation können optional Initialdaten in die Datenbank geladen werden. Dies dient im Wesentlichen dem Demonstrations- oder Testbetrieb. Dabei werden automatisch vordefinierte Beispieldaten erzeugt und gespeichert, um die Funktionalität der Anwendung präsentieren und testen zu können. Im produktiven Einsatz hingegen bleibt die Datenbank leer.

Jede Komponente enthält hierfür eigene Data-Provider-Implementationen. Eine Factory, exemplarisch in Abbildung 7.20 für die Incident-Komponente dargestellt, entscheidet zur Laufzeit, welche konkrete Implementierung verwendet wird. Die Factory nutzt dazu die Spring-Annotationen `@Configuration` und `@Bean`, wodurch ihre Factory-Methode automatisch vom Spring-Framework aufgerufen wird. Welche Data-Provider-Klasse instanziiert wird, ist über das Application-Property `incident.dataProvider.type` definiert und wird der Factory per `@Value`-Parameter übergeben. Dadurch kann flexibel zwischen verschiedenen Varianten gewechselt werden, ohne den Anwendungscode anzupassen.

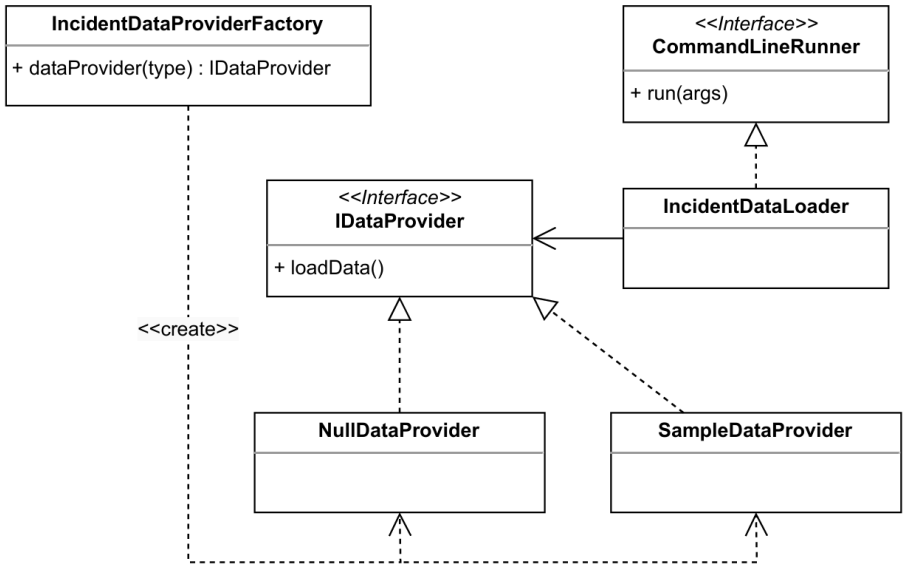


Abbildung 7.20.: Data-Provider-Factory

Für jede Komponente sind jeweils folgende Data-Provider verfügbar:

- **`NullDataProvider`**
  Eine No-Op-Implementierung, die keine Daten lädt und damit für produktive Umgebungen geeignet ist.
- **`SampleDataProvider`**
  Lädt automatisch vordefinierte Testdaten in die Datenbank.

Das Laden erfolgt beim Starten der Applikation über die Klasse `IncidentDataLoader`. Diese erhält über die in Abbildung 7.20 gezeigte Factory automatisch die konkrete Imple-

mentierung des Data-Providers injiziert. Da der Loader das Interface `CommandLineRunner` von `org.springframework.boot` implementiert, wird dessen `run`-Methode von Spring beim Start der Anwendung automatisch ausgeführt. Diese Methode ruft wiederum die Logik des Data-Providers auf, sodass die gewünschten Daten unmittelbar nach dem Start der Applikation zur Verfügung stehen. Alle übrigen Komponenten folgen demselben Konzept.

Durch den Einsatz der Factory ist das Design bewusst erweiterbar gestaltet. Neben den bestehenden Providern kann zukünftig z. B. ein `ArchiveDataProvider` implementiert werden, um historische Einsatzdaten einzuspielen.

Auf diese Weise wird eine klare Trennung zwischen Infrastruktur (Datenbank) und Business-Logik erreicht. Zugleich erleichtert der Data-Provider-Mechanismus sowohl das Testen als auch die Demonstration des Systems mit realistischen Daten. Dieser Ansatz folgt den SOLID-Prinzipien, insbesondere dem *Open-Closed-Prinzip* und dem *Dependency-Inversion-Prinzip* (Martin, 2017).

In diesem Kapitel wurden die zentralen Designentscheidungen für die Applikation vorgestellt. Im Vordergrund stand dabei, eine klare Trennung von Verantwortlichkeiten, eine hohe Wiederverwendbarkeit sowie die Erweiterbarkeit der einzelnen Komponenten sicherzustellen. Durch den Einsatz bewährter Design-Patterns und den SOLID-Prinzipien konnte ein flexibles und robustes Fundament geschaffen werden, das die spätere Weiterentwicklung der Anwendung erleichtert.

8. Implementation

Auch wenn letztlich der Quellcode die präziseste Form der Dokumentation technischer Details darstellt, werden in diesem Kapitel dennoch einige ausgewählte Aspekte der Umsetzungsphase hervorgehoben. Der Anspruch liegt nicht auf einer vollständigen Beschreibung sämtlicher Implementationsdetails, sondern auf der exemplarischen Darstellung von Überlegungen und Entscheidungen, welche die Umsetzungsphase geprägt haben. Ergänzend werden einzelne Implementationsdetails erläutert, soweit sie für das Verständnis des Gesamtsystems von Bedeutung sind.

8.1. Softwareprototyp

Die erste Iteration hatte das Ziel, einen Softwareprototypen im Sinne eines *Walking Skeleton* zu realisieren: einen schlanken, aber vollständigen Durchstich durch alle Schichten und zentralen Komponenten. Anhand einer geeigneten User Story sollte die Tauglichkeit der Architektur in der Praxis verifiziert und so technische Risiken früh minimiert werden.

Als User Story für den Prototypen wurde „*MV-01:* [redacted]“ gewählt. Trotz ihres geringen funktionalen Umfangs setzt diese Story das Vorhandensein sämtlicher Kernbausteine voraus: Persistenz (Datenbank), Event-Streaming-Plattform, Kommunikationsschnittstellen (APIs), *Mission-Subsystem*, *Backend-for-Frontend* sowie das *Operator Frontend*. Darüber hinaus mussten Projektstruktur und Schnittstellen definiert sowie Build- und Deploymentprozesse eingerichtet werden. Zusätzlich galt es, die Infrastruktur aufzusetzen und erste Schritte zur Qualitätssicherung in Form automatisierter Tests umzusetzen.

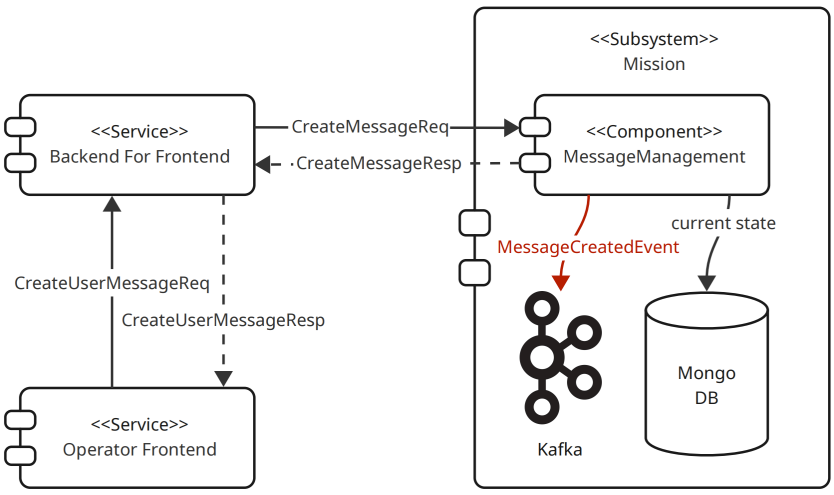


Abbildung 8.1.: Datenflüsse für den Softwareprototypen (informell)

Abbildung 8.1 zeigt die Datenflüsse für die Umsetzung der User Story *MV-01*. Es wird ersichtlich, wie die verschiedenen Systemkomponenten an der Umsetzung der User Story beteiligt sind und dass für deren Erfüllung von der Erfassung im Frontend bis zur Persistierung und Eventerfassung alles durchgängig funktionieren muss.

Mit diesem Prototypen ist es gelungen, die Tauglichkeit der Architektur anhand eines ersten Durchstichs zu belegen und somit die in der Risikoanalyse (Anhang E) identifizierten technischen Risiken des Projekts deutlich zu reduzieren.

- **Architekturrisiken:** Validierung einer sinnvollen Aufteilung des Systems in einzelne Komponenten und Sicherstellung konsistenter Schnittstellen. Dadurch wurde früh überprüft, ob die geplante Architektur und deren in der Designphase erarbeitete Umsetzung tragfähig ist und die Verantwortlichkeiten der einzelnen Systemkomponenten eingehalten werden können.
- **Technologie- und Integrationsrisiken:** Alle Teammitglieder sammelten praktische Erfahrung mit den eingesetzten Technologien. Gleichzeitig wurde gezeigt, dass sich die vorgesehene Architektur mit den gewählten Technologien umsetzen lässt. Das End-to-End-Zusammenspiel von Backend, Frontend, Persistenz und Event-Streaming-Plattform konnte erfolgreich validiert werden, wodurch Integrationsprobleme frühzeitig erkannt und behoben werden konnten.
- **Betriebsrisiken:** Durch den Aufbau reproduzierbarer Entwicklungs- und Demo-Umgebungen mittels Containerisierung wurde sichergestellt, dass Builds und Deployments stabil, nachvollziehbar und auf unterschiedlichen Systemen einsetzbar sind.
- **Qualitätssicherung:** Mit der frühen Etablierung automatisierter Tests wurden erste Sicherheitsnetze geschaffen. Fehler liessen sich dadurch schneller erkennen, Regressionen wurden vermieden und Änderungen sowie Erweiterungen in späteren Iterationen konnten mit geringerem Risiko durchgeführt werden.

Nach der Umsetzung des Softwareprototypen wurden in einer Phase der Reflexion wichtige Erkenntnisse gewonnen, welche in der nächsten Iteration umgesetzt werden konnten. So zeigte sich etwa, dass sich die Verwendung von *Avro-Schemas* nur mit erheblichem Aufwand mit *AsyncAPI* und der automatischen Codegenerierung kombinieren lässt. Daher wurde auf *JSON-Schemas* umgestellt. Darüber hinaus traten Inkonsistenzen in den *API-Definitionen* zutage, die bereinigt wurden. Auch wurde deutlich, dass es vorteilhaft ist, für Schnittstellenattribute wie Zeitstempel oder Adressen auf etablierte, technologieunabhängige Standards zurückzugreifen. Zusätzlich konnten Schwächen im Detaildesign einzelner Services identifiziert und behoben werden.

8.2. Projektstruktur

Die Wahl der Architektur und der eingesetzten Technologien hatte einen grossen Einfluss auf die Projektstruktur, Buildprozesse, Integration und das Deployment der Applikation. Siehe dazu auch den nachfolgenden Abschnitt 8.3 (Buildprozess) sowie Kapitel 11 (Continuous Integration und Deployment).

Bei der Suche nach einer geeigneten Projektstruktur wurden die *Backend-Services* und die *Frontends* getrennt betrachtet.

Backend-Services: Aufgrund der grossen Anzahl an Komponenten, welche aus der Architekturphase hervorgegangen sind, wurde entschieden, für die Backend-Services eine Monorepo-Struktur anstelle einer Polyrepo-Struktur zu verwenden.

Bei der Verwendung eines Monorepos wird für das Gesamtprojekt ein einzelnes Git-Repository verwendet. Die einzelnen Komponenten werden in einer gemeinsamen Ordnerstruktur gehalten, sind aber bezüglich des Sourcecodes unabhängig. Jeder Service und jede Komponente innerhalb des Repositories besitzt eigene Abhängigkeiten und Build-Konfigurationen.

Innerhalb desselben Repositories ist der aktuelle Entwicklungsstand jederzeit für alle sichtbar, was das Verständnis der Zusammenarbeit im Team erleichtert. Ein Nachteil ist der grössere Umfang des Repositories sowie potenziell komplexere Buildprozesse. Allerdings zeigen Beispiele wie der Linux-Kernel, dass auch sehr grosse Projekte erfolgreich als Monorepo organisiert werden können.

Frontend: Um die Komplexität durch unterschiedliche Technologien innerhalb eines Repositories zu vermeiden, wurde das Frontend in einem separaten Projekt abgelegt und nicht in das oben genannte Monorepo integriert. Somit werden nur Komponenten gemeinsam verwaltet, die auf demselben Tech-Stack basieren. Konsequenterweise würde ein als Mobile-App ausgeführtes Field-UI, welches auf einer anderen Technologie basiert als das Operator-UI, auch in ein eigenes Repository ausgelagert.

8.3. Buildprozess

8.3.1. Backend-Services

Mit der Entscheidung, das Backend (`hydra-services`) mit Java zu implementieren, standen die beiden Buildsysteme *Gradle* und *Maven* zur Auswahl. Ziel war es, einen möglichst einheitlichen Build-Prozess aufzubauen, der sowohl für die lokale Entwicklung als auch in der Integrations-Pipeline eingesetzt werden konnte. Zudem sollte das Buildsystem eine einfache Definition einzelner Buildschritte ermöglichen, die Interaktion mit verschiedenen Technologien unterstützen und den Umgang mit den zahlreichen Modulen erleichtern.

Unter Berücksichtigung dieser Punkte sowie der im Team vorhandenen Erfahrungen fiel die Wahl auf *Gradle*. Der Build wurde als *Multi-Project* realisiert, um die einzelnen Komponenten innerhalb des Monorepos möglichst unabhängig zu halten. Dieser Entscheid erfolgte zusammen mit der Definition der in Abschnitt 8.2 beschriebenen Projektstruktur.

Das Gradle-Root-Projekt besteht aus den folgenden Komponenten:

- **Services (blau):** Services, welche eigenständig bereitgestellt werden können. Als finale Buildartefakte entstehen Docker Container Images.
- **APIs (grün):** Definitionen der API-Endpoints (*OpenAPI* und *AsyncAPI*), welche beim Build in anderen Gradle-Projekten referenziert werden. Als finale Buildartefakte entstehen daraus Java-Klassen für die REST-Controller und -Clients.
- **Common (violett):** Gemeinsame Typendefinitionen, welche von anderen Gradle-Projekten benötigt werden.

- **buildSrc (gelb):** Alle Services verwenden die gleiche Build-Logik zur Erstellung ihrer Artefakte. Um diese zentral verfügbar zu machen, wurden zwei Gradle-Plugins entwickelt.

Abbildung 8.2 zeigt den Aufbau des Gradle-Multiprojekts. Zudem ist ersichtlich, welche Komponenten gemeinsame Bibliotheksobjekte aus `buildSrc` nutzen.

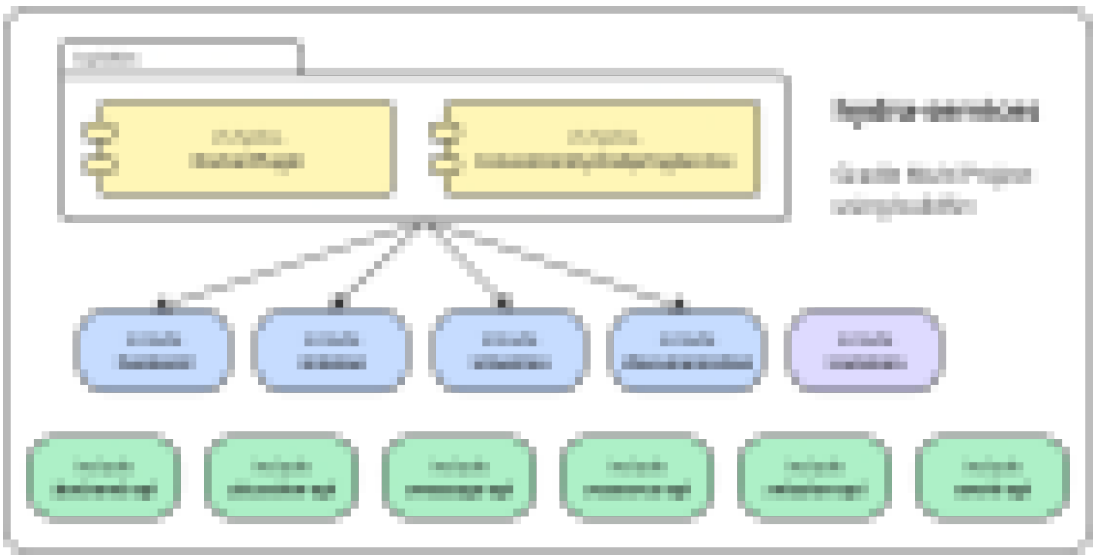


Abbildung 8.2.: Aufbau des Gradle-Multiprojekts (informell)

8.3.1.1. Gradle-Plugins

Gradle erlaubt die Erweiterung des Buildtools mit eigenen Plugins (Gradle, 2025). Für die gemeinsame Buildlogik wurden zwei Plugins implementiert, um Codeduplizierung zu vermeiden. Sie sind Teil des Repositories, können bei Bedarf jedoch in ein eigenständiges Projekt ausgelagert und als Artefakte publiziert werden.

- **DockerPlugin:** Beinhaltet die gemeinsame Buildlogik für das Erstellen der Docker Images.
- **ConventionHydraSpringService:** Beinhaltet die Definition der Importe, welche in allen Spring-Service Buildprojekten benötigt werden. Dadurch können diese Abhängigkeiten einheitlich an einem Ort definiert und verwaltet werden.

Die entsprechenden Build-Artefakte und Gradle-Plugins können somit sehr einfach in den Build-Dateien der einzelnen Gradle-Projekte referenziert werden:

```
1  import ch.hydra.ConventionHydraSpringService
2  import ch.hydra.DockerPlugin
3  dependencies {
4    implementation project(':common')
5    implementation project(':message-api')
6    ...
7    }
```

Listing 8.1: Beispielhafte Gradle Build-Datei.

8.3.2. Frontend

Mit der Wahl von TypeScript und React standen im Frontend (`hydra-operator`) eine Vielzahl von Tools für die Buildchain zur Auswahl. Auf ein React-Framework wie beispielsweise *Next.js* wurde bewusst verzichtet. Stattdessen kommen im Frontend *pnpm* als Paketmanager für die Abhängigkeiten und *Vite* als Build-Tool sowie Entwicklungsserver zum Einsatz (Vite, 2025).

Mit *Vite* kann lokal entwickelt werden, sofern *Node.js* installiert ist. Alternativ lässt sich das Frontend mittels Docker und dem bereitgestellten *Dockerfile* lokal ausführen. Um die Grösse des finalen Docker-Images zu minimieren, wurde ein auf Alpine Linux basierender Multi-Stage-Build umgesetzt.

Abbildung 8.3 zeigt, welche Artefakte am Build der beiden im Frontend verwendeten TypeScript-Bibliotheken beteiligt sind. Diese werden für die Verwendung der Backend-API sowie der WebSocket-Verbindung benötigt.

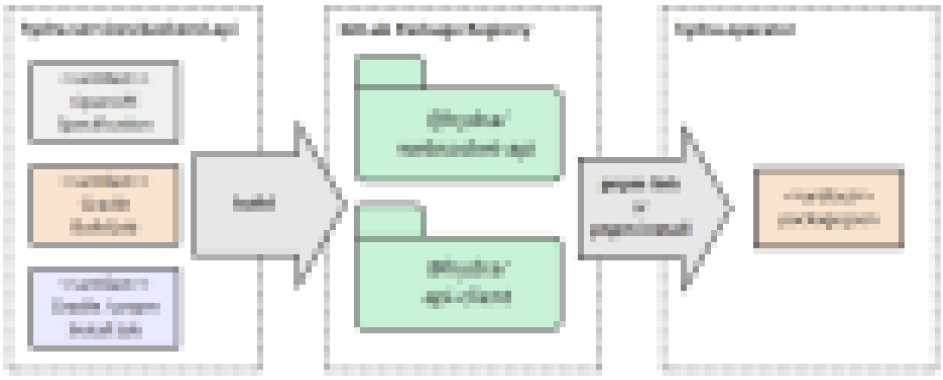


Abbildung 8.3.: Frontend TypeScript Packages

Die beiden Packages `@hydra/websocket-api` und `@hydra/api-client` enthalten die generierten TypeScript-Klassen für die DTOs sowie den REST-Client für die Kommunikation mit der Backend-API. Im Gradle-Projekt `hydra-services:backend-api` existieren dafür entsprechende Buildjobs. Für die lokale Frontend-Entwicklung können diese aus den OpenAPI-Spezifikationen generiert und lokal installiert werden. Anschliessend lassen sich die beiden Bibliotheken über *pnpm* im Frontend verlinken. Auf diese Weise können Änderungen an der API im Backend während dem Entwickeln schnell umgesetzt und direkt im Frontend verwendet werden. Der genaue Ablauf der Verlinkung und lokalen Entwicklung ist in den *README*-Dateien der Repositories dokumentiert. Der automatisierte Build-Prozess wird in Kapitel 11 beschrieben.

8.4. User Interface

Die Applikation *Hydra* bietet zwei verschiedene Benutzeroberflächen:

- **Operator-UI:** Wird von der Einsatzleitung und den Operators in der Einsatztentrale verwendet. Bietet einen Überblick über die aktuelle Lage sowie Managementwerkzeuge für die Einsatzleitung.

- **Field-UI:** Wird von Einsatzkräften während des Einsatzes im Feld verwendet. Es soll alle notwendigen Informationen für die Bearbeitung von Aufträgen bereitstellen und gleichzeitig den Einsatzkräften die Möglichkeit bieten, Informationen aus dem Feld an die Einsatzleitung zu übermitteln.

Die Architektur sowie das Deployment-Modell (siehe Abschnitt 6.9) sehen für das Field-UI grundsätzlich einen eigenständigen Service vor. Aufgrund des hohen Entwicklungsaufwands für eine dedizierte mobile Applikation wurde im Rahmen dieser Masterarbeit jedoch bewusst darauf verzichtet. Um dennoch aussagekräftige Tests mit Benutzern durchzuführen zu können und erstes Feedback einzuholen (siehe Abschnitt 12.1 und Anhang F), wurden im MVP ausgewählte Ansichten des Field-UI in das Operator-UI-Projekt integriert. Da das so implementierte Field-UI die Codebasis mit dem Operator-UI teilt, konnte ein grosser Teil der vorhandenen Logik direkt wiederverwendet werden.

Auf diese Weise konnten zentrale Benutzerinteraktionen erprobt und wertvolles Feedback für eine eigenständige Mobile-App gesammelt werden.

8.4.1. Technologien

Die folgenden verwendeten Technologien und Bibliotheken prägen das Frontend massgeblich:

UI-Design: Für das Theming und Layout der Applikation kommt *Material-UI (MUI)* zum Einsatz (Material UI, 2025). Diese Bibliothek stellt vordefinierte, visuell ansprechende Komponenten bereit. Dadurch entfällt grösstenteils das manuelle Styling mittels *CSS*, was den Aufwand für die Gestaltung einer benutzerfreundlichen Oberfläche erheblich reduziert.

Internationalisierung (i18n): Durch den Einsatz einer *i18n*-Bibliothek kann die Applikation in mehreren Sprachen genutzt werden. Im Rahmen der Masterarbeit wurden die beiden Sprachen Englisch und Deutsch integriert. Eine Erweiterung um zusätzliche Sprachen erfordert lediglich die Implementierung der entsprechenden Übersetzungsdateien im JSON-Format.

Karte: Ein wesentlicher Bestandteil des User Interface ist die Lagekarte. Für deren Integration wurde *OpenLayers* verwendet (OpenLayers, 2025). Diese Bibliothek ist weit verbreitet und bietet einen grossen Funktionsumfang. Das ist besonders wichtig, da viele der geplanten und gewünschten Features einen direkten Bezug zur Karte haben.

Das verwendete Kartenmaterial (Satellit, topografisch und Graustufen) wurde von *swisstopo* bezogen (swisstopo, 2025b). Für die Darstellung der Elemente anhand von Adressen wurde das amtliche Adressenverzeichnis von *swisstopo* genutzt (swisstopo, 2025a).

8.4.2. Aufbau

Das Frontend ist modular aufgebaut und orientiert sich an den folgenden Grundsätzen:

- **Komponentenorientierung:** Die Applikation wird aus möglichst wiederverwendbaren Komponenten aufgebaut. Jede Funktionseinheit (Incident, Mission, Squad, Message, Map) definiert darüber hinaus eigene, spezialisierte Komponenten wie

beispielsweise Formulare und Listen.

- **Globales State-Management:** Der globale Zustand wird über Stores organisiert, wodurch Datenkonsistenz über alle Seiten und Komponenten hinweg gewährleistet wird.
- **Theming & UI-Konsistenz:** Durch die zentrale Theme-Konfiguration wird ein einheitliches Erscheinungsbild sichergestellt.
- **Erweiterbarkeit:** Die klare Trennung in *pages*, *components*, *hooks*, *context* und *utils* erleichtert die Erweiterung des Systems um neue Features.

Im Folgenden wird die Projektstruktur beschrieben, welche die Einhaltung der obigen Grundsätze ermöglicht und fördert.

Die eigentlichen Seiten der Anwendung befinden sich im Verzeichnis `pages/`, welches in die Bereiche *Incidents*, *Missions*, *Squads*, *Messages* und *Map* gegliedert ist. Für die visuelle Darstellung werden die im Verzeichnis `components/` bereitgestellten wiederverwendbaren Bausteine genutzt, die nach Anwendungsbereich unterteilt sind (z. B. Layout-Elemente, Kartenkomponenten oder Formulare für spezifische Entitäten).

Für das Field-UI existiert mit `fieldUi/` ein paralleler Aufbau mit derselben Struktur. Die darin enthaltenen Komponenten sind speziell für die Anzeige auf mobilen Geräten und den Einsatz im Feld optimiert.

Die verschiedenen Stores zur Verwaltung des globalen Zustands von Incidents, Missions, Squads und der Map sind im Verzeichnis `context/` implementiert. Wiederkehrende Funktionalitäten wurden in Form eigener Hooks (`hooks/`) gekapselt, beispielsweise für API-Abfragen oder die Initialisierung der Karte.

Hilfsfunktionen, die keine direkten Manipulationen am Store vornehmen, sondern Validierungen, Datenaufbereitungen oder Navigationslogik beinhalten, sind im Verzeichnis `utils/` implementiert. In `theme/` sind alle Theming-bezogenen Ressourcen für eine konsistente Gestaltung des UIs auf Basis von Material-UI abgelegt. Ergänzend dazu werden statische Ressourcen (`assets/`) wie Icons und Kartenmarker sowie Sprachdateien (`locales/`) für die Mehrsprachigkeit bereitgestellt.

Darüber hinaus existieren im Verzeichnis `views/` einzelne Spezialansichten, die nicht direkt als Seiten fungieren, sondern grössere UI-Blöcke kapseln.

8.4.3. Operator-UI

Das Operator-UI besteht hauptsächlich aus den folgenden funktionalen Komponenten:

Hauptansicht Soll dem Operator einen groben Überblick über den aktuellen Einsatz verschaffen. Nebst statistischen Informationen wie beispielsweise dem Status der Ereignisse innerhalb eines Einsatzes sind auch die aktuellsten Meldungen ersichtlich.

Abbildung 8.4.: Operator-UI Hauptansicht

Ereignisboard Ermöglicht es dem Operator, den Status aller Ereignisse in einer einzigen Ansicht zu verwalten. Per Drag-and-Drop kann der Status eines Ereignisses angepasst werden, indem dieses in die entsprechende Spalte verschoben wird.

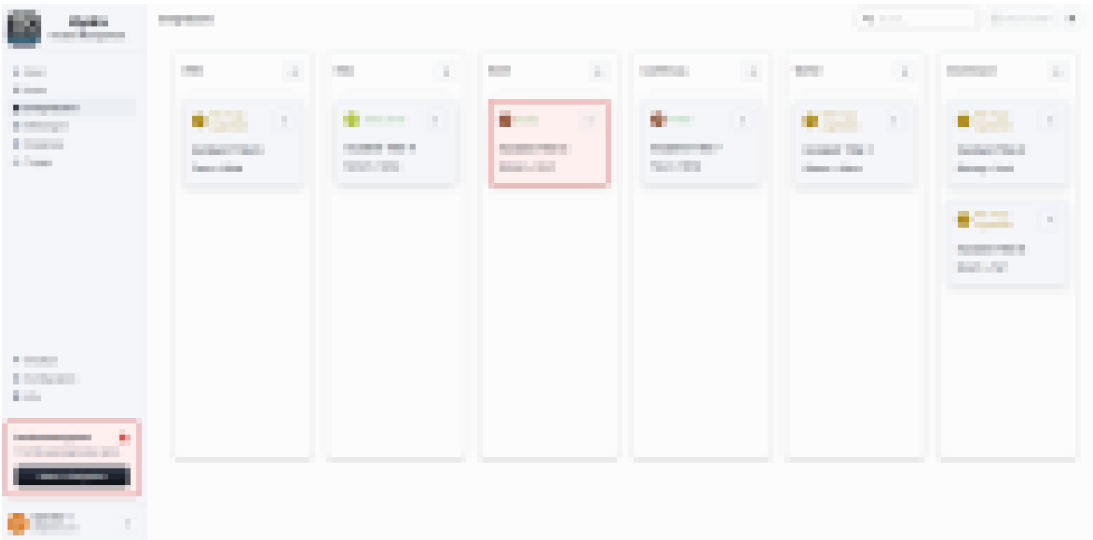


Abbildung 8.5.: Operator-UI Ereignisboard

Karte Verschafft dem Operator einen Überblick über die örtlich verteilten Ereignisse, Meldungen und Ressourcen. Sowohl Meldungen als auch Ereignissen kann eine Adresse zugewiesen werden, wodurch sie auf der Karte erscheinen. Sind einem Ereignis Einsatzkräfte zugeteilt, werden auch diese dargestellt. Unterschiedliche Symbole erleichtern die Identifikation der Elemente und sorgen für eine schnelle und präzise Orientierung.

Abbildung 8.6.: Operator-UI Karte

Listenansichten Für die verschiedenen Entitäten wie Einsätze, Ereignisse, Meldungen oder Squads wurden entsprechende Formulare und Anzeigeelemente implementiert, um deren Eigenschaften einzusehen und diese zu bearbeiten.

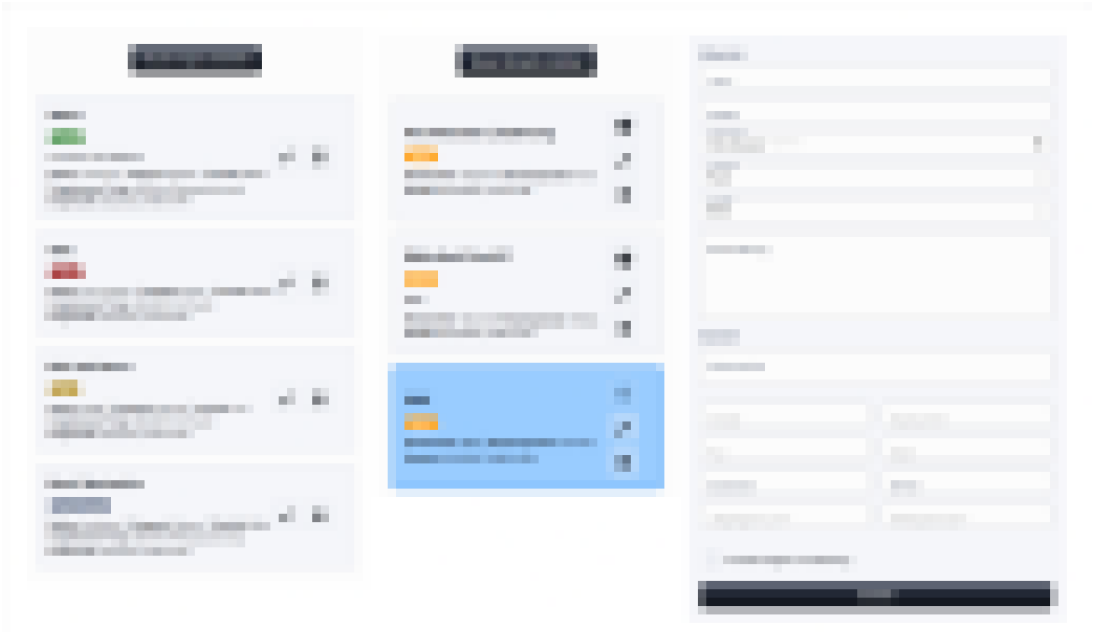


Abbildung 8.7.: Operator-UI Listen und Formulare

8.4.4. Field-UI

Für das Field-UI wurden eigene, auf die mobile Nutzung optimierte Seiten umgesetzt. Der Funktionsumfang ist im Vergleich zum Operator-UI bewusst reduziert und beschränkt sich auf das Nötigste, das Einsatztrupps im Feld für die Erledigung ihrer Aufgaben benötigen.

Interaktionen im Feld Um die Bedienung für Einsatzkräfte im Feld möglichst einfach zu gestalten, wurden die erforderlichen Bearbeitungsschritte in Form von einfachen Aktionen umgesetzt. So kann z. B. ein Reko-Trupp bei der Bearbeitung eines Ereignisses die notwendigen Felder ausfüllen und dann die gewünschte Aktion mit einem Klick ausführen. Beim *Abschliessen* wird dadurch nicht nur die Speicherung angestossen, sondern gleichzeitig der Status des Ereignisses automatisch auf *Bereit* gesetzt, sodass der Operator die weitere Bearbeitung übernehmen kann. Die Verwendung von Aktionen hat gegenüber einem manuellen Setzen von Zuständen den Vorteil, dass die Bedienung intuitiv und schnell ist und auch unter erschwerten Bedingungen keine falschen Zustände ausgewählt werden können.

Abbildung 8.8.: Field-UI Ereignis - Reko-Ansicht

Nebst der Möglichkeit, die ihnen zugewiesenen Ereignisse zu bearbeiten, stehen den Einsatzkräften im Field-UI weitere Funktionen zur Verfügung:

- Auswahl von Einsatz und Trupp, um Ereignisse im richtigen Kontext bearbeiten zu können.
- Anzeigen und Bearbeiten von Informationen zum eigenen Trupp.
- Karte mit allen aktuellen Ereignissen, Meldungen und Informationen abrufen.

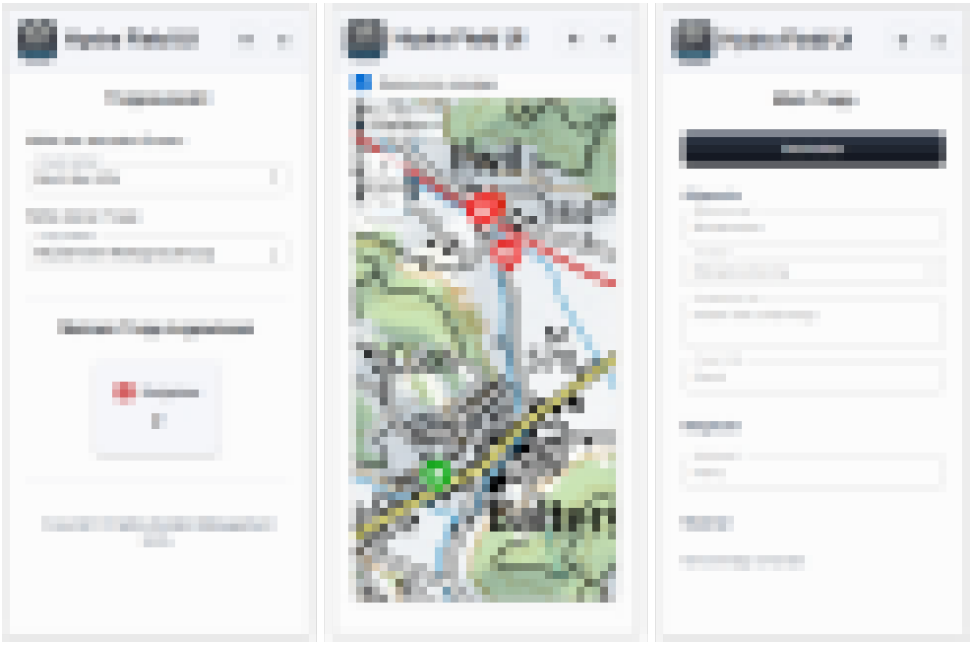


Abbildung 8.9.: Field-UI Ansichten

Weiterentwicklung Obwohl die Implementierung im Rahmen des MVP bewusst minimal gehalten wurde, fielen die Rückmeldungen aus den Benutzer-Tests insgesamt positiv aus. Eine genauere Analyse des Feedbacks soll aufzeigen, welche Funktionen im Feldeinsatz tatsächlich benötigt werden und in welcher Form diese bereitgestellt werden sollten. Auf Grundlage dieser Erkenntnisse kann entschieden werden, in welche Richtung die Weiterentwicklung des Field-UI erfolgen soll.

Mögliche Ansätze hierfür sind:

- Entwicklung einer eigenständigen Webanwendung
- Umsetzung als React-basierte Mobile App
- Verwendung von Kotlin Compose Multiplatform für Android und iOS

8.5. Spring

Für die auf Java basierenden Backend-Services wurde *Spring (Spring Boot)* als Framework gewählt. Diese Entscheidung wurde sorgfältig abgewogen, da die Verwendung eines Frameworks stets auch eine Abhängigkeit mit sich bringt. Gleichzeitig erleichtert Spring die

Implementierung von Datenbankzugriffen erheblich, bietet Dependency Injection (DI) und unterstützt die Nutzung von Profilen.

Durch Spring-Profile kann die Konfiguration der Services flexibel an die jeweilige Umgebung angepasst werden. Diese Möglichkeit wurde genutzt, um verschiedene Konfigurationen für die lokale Entwicklung sowie die produktive Umgebung zu definieren. Für die Services *Situation*, *Backend for Frontend* und *Mission* wurden die folgenden Profile implementiert:

- **application:** Allgemeine Spring- und Applikationskonfiguration, die beim Start immer eingelesen wird. Enthält Definitionen, welche über alle Umgebungen gleich sind. Hier werden beispielsweise auch die erforderlichen Profilgruppen für das Logging definiert.
- **application-docker:** Produktive Konfiguration für das starten oder builden der Services in einer Docker-Umgebung.
- **application-local:** Produktive Konfiguration für das starten oder builden der Services als eigenständige Java-Applikation *(.jar)*.
- **application-cicd:** Testkonfiguration für das Ausführen der Unit- und Integrationstests innerhalb der CI/CD-Pipeline.
- **application-test:** Testkonfiguration für das Ausführen der Unit- und Integrationstests in der lokalen Entwicklungsumgebung.

Diese finden sich in den jeweiligen Services im Repository *hydra-services* unter: `src/{main, test}/resources/*.properties`.

In diesem Kapitel wurde nur eine Auswahl von Implementationsdetails beschrieben. Die vorgestellten Themen und Entscheidungen haben das Projekt wesentlich geprägt und zeigen exemplarisch auf, wie die Architekturvorgaben in konkrete Lösungen überführt wurden. Nicht alle technischen Aspekte konnten im Detail behandelt werden, doch die zentralen Entscheidungen und deren Umsetzung geben einen Einblick in die Implementationsdetails von *Hydra*.

9. Qualitätsmassnahmen

Im Projekt *Hydra* kamen unterschiedliche Massnahmen zur Anwendung, um die Qualität der entwickelten Software sicherzustellen. Dazu zählen sowohl technische als auch organisatorische Ansätze, wie die Verwendung von Coding-Guidelines, die Durchführung von Code-Reviews sowie der Einsatz automatisierter Tests und Build-Prozesse. Die folgenden Abschnitte geben einen Überblick über die umgesetzten Massnahmen zur Qualitätssicherung.

9.1. Coding-Guidelines

Während der Implementierung wurden grundsätzlich die etablierten Java Coding-Guidelines von Oracle beachtet (Oracle, n. d.).

Zusätzlich wurden projektspezifische Konventionen eingeführt, um die Lesbarkeit und Wartbarkeit weiter zu verbessern. Die projektspezifischen Coding-Guidelines sind als beiliegendes Dokument *Coding-Guidelines.pdf* zu finden. Sie dienten während der gesamten Entwicklung als Referenz und wurden auch bei den Code-Reviews überprüft.

9.2. Merge Requests

Während der Implementierung wurde konsequent nach dem Vier-Augen-Prinzip gearbeitet. Nach der Umsetzung eines neuen Features oder einer Anpassung im bestehenden Code erstellte der verantwortliche Entwickler einen Merge Request im GitLab-Repository. Dieser wurde nicht direkt in den Develop-Branch gemergt, sondern zunächst von einem anderen Teammitglied überprüft.

Das Review umfasste sowohl technische Aspekte (z. B. Einhaltung des Architekturkonzepts, der Coding-Guidelines, Testabdeckung, Lesbarkeit und Wartbarkeit des Codes) als auch fachliche Kriterien im Hinblick auf die korrekte Umsetzung der Anforderungen. Erst nach Freigabe durch den Reviewer wurde der Merge Request in den Develop-Branch übernommen.

Durch dieses Vorgehen wurde die Codequalität gesteigert, potenzielle Fehler oder Missverständnisse frühzeitig erkannt und gleichzeitig ein gemeinsames Verständnis für den Code im Team gefördert.

9.3. Automatisierter Build-Prozess

Um den Entwicklungsprozess effizient zu gestalten, wurde ein automatisierter Build-Prozess eingerichtet. Mithilfe der CI/CD-Funktionalität von GitLab wurde bei jedem Push

und jedem Merge Request automatisch eine Pipeline ausgeführt. Die hierfür verwendete Infrastruktur ist in Kapitel 10 beschrieben.

Der Build-Prozess basierte auf Gradle und umfasste Standard-Tasks wie *clean*, *build* und *test*. Darüber hinaus wurden eigene Build-Tasks ergänzt, welche während des Builds die Generierung von Artefakten und deren Verteilung anstossen. So wurde sichergestellt, dass der Code fehlerfrei kompiliert, alle Unit- und Integrationstests erfolgreich ausgeführt und die resultierenden Artefakte konsistent erstellt wurden.

Durch den automatisierten Ablauf konnten Fehler frühzeitig erkannt und manuelle Arbeitsschritte vermieden werden. Da bei Änderungen am Code sofort ein Build gestartet und die Ausführung aller Tests angestossen wurde, erhielt jedes Teammitglied zeitnah Feedback. So konnte die Qualität des aktuellen Entwicklungsstands jederzeit sichergestellt werden.

9.4. Automatisierte Tests

Ein zentrales Element zur Sicherstellung der Qualität und Stabilität der entwickelten Applikation sind automatisierte Tests. Sie ermöglichen es, Funktionalitäten kontinuierlich zu überprüfen, Regressionen frühzeitig zu erkennen und die Wartbarkeit des Systems zu erhöhen. Im Projekt *Hydra* kamen für die Backend-Services sowohl Unit- als auch Integrationstests zum Einsatz.

Die automatisierten Tests wurden direkt in die Build-Pipeline integriert, sodass bei jedem Commit oder Merge Request automatisch sichergestellt wurde, dass alle Unit- und Integrationstests erfolgreich durchlaufen wurden.

Eine detaillierte Übersicht über die ausgeführten Unit- und Integrationstests ist in den beiliegenden Dokumenten *Unit-Test-Results.html* und *Integration-Test-Results.html* zu finden.

9.4.1. Unit-Tests

Unit-Tests prüfen einzelne Klassen isoliert auf Funktionalität. Bei Bedarf werden Mocks eingesetzt, um einzelne Abhängigkeiten zu ersetzen und so die Tests zu vereinfachen. Auf die Verwendung von Spring-Boot-Tests wurde in den Unit-Tests bewusst verzichtet, damit die Tests leichtgewichtig und schnell bleiben. Die Unit-Tests sind jeweils in den Subprojekten unter dem Verzeichnis `src/test/java/*/unit` zu finden.

Für die Implementierung der Unit-Tests wurde das Java-Testframework *JUnit 5* verwendet. Abhängigkeiten wurden mit *Mockito* gemockt.

Bei der Implementierung wurde darauf geachtet, nicht nur den Happy Path zu testen, sondern das System gezielt mit Fehlerfällen zu konfrontieren. Gerade bei Unit-Tests lassen sich solche Fehlerfälle vergleichsweise einfach prüfen. Es wurde darauf geachtet, alle relevanten Fehlerpfade in den einzelnen Klassen abzudecken. So wurden im *Mission-* und *BFF*-Service total 366 Unit-Tests implementiert, welche die korrekte Funktionsweise der Applikation sicherstellen.

9.4.2. Integration-Tests

Im Gegensatz zu den Unit-Tests überprüfen die Integration-Tests das Zusammenspiel mehrerer Komponenten innerhalb der Applikation. Dabei werden keine (oder nur wenige) Abhängigkeiten gemockt. Wo immer möglich wird die reale Zusammenarbeit zwischen den Systemkomponenten getestet.

Im Projekt wurden dazu insbesondere die Integration mit der Mongo-Datenbank und der REST-Kommunikation zwischen dem *Backend-for-Frontend-Service (BFF)* und dem Mission-Service getestet, inklusive des Error-Handlings. Dabei wird die vollständige Spring-Applikation in einem Testkontext hochgefahren und über direkte Methodenaufrufe im *BFF*- oder *Mission*-Service getestet, ob die Komponenten wie vorgesehen zusammenspielen. So konnte beispielsweise überprüft werden, ob Daten korrekt über die Repository-Schicht gespeichert, von Services verarbeitet und von der REST-APIs zurückgegeben werden.

Die Integration-Tests sind jeweils in den Subprojekten unter dem Verzeichnis `src/test/java/*/integration` zu finden. Sie ergänzen die Unit-Tests, indem sie sicherstellen, dass die einzelnen Komponenten nicht nur isoliert korrekt arbeiten, sondern auch in Kombination als stabiles Gesamtsystem funktionieren. Dies wird mit insgesamt 86 Integration-Tests im Mission- und BFF-Service getestet.

9.4.3. Code-Coverage

Zur Bewertung der Qualität der automatisierten Tests wurde im Projekt das Tool *JaCoCo (Java Code Coverage)* eingesetzt. Es misst, welcher Anteil des Sourcecodes während der Ausführung der Testfälle tatsächlich durchlaufen wird. Die Messung erfolgt automatisiert im Rahmen des Build-Prozesses und liefert Berichte über die Testabdeckung auf Klassen-, Paket- und Gesamtprojektebene.

JaCoCo unterscheidet verschiedene Metriken:

- **Branch Coverage**: Anteil der geprüften Verzweigungen (z. B. `if`-Anweisungen).
- **Instruction Coverage**: Anteil der ausgeführten Bytecode-Instruktionen.

Die Ergebnisse geben Hinweise auf mögliche Lücken in den Tests. Ein hoher Coverage-Wert bedeutet, dass grosse Teile des Codes ausgeführt wurden, garantiert jedoch noch keine Fehlerfreiheit. Umgekehrt muss eine Abdeckung unter 100 % nicht zwingend problematisch sein, wenn nicht kritische und schwer testbare Pfade (z. B. Logging, Fehlerbehandlungen) bewusst ausgeschlossen wurden.

Im Projekt diente die Code Coverage daher als unterstützende Kennzahl, um die Qualität der Tests zu überwachen und gezielt Stellen mit geringer Abdeckung zu identifizieren. Die Werte wurden nicht als fixes Qualitätsziel verstanden, sondern als Anhaltspunkt zur kontinuierlichen Verbesserung der Tests. Ausserdem wurden gewisse Teile des Codes (z. B. mit OpenAPI generierte Klassen) bewusst von der Messung ausgeschlossen.

Im Projekt *Hydra* wurde im Backend-Code eine Instruction Coverage von 96 % und eine Branch Coverage von 98 % gemessen. Diese Werte zeigen, dass der Grossteil des produktiven Codes von Tests ausgeführt wird. Somit ist eine gute Ausgangslage geschaffen, um die Stabilität der Logik durch zusätzliche Massnahmen wie die in Abschnitt 9.7.1 erwähnten Mutation-Tests weiter zu verbessern.

Eine detaillierte Auswertung der Code Coverage ist im beiliegenden Dokument *Code-Coverage.html* zu finden.

9.5. Manuelle Tests

Für das User-Interface wurde im Rahmen der Masterarbeit auf automatisierte Tests verzichtet. Die implementierten Use Cases sind noch nicht sehr komplex und können gut manuell getestet werden. Zudem wurden während der Entwicklung Refactorings an den Benutzeroberflächen vorgenommen, was laufende Anpassungen der automatisierten UI-Tests zur Folge gehabt hätte.

Stattdessen wurden für das Operator-UI und das Field-UI manuelle Testpläne erstellt. Diese definieren die wichtigsten Abläufe und können bei Bedarf systematisch und wiederholbar ausgeführt werden. Auf diese Weise wurde sichergestellt, dass die Kernfunktionen trotz fehlender Testautomatisierung überprüft werden.

Die zugehörigen Testprotokolle finden sich in den beiliegenden Dokumenten *Operator-UI-Test-Results.pdf* und *Field-UI-Test-Results.pdf*.

Eine mögliche Ergänzung durch automatisierte Frontend-Tests wird in Abschnitt 9.7.2 beschrieben.

9.6. Refactorings

Ein wichtiger Bestandteil der Qualitätssicherung im Projekt *Hydra* waren Refactorings. Dabei war das Ziel, die Verständlichkeit, Wartbarkeit und Erweiterbarkeit des Codes zu erhöhen und technische Schulden zu vermeiden.

Refactorings kamen insbesondere in folgenden Bereichen zur Anwendung:

- **Strukturverbesserungen:** Aufbrechen von zu grossen Klassen oder Methoden, um die Lesbarkeit zu erhöhen und Verantwortlichkeiten klarer zu trennen.
- **Benennung und Konsistenz:** Vereinheitlichung und Verbesserung der Namensgebung, damit der Code für alle Teammitglieder leichter zu verstehen ist.
- **Vereinfachung von Logik:** Entfernen von Wiederholungen oder unnötig komplexem Code.
- **Technische Anpassungen:** Einführung von Interfaces oder Abstraktionen an geeigneten Stellen, um Abhängigkeiten zu reduzieren und spätere Erweiterungen zu erleichtern.

Refactorings wurden stets im Rahmen von Merge Requests durchgeführt und von mindestens einem Teammitglied beim Review geprüft. Durch den CI/CD-Prozess und die darin vorhandene Testautomatisierung konnte sichergestellt werden, dass Refactorings die Funktionalität der Anwendung nicht unbeabsichtigt beeinträchtigen. So konnte die Codebasis über den gesamten Projektverlauf stabil, verständlich und wartbar gehalten werden.

9.7. Mögliche Erweiterungen

9.7.1. Mutation-Tests

Eine sinnvolle Ergänzung zur Qualitätssicherung wären Mutation-Tests. Während klassische Code-Coverage lediglich misst, welche Teile des Codes beim Testen ausgeführt werden, prüfen Mutation-Tests die Aussagekraft der Tests selbst. Dabei werden gezielt kleine Änderungen am Bytecode vorgenommen, z. B. das Vertauschen von Vergleichsoperatoren oder das Entfernen von Anweisungen. Anschliessend wird überprüft, ob die bestehenden Tests diese Änderungen entdecken. Schlagen die Tests fehl, ist dies ein Hinweis auf gute Testfälle. Bleiben die Tests jedoch erfolgreich, deutet dies auf Lücken in den Testfällen hin.

Im Projekt *Hydra* wurde aus zeitlichen Gründen auf den Einsatz von Mutation-Tests verzichtet. Für eine Weiterentwicklung könnte deren Einführung jedoch wertvolle Erkenntnisse zur Qualität der vorhandenen Tests liefern und die Robustheit des Systems weiter verbessern.

Ein geeignetes Tool hierfür wäre *PITest*, welches als Gradle-Plugin direkt in der Build-Konfiguration eingebunden werden kann.

9.7.2. Automatisierte Frontend-Tests

Während der Fokus der automatisierten Tests im Projekt *Hydra* auf der Backend-Logik lag, wurden für das Frontend keine automatisierten UI-Tests implementiert (siehe Abschnitt 9.5).

Als zukünftige Erweiterung bietet sich daher der Einsatz von automatisierten End-to-End-Tests an, um die Funktionalität der grafischen Oberfläche und deren Interaktion mit dem Backend zu überprüfen. Hierfür würde sich *Selenium* anbieten. Dabei werden die Benutzereingaben simuliert und das Resultat mit einem erwarteten Ergebnis verglichen, um das Verhalten der Anwendung im Browser testen zu können. Damit liesse sich die Stabilität des Frontends insbesondere bei Refactorings oder Änderungen der UI-Logik nachhaltig absichern.

Die beschriebenen Qualitätsmassnahmen schaffen eine belastbare Grundlage für die Weiterentwicklung von *Hydra*. Verbindliche Coding-Guidelines, konsequente Merge-Requests, ein automatisierter Build-Prozess und integrierte Unit- sowie Integration-Tests sorgen für nachvollziehbare Änderungen, reproduzierbare Builds und eine stabile Codebasis. Durch regelmässige Refactorings bleibt die Architektur erweiterbar und technische Schulden werden früh adressiert. Insgesamt ist damit eine robuste Basis gelegt, auf der neue Funktionen, Services und Schnittstellen mit überschaubarem Risiko implementiert, getestet und ausgerollt werden können. Optional ergänzende Verfahren (z. B. Mutation- oder End-to-End-Tests) können diese Grundlage künftig weiter festigen, ohne die bestehenden Massnahmen zu verändern.

10. Infrastruktur

Zur effektiven Zusammenarbeit im Team wurde eine passende Infrastruktur aufgebaut. Diese umfasst unter anderem Sourcecodeverwaltung, Build- und Laufzeitumgebungen, Projektmanagementtools und Registries zur Speicherung der Buildartefakte. Die dafür verwendeten Software- und Hardware-Komponenten werden in diesem Kapitel beschrieben.

Im Hinblick auf eine mögliche spätere Kommerzialisierung der Applikation wurde zudem eine belastbare Grundlage für die Automatisierung von Integration und Deployment geschaffen.

Tabelle 10.1 bietet eine Übersicht über die eingesetzte Infrastruktur.

Komponente	**Art**	**Verwendung**
Sourcecode-Verwaltung	GitLab (Cloud)	Verwaltung des Sourcecodes
Container-Registry GitLab	GitLab (Cloud)	Speicherung der erstellten Docker-Container-Images
Container-Registry Dockerhub	Dockerhub (Cloud)	Bezug der öffentlichen Docker-Container-Images für die Integration
Package-Registry	GitLab (Cloud)	Speicherung der Buildartefakte
Buildpipelines	GitLab (Cloud)	Konfiguration der Build-Pipeline zur Integration und anschliessendem Deployment der Applikation, Infrastruktur und Tools.
Buildserver	Server (On-Prem)	Ausführung der in GitLab konfigurierten Pipeline mittels Runner
Webserver (IaaS)	Server (Cloud)	Public Cloud und Webserver, über den die Applikation via Pipeline bereitgestellt wird.
Issue- und Projektmanagement	YouTrack (Cloud)	Verwaltung der Issues während der Entwicklung und für das Projektmanagement.

Tabelle 10.1.: Verwendete Infrastruktur-Komponenten

10.1. Sourcecodeverwaltung

Für die Verwaltung des Sourcecodes wurde *Git* gegenüber *SVN* gewählt. Ausschlaggebend waren die Eigenschaften eines verteilten Versionsverwaltungssystems, der ausgereifte

Branch-/Merge-Workflow, die weitere Verbreitung von Git sowie die im Lehrgang bereits erworbenen Kenntnisse.

Für eine zentrale, allen Teammitgliedern zugängliche Sourcecodeverwaltung wurde entschieden, einen Clouddienst zu verwenden. Aufgrund des umfangreicheren Angebots im Vergleich mit Konkurrenzprodukten und der bereits vorhandenen Erfahrung des Teams mit dieser Plattform wurde *GitLab* gewählt.

10.1.1. Repositories

Im Rahmen der Masterarbeit wurden mehrere Repositories in einer gemeinsamen Gruppe zusammengefasst. Dies ermöglicht eine zentrale und einfachere Konfiguration der Zugriffsrechte, sowohl für Teammitglieder als auch für die Build-Pipeline. Alle erstellten Repositories wurden als *privat* angelegt, sodass sie nicht von extern eingesehen werden können.

Die folgenden, in der Gruppe *MAS-SE23-Masterarbeit* erstellten Repositories sind für die Masterarbeit relevant:

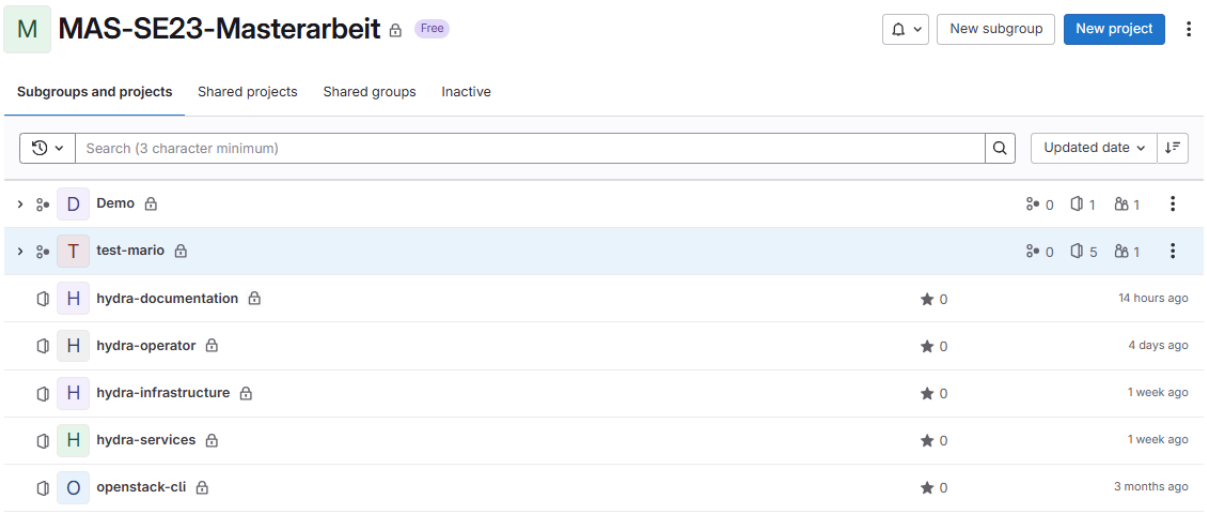


Abbildung 10.1.: Code-Repositories in der Masterarbeit-Gruppe

- **hydra-services:** Enthält den Sourcecode und die Pipeline für die Backend- und Application-Services der Applikation in Java unter Verwendung von Spring.
- **hydra-operator:** Sourcecode für Operator-UI (und Field-UI) in TypeScript auf Basis von React.
- **hydra-documentation:** Enthält die im Rahmen der Masterarbeit erstellte Dokumentation, welche als Sourcecode gepflegt wird, z. B. die vorliegende Arbeit in LaTeX.
- **hydra-infrastructure:** Sourcecode für den Betrieb der Infrastruktur sowie die zugehörige Pipeline, mit der die Applikation zur Verfügung gestellt wird.
- **OpenStack CLI:** Sourcecode und Pipeline für den Docker-Container, welcher für das Deployment der Applikation verwendet wird.

10.2. Container-Registry

Die für das Deployment erforderlichen Docker Container-Images werden in einer *GitLab*-Container-Registry abgelegt. Da die Registry auf Gruppenebene existiert, sind sämtliche Images aus den unterschiedlichen Repositories zentral ersichtlich.

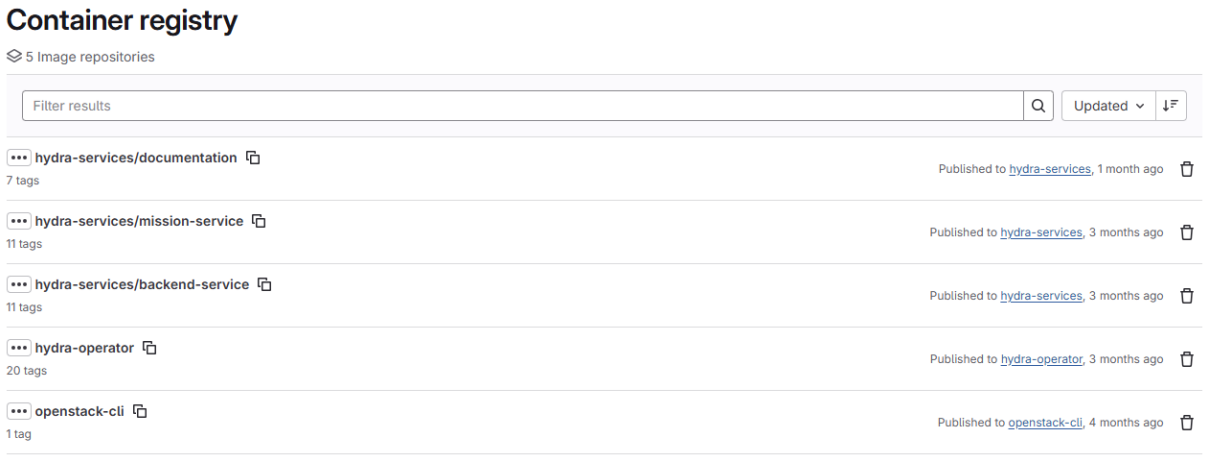


Abbildung 10.2.: GitLab Container-Registry

Die für die Integration benötigten Docker Container-Images werden von der *Docker Hub*-Container-Registry bezogen (Dockerhub, 2025). Diese werden vom *Runner* (siehe Abschnitt 10.5) heruntergeladen und beinhalten die für den Buildprozess notwendigen Programme (siehe Kapitel 11).

10.3. Package-Registry

Einige der in den automatischen Pipelines erstellten Buildartefakte werden von anderen Pipelines verwendet. Dies betrifft z. B. das TypeScript-Package mit den Backend-API-Typen, welches vom Frontend für die Verwendung des *Backend-APIs* benötigt wird. Dieses Package wird als Buildartefakt innerhalb des Repositories *hydra-services* erstellt und im Repository *hydra-operator* verwendet.

Abbildung 10.3.: GitLab Package Registry

Durch die Verwendung der *GitLab*-Package-Registry, welche sich in der selben Gruppenebene befindet, können sämtliche Repositories und Pipelines zentral auf diese Packages zugreifen.

10.4. Buildpipeline

Im Hinblick auf eine mögliche spätere Kommerzialisierung der Applikation war eine automatisierte Integration und ein (teil-)automatisiertes Deployment erforderlich. Ein wesentlicher Vorteil dieser Automatisierung besteht darin, dass die Abläufe von allen Teammitgliedern, auch ohne spezielle DevOps-Kenntnisse, durchgeführt werden können und reproduzierbar sind. Zudem können die benötigten Geheimnisse *(Secrets)* wie Passwörter und Tokens zentral und sicher abgelegt werden, wodurch das Risiko eines falschen Umgangs mit sensitiven Zugangsdaten deutlich reduziert wird. Für jedes Repository wurde daher eine eigene *GitLab-Pipeline* eingerichtet.

10.5. Buildserver

Die beschriebenen Pipelines werden jeweils von einem *Runner* ausgeführt. Aus Sicherheitsgründen und aufgrund der eingeschränkten Verfügbarkeit wurde auf die Nutzung der von *GitLab* bereitgestellten *Instance Runner* verzichtet. Stattdessen wurde ein eigener Runner für die Abarbeitung der Jobs konfiguriert.

Der Buildserver prüft kontinuierlich, ob neue Build-Jobs vorhanden sind und führt diese aus. Dafür wurde ein Linux-Server mit 64-Bit-Betriebssystem und 64 GB Arbeitsspeicher verwendet.

Executor

Bei der Konfiguration eines Runners muss einer von mehreren möglichen sogenannten Executors gewählt werden. Die Art des Executors bestimmt, wie die Build-Jobs ausgeführt werden. Dies bringt jeweils unterschiedliche Vor- und Nachteile mit sich. Im Rahmen der Masterarbeit kamen auf dem Buildserver zwei verschiedene *Executors* zum Einsatz. Für Integration und Deployment der Applikation wurde bevorzugt der *Docker Executor* verwendet.

- **Shell Executor:** Die einfachste Konfiguration eines *Runners*. Build-Jobs werden lokal auf dem Server ausgeführt, weshalb alle benötigten Programme und Abhängigkeiten ebenfalls lokal installiert sein müssen. Entsprechend bietet dies eine limitierte Isolation zwischen einzelnen Build-Jobs.
- **Docker Executor:** Aufgrund der Verwendung von Docker-Containern für die Ausführung werden die einzelnen Build-Jobs isoliert und konsistent ausgeführt. Ein weiterer Vorteil besteht darin, dass Services wie MongoDB, Kafka etc. ohne grossen Aufwand eingebunden werden können.

Grundsätzlich bietet der *Docker Executor* die meisten Vorteile und sollte daher bevorzugt werden. Sollen jedoch *Docker*-bezogene Befehle (z. B. das Starten eines Docker-Containers)

auf einem Runner mit Docker Executor ausgeführt werden, spricht man von *Docker-in-Docker*. Dies ist nur möglich, wenn die Privilegien des Docker Executors sowie der entsprechenden Build-Container erhöht werden. Die Erhöhung der Privilegien hebt allerdings den Sicherheitsvorteil bei der Verwendung von Containern auf.

Um eine konsistente, reproduzierbare und sichere Buildumgebung zu schaffen, wurde der Docker Executor für Build-Jobs mit vielen Abhängigkeiten eingesetzt. Für alle *Docker*-bezogenen Build-Jobs wurde hingegen der *Shell Executor* verwendet. Dieser ist lediglich abhängig von einer Docker-Runtime, die ohnehin für den Docker Executor bereits vorhanden ist.

Die Kombination beider Executor-Arten ermöglichte den Aufbau einer sicheren, reproduzierbaren und gut betreibbaren Integrations- und Bereitstellungsumgebung.

10.6. Webserver und IaaS

Für die Bereitstellung der Applikation wurde ein schlanker Webserver eingesetzt, über den die Anwendung für Tests und Demos erreichbar gemacht wurde.

Im Rahmen der Masterarbeit erfolgte kein produktives Deployment. Stattdessen wurde eine Testumgebung (*Test*) für Demonstrationen und Erprobungen durch die Feuerwehr eingerichtet. Auf eine produktive Umgebung (*Prod*) wurde bewusst verzichtet. Die bestehende Konfiguration kann bei Bedarf mit geringem Aufwand um weitere Umgebungen wie *Staging* und *Prod* erweitert werden.

Die Testumgebung wurde in einer *Public Cloud* gehostet. Dabei wurde *Infrastruktur as a Service (IaaS)* auf *OpenStack*-Basis verwendet (openstack, 2025). Dieser Ansatz reduziert Anbieterabhängigkeiten (Vendor Lock-in), denn abhängig von den genutzten Diensten der verschiedenen Cloudanbieter (AWS, Azure, Hetzner, Google Cloud etc.) sind diese nicht ohne Weiteres austauschbar. In Kombination mit einem einfach gehaltenen Webserver ergibt sich eine flexible, übertragbare Lösung, die sich an unterschiedliche Provider anpassen lässt, auch wenn der initiale Konfigurationsaufwand höher ist.

Für den Betrieb der Testumgebung kamen folgende zentrale *OpenStack*-Services zum Einsatz:

- **Compute:** Serverinstanz mit 4 GB RAM, 2 vCPUs, Linux Server LTS
- **Volume:** 20 GB Ceph Volume als Root Disk für den Webserver
- **Network:** Internes IPv4-Netz sowie eine fixe externe IPv4-Adresse für den externen Zugriff auf den Webserver
- **DNS:** DNS-Zone für den externen Zugriff sowie für die zugehörige TLS-Konfiguration

Konfiguration

Die Konfiguration der *OpenStack*-Services wurde bewusst pragmatisch gehalten. Auf eine Umsetzung als *Infrastructure as Code (IaC)* wurde verzichtet, da der zusätzliche Aufwand in keinem Verhältnis zum Nutzen stand und das endgültige produktive Deployment-Modell noch nicht festgelegt war. Für die Testumgebung genügte ein einzelner Webserver, sodass sich durch eine IaC-Lösung kein relevanter Zeitgewinn ergeben hätte. Die Einrichtung der

Infrastruktur erfolgte daher hauptsächlich über die Weboberfläche *Horizon*.

OpenStack CLI

Während die grundlegende Konfiguration der Infrastruktur über die Weboberfläche *Horizon* erfolgte, wurde für wiederkehrende und automatisierte Aufgaben das *OpenStack CLI* eingesetzt. Damit liessen sich insbesondere Deployment-Schritte direkt aus der Pipeline heraus ausführen, ohne dass manuelle Eingriffe notwendig waren.

Zu diesem Zweck wurde ein eigenes Docker-Image erstellt, welches das CLI zusammen mit einer minimalen *Python*-Laufzeitumgebung und weiteren Hilfsprogrammen (z. B. *openssl*, *openssh*, *curl*, *bash*) enthält. Das Image wird über eine GitLab-Pipeline automatisch gebaut, getestet und in der GitLab-Registry veröffentlicht. Es kann dann in der Deployment-Pipeline für die Interaktion mit *OpenStack* verwendet werden. Die dafür benötigten Definitionen (*Dockerfile* und *Pipeline*) sind im Repository *OpenStack CLI* abgelegt.

Das Repository umfasst im Wesentlichen die folgenden Komponenten:

- **.gitlab-ci:** Build-Pipeline für Build und Release des *OpenStack CLI* Container-Images sowie dessen Integration in die GitLab Container-Registry. Diese wird in Abschnitt 11.1.2 detaillierter beschrieben.
- **Dockerfile:** Definition des Container-Images, basierend auf einem *Alpine*-Image. Zudem beinhaltet die Build-Datei weitere Programme wie *openssl*, *openssh*, *curl*, *bash* etc.
- **test.sh:** Bash-Skript für den Test des Containers innerhalb der Pipeline.

10.7. Issue- und Projektmanagement

Für das Issue- und Projektmanagement kam eine Cloud-Instanz von *YouTrack* zum Einsatz (YouTrack, 2025). So konnten entdeckte Bugs, neue Ideen oder Anforderungen unmittelbar als für alle Teammitglieder sichtbare Issues erfasst werden. Eine kontinuierliche Priorisierung, die Verwaltung der Issues sowie deren Zuweisung an Teammitglieder liessen sich damit problemlos umsetzen. Zudem war auch die Zeiterfassung pro Teammitglied und Issue einfach möglich.

Zur Verknüpfung von Entwicklung und Planung wurde die verwendete *GitLab*-Instanz mit *YouTrack* integriert. Dadurch erschienen Commits und Merge Requests direkt beim jeweiligen Issue, was den Wechsel zwischen Aufgaben und den zugehörigen Änderungen wesentlich erleichterte.

Die beschriebene Infrastruktur stellt die technische Grundlage für Entwicklung, Integration und Betrieb von *Hydra* bereit. Quellcodeverwaltung, Artefakt- und Container-Registries sowie automatisierte Buildpipelines werden in einem Projekt auf GitLab gebündelt. Für die Bereitstellung einer Testapplikation wurde eine Cloud-basierte Laufzeitumgebung auf OpenStack-Basis eingerichtet. Die Anbindung von YouTrack an GitLab schafft zudem durchgängige Nachvollziehbarkeit von Anforderungen bis zu den Änderungen im Code.

11. Continuous Integration und Deployment

Ohne automatisierte Continuous-Integration- und -Deployment-Umgebungen (*CI/CD*) ist der professionelle Betrieb von Softwareprodukten heute kaum mehr möglich. Dies gilt vor allem für verteilte Systeme mit vielen Komponenten.
Für das System *Hydra* bietet eine solche Umgebung unter anderem folgende Vorteile:

- **Konsistenz und Nachvollziehbarkeit:** Bei korrekt konfiguriertem Build-Server (Runner) laufen Builds automatisiert und reproduzierbar ab. Siehe dazu die Beschreibung des *Docker-Executors* unter Abschnitt 10.5.
- **Sicherheit:** Passwörter und Tokens für Zugriffe auf Umgebungen und Registries werden zentral und sicher verwaltet und in der Pipeline referenziert. Zu schützende Werte werden als *GitLab CI/CD Variables* hinterlegt. So landen keine sensitiven Geheimnisse in Skripten oder im Quellcode.
- **Geschwindigkeit und Qualität:** Eine automatisierte CI/CD-Umgebung erhöht die Geschwindigkeit, mit der neue *Features* entwickelt, getestet und ausgerollt werden können und kann gleichzeitig eine hohe Softwarequalität sicherstellen.

Grundlage für ein zuverlässiges Deployment der Applikation ist eine korrekt funktionierende, weitgehend automatisierte Integrationspipeline (*CI*). Sie testet, baut den Code und erzeugt deploybare Artefakte (z. B. Docker-Images). Darauf aufbauend verteilt die Deployment-Pipeline (*CD*) diese Artefakte in die Zielumgebungen. Beide Pipelines können sowohl automatische als auch manuelle Jobs beinhalten. Die Ausgestaltung dieser Jobs hängt oftmals von der Zielumgebung ab.

In den folgenden Repositories wurden Pipelines implementiert:

- **openstack-cli:** Integrations- und Deployment-Pipeline für den *OpenStack*-Container, welcher für das Deployment im Repository *hydra-infrastructure* verwendet wird (siehe Abschnitt 10.6).
- **hydra-operator:** Integrations-Pipeline für das Frontend sowie den Build des Docker-Images für das Deployment.
- **hydra-services:** Integrationspipeline für die Backend-Services, inklusive Build der Docker-Images für das Deployment sowie der Artefakte, die im Frontend-Service verwendet werden.
- **hydra-infrastructure:** Deployment-Pipeline für die Infrastruktur und die Applikation selbst.

11.1. Continuous Integration

Für die Umsetzung der Continuous Integration wurden verschiedene von GitLab bereitgestellten Services eingesetzt: GitLab CI, GitLab Runner (Docker-Executor), die GitLab Artifact Registry sowie die GitLab Container Registry. In Kombination mit den GitLab-Pipelines konnte so eine einfache und konsistente Implementierung der Integration erreicht werden (siehe auch Kapitel 10 Infrastruktur).

Abbildung 11.1 zeigt die Interaktion zwischen den verwendeten Gitlab-Services, den Pipeline-Jobs sowie den erzeugten Artefakten.

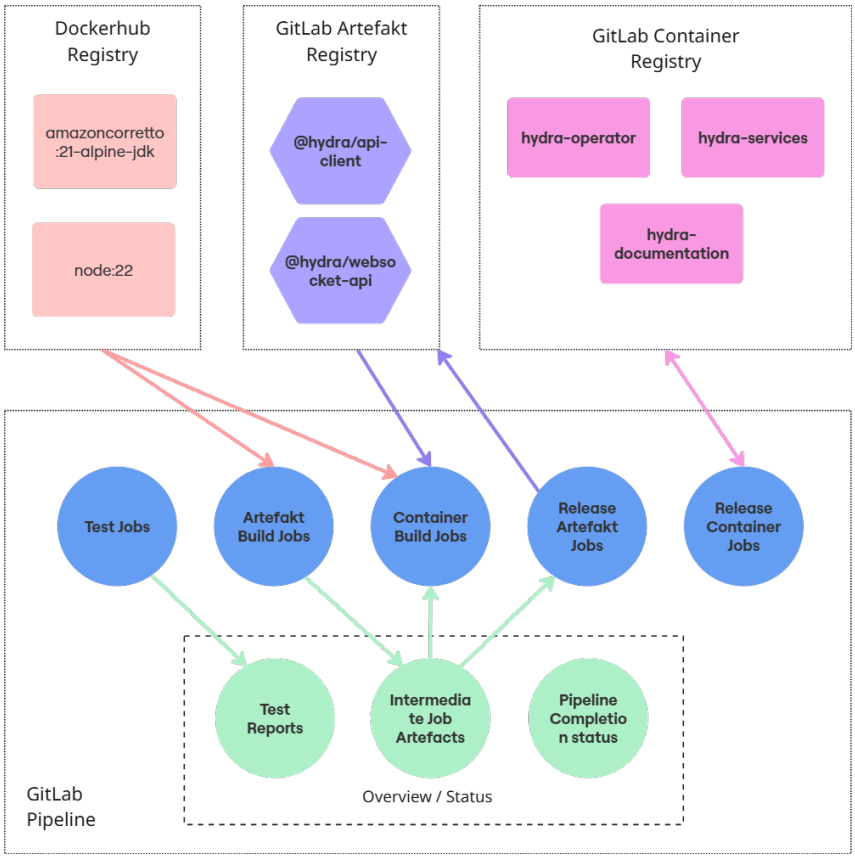


Abbildung 11.1.: Interaktion von CI-Pipeline und GitLab-Services

Die in Abbildung 11.1 dargestellte Aufteilung der Pipeline in einzelne, voneinander unabhängige Jobs ermöglicht eine effiziente Verarbeitung, da mehrere Jobs parallel auf unterschiedlichen Runnern ausgeführt werden können.

In der Pipeline-Ansicht (und in Merge-Requests) zeigt GitLab neben dem Gesamtstatus auch die Detailausgaben und Berichte einzelner Jobs. Testberichte können einfach über `artifacts:reports:junit` integriert werden, wie das untenstehende Code-Snippet zeigt.

```
1  artifacts:
2    reports:
3      junit:
4      - domain-service/mission/build/test-results/test/*.xml
5    expire_in: 1 week
```

Die erstellten Testberichte werden dann nach Beendigung der Test-Jobs als Build-Artefakte hochgeladen und erscheinen anschliessend in der Pipeline-Übersicht. In Abbildung 11.2 ist ein solcher Testreport ersichtlich.

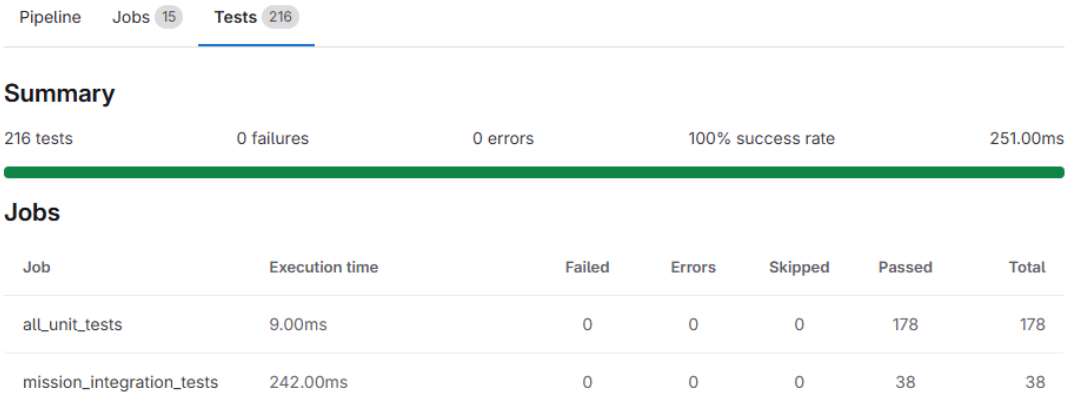


Abbildung 11.2.: CI-Pipeline Test-Report

11.1.1. Versionierung

Für die Versionierung der Artefakte und Container-Images werden die ersten acht Zeichen des Commit-Hashes verwendet. Dieser wird in der Pipeline automatisch ausgelesen und als Versionsnummer genutzt.

```
variables:
  VERSION: $CI_COMMIT_SHORT_SHA
...
-t $CI_REGISTRY_IMAGE/backend-service:$VERSION
...
- SNAPSHOT_VERSION=0.0.0-snapshot.$CI_COMMIT_SHORT_SHA
- npm version $SNAPSHOT_VERSION --no-git-tag-version
```

Der Vorteil dieser Vorgehensweise besteht darin, dass erstellte Artefakte und Container immer eindeutig einem Commit zugeordnet werden können. So ist ausgeschlossen, dass zwei verschiedene Artefakte dieselbe Version besitzen. Die Versionsreferenz entspricht jeweils der in GitLab angezeigten Commit-ID, was die Identifizierung zusätzlich erleichtert.

Eine Versionierung im klassischen Sinne, beispielsweise nach dem gängigen *Major/Minor/Patch*-Schema nach *Semantic Versioning* (SemVer, 2025) war während der Masterarbeit nicht erforderlich, da der Fokus auf einem lauffähigen MVP lag. Ein vollständiges Release-Pattern könnte jedoch in einer späteren Phase im Rahmen einer allfälligen Release-Planung implementiert werden.

11.1.2. Pipelines

Zur Optimierung des Entwicklungsprozesses unterscheiden wir zwischen Merge-Request-Pipelines und Branch-Pipelines. Mittels `workflow:rules`-Einstellungen wird sichergestellt, dass pro Push genau *eine* Pipeline läuft: Entweder die Merge-Request-Pipeline (bei

einem offenen Merge-Request) oder, falls kein MR existiert, die Branch-Pipeline. So wird vermieden, dass ein Push auf einen Feature-Branch parallel zur MR-Pipeline zusätzlich eine Branch-Pipeline startet. Die dafür nötige Konfiguration wird im untenstehenden Codesnippet verdeutlicht.

```
workflow:
  rules:
    #Run merge request pipelines
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    # Prevent branch pipelines when MR is open
    - if: $CI_COMMIT_BRANCH && $CI_OPEN_MERGE_REQUESTS
      when: never
    # Run branch pipelines when no MR is open
    - if: $CI_COMMIT_BRANCH
```

Darüber hinaus wurden einige der Build-Jobs als *manuell* deklariert. Diese werden nicht automatisch ausgeführt, sondern müssen in GitLab manuell gestartet werden. Dadurch laufen nicht bei jedem Push alle rechenintensiven Schritte, sondern nur diejenigen, die tatsächlich benötigt werden. Dies reduziert die Laufzeit der Pipeline und verbessert so das Feedback an die Entwickler.

Nachfolgend werden die implementierten CI-Pipelines kurz beschrieben.

hydra-services

Die Pipeline im Repository *hydra-services* ist die umfangreichste. Darin werden neben den Services auch die TypeScript-Bibliotheken für das Frontend (*hydra-operator*) erstellt und in die Artefakt-Registry hochgeladen. Die Stages `build_container`, `release_npm_latest` und `release_container` sind als manuelle Jobs konfiguriert und werden daher nicht automatisch ausgeführt, können aber bei Bedarf gestartet werden.

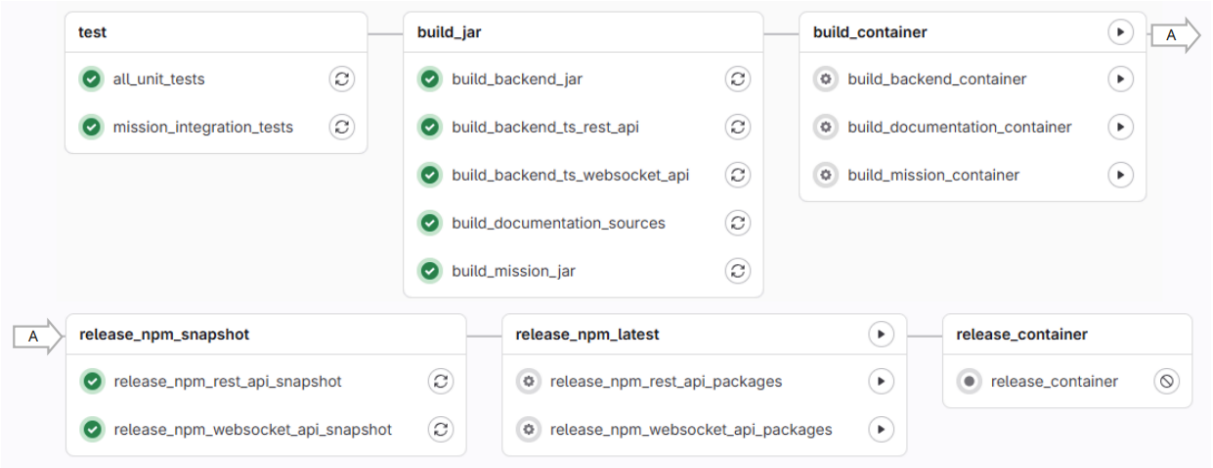


Abbildung 11.3.: Pipeline *hydra-services*

hydra-operator

Die Pipeline im Repository *hydra-operator* ist weniger komplex und umfasst weniger Stages. Sie ist abhängig von den im Repository *hydra-services* produzierten Artefakten und muss bei gleichzeitigen Anpassungen an beiden Repositories zuletzt gestartet werden. Auf die Konfiguration einer konditionalen Ausführung wurde aufgrund des knappen Zeitbudgets verzichtet.

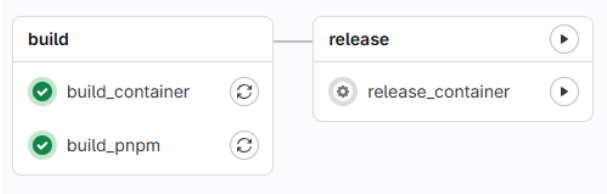


Abbildung 11.4.: Pipeline *hydra-operator*

openstack-cli

Die Pipeline im Repository *openstack-cli* ist wie bereits im Abschnitt 10.6 beschrieben für das Erstellen eines *OpenStack-CLI*-Docker-Images für den Deploymentprozess zuständig. Im `build`-Job wird das Docker Container Image erstellt. Anschliessend wird der Build im `test`-Job getestet und danach in einem automatischen `release`-Job veröffentlicht.

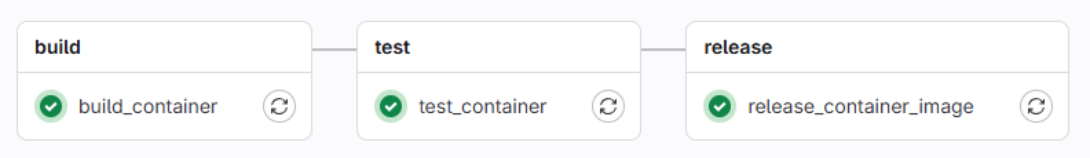


Abbildung 11.5.: Pipeline *openstack-cli*

11.1.3. Umgang mit Secrets

Ein zentraler Aspekt bei der Verwendung automatisierter Prozesse ist der sichere Umgang mit sogenannten Secrets, also Passwörtern und Tokens. Diese dürfen keinesfalls im Sourcecode hinterlegt werden, da dadurch alle Personen mit Zugriff auf das Repository unkontrollierten Zugang erhalten würden. Darüber hinaus ist es nahezu unmöglich, einmal eingetragene Secrets nachträglich aus der Versionshistorie zu entfernen.

Im Projekt *Hydra* kommen folgende Arten von Secrets zum Einsatz:

- **GitLab-Service-Tokens:** Vordefinierte Tokens für den Zugriff auf GitLab-Services aus den Pipelines in den verschiedenen Repositories, z. B. für den Zugriff auf die Package- oder Container-Registry.
- **Zugangsdaten für externe Systeme:** API-Keys, Tokens und Cloud-Zugangsdaten (z. B. OpenStack) sowie private ssh-Keys, welche in Skripten benötigt werden.
- **Sensitive Variablen:** Nicht geheime, aber sensitive Variablen wie DNS-Namen, IP-Adressen oder Account-Namen.

Innerhalb der Pipelines werden die benötigten Secrets als CI/CD-Variablen genutzt. Für GitLab-Services werden vordefinierte Variablen bereitgestellt. Diese sind in der GitLab-Dokumentation detailliert beschrieben. Eigene Variablen können in GitLab unter den CI/CD-Einstellungen definiert werden. Abbildung 11.6 zeigt exemplarisch einige der für *hydra-infrastructure* definierten Secret-Variablen.

Abbildung 11.6.: Auszug aus den CI/CD-Variablen für *hydra-infrastructure*

Die so konfigurierten Variablen werden in den Pipeline-Skripten wie im untenstehenden Codesnippet gezeigt direkt verwendet.

```
1  - cat $OS_AUTH > traefik/os_auth.sh
2  -v $ID_RSA:/.ssh/id_rsa
3  - ... $CI_PROJECT_DIR/traefik/* $SERVER_USER@$SERVER_IP:
       $TEMP_SERVER_PATH
```

11.2. Continuous Deployment

Dieses Kapitel beschreibt das Continuous Deployment (CD) von *Hydra*. Alle deployten Komponenten werden als Docker-Container bereitgestellt. Deren Konfiguration wird zentral im Repository *hydra-infrastructure* verwaltet. Das CD setzt sich aus drei Hauptteilen zusammen:

- **Applikation:** Ausrollen des *Hydra*-Applikationsstacks.
- **Reverse-Proxy:** Ausrollen von Traefik für Updates in der Reverse-Proxy Konfiguration.
- **Infrastruktur:** Ausrollen von zusätzlicher, unterstützender Infrastruktur mittels einer vordefinierten Konfiguration.

Die folgenden Unterkapitel führen diese Aspekte detaillierter aus.

11.2.1. hydra-Applikation

Das Deployment des *Hydra*-Applikationsstacks erfolgt auf der in Abschnitt 10.6 beschriebenen *Public-Cloud*-Infrastruktur. Dies geschieht bei Bedarf automatisiert über

eine GitLab-Pipeline im Repository *hydra-infrastructure*. Alle Services werden dabei als Docker-Container veröffentlicht und gestartet.

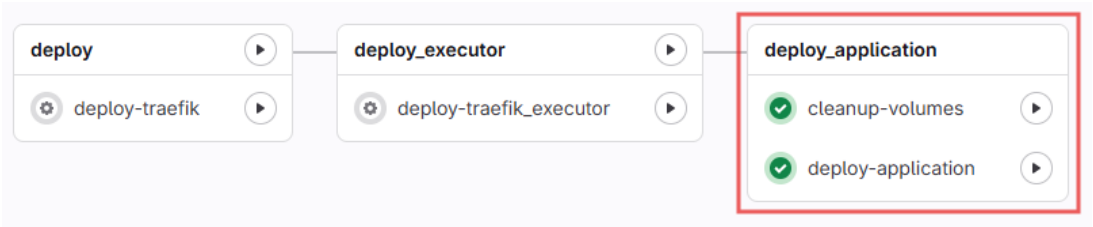


Abbildung 11.7.: CD-Pipeline Applikation

Die Deployment-Gruppe `deploy_application` wird verwendet, um den *Hydra*-Applikationsstack zu deployen. Darin stehen zwei Jobs zur Verfügung, welche beide manuell gestartet werden können:

- **cleanup-volumes:** Entfernt vorhandene Daten (Docker-Volumes) der aktuell deployten Applikation. Dieser Job kann bei Bedarf manuell vor dem eigentlichen Deployment ausgeführt werden, falls die vorhandenen Daten bereinigt werden sollen.
- **deploy-application:** Führt das Deployment der Applikation gemäss den im Repository definierten Versionsvariablen durch. Bestehende Applikationsdaten bleiben erhalten.

Der *Hydra*-Applikationsstack, welcher von dieser Deploymentpipeline veröffentlicht wird, setzt sich aus den folgenden Services zusammen:

Service	**Beschreibung**	**URL**
Frontend	Benutzeroberfläche für Operator sowie Einsatzkräfte im Feld.	[redacted]
Backend	Backend-for-Frontend mit APIs für die Frontends.	[redacted]
Mission	Mission-Service, welcher die Businesslogik enthält und vom BFF verwendet wird.	keine externe Erreichbarkeit
Dokumentation	Interaktive Dokumentation aller Schnittstellen.	[redacted]

Tabelle 11.1.: Services des Hydra-Applikationsstacks

Die zu deployende Version kann einfach über eine Umgebungsvariable definiert werden, wie nachfolgend gezeigt wird. Hierzu wird die Datei `application/env.sh` im Repository angepasst. Gültige Referenzen sind entweder `latest` oder der kurze Commit-Hash (z. B. `6f19e449`), mit dem der Container released wurde (siehe Abschnitt 11.1.1).

```
export FRONTEND_VERSION=latest
export BACKEND_VERSION=latest
export MISSION_VERSION=latest
export DOCUMENTATION_VERSION=latest
```

11.2.2. Traefik Reverse-Proxy

Der Zugriff auf die bereitgestellte Applikation sowie die Infrastruktur-Services erfolgt über den Reverse-Proxy *Traefik*. Traefik terminiert TLS und übernimmt die Verwaltung der Zertifikate. Somit entfällt die TLS-Implementierung in den einzelnen Services. Sämtliche extern erreichbaren Endpunkte werden über Traefik bereitgestellt und sind mit kostenlosen Let's-Encrypt-Zertifikaten abgesichert.

Die Konfiguration von Routing (Hostnames/Endpunkte) und Zugriffsberechtigungen ist, wie nachfolgend gezeigt, mittels *Traefik-Labels* direkt in den für das Deployment verwendeten Docker-Compose-Dateien der jeweiligen Services hinterlegt. Dies ermöglicht eine konsistente, servicegebundene Konfiguration für ein automatisiertes Deployment.

```
1  "traefik.http.routers.frontend.rule=
2    Host(`app.hydra-xxx`)"
3  "traefik.http.routers.frontend.middlewares=
4    appAuth@file"
```

Ein neues Deployment von Traefik ist nur erforderlich, wenn Änderungen an einer der folgenden Konfigurationen vorgenommen wurden:

- Routing- oder Protokolländerungen bestehender Services
- Anpassungen an den Zugriffsberechtigungen bestehender Services
- Änderungen an der Traefik-Konfiguration

Für das Deployment stehen die in Abbildung 11.8 rot markierten Pipeline-Jobs zur Verfügung. Der Job `deploy-traefik` wird auf einem Shell-Runner ausgeführt, während `deploy-traefik_executor` auf einem Docker-Executor-Runner läuft. Bei Bedarf kann der jeweils benötigte Job manuell ausgeführt werden.

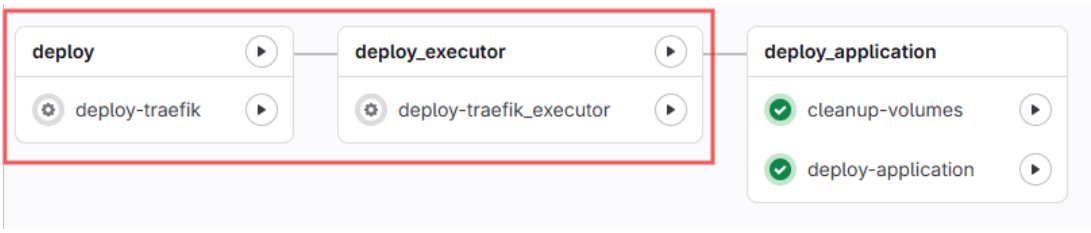


Abbildung 11.8.: CD-Pipeline Traefik

11.2.3. Infrastruktur

Neben der Applikation und dem Reverse Proxy werden für den Betrieb von *Hydra* noch zusätzliche Infrastrukturservices benötigt. Dies sind beispielsweise der Kafka-Broker, MongoDB oder Services für Logging und Monitoring mit Loki und Grafana.

Für jede zusammengehörige Service-Gruppe existiert jeweils eine eigene Docker-Compose-Datei (z. B. `application/docker-compose.kafka.yaml`). Diese konfiguriert den produktiven Dienst, zusätzlich benötigte unterstützende Dienste und, falls vorhanden, das zugehörige Administrationstool.

Diese unterstützenden Services erfahren eher selten Updates. Diese geschehen deshalb manuell, indem die in der Docker-Compose-Datei definierte Konfiguration manuell angepasst wird. Das Deployment erfolgt dann in der *hydra-infrastructure*-Pipeline zusammen mit den Applikationsservices. Die folgenden Services werden auf diese Weise ausgerollt:

Service	**Beschreibung**	**URL**
MongoDB	Datenbank-Service.	keine externe Erreichbarkeit
Mongo-Express	Weboberfläche zur Verwaltung von MongoDB.	[redacted]
Kafka	Message-Broker.	keine externe Erreichbarkeit
Kafka Schema Registry	Verwaltung der Kafka-Schemas.	keine externe Erreichbarkeit
Kafka KSQL	KSQL-Server für spätere Streaming-Anwendungen.	keine externe Erreichbarkeit
Kafbat-UI	Weboberfläche zur Verwaltung von Kafka.	[redacted]
Loki	Server zur Log-Aggregation.	keine externe Erreichbarkeit
Grafana	Weboberfläche zur Analyse und Visualisierung von Logs.	[redacted]

Tabelle 11.2.: Infrastrukturservices

11.3. Release- und Deploymentprozess

Wie vorangehend dokumentiert, läuft ein grosser Teil des Deploymentprozesses automatisiert ab. Um Abweichungen in den manuell ausgeführten Schritten zu vermeiden und Reproduzierbarkeit sicherzustellen, wurde für *Hydra* ein detaillierter Release- und Deployment-Prozess dokumentiert. Auf hoher Ebene umfasst dieser das Erstellen und Versionieren der Artefakte, die Freigabe und das Ausrollen auf der Zielumgebung sowie eine abschliessende Funktions- bzw. Versionsprüfung.

Eine detaillierte Schritt-für-Schritt-Anweisung des Release- und Deploymentprozesses findet sich im Anhang D.

Zusammenfassend ermöglicht die beschriebene CI/CD-Umgebung einen zuverlässigen, reproduzierbaren und sicheren Betrieb von *Hydra*. Automatisierte Builds und Tests, eine eindeutige Versionierung der Artefakte und Container-Images über den Commit-Hash sowie klar definierte Pipelines pro Repository sichern Nachvollziehbarkeit und Qualität. Die Verwaltung von Secrets über CI/CD-Variablen und der zentrale Ingress über Traefik reduzieren operative Risiken. Deployments der Applikation erfolgen automatisiert mit manuellem Auslöser, unterstützende Infrastruktur wird gezielt manuell ausgerollt. Der dokumentierte Release- und Deploymentprozess stellt konsistente Ausrollungen sicher und schafft eine gemeinsame Arbeitsgrundlage. All dies bildet die Basis für eine einfache und sichere Erweiterung von *Hydra*.

12. Fazit

12.1. Benutzer-Feedback

Kurz vor Abschluss der Masterarbeit wurde den Stakeholdern das MVP der Applikation präsentiert. Diese waren mit dem Ergebnis sehr zufrieden und konnten sich bereits vorstellen, *Hydra* unter realen Bedingungen einzusetzen. Obwohl während der Vorführung einige Bugs und Feature-Wünsche aufgenommen wurden, schlug der Kommandant vor, die Applikation bei der nächsten Feuerwehrübung einzusetzen.

Die Rückmeldungen fielen durchwegs sehr positiv aus. Obwohl es sich beim aktuellen Stand „nur“ um ein MVP handelt, zeigte sich bereits ein erheblicher praktischer Nutzen. Die Applikation reduziert den administrativen Aufwand während eines Einsatzes spürbar und trägt zu einer effizienteren Einsatzführung bei. Der Kommandant betonte, dass die *Hydra*-Applikation in ihrem jetzigen Entwicklungsstand bereits produktiv eingesetzt werden könnte. Eine Weiterentwicklung wird ausdrücklich gewünscht und begrüsst. Damit kann der erhoffte Nutzen für die Feuerwehr definitiv nachgewiesen werden.

Die beim Test gefundenen Bugs, Verbesserungsvorschläge und Feature-Wünsche sind in Anhang F aufgelistet.

12.2. Allgemeines Fazit

Mit *Hydra* ist es gelungen, Architektur- und Technologiewahl so zu kombinieren, dass eine verteilte Anwendung entstanden ist, die bereits in der MVP-Version echten Nutzen für die Feuerwehr bringt. Die eingesetzten Technologien haben sich durchgehend bewährt. Eine Anpassung der Kerntechnologien, Frameworks oder der Architektur ist daher nicht erforderlich. Die Applikation adressiert ein konkretes Problem der Feuerwehr bei der Bewältigung von Mehrfachereignissen und hat gezeigt, dass sich der gewählte Ansatz in der Praxis funktioniert.

Auch bei den Tests unter realen Bedingungen wurde bestätigt, dass die gewählten Architektur- und Designentscheidungen tragfähig sind und eine gute Grundlage für weitere Entwicklungen bieten. Dank der eingesetzten Technologien und Tools ist eine Lösung entstanden, die wartbar ist und sich flexibel erweitern lässt.

Damit konnte die Masterarbeit nicht nur die gesteckten Ziele erreichen, sondern auch den Mehrwert einer Software-Lösung für den Einsatzbereich der Feuerwehren klar aufzeigen.

12.3. Fazit der Gruppe

Der abschliessende Praxistest hat gezeigt, dass wir mit *Hydra* eine Applikation entwickelt haben, die selbst als MVP einen gewaltigen Mehrwert für die Feuerwehr bringt. Damit konnten wir unter Beweis stellen, dass unsere Lösung nicht nur konzeptionell, sondern auch praktisch überzeugt.

Wir haben moderne Technologien wie Spring Boot, Docker, GitLab CI/CD-Pipelines und React in einem verteilten System erfolgreich kombiniert. Das war für uns alle technisch anspruchsvoll. Die robuste und flexible Architektur zeigt jedoch, dass wir eine Lösung geschaffen haben, die nicht nur heute funktioniert, sondern sich auch in Zukunft weiterentwickeln lässt und professionell betrieben und genutzt werden kann.

Besonders hervorheben möchten wir die ausgezeichnete Zusammenarbeit im Team. Der enge Austausch hat geholfen, ein gemeinsames Verständnis für die Thematik zu schaffen und organisatorische und technische Fehler zu vermeiden. So konnten wir unsere unterschiedlichen Stärken optimal einbringen und voneinander profitieren.

Trotz des intensiven Arbeitspensums stand die Freude am „Hydrieren“ immer im Vordergrund. So konnten wir die gemeinsame Arbeit stets mit Motivation, Humor und Begeisterung vorantreiben. Dieser Teamgeist war ein entscheidender Faktor für das Gelingen der Masterarbeit.

Einziger Wermutstropfen war, dass der Austausch im Team und mit dem Referenten fast ausschliesslich online erfolgte, wodurch persönliche Gespräche abseits der Arbeit zu kurz kamen.

Zusammenfassend sind wir stolz darauf, im Rahmen dieser Masterarbeit eine innovative und praxistaugliche Applikation entwickelt zu haben, die bereits in der MVP-Version einen deutlichen Nutzen für die Anwender bringt und gleichzeitig eine solide Basis für zukünftige Weiterentwicklungen bildet.

12.4. Ausblick

Die Arbeit an *Hydra* endet nicht mit dem Abschluss dieser Masterarbeit. Nach der Behebung der beanstandeten Punkte werden wir uns erneut mit der Feuerwehr zusammensetzen und gemeinsam die nächsten Schritte planen. Ein konkreter Plan für die Weiterentwicklung besteht derzeit noch nicht, doch das Interesse und die Motivation sind vorhanden. Wir sind gespannt, wohin sich *Hydra* entwickeln wird und welchen weiteren Beitrag die Applikation künftig zur Unterstützung der Einsatzkräfte leisten kann.

Glossar

Begriffe

Begriff	Beschreibung
Administrator	Verantwortlich für Stammdaten, Benutzer- und Rollenverwaltung.
Backend-for-Frontend	(BFF) Spezielle Backend-Schicht, die UI-spezifische Daten aufbereitet.
Bounded Context	Abgegrenzter Modellbereich mit eigener Sprache (DDD-Konzept).
CRUD-Funktionen	Basisoperationen Create, Read, Update, Delete für Entities.
Einsatz	Das Mehrfachereignis selbst. Beinhaltet sämtliche Meldungen, Ereignisse und Ressourcen.
Einsatzkraft	Sammelbegriff für Kräfte im Feld (Reko- und Bewältigungstrupps).
Einsatzleitung	Koordiniert den Einsatz, priorisiert Ressourcen und verteilt Aufträge.
Entität	Fachliches Objekt im System, das gespeichert und verarbeitet wird.
Ereignis	Konkreter Vorfall innerhalb eines Einsatzes, der bearbeitet wird.
Event Sourcing	Persistenzmuster: Änderungen werden als Events gespeichert.
Exception-Handler	Zentrale Komponente für die Fehlerbehandlung.
Git-Repository	Versionsverwaltungs-Container, in dem Quellcode, Dokumente oder Konfigurationen gespeichert und versioniert werden. Grundlage für Zusammenarbeit mit GitLab.
Journal	Kanban-artige Übersicht aller Ereignisse und Aufgaben eines Einsatzes.
Lagekarte	Kartendarstellung des Einsatzgebietes mit Ereignissen, Meldungen und Ressourcen.
Meldung	Eingehende Information (z. B. „Baum auf Strasse“). Kann zu einem Ereignis führen.
Operator	Bedient das System in der Einsatzzentrale und unterstützt die Einsatzleitung.
Rekognoszierender	Bewertet Schadenslagen, liefert Einschätzungen und Handlungsempfehlungen.

Repository	Abstraktion für den Zugriff auf persistente Daten.
Soft-Delete	Entities als gelöscht markieren, anstatt permanent zu löschen.
Stammdaten	Einsatz-unabhängige Daten wie Personal, Material, Objekte oder Benutzer.
Trupp	Gruppe von Einsatzkräften, die im Feld Aufträge ausführen.
Truppchef	Führt einen Einsatztrupp im Feld.

Tabelle 12.1.: Begriffe

Abkürzungen

Abkürzung	Beschreibung
API	*Application Programming Interface*, Schnittstelle für Software-Kommunikation.
BFF	*Backend-for-Frontend*, spezielle Backend-Schicht für UI-spezifische Daten.
CI/CD	*Continuous Integration / Continuous Deployment*, automatisierte Build- und Deployment-Prozesse.
CRUD	*Create, Read, Update, Delete*, Basisoperationen für Daten.
DDD	*Domain-Driven Design*, Architektur- und Designansatz für komplexe Domänen.
DTO	*Data Transfer Object*, Transportobjekt für Daten.
GIS	*Geoinformationssystem*, Karten- und Standortdaten.
IaaS	*Infrastructure as a Service*, Cloud-Infrastrukturmodell.
JSON	*JavaScript Object Notation*, textbasiertes Datenformat.
LTS	*Long-Term Support*, Version mit langfristigem Support (z. B. Java 21 LTS).
MVP	*Minimum Viable Product*, funktionsfähiges Produkt mit minimalem Funktionsumfang.
Reko	*Rekognoszierung*, Bewertung und Einschätzung von Schadenslagen für weitere Massnahmen.
REST	*Representational State Transfer*, Architekturprinzip für Web-APIs.
STOMP	*Simple Text Oriented Messaging Protocol*, Subprotokoll für WebSockets.
UI	*User Interface*, grafische Benutzerschnittstelle.

Tabelle 12.2.: Abkürzungstabelle

Übersetzungen Domänenbegriffe

Deutsch	**Englisch**
Archivierung	Archiving
Aufgabe	Task
Bewältigung	Execution
Dokumentation	Documentation
Einsatz	Mission
Einsatzführung	Mission Management
Einsatzleitung	Mission Control
Einsatzobjekt	Facility
Entität	Entity
Ereignis	Incident
Ereignisverwaltung	Incident Management
Feld	Field
Karte	Map
Lage	Situation
Lagebild	Operation Overview
Material	Material
Meldung	Message
Meldungsverwaltung	Message Management
Melder	Reporter
Operator	Operator
Ort	Location
Personal	Staff
Priorität	Priority
Ressourcenverwaltung	Resource Management
Stammdaten	Master Data
Trupp	Squad
Truppchef	Squad Leader
Truppmitglied	Squad Member

Tabelle 12.3.: Übersetzungstabelle Domänenbegriffe

Hilfsmittel

Einsatz von KI-Werkzeugen

Im Rahmen der Masterarbeit wurden KI-basierte Werkzeuge eingesetzt. Diese dienten ausschliesslich der Unterstützung und nicht der inhaltlichen Erstellung der Arbeit.

Konkret umfasste dies:

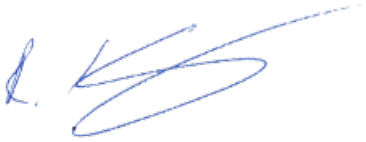
- Auto-Completion in der Entwicklungsumgebung zur Unterstützung bei der Implementierung
- Unterstützung bei der Recherche
- Generierung des *Hydra*-Logos und Porträtbilder der Personas
- Rechtschreib- und Formulierungsprüfungen in der Dokumentation

Die inhaltliche Ausarbeitung, Architekturentscheidungen, Implementierung und Analyse wurden vollständig in Eigenleistung der Autoren der Arbeit erbracht.

Selbständigkeitserklärung

Hiermit erklären wir, dass wir die vorliegende Masterarbeit im MAS Software Engineering mit dem Titel «Hydra - Multi-Incident Management - Simplified» selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht haben. Weiterhin erklären wir, dass wir keine durch Copyright geschützten Materialien (z. B. Bilder) in dieser Arbeit in unerlaubter Weise verwendet haben und in dieser Arbeit keine Adressen, Telefonnummern und andere persönliche Daten von Personen, die nicht zum Kernteam gehören, publizieren.

Ort, Datum: Rothenburg, 14.09.2025

Reto Kunz, Unterschrift: 

Mario Leonini, Unterschrift: 

Fabian Ott, Unterschrift: 

Abbildungsverzeichnis

Tabellenverzeichnis

Quellenverzeichnis

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional.

Cohn, M. (2004). *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional.

Dockerhub. (2025). *Dockerhub Registry*. Verfügbar 6. September 2025 unter https://hub.docker.com/

Einstein, A. (1933). On the Method of Theoretical Physics.

Evans, E. (2003). *Domain-Driven Design*. Addison-Wesley.

Everest, G. C. (1976). BASIC DATA STRUCTURE MODELS EXPLAINED WITH A COMMON EXAMPLE. *Computing Systems, pages 39-46*(1976).

Fowler, M. (2005). *Event Sourcing*. Verfügbar 31. August 2025 unter https://martinfowler.com/eaaDev/EventSourcing.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

GitLab. (2025). *GitLab Documentation*. Verfügbar 28. August 2025 unter https://docs.gitlab.com/

Gradle. (2025). *Gradle - Multi-Project Builds*. Verfügbar 6. September 2025 unter https://docs.gradle.org/current/userguide/intro_multi_project_builds.html

Highsmith, J. (2004). *Agile Project Management: Creating Innovative Products*. Addison-Wesley Professional.

Kecher, C., & Salvanos, A. (2015). *UML 2.5: Das umfassende Handbuch*. Rheinwerk Verlag GmbH.

Martin, R. C. (2008, August). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education Limited.

Martin, R. C. (2012, August). *The Clean Architecture*. Verfügbar 10. September 2025 unter https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Martin, R. C. (2017, September). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson Education Limited.

Material UI. (2025). *Material UI - Documentation*. Verfügbar 6. September 2025 unter https://mui.com/material-ui/getting-started/

OpenLayers. (2025). *OpenLayers - Documentation*. Verfügbar 6. September 2025 unter https://openlayers.org/doc/

openstack. (2025). *openstack Documentation*. Verfügbar 28. August 2025 unter https://docs.openstack.org/2025.1/

Oracle. (n. d.). *Code Conventions for the Java Programming Language*. Verfügbar 2. September 2025 unter https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

Qvortrup, M. (n. d.). *Vorlesung Softwarearchitektur* [MAS Software Engineering, Wintersemester 2023/24] [Vorlesungsunterlagen].

SemVer. (2025). *Semantic Versioning 2.0.0*. Verfügbar 14. September 2025 unter https://semver.org/

spring. (2025). *spring Boot - Profiles*. Verfügbar 6. September 2025 unter https://docs.spring.io/spring-boot/reference/features/profiles.html

Stopford, B. (2018). *Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka*. O'Reilly.

swisstopo. (2025a). *swisstopo - Amtliches Verzeichnis Adressen*. Verfügbar 6. September 2025 unter https://www.swisstopo.admin.ch/de/amtliches-verzeichnis-der-gebaeudeadressen

swisstopo. (2025b). *swisstopo - Karten*. Verfügbar 6. September 2025 unter https://www.swisstopo.admin.ch/de/karten

Thurgaufire. (n. d.). *Einsatzablauf Mehrfachereignis*. Verfügbar 7. September 2025 unter https://www.thurgaufire.ch/uploads/doc/Infos-Kommandanten/Einsatzablauf%20Mehrfachereignis.pdf

Vernon, V. (2013, Februar). *Implementing Domain-Driven Design*. Addison-Wesley Educational Publishers Inc.

Vite. (2025). *Vite*. Verfügbar 6. September 2025 unter https://vite.dev

YouTrack. (2025). *YouTrack Documentation*. Verfügbar 28. August 2025 unter https://www.jetbrains.com/help/youtrack/

Anhang

Dieser Inhalt wurde aus Vertraulichkeitsgründen aus der publizierten Version entfernt.

B. Eingesetzte Technologien und Tools

B.1. Projektmanagement

Teilsystem	**Technologie**
Projektmanagement	YouTrack: Agile / Kanban
Dokumentation	YouTrack: Wiki & LaTeX(MikTeX)
Diagramme	Miro, Mermaid & DrawIo
Versionskontrolle	Git
Sourcecode-Verwaltung	GitLab (Cloud)
API-Dokumentation	OpenAPI & AsyncAPI

Tabelle B.1.: Projektmanagement-Technologien

B.2. Infrastruktur

Teilsystem	**Technologie**
Verwaltung Sourcecode	Gitlab (Cloud)
CI/CD Pipelines	Gitlab (Cloud)
Package / Artefakt- Registry	Gitlab (Cloud)
Container Registry	Private: Gitlab (Cloud) / Public: Dockerhub
Cloud / Deployment	Openstack (IaaS)
Reverse Proxy	Traefik
Container Runtime	Docker
Webserver	Linux ubuntu LTS / Openstack (IaaS)
Buildserver	Linux popOS 64bit (Private HW)
Log-Aggregator	Loki & Grafana (Admin UI)

Tabelle B.2.: Infrastruktur-Technologien

B.3. Frontend

Teilsystem	Technologie
Programmiersprache	Typescript
Platform	NodeJs
Framework	REACT
Buildtool	Vite
Bibliothek Karte	Open Layers
Bibliothek UI	Material UI

Tabelle B.3.: Frontend-Technologien

B.4. Backend

Teilsystem	Technologie
Programmiersprache	Java 21
Main-Framework	Spring
Test-Framework	JUnit
Mocking-Framework	Mockito
Code Coverage	JaCoCo
Buildtool	Gradle
Datenbank	MongoDB & Mongo-Express (Admin UI)
Event Streaming Platform	Kafka & kafbat (Admin UI)
Schema Registry	Confluent Schema Registry
Codegeneratoren AsyncAPI	AsyncAPI Generator (CLI Tool)
Codegeneratoren OpenAPI	OpenAPI Generator (CLI Tool) & Swagger Generartor (Gradle Plugin)

Tabelle B.4.: Backend-Technologien

Dieser Inhalt wurde aus Vertraulichkeitsgründen aus der publizierten Version entfernt.

E. Risikoanalyse

Im Rahmen der Masterarbeit war es wichtig, potenzielle Risiken frühzeitig zu identifizieren, zu bewerten und geeignete Massnahmen zur Risikominimierung zu definieren. Die vorliegende Risikoanalyse diente dazu, mögliche Unsicherheiten zu erfassen, die den Projekterfolg gefährden könnten. Ziel war es, ein Risikomanagement zu haben, um die Qualität, den Zeitplan und die Zielerreichung der Arbeit sicherzustellen.

ID	Risiko	Kategorie	Auswirkung	Massnahme
R1	Zu komplexe Lösung	Technisch	Hoher Aufwand, schwer wartbar	Fokus auf Einfachheit, MVP-Ansatz
R2	Fehlerhaftes Design	Technisch	Hoher Aufwand, schwer wartbar	Refactoring
R3	Zu viele User Stories	Scope	Zeitverzug, Qualitätsverlust	Klarer Fokus, Priorisierung
R4	Dokumentation vernachlässigen	Organisatorisch	Stress gegen Ende der Masterarbeit	Dokumentation laufend pflegen
R5	Zu viel Zeit in CI/CD investieren	Technisch	User Stories werden nicht fertig	Nur so viel CI/CD wie nötig
R6	Technologie- Lock-in (Kafka, Spring)	Strategisch	Eingeschränkte Weiterentwicklung	Technologien bewusst wählen, Alternativen prüfen
R7	Fehlende Skalierbarkeit	Technisch	Eingeschränkter produktiver Betrieb	Früh an Skalierung denken, testen

Tabelle E.1.: Risiken

Die identifizierten Risiken wurden anhand der geschätzten Eintrittswahrscheinlichkeit und der erwarteten Auswirkungen beurteilt und mit der Risikomatrix in Abbildung E.1 visualisiert.

Die Risiken R6 und R7 sind in der Matrix einmal als kurzfristige (schwarz) und einmal als langfristige Risiken (rot) eingetragen.

Schlussfolgerung

Die Risiken *zu komplexe Lösung (R1)*, *fehlerhaftes Design (R2)*, *zu viele User Stories (R3)*, *vernachlässigte Dokumentation (R4)* sowie *zu viel Zeit in CI/CD (R5)* befinden sich

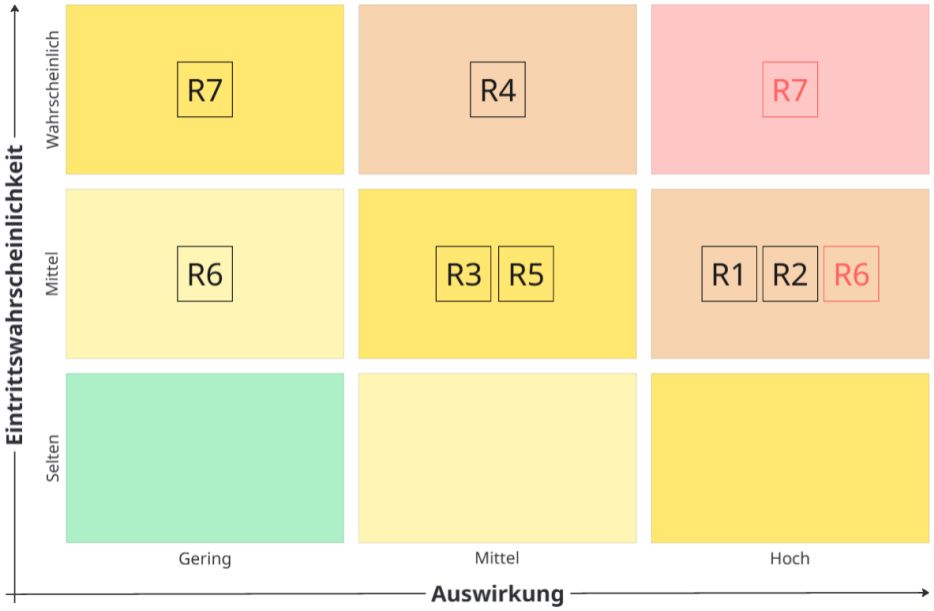


Abbildung E.1.: Risikomatrix

im mittleren bis hohen Auswirkungsbereich. Diese erfordern während der Masterarbeit erhöhte Aufmerksamkeit, um Verzögerungen und Qualitätsprobleme zu vermeiden.

Die Risiken *Technologie-Lock-in (R6)* und *fehlende Skalierbarkeit (R7)* sind für den Abschluss der Masterarbeit nicht entscheidend, gewinnen jedoch an Relevanz, falls das Projekt kommerziell weitergeführt werden soll.

Für den Erfolg der Masterarbeit lassen sich daraus folgende Bereiche definieren, auf welche besonderen Fokus gelegt werden sollte:

- **Einfachheit und Priorisierung:** Die Umsetzung sollte ein MVP zum Ziel haben, um Komplexität und Überladung mit User Stories zu vermeiden.
- **Qualitätssicherung:** Durch laufendes Refactoring, eine angemessene Testabdeckung sowie einen zweckmässigen, aber nicht übertriebenen Einsatz von CI/CD wird die technische Stabilität gesichert.
- **Kontinuierliche Dokumentation:** Eine kontinuierliche Dokumentation verhindert Wissensverlust, reduziert den Stress gegen Ende des Projekts und stellt sicher, dass Architektur- und Designentscheidungen auch später noch nachvollzogen werden können.
- **Bewusstes Technologiemanagement:** Auch wenn Lock-in und Skalierbarkeit langfristige Themen sind, sollte bereits jetzt bei Architekturentscheidungen eine gewisse Flexibilität eingeplant werden.

Die Ergebnisse der Risikoanalyse zeigen auf, welche potentiellen Probleme während der Masterarbeit besondere Aufmerksamkeit erfordern. So können frühzeitig geeignete Massnahmen getroffen werden, um den Erfolg des Projektes sicherzustellen.

Dieser Inhalt wurde aus Vertraulichkeitsgründen aus der publizierten Version entfernt.