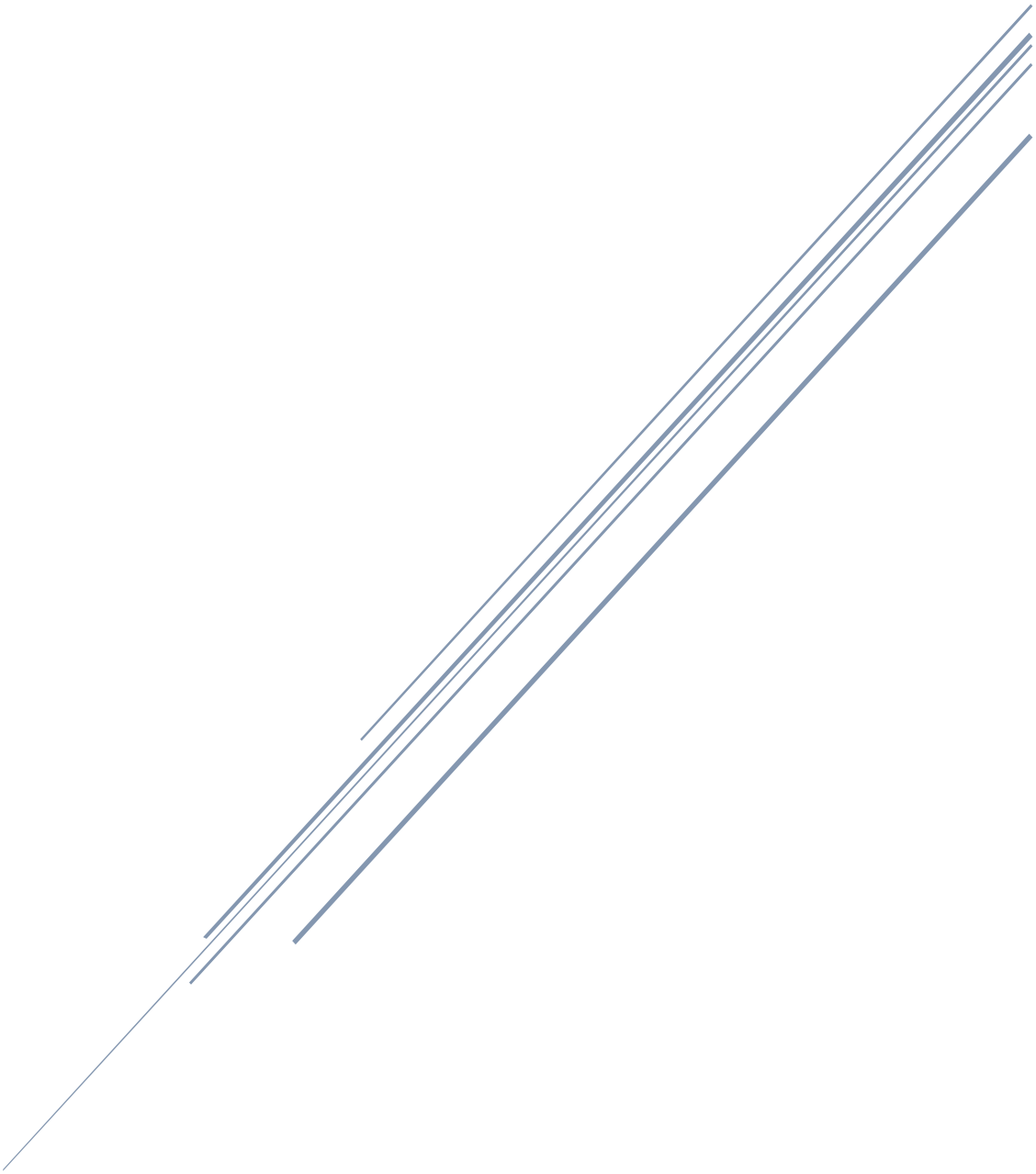
SKI-SERVICE MANAGEMENT NOSQL

Projektdokumentation nach IPERKA



IBZ - Basel
Fokko Vos

Inhaltsverzeichnis

1 Versionsverlauf

Version	Autor	Datum	Änderung
1.0	Fokko Vos	31.01.2024	Erstellung des Dokumentes und der Planung
1.1	Fokko Vos	01.02.2024	Minimale Anpassungen des Plans
1.2	Fokko Vos	02.02.2024	Erstellung der Realisierungs Abschnitte
1.3	Fokko Vos	02.02.2024	Finalisierung der Dokumentation

2 Informieren

2.1 Ausgangsituation

Unternehmen
Jetstream-Service, ein KMU, das sich auf Skiservicearbeiten in der Wintersaison spezialisiert hat.

Ziel
Das Hauptziel des Projekts ist die Migration des bestehenden Backend-Systems zu einer NoSQL-Datenbanklösung. Dieser Schritt zielt darauf ab, die Datenverwaltung effizienter, flexibler und skalierbarer zu gestalten, um die Verwaltung von Ski-Service-Aufträgen zu optimieren.

Integration
Im Rahmen dieses Projekts liegt der Fokus ausschließlich auf der Entwicklung und Implementierung des Backend-Systems mit NoSQL. Die Integration in das bestehende Frontend oder andere Systemkomponenten ist nicht Teil dieses Projektauftrags.

2.2 Anforderungsanalyse

2.2.1 Funktionale Anforderungen

- Request Protokollierung (Serilog)
- Übersicht der ausstehenden Service-Aufträge
- Login für Änderungen an einem Auftrag (Status, Notiz, Löschen)

2.2.2 Nicht-Funktionale Anforderungen

- Starke Authentifizierung und Autorisierung.
- Schnelle und effiziente Verarbeitung von Anfragen.
- Einfache und intuitive Bedienung für Mitarbeiter.

2.3 Technische Anforderungen

Technologie
Backend Entwicklung von einer ASP.NET Web API mit einer NoSQL Anbindung.

Datenbankanforderungen
Robustes Datenbank Design für die Verwaltung von Auftrags- und Kundendaten

Integration
Kompatibilität und reibungslose Integration mit der bestehenden Infrastruktur.

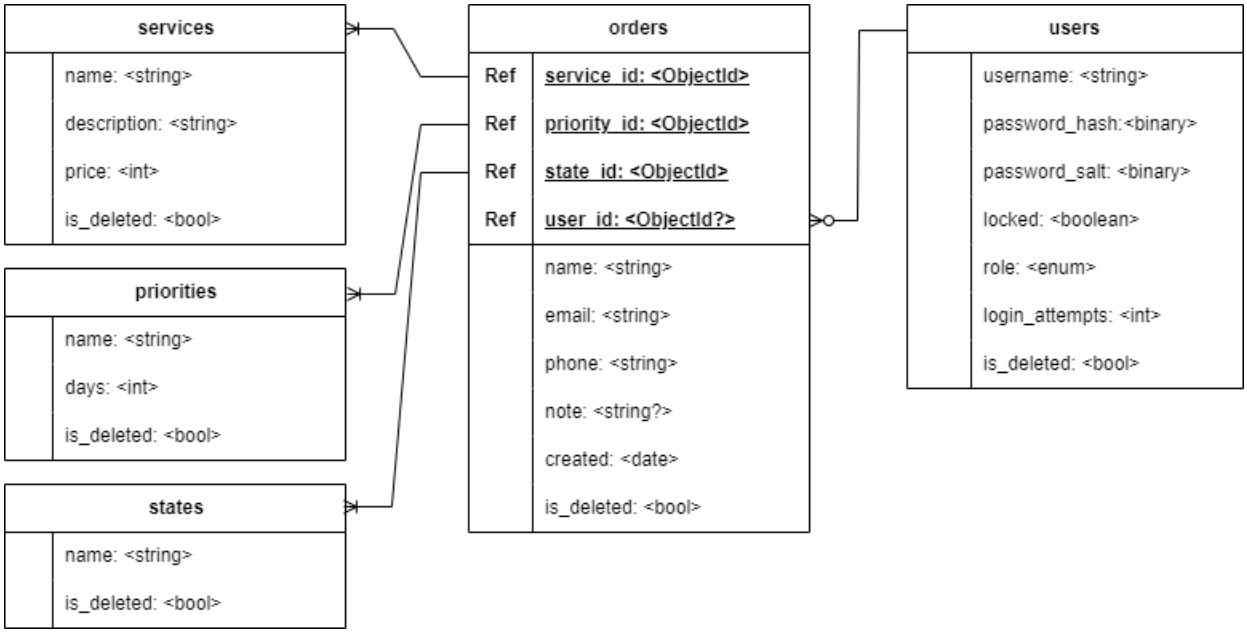
3 Planen

3.1 Zeitplanung

Nr.	Beschreibung	SOLL-Zeit	IST-Zeit
1	**Informieren**	**1.5**	**2**
1.1	Situationsanalyse	1	1.5
1.2	Tool-Installation (MongoDB)	0.5	0.5
2	**Planen**	**6.5**	**5.5**
2.1	Anpassung des Zeitplans	0.5	1
2.2	Anpassung des Datenbankdesigns für NoSQL	1.5	1
2.3	Anpassung der Systemarchitektur	0.5	1
2.4	Planen des Nutzerkonzepts	1	1.5
2.5	Überprüfung der API-Endpunkte	1	1
3	**Entscheiden**	**1**	**1**
3.1	Test-Strategie anpassen	0.5	0.5
3.2	Festlegen der Migrationstechnik (NoSQL)	0.5	0.5
4	**Realisieren**	**27.5**	**29.5**
4.1	Grundlegendes Projekt erstellen	1	1
4.2	Git-Repository aufsetzen	0.5	0.5
4.3	Erstellung des Initialisierung Skripts	3	6
4.4	Implementierung der NoSQL-Datenbank	4	5
4.5	Entwicklung des Backend (NoSQL-Anpassungen)	7	10
4.6	Erstellung von Backup- und Restore-Strategien	3	5
4.7	Entwicklung angepasster Tests	3	2
5	**Kontrollieren**	**3**	**2.5**
5.1	Test-Strategie ausführen	1	0.5
5.2	Anforderungen mit dem Produkt abgleichen	2	2
6	**Auswerten**	**4**	**4**
6.1	Finalisierung der Dokumentation	2	1
6.2	Lessons-Learned identifizieren	1	2
6.3	Abschlusspräsentation vorbereiten	1	1
Gesamt		**40.5**	**44.5**

1 - PSP Zeitplanung

3.2 Datenbankentwurf



1 - Datenbankentwurf

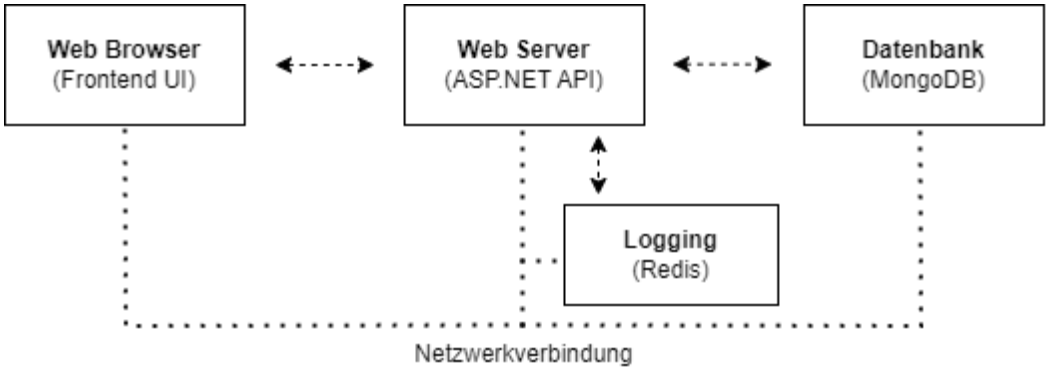
In der Planungsphase meines Projekts habe ich mich für einen NoSQL-Datenbankentwurf entschieden, der die Flexibilität und Schnelligkeit eines dokumentenorientierten Systems nutzt. Jedes Element, von Services über Benutzer bis hin zu Prioritäten, ist als eigenständiges Dokument modelliert. Diese Struktur erlaubt es mir, Beziehungen zwischen verschiedenen Entitäten effizient abzubilden und dabei die Erweiterbarkeit des Systems im Blick zu behalten.

Ich habe besonderen Wert daraufgelegt, dass jedes Dokument durch spezifische Indizes unterstützt wird, um die Abfragegeschwindigkeit zu maximieren. Beispielsweise gewährleisten Indizes auf Benutzernamen und Dienstleistungsnamen die Einzigartigkeit dieser Elemente und erleichtern schnelle Suchoperationen. Zudem ermöglicht die Indizierung von Aufträgen nach Erstellungsdatum, Priorität und Zustand eine effiziente Organisation und Verwaltung der Daten.

Die Implementierung eines Soft-Delete-Mechanismus durch das **is_deleted** Attribut in jedem Dokument spiegelt meine Intention wider, Daten nicht physisch zu löschen, sondern sie als gelöscht zu markieren. Dieser Ansatz unterstützt nicht nur die Integrität und Nachvollziehbarkeit der Daten, sondern ermöglicht auch eine problemlose Wiederherstellung im Bedarfsfall.

Insgesamt reflektiert der Entwurf meines Datenbankschemas eine Balance zwischen Leistung und Erweiterbarkeit, wobei ich stets darauf bedacht war, ein System zu schaffen, das sowohl den aktuellen Anforderungen gerecht wird als auch Raum für zukünftiges Wachstum bietet.

3.3 Systemarchitekturentwurf



2- Systemarchitekturentwurf

Das System ist auf eine Drei-Schichten-Architektur ausgelegt, die eine klare Trennung zwischen Benutzeroberfläche, Geschäftslogik und Datenpersistenz ermöglicht. Diese Strukturierung erlaubt es, jede Schicht unabhängig von den anderen zu entwickeln und zu warten, was eine effiziente und zielgerichtete Entwicklung fördert.

Web Browser (Frontend UI)
Die Präsentationsschicht ist als Webanwendung realisiert, die im Browser des Benutzers läuft. Sie ist verantwortlich für die Darstellung der Benutzeroberfläche und die Interaktion mit dem Nutzer.

Web Server (ASP.NET API)
Die Geschäftslogikschicht, implementiert durch eine ASP.NET-basierte API, behält ihre Rolle bei der Verarbeitung von Benutzeranfragen und der Ausführung von Geschäftsregeln bei. Die Anpassungen betreffen die Integration mit MongoDB.

Datenbank (MongoDB)
Die Datenhaltungsschicht wird von MongoDB, einem NoSQL-Datenbankmanagementsystem, realisiert. MongoDB bietet eine dynamische, dokumentenorientierte Datenspeicherung, die eine flexible Handhabung verschiedenster Datenformate ermöglicht.

Logging (Redis)
Diese Schicht ist verantwortlich für die effiziente Erfassung, Speicherung und möglicherweise auch für die Analyse von Log-Daten. Die Nutzung von Redis ermöglicht eine schnelle und skalierbare Lösung für das Erfassen von Systemlogs, was für die Überwachung und Fehlersuche von Bedeutung ist.

Die Kommunikation zwischen den Schichten erfolgt über HTTPS, wobei JSON als Datenaustauschformat verwendet wird, um die Interoperabilität und das leichte Parsing der Daten zu gewährleisten. Diese Architektur bietet eine solide Basis für die Realisierung unseres Projekts und unterstützt dessen Wachstum und Anpassungsfähigkeit an zukünftige Anforderungen.

3.4 API-Endpunkte

Jeder Endpunkt wird hinter dem Präfix /api liegen

Endpunkt	Methode	Beschreibung	AUTH	Rolle
/users	GET	Abfrage aller Nutzer	X	-
/users	POST	Erstellung eines neuen Nutzers	X	superadmin
/users/login	POST	Login Endpunkt		-
/users/me	GET	Informationen des eingeloggten Nutzers	X	-
/users/<id>	GET	Nutzer abfragen	X	superadmin, self
/users/<id>	PUT	Nutzer ändern	X	superadmin
/users/<id>	DELETE	Nutzer löschen	X	superadmin, self
/users/<id>/orders	GET	Alle zugewiesenen Aufträge für diesen Nutzer		-
/users/revoke	POST	Refresh Token entfernen	X	self
/users/refresh	POST	Mittels Refresh Token und abgelaufenem Access Token einen neuen Access Token erhalten	X	self
/users/<id>/unlock	POST	Einen Benutzer entsperren	X	superadmin
/orders	GET	Abfrage aller Aufträge	X	-
/orders	POST	Neuen Auftrag erstellen	X	-
/orders/<id>	GET	Auftrag abfragen	X	-
/orders/<id>	PUT	Auftrag ändern	X	-
/orders/<id>	DELETE	Auftrag löschen	X	-
/orders/user/<user_id>	GET	Alle Aufträge eines Nutzers	X	-
/orders/state/<state_id>	GET	Alle Aufträge mit einem bestimmten State	X	-
/orders/service/<service_id>	GET	Alle Aufträge für einen bestimmten Service-Typ	X	-
/orders/priority/<priority_id>	GET	Alle Aufträge mit einer bestimmten Priorität	X	-
/services	GET	Abfrage aller Services	X	-
/services/<id>	GET	Abfrage eines Service	X	-
/states	GET	Abfrage aller Stati	X	-
/states/<id>	GET	Status abfragen	X	-
/priority	GET	Abfrage aller Prioritäten	X	-
/priority/<id>	GET	Priorität abfragen	X	-

2 - API Endpunkte

Dies ist eine Grundlegende Initial-Definition der API-Endpunkte, kann sich bei der Entwicklung leicht ändern. Standard endpoints wie POST, PUT und DELETE wurden für States, Services, und Priorität weggelassen auf Grund der Tabellen Länge diese benötigen allerdings superadmin um Änderungen vor zu nehmen.

4 Entscheiden

4.1 Test-Strategie

Für das Projekt habe ich geplant, Integrationstests mit einer Postman Collection durchzuführen, um die Interaktionen und Funktionalitäten der verschiedenen API-Endpoints zu überprüfen. Diese Methode ermöglicht es, einen umfassenden Überblick über die Systemintegration und das Zusammenspiel der Komponenten zu erhalten. Durch den Einsatz von Postman als Testwerkzeug kann effizient sichergestellt werden, dass die API wie erwartet funktioniert und korrekt auf Anfragen reagiert. Diese praxisnahe Teststrategie bietet einen effektiven Ansatz, um die Anforderungen des Projekts ohne die Implementierung von Unit Tests zu erfüllen.

4.2 Code-First Anpassung an NoSQL

Für die Datenbankentwicklung im Rahmen des Ski-Service Management-Projekts wird nun ein Database-First-Ansatz verwendet. Dieser Ansatz beginnt mit der Definition und Initialisierung der Datenbankstruktur durch ein Skript vor dem ersten Start der Anwendung. Dabei werden alle notwendigen Indizes und Schemas konfiguriert, um eine optimale Performance und Skalierbarkeit zu gewährleisten. Dieser Prozess erlaubt eine präzise Kontrolle über die Datenbankarchitektur und erleichtert die Integration mit bestehenden Datenbanksystemen.

Die genauen Models können im NuGet Projekt gefunden werden

https://github.com/fokklz/nuget-ski-service-models/tree/main/SkiServiceModels.BSON

4.3 Benutzer Konzept

Dieses Konzept definiert die Zugriffsrechte und Rollen innerhalb des Systems, um eine sichere und effiziente Nutzung der Anwendung zu gewährleisten. Im Folgenden wird das Benutzerkonzept, einschließlich der Rollenverteilung und der Zugriffsrechte, dargelegt.

4.3.1 Rollen innerhalb des Systems

Das System unterscheidet zwischen zwei Hauptrollen:

Superadmin
Der Superadmin besitzt die höchsten Zugriffsrechte innerhalb des Systems. Diese Rolle ist in der Lage, sämtliche administrative Aufgaben durchzuführen, einschließlich der Verwaltung von Benutzerkonten, der Erstellung von Orders und der Anpassung von Systemeinstellungen.
User
Der User repräsentiert die Standardbenutzerrolle im System. Benutzer dieser Rolle können neue Orders erstellen und ihre eigenen Aufträge nach der Authentifizierung aktualisieren.

4.3.2 Zugriffsrechte und Funktionalitäten

Login
Die Login-Funktion ist für alle Benutzer zugänglich, um Zugang zu ihren jeweiligen Konten und den damit verbundenen Funktionen zu erhalten.
Erstellung neuer Orders
Die Möglichkeit zur Erstellung neuer Orders ist sowohl für authentifizierte als auch für nicht authentifizierte Benutzer verfügbar. Dies ermöglicht es auch nicht registrierten Nutzern, Dienstleistungen des Systems in Anspruch zu nehmen.
Zugriff auf weitere Routen
Für den Zugriff auf weitere Funktionen des Systems, wie das Anzeigen oder Bearbeiten bestehender Orders, ist eine Authentifizierung erforderlich. Dies dient dem Schutz sensibler Benutzerdaten und der Sicherstellung, dass nur berechtigte Nutzer auf bestimmte Informationen zugreifen können.
Erstellen von Inhalten
Das Erstellen und Verwalten von Inhalten im System (außer der Erstellung neuer Orders) ist ausschließlich dem Superadmin vorbehalten. Dies umfasst unter anderem die Möglichkeit, neue Benutzerrollen zu definieren oder Systemeinstellungen anzupassen.
Aktualisierung von Orders
Die Aktualisierung bestehender Orders ist sowohl für den Superadmin als auch für den jeweiligen Owner des Orders möglich. Als Owner gilt dabei der Benutzer, der den Order erstellt hat, oder ein Benutzer, der explizit einem Order zugewiesen wurde.

5 Realisierung

5.1 Grundlegendes Projekt erstellen

Ich habe mit Visual Studio ein ASP.NET Web API Projekt für das Ski-Service-Management-System erstellt. Dieses Projekt ist nicht nur mit grundlegenden Bausteinen ausgestattet, sondern auch mit erweiterten Funktionen für eine robuste und sichere Anwendung:

JWT-Authentifizierung
Implementiert im TokenService und AuthService, um sichere und effiziente Benutzerauthentifizierung und -autorisation mit Rollen zu gewährleisten.
Serilog Logging
Konfiguriert im appsettings für fortgeschrittene Logging-Fähigkeiten, einschliesslich der Rotation von Logdateien.
Datenbankanbindung
Verwendung von Entity Framework Core für die Datenbankinteraktion, konfiguriert in Programm.cs und appsettings.
Automapper
Eingesetzt für die effiziente Umwandlung zwischen Datenmodellen und Data Transfer Objects (DTOs), konfiguriert im SkiServiceModels NuGet.
Swagger Integration
Ermöglicht eine interaktive Dokumentation und einfache Testmöglichkeiten der Web API.
Docker-Unterstützung
Bereitstellung von Dockerfiles und Skripten für die Containerisierung der Anwendung und der Datenbank.

5.2 Git-Repository Aufsetzen

Für die Versionskontrolle habe ich GitHub verwendet. Das Initial Repository wurde direkt aus Visual Studio heraus erstellt. Wichtige Dateien und Konfigurationen, die im Repository enthalten sind:

.gitignore
Eine sorgfältig konfigurierte Datei, die sicherstellt, dass nur relevante Dateien und Verzeichnisse im Repository verfolgt werden. Sie schließt unnötige oder vertrauliche Dateien aus, wie z.B. lokale Konfigurationsdateien und Build-Artefakte.
.gitattributes
Diese Konfiguration hilft, korrekte Zeilenendungen sicherzustellen und verhindert potenzielle Probleme bei der Ausführung auf unterschiedlichen Betriebssystemen.
README.md
Eine zentrale Dokumentationsdatei, die eine Übersicht über das Projekt, Installationsanweisungen, Nutzungshinweise und andere wichtige Informationen enthält.
Lizenzdatei
Die LICENSE-Datei definiert, wie andere das Projekt verwenden dürfen. Dies ist wichtig für die Festlegung von Urheberrechten und Nutzungsbedingungen.

5.3 Erstellung des Initialisierung Skripts

Im Rahmen der Entwicklung des Ski-Service-Management-Systems, welches auf einer NoSQL-Datenbankstruktur basiert, wurde ein PowerShell-Skript namens initialize.ps1 implementiert. Dieses Skript spielt eine zentrale Rolle in der Vorbereitung und Initialisierung der Datenbankinfrastruktur, indem es die erforderlichen Datenstrukturen und Berechtigungen einrichtet und eine solide Grundlage für die Aufnahme der Geschäftsdaten schafft.

Das Initialisierungsskript führt eine Reihe von kritischen Aufgaben durch, um die Datenbank für den Einsatz bereitzustellen:

Bereinigung der Datenbank:
Zu Beginn entfernt das Skript bestehende Daten und Strukturen, um eine saubere Basis für die Initialisierung zu schaffen.
Erstellung von Sammlungen
Anschließend werden die für das System erforderlichen Sammlungen angelegt, wobei für jede Sammlung spezifische Schemata verwendet werden, die die Datenintegrität und -struktur gewährleisten
Indexierung
Um die Leistung der Datenabfragen zu optimieren, werden Indizes für die wichtigsten Felder innerhalb der Sammlungen erstellt.
Datenmigration
Das Skript führt die Migration bestehender Daten durch, wobei für jede Sammlung spezielle Migrationsskripte zum Einsatz kommen.
Einrichtung von Benutzerrollen und -berechtigungen
Zum Abschluss werden administrative Benutzerkonten und Berechtigungen eingerichtet, darunter ein Superadmin-Konto für die Datenbankverwaltung.

Technische Implementierung

Das Skript kann mit dem Parameter $Uri aufgerufen werden, der die Verbindungszeichenfolge zur MongoDB-Datenbank angibt. Standardmäßig wird mongodb://localhost:27017 verwendet. Es wird versucht, mit Superadmin-Rechten auf die Datenbank zuzugreifen. Falls kein Superadmin-Benutzer vorhanden ist, versucht das Skript, die Operationen ohne Authentifizierung durchzuführen. Jeder Schritt des Skripts ist mit einer Fehlerprüfung versehen, um sicherzustellen, dass alle Operationen erfolgreich durchgeführt wurden. Im Fehlerfall gibt das Skript eine entsprechende Meldung aus.

5.4 Datenbank Implementierung

In diesem Abschnitt habe ich die Datenbank für das Ski-Service-Management-System implementiert, wobei ich mich auf Robustheit, Effizienz und Skalierbarkeit konzentriert habe:

Datenbanktechnologie und Containerisierung
Verwendung von MongoDB, wie im Docker Compose file ersichtlich. Die Containerisierung ermöglicht eine einfache und konsistente Bereitstellung der Datenbank.
Initialisierungsskript
Ein initialize.ps1 Skript wurde erstellt, um die Datenbank und erforderlichen Benutzer bei Ausführung zu initialisieren.
Datenbankmodelle
Die Datenbankmodelle, wie User, Order, Priority, Service und State, sind in den entsprechenden Modellklassen definiert. Diese Modelle repräsentieren die Struktur der Datenbanktabellen und deren Beziehungen.

5.5 Entwicklung des Backend

In diesem Abschnitt habe ich die Haupt-Logik des Programms geschrieben:

Architektur und Design
In der Entwicklung des Backends meines ASP.NET Web API Projekts habe ich eine Architektur geschaffen, die generische Controller und Services nutzt. Diese ermöglichen eine effiziente und flexible Handhabung verschiedener Entitätstypen. Die Architektur unterstützt die Wiederverwendbarkeit des Codes und erleichtert die Wartung, während sie gleichzeitig eine solide Grundlage für die Sicherheits- und Autorisierungsmechanismen bietet.

API-Entwicklung
Ich habe mich bei der API-Entwicklung streng an REST-Prinzipien gehalten. Durch die Verwendung von AutoMapper, implementiert im SkiServiceModels NuGet, konnte ich eine effiziente Konvertierung zwischen Datenmodellen und DTOs erreichen. Die Controller, wie OrdersController und UsersController, sind konsistent in ihrer Struktur und bieten eine klare und intuitive Schnittstelle für CRUD-Operationen.

Authentifizierung mit JWT
Die Implementierung der JWT-basierten Authentifizierung erfolgte durch eine Kombination von Anpassungen in Program.cs, dem TokenService, AuthService und dem UserService, sowie durch die Integration in den UsersController. Diese Implementierung stellt eine sichere und effiziente Authentifizierung sicher und ist ein Kernbestandteil der Sicherheitsarchitektur der Anwendung.

Fehlerbehandlung und Logging
Für das Logging habe ich Serilog integriert, um detaillierte und aussagekräftige Log-Informationen zu erfassen. Die selbst entwickelte ExceptionHandlingMiddleware spielt eine zentrale Rolle in der Fehlerbehandlung, indem sie Ausnahmen abfängt und in einer benutzerfreundlichen Form zurückgibt.

Dokumentation und Swagger
Die Integration von Swagger/OpenAPI in das Projekt ermöglicht eine klare und interaktive Dokumentation der API. Die Response-DTOs sind detailliert definiert, um eine strukturierte und verständliche Rückgabe von Daten zu gewährleisten.

Performance und Skalierung
Ich habe mich auf generische Implementierungen, effiziente Datenbankzugriffe und asynchrone Programmierung konzentriert, um eine hohe Performance und Skalierbarkeit des Backends zu erreichen. Regelmäßiges Performance-Monitoring und gezielte Optimierungen tragen zur langfristigen Leistungsfähigkeit des Systems bei.

5.6 Erstellung von Backup- und Restore-Strategien

Für das Ski-Service-Management-System, das eine NoSQL-Datenbank verwendet, sind zuverlässige Backup- und Restore-Strategien unerlässlich, um die Datensicherheit und Wiederherstellungsfähigkeit im Falle eines Datenverlusts oder Systemausfalls zu gewährleisten. Im Projektverzeichnis scripts wurden hierfür spezielle PowerShell-Skripte entwickelt: backup.ps1 für Backups und restore.ps1 für die Wiederherstellung, einschließlich ihrer Docker-Varianten backup.docker.ps1 und restore.docker.ps1.

5.6.1 Backup-Strategie

Das Skript **backup.ps1** automatisiert den Prozess der Erstellung von Datenbankbackups. Es generiert eine komprimierte Archivdatei der Datenbank, die mit dem aktuellen Datum und der Uhrzeit benannt wird, um eine einfache Identifizierung und Versionierung der Backups zu ermöglichen. Das Skript akzeptiert einen optionalen Port-Parameter, um die Flexibilität bei der Verbindung zu verschiedenen MongoDB-Instanzen zu erhöhen. Bei Ausführung ohne Fehlermeldung wird das Backup im Verzeichnis backups abgelegt. Es wird zunächst versucht, das Backup mit Superadmin-Rechten durchzuführen, um alle Benutzerdaten und Rollen zu sichern. Bei Fehlern wird eine entsprechende Fehlermeldung ausgegeben.

Die Docker-Variante **backup.docker.ps1** ruft das Basis-Backup-Skript mit einem spezifischen Port auf, der für die Docker-Container-Umgebungen angepasst ist.

5.6.2 Restore-Strategie

Das Skript **restore.ps1** ermöglicht die Wiederherstellung der Datenbank aus einem zuvor erstellten Backup. Der Nutzer kann aus einer Liste von verfügbaren Backups auswählen, welche Datenbankversion wiederhergestellt werden soll. Ähnlich dem Backup-Skript akzeptiert auch dieses einen Port-Parameter für die Verbindung zur Datenbank. Die Wiederherstellung erfolgt unter Berücksichtigung der Benutzerrechte und Rollen, um die Integrität der Datenbankzugriffsrechte zu wahren.

Die Docker-Variante **restore.docker.ps1** ruft das Basis-Backup-Skript mit einem spezifischen Port auf, der für die Docker-Container-Umgebungen angepasst ist.

5.6.3 Automatisierte Backup-Einrichtung

Das Skript **register-auto-backup.ps1** wurde speziell entwickelt, um die Automatisierung von Backup-Prozessen für das Ski-Service-Management-System zu ermöglichen. Es nutzt die Windows-Aufgabenplanung, um täglich automatische Backups der Datenbank durchzuführen. Diese Strategie gewährleistet eine konstante Datensicherheit durch regelmäßige Backups, ohne dass ein manueller Eingriff erforderlich ist.

Bei der Ausführung prüft das Skript zunächst, ob es mit Administratorrechten läuft. Ist dies nicht der Fall, fordert es die benötigten Berechtigungen an, um die Aufgabenplanung erfolgreich konfigurieren zu können. Anschließend wird eine geplante Aufgabe erstellt, die täglich um 2 Uhr morgens das Backup-Skript backup.ps1 ausführt. Diese Aufgabe wird unter dem SYSTEM-Konto registriert, um höchste Ausführungsprivilegien zu gewährleisten und potenzielle Zugriffsprobleme auf Systemressourcen zu vermeiden.

Die Einführung dieser automatisierten Backup-Lösung stellt einen signifikanten Vorteil für das Systemmanagement dar, indem sie die Zuverlässigkeit der Datensicherung erhöht und gleichzeitig den administrativen Aufwand reduziert. Durch die Nutzung der Windows-Aufgabenplanung bietet das Skript eine einfache und effektive Methode, um sicherzustellen, dass die Daten des Ski-Service-Management-Systems regelmäßig und ohne menschliches Zutun gesichert werden.

Die Docker-Variante **register-auto-backup.docker.ps1** ruft das Basis-Registrierungs-Skript mit einem spezifischen Port für das Backup Skript auf, der für die Docker-Container-Umgebungen angepasst ist.

5.7 Entwicklung der Tests

In der Entwicklungsphase meines Projekts habe ich mich auf Integrationstests mit einer spezifischen Postman Collection konzentriert, um die Funktionalität und Zuverlässigkeit der Anwendung zu gewährleisten. Dieser Ansatz ermöglicht eine direkte Überprüfung aller API-Endpunkte, wodurch spezifische Controller-Tests und Mock-Daten überflüssig werden. Durch automatisierte Tests mit der Postman Collection kann ich die Qualität der Anwendung effizient und kontinuierlich überwachen.

6 Kontrollieren

6.1 Test-Strategie ausführen

In dieser Phase des Projekts lag der Fokus auf der Durchführung und Dokumentation von Tests, um die Funktionalität und Zuverlässigkeit der entwickelten API sicherzustellen. Die Teststrategie umfasste Integrationstests.

Integrationstests mit Postman
Die Integrationstests dienten dazu, die Interaktionen zwischen verschiedenen Komponenten der API zu überprüfen. Dies bot einen realistischeren Kontext für die Funktionsweise der API.

Die Ergebnisse der Integrationstests bestätigten die korrekte Funktionsweise der API bei der Verarbeitung integrierter Anfragen. Details zu diesen Tests sind ebenfalls im Testbericht enthalten.

Ort: files/Berichte/SkiService-Management.postman_test_run.json

Testbericht
Die Testergebnisse wurden sorgfältig dokumentiert und sind im angehängten Testbericht einsehbar. Dieser Bericht enthält detaillierte Informationen über die durchgeführten Tests, einschließlich der Testfälle, der erzielten Ergebnisse und der festgestellten Probleme.

6.2 Anforderungen mit dem Produkt abgleichen

In diesem Abschnitt führe ich einen Abgleich der Projektanforderungen mit den tatsächlich implementierten Lösungen durch. Mein Ziel war es, ein robustes, effizientes und benutzerfreundliches Ski-Service-Management-System zu entwickeln, das sich durch die Verwendung von NoSQL-Technologien auszeichnet.

Datenmigration und Benutzerkonzept (A1 & A2)
Erfolgreiche Migration der Datenbasis von relational zu NoSQL, ergänzt durch ein ausgefeiltes Benutzerkonzept mit verschiedenen Berechtigungsstufen, das Datenschutz und Datenintegrität gewährleistet.

Sicherheit und Datenzugriff (A3 & A4)
Implementierung von Sicherheitsmechanismen für Datenbankbenutzerzugänge und Einrichtung eines Schemas zur Sicherstellung der Datenkonsistenz, um die Integrität und Sicherheit der Daten zu garantieren.

Performanz und Skalierbarkeit (A5-A8)
Fokussierung auf Performanz Optimierung und Skalierbarkeit der Anwendung durch effiziente Indexierung, Optimierung von Abfragen und den Einsatz von Caching-Strategien.

Dokumentation und Qualitätssicherung (A9-A12)
Umfassende Dokumentation des Projekts und der Codebasis sowie Implementierung von Qualitätssicherungsmaßnahmen wie Tests zur Sicherstellung der Anwendungsqualität.

Zusatzanforderungen (AO1 & AO2)
Einrichtung eines automatisierten Backup-Konzepts zur Datenwiederherstellung und komplexe Schema-Validierungen zur Wahrung der Datenintegrität und -qualität bereitgestellt.

7 Auswerten

7.1 Fazit

Das Projekt basierte größtenteils auf dem vorherigen Backend, weshalb meine Möglichkeiten, neue Kenntnisse zu erwerben, begrenzt waren. Dennoch konnte ich sinnvolle Anpassungen am Backend-System vornehmen, die sich bei der Integration mit dem UI-Modul als vorteilhaft erwiesen haben.

Einen bedeutenden Lerngewinn erzielte ich jedoch im Umgang mit MongoDB. Bisher hatte ich diese Datenbank stets schema-los genutzt. Die Notwendigkeit, Schemas zu definieren, vertiefte mein Verständnis für die Datenbank erheblich.

Ein weiterer Lernbereich war der Umgang mit dem .NET-Paketmanager durch das Auslagern der Modelle in ein separates NuGet-Paket. Die Herausforderung, Modelle so zu gestalten, dass sie sowohl mit SQL als auch mit NoSQL kompatibel sind, war zwar keine explizite Anforderung, stellte aber eine interessante persönliche Herausforderung dar.

Insgesamt hat das Projekt meine Fähigkeiten im Umgang mit MongoDB sowie mein Verständnis von Interfaces und Klassen gestärkt.

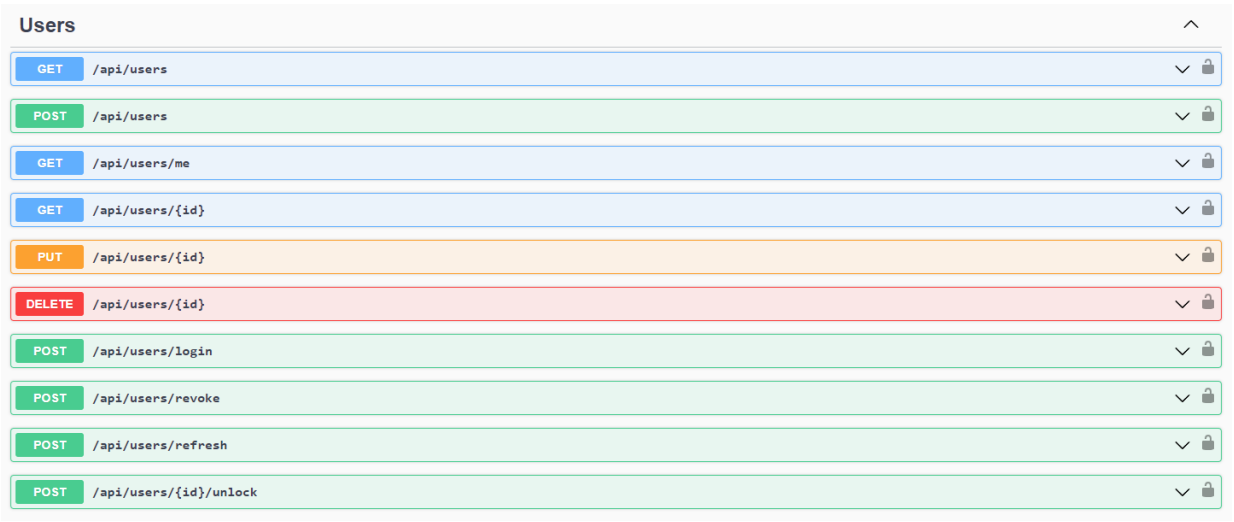
8 Anhänge

8.1 Verwendete NuGet-Pakete

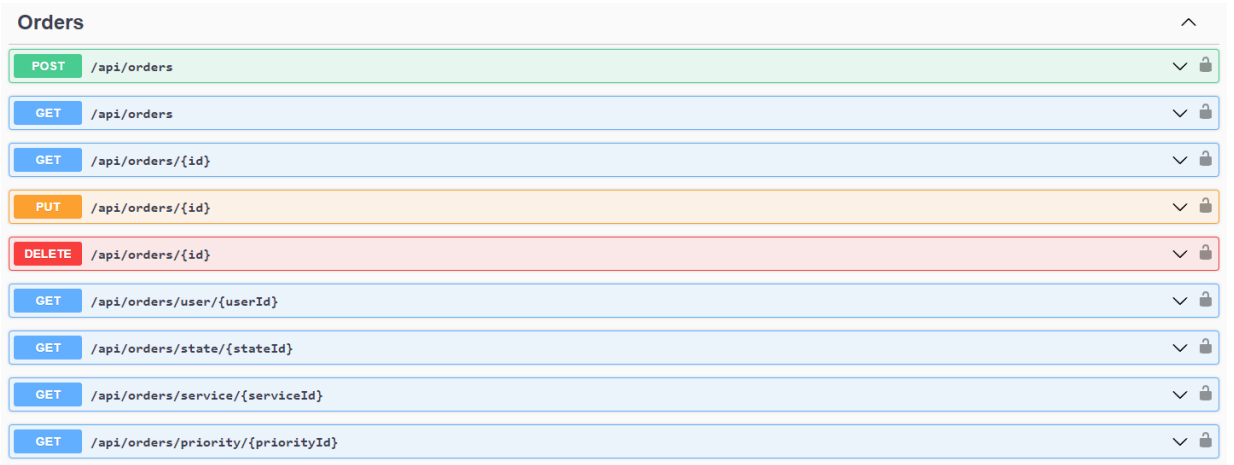
Paket	Version
AutoMapper	12.0.1
AutoMapper.Extensions.Microsoft.DependencyInjection	12.0.1
Microsoft.AspNetCore.Authentication.JwtBearer	8.0.1
Microsoft.AspNetCore.Mvc.NewtonsoftJson	8.0.1
Microsoft.IdentityModel.Tokens	7.3.0
MongoDB.Driver	2.23.1
Newtonsoft.Json	13.0.3
Serilog.AspNetCore	8.0.1
Serilog.Sinks.Console	5.0.1
Serilog.Sinks.File	5.0.0
SkiServiceModels.BSON	2.3.15
SkiServiceModels.BSON.AutoMapper	2.3.15
SkiServiceModels.BSON.Extensions	2.3.15
StackExchange.Redis	2.7.17
Swashbuckle.AspNetCore	6.5.0

3 - NuGet Paket Versionen

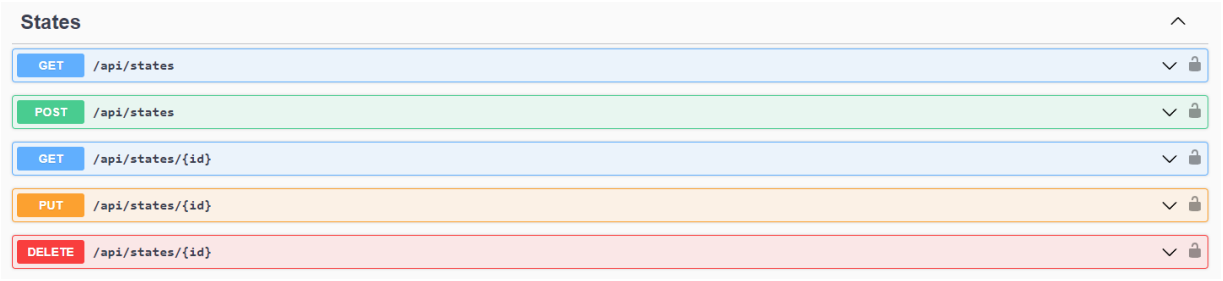
8.2 Swagger



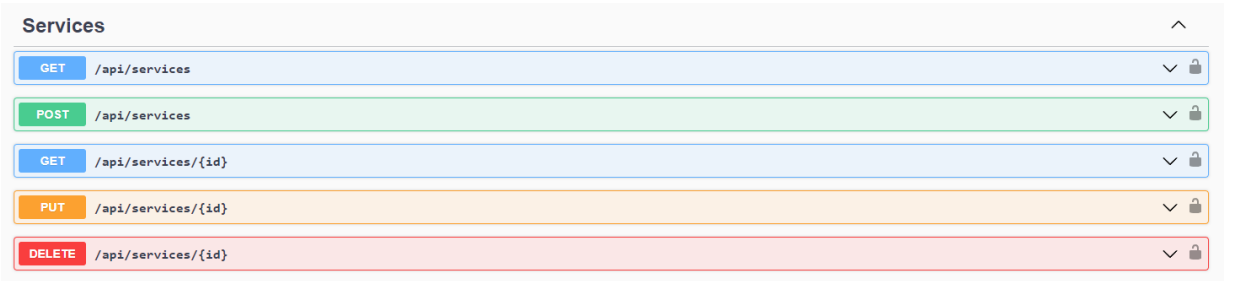
3 - Swagger UI - /api/users



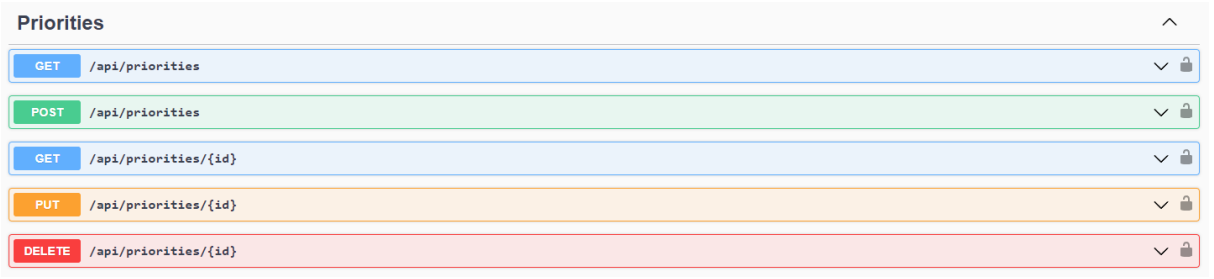
4 - Swagger UI - /api/orders



5 - Swagger UI - /api/states

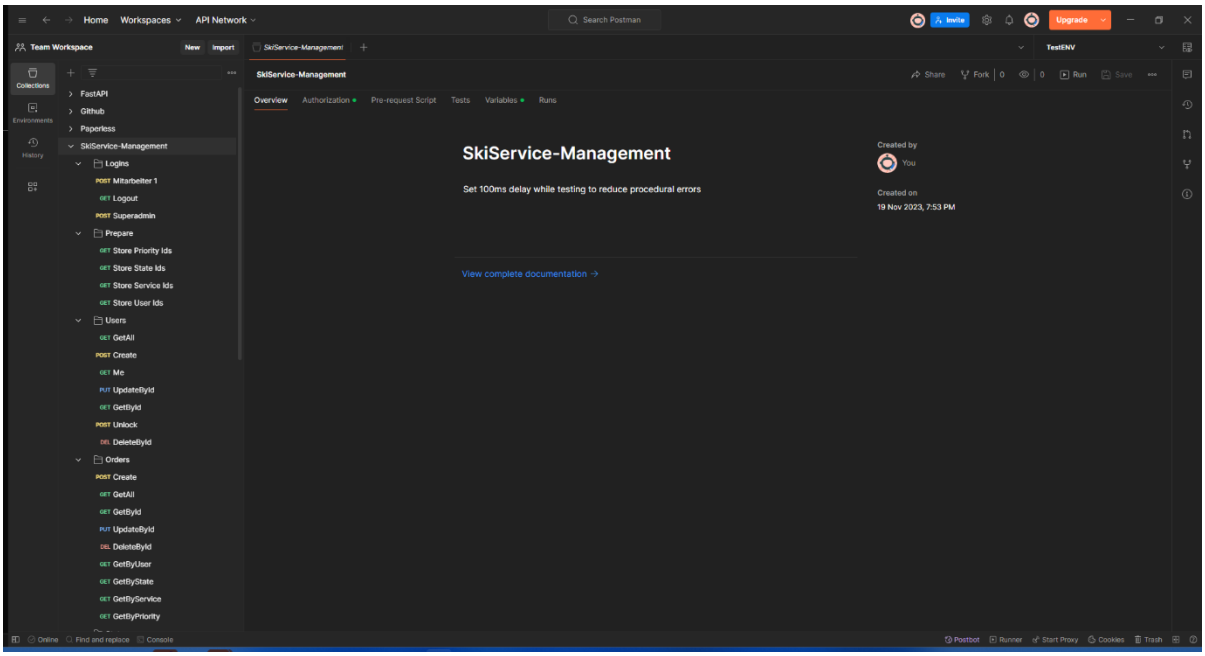


6 - Swagger UI - /api/services



7 - Swagger UI - /api/priorities

8.3 Postman



8 - Postman Collection - Übersicht

8.4 Glossar

Abkürzung	Ausgeschrieben	Erklärung
KMU	Kleine und mittlere Unternehmen	Unternehmen mit einer geringen Anzahl von Mitarbeitern und einem begrenzten Umsatzvolumen.
NoSQL	Not Only SQL	Eine Kategorie von Datenbankmanagementsystemen, die eine flexible Schemafreie Organisation von Daten ermöglicht.
API	Application Programming Interface	Eine Schnittstelle, die es ermöglicht, dass verschiedene Softwareanwendungen miteinander kommunizieren können.
ASP.NET	Active Server Pages .NET	Ein Framework für Webanwendungen, das von Microsoft entwickelt wurde.
UI	User Interface	Die Benutzeroberfläche einer Anwendung, durch die der Nutzer mit der Software interagiert.
JWT	JSON Web Token	Ein offener Standard zur sicheren Übertragung von Informationen zwischen Parteien als JSON-Objekt.
DTO	Data Transfer Object	Ein Objekt, das Daten zwischen Softwareanwendungsschichten überträgt, ohne Logik.
CRUD	Create, Read, Update, Delete	Grundlegende Operationen, die in Datenbankinteraktionen verwendet werden.
SQL	Structured Query Language	Eine standardisierte Programmiersprache, die zum Verwalten und Manipulieren relationaler Datenbanken verwendet wird.
HTTPS	Hypertext Transfer Protocol Secure	Eine Erweiterung des HTTP-Protokolls, die eine sichere Kommunikation über ein Computernetzwerk ermöglicht.
JSON	JavaScript Object Notation	Ein leichtgewichtiges Daten-Austauschformat, das für Menschen einfach zu lesen und zu schreiben ist.
Redis	Remote Dictionary Server	Ein In-Memory-Datenstrukturspeicher, der als Datenbank, Cache und Message Broker verwendet wird.
Serilog	-	Eine Bibliothek zum Protokollieren von Ereignissen in .NET-Anwendungen.
MongoDB	-	Eine NoSQL-Datenbank, die flexible Dokumentendatenmodelle verwendet.
Docker	-	Eine Plattform für die Entwicklung, den Versand und die Ausführung von Anwendungen in Containern.

4 - Glossar

9 Verweise

9.1 Quellen

SwaggerGen. (02. 02 2024). *Swagger*. Von Swagger Docs: https://localhost:7137 abgerufen

9.2 Tabellen

9.3 Abbildung