Ski-Service Management noSQL

Projektdokumentation nach IPERKA

IBZ - Basel

Fokko Vos

Inhaltsverzeichnis

[1 Versionsverlauf 1](#_Toc157594338)

[2 Informieren 1](#_Toc157594339)

[2.1 Ausgangsituation 1](#_Toc157594340)

[2.2 Anforderungsanalyse 1](#_Toc157594341)

[2.2.1 Funktionale Anforderungen 1](#_Toc157594342)

[2.2.2 Nicht-Funktionale Anforderungen 1](#_Toc157594343)

[2.3 Technische Anforderungen 1](#_Toc157594344)

[3 Planen 2](#_Toc157594345)

[3.1 Zeitplanung 2](#_Toc157594346)

[3.2 Datenbankentwurf 3](#_Toc157594347)

[3.3 Systemarchitekturentwurf 4](#_Toc157594348)

[3.4 API-Endpunkte 5](#_Toc157594349)

[4 Entscheiden 6](#_Toc157594350)

[4.1 Test-Strategie 6](#_Toc157594351)

[4.2 Code-First Anpassung an NoSQL 6](#_Toc157594352)

[5 Verweise 7](#_Toc157594353)

[5.1 Quellen 7](#_Toc157594354)

[5.2 Tabellen 7](#_Toc157594355)

[5.3 Abbildung 7](#_Toc157594356)

# Versionsverlauf

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Autor | Datum | Änderung |
| 1.0 | Fokko Vos | 31.01.2024 | Erstellung des Dokumentes und der Planung |

# Informieren

## 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.

## Anforderungsanalyse

### Funktionale Anforderungen

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

### Nicht-Funktionale Anforderungen

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

## 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.

# Planen

## 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** | **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 | Überprüfung der API-Endpunkte | 1 | 1 |
| **3** | **Entscheiden** | **1** |  |
| 3.1 | Test-Strategie anpassen | 0.5 | 0.5 |
| 3.2 | Festlegen der Migrationstechnik (NoSQL) | 0.5 | 0.5 |
| **4** | **Realisieren** | **25.5** |  |
| 4.1 | Grundlegendes Projekt erstellen | 1 | 1 |
| 4.2 | Git-Repository aufsetzen | 0.5 | 0.5 |
| 4.3 | Erstellung des Initialisierung Skripts | 3 | 4 |
| 4.4 | Implementierung der NoSQL-Datenbank | 4 | 5 |
| 4.5 | Entwicklung des Backend (NoSQL-Anpassungen) | 5 |  |
| 4.6 | Erstellung von Backup- und Restore-Strategien | 3 |  |
| 4.7 | Entwicklung angepasster Tests | 3 |  |
| **5** | **Kontrollieren** | **3** |  |
| 5.1 | Test-Strategie ausführen | 1 |  |
| 5.2 | Anforderungen mit dem Produkt abgleichen | 2 |  |
| **6** | **Auswerten** | **4** |  |
| 6.1 | Finalisierung der Dokumentation | 2 |  |
| 6.2 | Lessons-Learned identifizieren | 1 |  |
| 6.3 | Abschlusspräsentation vorbereiten | 1 |  |
| **Gesamt** | | **37.5** |  |

- PSP Zeitplanung

## Datenbankentwurf

1 - Datenbankentwurf

Die überarbeiteten NoSQL-Modelle reflektieren ein dokumentenorientiertes Design, das für hohe Leistung und Erweiterbarkeit optimiert ist. Jede Entität wie Services, Prioritäten und Zustände wird als eigenständiges Dokument gespeichert, wobei Referenzen auf andere Dokumente die Beziehungen abbilden.

Die Einzigartigkeit der Benutzer wird durch das Username-Feld sichergestellt, während Rollen und Zugriffsberechtigungen durch das Rollenattribut definiert sind. Aufträge können mit Benutzern verbunden sein, wobei diese Zuordnung flexibel ist und die Möglichkeit besteht, dass ein Auftrag keinen oder einen spezifischen Benutzer hat. Jeder Auftrag muss einem Service, einer Priorität und einem Zustand zugeordnet werden, wobei die Möglichkeit besteht, dass mehrere Aufträge dieselben Referenzen teilen.

Soft-Delete wird durch das Attribut 'is\_deleted' in jedem Dokument umgesetzt, was das Markieren von Dokumenten als gelöscht ermöglicht, ohne sie physisch zu entfernen, was wiederum Audits und eventuelle Wiederherstellungen unterstützt.

## 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.

## 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 | - |

- 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.

# Entscheiden

## Test-Strategie

Für das Projekt habe ich geplant, für die verschiedenen API-Endpoints spezifische Unit Tests zu implementieren. Diese Tests werden sich darauf konzentrieren, die Funktionalität der einzelnen Komponenten der API zu überprüfen. Dabei wird überprüft, ob die Endpoints die erwarteten Ergebnisse liefern, korrekt mit Eingabeparametern umgehen und geeignete Reaktionen auf unterschiedliche Anfrageszenarien zeigen.

Im Rahmen des schulischen Projekts liegt der Schwerpunkt auf Unit Tests, um die Funktionalität der einzelnen Komponenten zu validieren. Dabei wird das in Visual Studio integrierte Testframework genutzt. Zusätzlich werden grundlegende Integrationstests mit Postman durchgeführt, um die Interaktionen zwischen verschiedenen Komponenten zu überprüfen. Dies bietet einen realistischeren Kontext für die Tests, ohne den Rahmen des Projekts durch umfassende End-to-End-Tests zu überschreiten.

## Code-First Anpassung an NoSQL

Für das Ski-Service Management Projekt, das ursprünglich mit einem Code-First-Ansatz unter Verwendung des Entity Frameworks entwickelt wurde, wird nun ein Wechsel zu NoSQL, spezifisch MongoDB, vollzogen. Dieser Ansatz passt sich dem veränderten Anforderungsprofil an und bietet eine hohe Flexibilität und Effizienz in der Datenhandhabung.

Beim Start der Anwendung wird nun der MongoDB-Treiber verwendet. Dieser überprüft zunächst, ob Initialdaten in der Datenbank vorhanden sind. Falls keine Daten vorhanden sind, werden diese automatisch erstellt. Diese Vorgehensweise ermöglicht eine dynamische Datenverwaltung und erleichtert den Umgang mit Daten im NoSQL-Umfeld.

Zusätzlich müssen vor dem ersten Start der Anwendung Validator-Schemas und Indizes erstellt werden. Dies erfolgt durch ein PowerShell-Skript, das eine effiziente und automatisierte Einrichtung der Datenbankstruktur ermöglicht. Dieser Schritt ist notwendig, um die Datenintegrität und optimierte Abfrageleistung in MongoDB zu gewährleisten.

Durch diese Änderungen wird das Backend des Projekts effizienter und besser an die Anforderungen eines modernen NoSQL-basierten Systems angepasst, was eine schnellere Entwicklung und flexiblere Datenmodellierung ermöglicht.

# Verweise

## Quellen

SwaggerGen. (21. 11 2023). *Swagger*. Von Swagger Docs: https://localhost:7155 abgerufen

## Tabellen

[1 - PSP Zeitplanung 2](#_Toc151496444)

[2 - API Endpunkte 5](#_Toc151496445)

[3 - NuGet Paket Versionen 14](#_Toc151496446)

## Abbildung

[1 - Datenbankentwurf 3](file:///C:\Github\295-ski-service-management\files\Dokumentation.docx#_Toc151496447)

[2- Systemarchitekturentwurf 4](file:///C:\Github\295-ski-service-management\files\Dokumentation.docx#_Toc151496448)

[3 - Swagger UI - /api/users 15](#_Toc151496449)

[4 - Swagger UI - /api/orders 15](#_Toc151496450)

[5 - Swagger UI - /api/states 15](#_Toc151496451)

[6 - Swagger UI - /api/services 15](#_Toc151496452)

[7 - Swagger UI - /api/priorities 16](#_Toc151496453)

[8 - Postman Collection - Übersicht 16](#_Toc151496454)