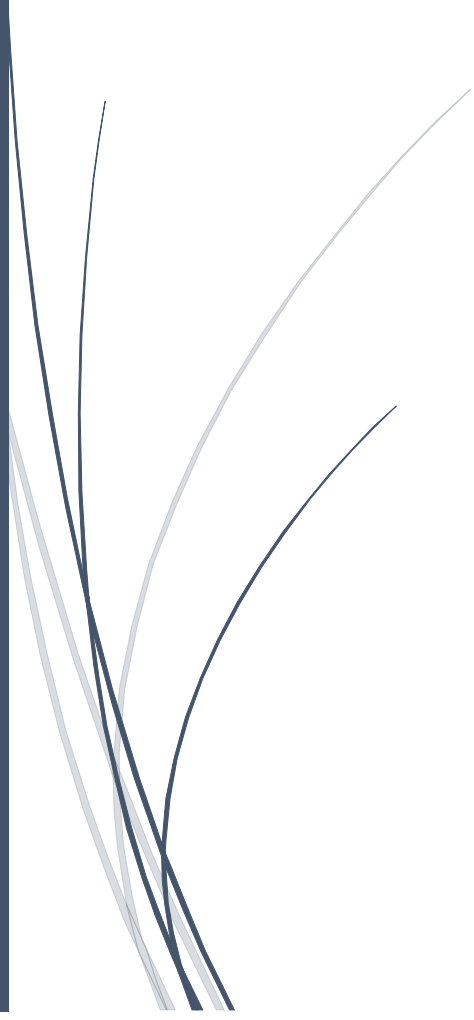
4.2.2023

Dokumentation: Ski-Service API

Modul 165 – NoSQL-Datenbanken einsetzen



Simon Stauffer

Inhaltsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis (Chronologisch)

API	Application Programming Interface
SQL	Structured Query Language
NoSQL	Not (only) SQL
VS	Visual Studio
JSON	JavaScript Object Notation
JWT	JSON Web Tokens

Einleitung

Das Projekt «Ski-Service API» zielte darauf ab, ein Backend zu erstellen. Dieses Projekt umfasst folgenden Punkte:

- Datenbankdesign und Implementierung (NoSQL)
- Datenmigration (SQL ➔ NoSQL)
- Migration WebAPI Projekt
- Testprojekt / Testplan
- Realisierung der kompletten Anwendung, gemäss den Anforderungen
- Durchführung Integrationstest mit bestehender Frontend Lösung.

Das Projekt wurde von Simon Stauffer realisiert und dokumentiert. Die Dokumentation, sowie die Umsetzung des Ski-Service API basieren auf dem IPERKA-Modell. Das Vorgehen wurde phasenweise dokumentiert.

Verwendet Tools:

Als IDE wurde Visual Studio 2022 (VS) 17.4.4 genutzt. Zusätzlich wurden mehrere NuGets von VS verwendet:

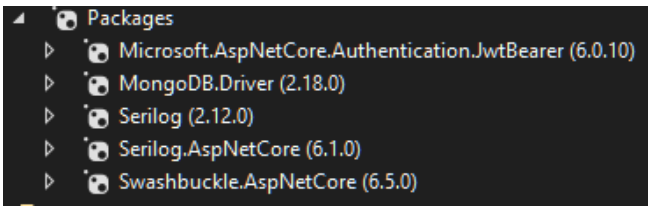


Abbildung 1: Nuget Liste

Für die Datenbank wurde MongoDB mit allen Tools, die MongoDB zur Verfügung stellt, verwendet. Für die Dokumentation der API Swagger und um diese zu testen, Postman. Als Plattform, um den Code versionieren zu können, habe ich GitHub benutzt. Als Client wurde Windows 11Pro Build 22621 verwendet.

Informieren

Als Erstes las ich den Arbeitsauftrag durch und besprachen bestehende Unklarheiten mit Herrn Müller. Nachdem die Unklarheiten geklärt waren, inspizierten ich die im Arbeitsauftrag beschriebenen Schritte zur Umsetzung des Projektes. Danach überlegte ich mir Möglichkeiten, um diese umzusetzen. Weiter haben wir uns über verschiedene Tools informiert, die wir für die Umsetzung des Projekts benutzen werden. Zusätzlich setzten wir uns mit den Bewertungskriterien auseinander. Anschliessend habe ich mich im Internet nach Möglichkeiten gesucht, wie ich dieses Projekt umsetzen könnte.

Projektumfeld

Die Firma Jetstream-Service führt als KMU in der Wintersaison Skiservicearbeiten durch und hat in den letzten Jahren grosse Investitionen in eine durchgängige digitale Auftragsanmeldung und Verwaltung, bestehend aus einer datenbankbasierenden Web-Anmeldung und Auftragsverwaltung getätigt. Aufgrund guter Auftragslage hat sich die Geschäftsführung für eine Diversifizierung mit Neueröffnungen an verschiedenen Standorten entschieden.

Die bis anhin eingesetzte relationale Datenbank genügt den damit verbundenen Ansprüchen an Datenverteilung und Skalierung nicht mehr. Um einerseits den neuen Anforderungen gerecht zu werden sowie anderseits Lizenzkosten einzusparen, soll im Backend der Anwendung die Datenbank auf ein NoSQL Datenbanksystem migriert werden.

Projektorganisation

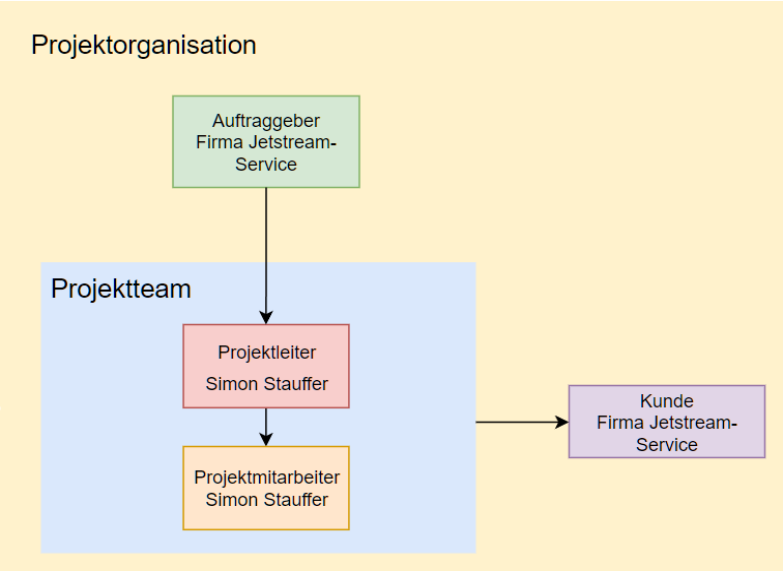


Abbildung 2: Projektorganisation

Projektziele

Das Projekt wird mit dem Ziel durchgeführt, dass die Firma Jetstream-Service einen einfacheren Umgang mit ihren Kundendaten haben. Die horizontale Skalierungsmöglichkeit wird durch die Verwendung von MongoDB gewährleistet. So wird auch garantiert, dass die neu eröffneten Filialen kein Problem mit der Datenbank haben werden.

Planen

Die Projektplanung werde ich nach dem IPERKA Modell machen. Der Grund dafür ist, dass ich schon viel mit diesem Modell gearbeitet habe und weil es für mich die beste Methode ist, um ein Projekt zu planen.

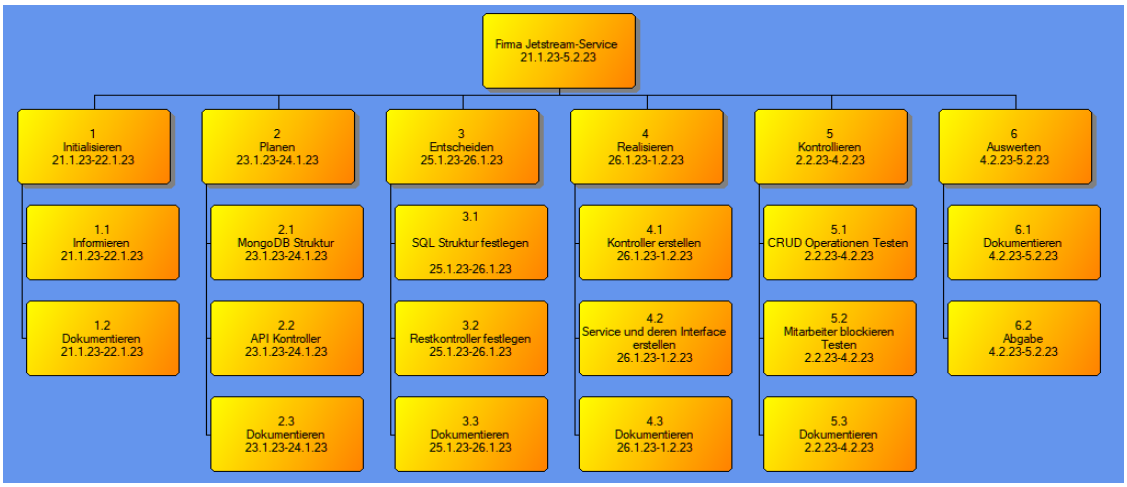


Abbildung 3: Projektplanung

Ablauf- und Terminplanung

1. Arbeitspakete erstellen
2. Reihenfolge der Arbeitspakete erstellen
3. Vorgangsdauer und Aufwand pro Arbeitspaket definieren.
4. Gesamtlänge und Gesamtaufwand des ganzen Projektes schätzen.

Nachdem ich diese Arbeitstakte erstellt, deren Reihenfolge und Aufwand geschätzt habe, entsteht folgende Planung:

Tabelle 1: Grober Zeitplan der einzelnen Phasen

Projektphase	Geplante Zeit
Informieren	**2 h**
Planen	**1 h**
Entscheiden	**0.5 h**
Realisieren	**4 h**
Kontrollieren	**1 h**
Auswerten	**4 h**
Gesamt	**12.5 h**

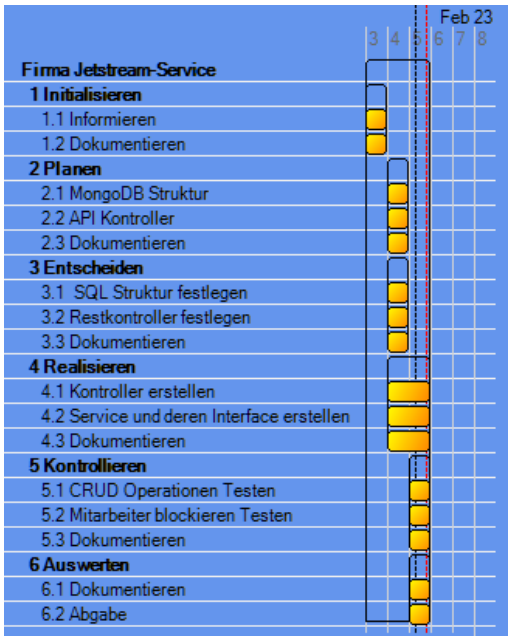


Abbildung 4:GANTT für das Projekt

Datenbank Möglichkeiten

MongoDB ist eine NoSQL-Datenbank, die auf Dokumentenorientierung basiert. Das bedeutet, dass Daten in Form von Dokumenten gespeichert werden. Hierdurch ist es möglich, Daten einfach und flexibel zu modellieren, ohne dass es einer vorherigen Festlegung eines Schemas bedarf. Ausserdem bietet MongoDB eine hohe Skalierbarkeit.

Im Gegensatz dazu ist Neo4j eine Graphen-Datenbank, die auf der Verwendung von Knoten und Beziehungen zur Modellierung von Daten basiert. Es ist hervorragend für die Verwaltung von Beziehungen und Netzwerken geeignet, aber kann Schwierigkeiten haben, bei der Verarbeitung von Datenmengen oder bei der Skalierung auf mehrere Knoten.

SQL-Daten in NoSQL Datenbank migrieren

Da ich schon eine laufende API habe, welches mit einer SQL-Datenbank verbunden ist, konnte ich so planen, dass ich die Daten einfach über die API abrufe, sie kopiere und anschliessend in die NoSQL-Datenbank einzufügen.

Entscheiden

Aufgrund der Anforderungen des Projekts, bei dem es darum geht, grosse Datenmengen flexibel zu speichern und skalierbar zu sein, habe ich mich für MongoDB entschieden. Bei der Struktur der Dokumente habe ich mich für die simpelste Möglichkeit entschieden. So wird für jede Bestellung ein eigenes Dokument erstellt. Ich bin davon ausgegangen, dass die wenigsten Kunden mehrere Bestellungen machen werden und somit sich die Redundanz in Grenzen hält.

Für das Backup und Restore habe ich mich für die Tools "mongoimport" und "mongorestore" von MongoDB entschieden. Um die Tools aufrufen zu können, habe ich mich für das Benutzen von PowerShell entschieden.

Ausserdem habe ich mich dafür entschieden, dass ich den Grossteil der Logik in den Service-Klassen machen werde, um es übersichtlicher und auch einheitlicher zu halten. Für die Sicherheit habe ich mich gegen einen API-Key und für das JSON Web Token entschieden.

Realisieren

Datenbank

In dieser Phase begann ich mit dem eigentlichen Programmieren. Ich habe ein neues Web API Projekt erstellt und dieses sogleich mit GitHub synchronisiert. Die Struktur der Datenbank ist wie Folgt:

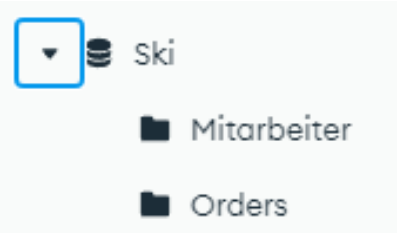


Abbildung 5: Datenbank mit zwei Collections

```
//Index erstellen
db.Orders.createIndex ({"Status": 1,})
db.Orders.createIndex ({"Priorität": 1,})
```

Abbildung 6: Index für die Collection Orders

```
// Collection erstellen und Validierung einfügen
db.createCollection( "Orders", { validator: {
    "$jsonSchema": {
        "bsonType": "object",
        "required": ["Kundenname", "Email", "Telefon", "Priorität", "Dienstleistung", "Status", "Erfassungsdatum", "Abholdatum"],
        "properties": {
            "Kundenname": {
                "bsonType": "string",
                "minLength": 1
            },
            "Email": {
                "bsonType": "string",
                "pattern": "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
            },
            "Telefon": {
                "bsonType": "string",
                "pattern": "^[0-9]*$"
            },
            "Priorität": {
                "bsonType": "string",
                "enum": ["Tief", "Standart", "Express"]
            },
            "Dienstleistung": {
                "bsonType": "string",
                "enum": ["Kleiner Service", "Grosser Service", "Rennski-Service", "Bindung montieren und einstellen", "Fell zuschneiden", "Heisswachsen"]
            },
            "Status": {
                "bsonType": "string",
                "enum": ["Offen", "In Bearbeitung", "Abgeschlossen"]
            },
            "Erfassungsdatum": {
                "bsonType": "date"
            },
            "Abholdatum": {
                "bsonType": "date"
            }
        }
    }
} } )
```

Abbildung 7: Schema für die Collection "Orders"

Für die Administration der Datenbank habe ich drei User erstellt.

```
//User erstellen und Rechte vergeben
use admin
db.createUser({user: "admin", pwd: "admin", roles: ["userAdminAnyDatabase", "readWriteAnyDatabase"]})
use urs
db.createUser({user: "urs", pwd: "1234", roles: [{role: "readWrite", db: "Ski"}]})
use peter
db.createUser({user: "peter", pwd: "1234", roles: [{role: "read", db: "Ski"}]})
```

Abbildung 9: MongoDB User-erstellen

Um die Autorisation zu aktivieren, musste ich die Konfiguration von MongoDB anpassen. Diese findet man im MongoDB Installationsordner unter " MongoDB\Server\6.0\bin\mongod.cfg".

```
security:
  authorization: enabled
```

Abbildung 8: Autorisation aktivieren

API

Der nächste Schritt war das Erstellen der Kontroller Klassen. Jeweils eine für die Registrationen und die Mitarbeiter. Anschliessen habe ich Interfaces der Services und die einzelnen Services erstellt und deren Logik geschrieben. Hier ein Beispiel eines Kontrollers:

```
[Authorize]
[Route("[controller]")]
[ApiController]
3 references | 0 changes | 0 authors, 0 changes
public class OrdersController : ControllerBase
{
    private readonly ILogger<OrdersController> _logger;
    private readonly IOrdersService _ordersService;
    0 references | 0 changes | 0 authors, 0 changes
    public OrdersController(IOrdersService ordersService, ILogger<OrdersController> logger)
    {
        _ordersService = ordersService;
        _logger = logger;
    }

    /// <summary> Alle Registrationen
    [AllowAnonymous]
    [HttpGet ("All")]
    2 references | 0 changes | 0 authors, 0 changes
    public ActionResult<List<Orders>> Get()
    {
        try
        {
            return _ordersService.Get();
        }catch (Exception ex)
        {
            _logger.LogWarning($"Warning --> {ex.Message}");
            return NotFound($"Warning --> {ex.Message}");
        }
    }
```

Abbildung 10: Kontroller für Orders

Und die dazu passende Methode in der Service-Klasse, die im Kontroller aufgerufen wird:

```
private readonly IMongoCollection<Orders> _orders;

0 references | 0 changes | 0 authors, 0 changes
public OrdersService(ISkiDatabaseSettings settings)
{
    var client = new MongoClient(settings.ConnectionString);
    var database = client.GetDatabase(settings.DatabaseName);

    _orders = database.GetCollection<Orders>(settings.OrdersCollectionName);
}

2 references | 0 changes | 0 authors, 0 changes
public List<Orders> Get() =>
    _orders.Find(order => true).ToList();
```

Abbildung 11: Methode, um alle Registrationen abzurufen.

In dem Konstruktor des Kontrollers gebe ich über Dependency Injection das Interface der dazugehörigen Service-Klasse mit, um eine Verbindung der beiden Klassen herzustellen. Im Kontroller rufe ich dann die Methode Get() in der Klasse OrdersService auf. In dieser Methode wird eine Verbindung zu der Datenbank hergestellt. Dieses Spiel habe ich für alle CRUD-Methoden wiederholt und ich werde hier nicht weiter darauf eingehen.

Damit die erstellten Services geladen werden, müssen wir diese in program.cs zuerst erstellen. Dies sieht wie folgt aus.

```
builder.Services.Configure<SkiDatabaseSettings>(
    builder.Configuration.GetSection(nameof(SkiDatabaseSettings)));

builder.Services.AddSingleton<ISkiDatabaseSettings>(sp =>
sp.GetRequiredService<IOptions<SkiDatabaseSettings>>().Value);

builder.Services.AddSingleton<IMongoClient>(s =>
new MongoClient(builder.Configuration.GetValue<string>("SkiDatabaseSettings:ConnectionString")));

builder.Services.AddSingleton<IOrdersService, OrdersService>();
builder.Services.AddSingleton<ITokenService, TokenService>();
builder.Services.AddSingleton<IMitarbeiterService, MitarbeiterService>();
```

Abbildung 12: Erstellen der Services

Für die Sicherheit habe ich mich, wie bereits beschrieben, für ein JWT entschieden. Dies konnte ich als vorgefertigte Klasse übernehmen und musste es nur noch in das Projekt implementieren, deshalb werde ich auch auf das nicht weiter eingehen. Wichtig ist auch hier das Initialisieren in program.cs.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"])),
            ValidAudience = builder.Configuration["Jwt:Audience"],
            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidateIssuer = false,
            ValidateAudience = false
        };
    });
```

Abbildung 13: JWT-Initialisierung

Um mich mit der Datenbank verbinden und mich anmelden zu können, habe ich folgendes in das appsettings.json geschrieben. Hier habe ich das Passwort im Klartext. Ich habe mich gegen die Methode, die ich das letzte Mal verwendet habe, entschieden, da diese nicht zu meiner Zufriedenheit funktionierte.

```
"SkiDatabaseSettings": {
  "OrdersCollectionName": "Orders",
  "MitarbeiterCollectionName": "Mitarbeiter",
  "ConnectionString": "mongodb://admin:admin@localhost:27017",
  "DatabaseName": "Ski"
},
"Jwt": {
  "Key": "slajbdaksfjbalkblakscblaksjbkasjblksjblakjbsalkbkjbaksjbcjbsakjcbalks",
  "Issuer": "JWTAuthenticationServer",
  "Audience": "JWTServicePostmanClient"
},
```

Abbildung 14: appsettings.json Beinhaltet die benötigten Informationen für die Verbindung und den JWT

Für die Anmeldung der Mitarbeiter, habe ich dieselbe Methode wie im letzten Projekt verwendet und musste diese nur anpassen, dass die Manipulationen gegen eine MongoDB geht und nicht mehr gegen eine SQL-DB.

Hier ein Überblick über das ganze API:

Abbildung 15: Klassen-Diagramm von API

Backup & Restore

Für das Backup und Restore habe ich jeweils ein PowerShell-Skript gemacht, bei dem der User entscheiden kann, welche Datenbank wohin gespeichert werden soll oder welches Backup in welche Datenbank importiert werden soll. Diese Skripts laufen leider, nur wenn ich sie in der PowerShell ISE ausführe, wenn ich sie einfach mit der PowerShell ausführe, gelingt mir die Verbindung zu der Datenbank nicht. Ich habe keine Ahnung, woran das liegt.

```
# Abfrage nach Datenbankname
$allDatabases = mongosh -u "admin" -p "admin" --authenticationDatabase "admin" --quiet --eval "show dbs"
$databaseList = $allDatabases -split "`n" | Select-Object -Skip 2
Write-Host "Vorhandene Datenbanken auf localhost:"
foreach ($database in $databaseList) {
    Write-Host $database
}
$database = Read-Host "Welche Datenbank möchten Sie sichern?"

# Öffne Explorer, um Pfad und Namen für Backup-Datei auszuwählen
$saveFile = [System.Windows.Forms.SaveFileDialog]::new()
$saveFile.Filter = "Bson-Files (*.bson)|*.bson|All Files (*.*)|*.*"
$saveFile.ShowDialog() | Out-Null
$backupPath = $saveFile.FileName

# Erstelle Backup mit mongodump-Befehl
mongodump  -u "admin" -p "admin" --authenticationDatabase "admin" --db $database --out $backupPath
Write-Host "Das Backup der Datenbank $database wurde erfolgreich im Pfad $backupPath gespeichert."
```

Abbildung 16: Backup-Skript für den User

```
$fbd = New-Object System.Windows.Forms.FolderBrowserDialog
$fbd.Description = "Wählen Sie den Restore-Ordner"
$fbd.RootFolder = "Desktop"
$fbd.ShowNewFolderButton = $false
if ($fbd.ShowDialog() -eq "OK")
{
  $folderPath = $fbd.SelectedPath
  Write-Host "Der ausgewählte Ordnerpfad ist: $folderPath"
}
$database = Read-Host "Wie heisst die Datenbank die Sie wiederherstellen möchten?"

if ($folderPath) {
  mongorestore -u "admin" -p "admin" --authenticationDatabase "admin" --db $database $folderPath
}
```

Abbildung 17: Restore-Skript für den User

Damit ich eine automatische Backuplösung machen konnte habe ich noch ein weiteres, einfacheres Skript geschrieben, welches auch auf der PowerShell läuft.

```
$date = Get-Date -Format "dd/MM/yyyy"
$backup = $Env:HOMEPATH + "\MongoBackup" + $date
mongodump -u "admin" -p "admin" --authenticationDatabase "admin" --db Ski --out $backup
```

Abbildung 19: Backup-Skript für das automatische Backup

Um dieses Skript zu automatisieren, musste ich unter Aufgabenplanung eine neue Aufgabe erstellen.

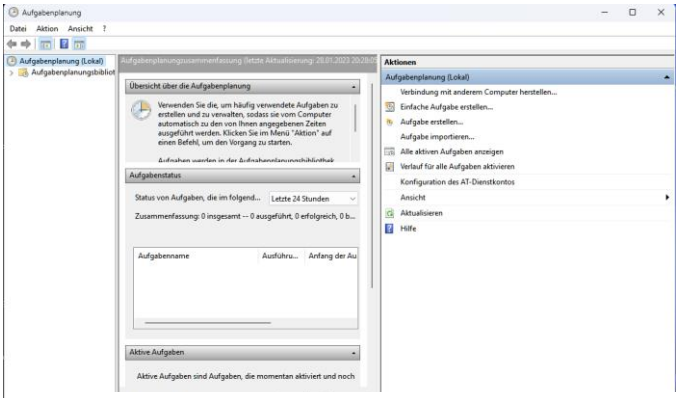


Abbildung 18:Aufgabenplanung

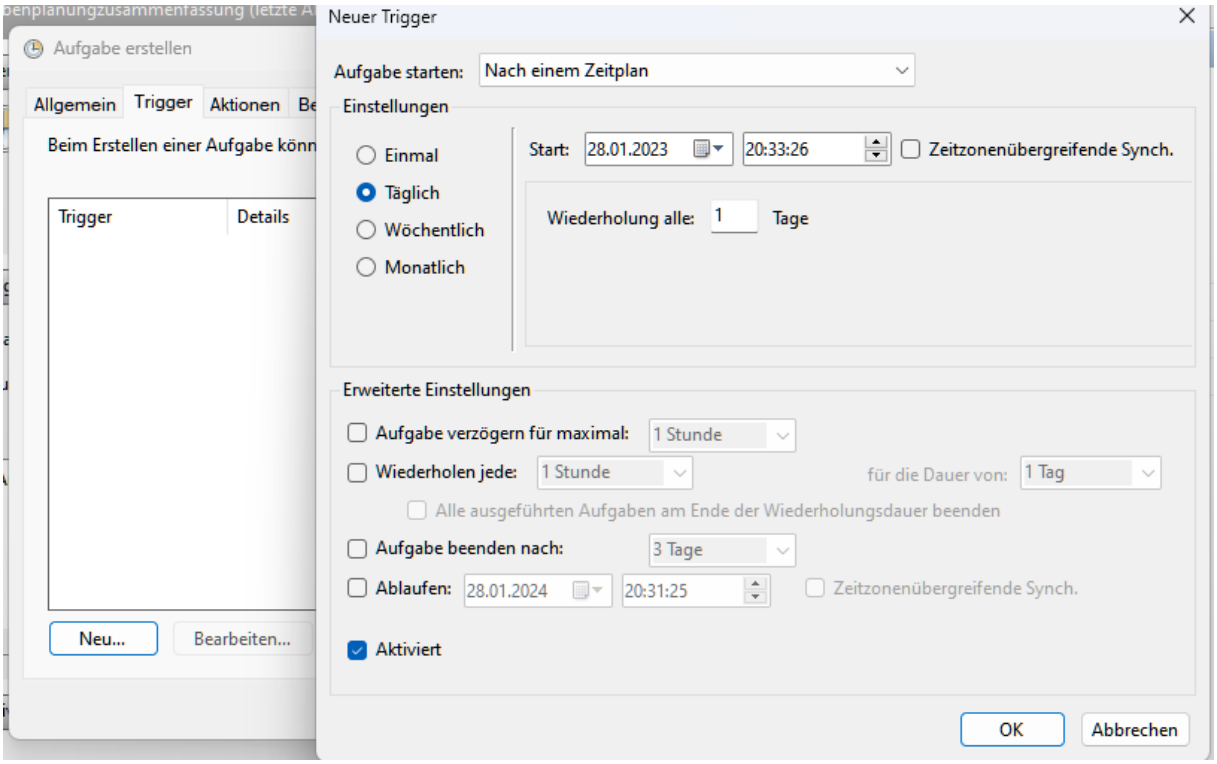


Abbildung 20: Trigger Einstellungen für die Aufgabe

Unter dem Reiter Trigger konnte ich einen neuen Auslöser des Skripts einstellen. Hier habe ich mich für ein tägliches um 20.33 Uhr Ausführen des Skripts entschieden.

Zum Schluss musste ich unter dem Reiter Aktion einstellen, was den ausgeführt werden soll. Unter Programm habe ich die PowerShell angegeben und unter Argument den Pfad des Backup-Skriptes.

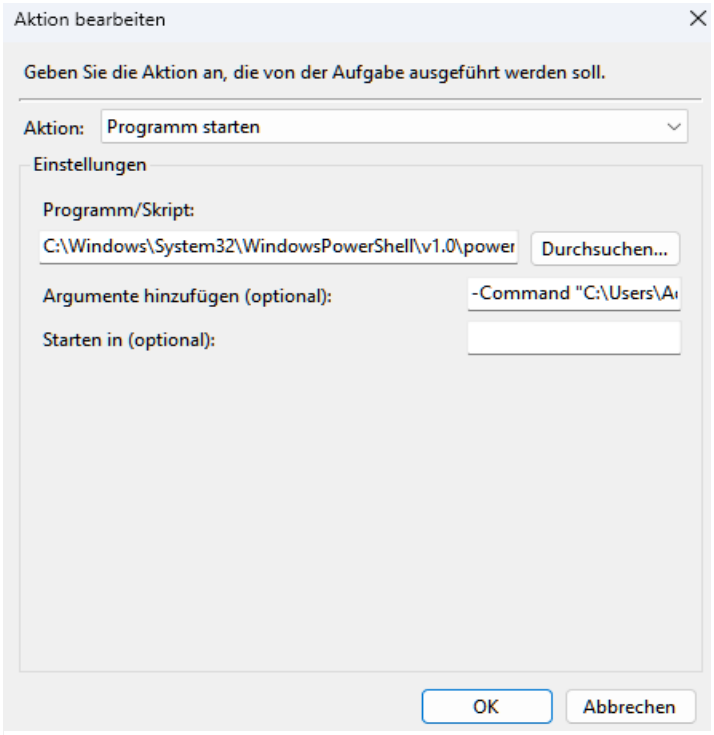


Abbildung 21: Aktion die die Aufgabe auslösen soll

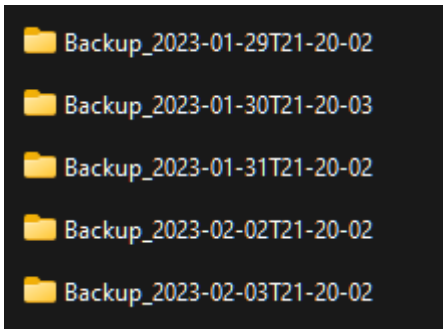


Abbildung 22: Backups die automatisch gemacht wurden

Kontrollieren

Nachdem ich eine neue Methode implementiert habe, habe ich diese sogleich getestet. Dies mit allen CRUD-Methoden und den zusätzlichen, wie zum Beispiel das Blockieren eines Mitarbeiters oder auch das Erstellen des JWT. Zum Testen habe ich anfangs nur Swagger über den Browser und später Postman mit dem jeweiligen JWT benutzt.

Alle Skripte wurden getestet und wie bereits erwähnt, liefen die User-Skripte nur in der PowerShell ISE und ich weiss bis heute nicht wieso. Das Skript für das automatische Backup wurde durch die von mir erstellte Aufgabe getestet und wie auf Abb. 21 ersichtlich ist, hat dies auch funktioniert.

Auswerten

Nach der Fertigstellung des Projektes habe ich mir Gedanken zu den positiven und negativen Momenten bei der Realisierung gemacht.

Ich hatte Anfangs grosse Probleme mit dem Management der Datenbank. Ich konnte keine Backups von einzelnen Datenbanken machen, weil ich anscheinend die Berechtigung nicht hatte, obwohl ich mich als "admin" angemeldet hatte. Erst als ich MongoDB erneut installiert und die Datenbanken und User erstellt hatte ging es ohne Probleme. Dieses Problem hatte ich nur am PC auf dem Laptop, hat es immer funktioniert.

Ich habe viel Zeit für die PowerShell Skripts verwendet und weniger auf die API selber, da ich diese schon einmal gemacht hatte. Deshalb, habe ich das Ändern des Status nicht mehr separat gemacht wie das letzte Mal. Da dies einfach mehr Zeit für wenig neues gebraucht hätte.

Fazit

Wie bei jedem Projekt, bei dem das Programmieren die Hauptsache ist, hatte ich auch bei diesem viel Spass. Leider konnte ich nicht so viel machen wie ich es gerne gemacht hätte, da wir in den letzten Wochen sehr viele andere Arbeiten zu erledigen hatten. Das Arbeiten mit NoSQL Datenbanken war für mich eine Freude, da dies viel eher meinem Verstand entspricht als SQL-Datenbanken, bei denen ich mich nicht so wohl fühle. Für mich ist das Arbeiten mit NoSQL Datenbanken, sei es MongoDB oder auch Neo4J viel einfacher als mit SQL-Datenbanken. Ich hoffe, dass ich in Zukunft noch mehr mit NoSQL-Datenbanken arbeiten werde.

Testprotokoll

Tabelle 2: Testprotokoll

ID	Was wir Testen	Wie wir testen	Erwartetes Ergebnis	Tatsächliches Ergebnis	Testergebnis	Wurde es gelöst	Wie wurde es gelöst
1	Login der Mitarbeiter	Falsche Angaben	Fehlermeldung bei falsch Eingabe	Fehlermeldung "Falsche Angaben"	Fehlerfrei	-	
2	Login der Mitarbeiter	Falsches Passwort	Falsche Angaben und +1 Zähler	Zähler blieb bei Null	Fehlerhaft	Ja	Zähler in DB gespeichert
3	Login der Mitarbeiter	Richtige Angaben	JWT erhalten	JWT erhalten	Fehlerfrei	-	
4	Mitarbeiter deblockieren	Postman Put Mitarbeiter	Zähler wird auf 0 gesetzt	Zähler wurde auf 0 gesetzt	Fehlerfrei	-	
5	Get All Mitarbeiter	Postman Get Mitarbeiter	Liste aller Mitarbeiter	Liste mit allen Mitarbeitern aber mit Passwörtern	Fehlerhaft	Ja	Passwörter vor der Ausgabe mit * überschrieben
6	Get Priority Registration	Postman Get Abfrage	Ganzer Datensatz nach Priorität geordnet	Datensatz unvollständig	Fehlerfrei	-	Einbauen der Model-Klasse
7	Get All Registrationen	Postman Get Abfrage	Die Ausgabe aller Registrationen	Ausgabe aller Registrationen	Fehlerfrei	-	
8	Post Registration	Postman Post	Eintrag in die Datenbank	Eintrag wurde in Datenbank gespeichert	Fehlerfrei	-	
9	Get by ID Registration	Postman Get mit ID	Einzelner Datensatz der gewünschten ID	Einzelner Datensatz der gewünschten ID	Fehlerfrei	-	
10	Put Registration	Postman Put mit ID	Datensatz wird geändert	Datensatz wurde geändert	Fehlerfrei	-	
11	Delete Registration	Postman Delete mit ID	Datensatz wird gelöscht	Datensatz wurde gelöscht	Fehlerfrei	-	
12	Autorisierung	CRUD-Abfragen	Nur mit gültigem JWT Möglich	War nur mit gültigem JWT möglich	Fehlerfrei	-	
13	Backup-Skript	Skript ausführen	Backup einer bestimmten Datenbank wird erstellt	Nur Backup von allen Datenbanken konnte erstellt werden	Fehlerhaft	Ja	Neuinstallation von MongoDB konnte das Problem beseitigen
14	Automatisches Backup	Wird die Aufgabe ausgeführt	Erstellen eines Backups zu einer bestimmten Zeit	Backup wurde erstellt	Fehlerfrei	-	
15	Restor-Skript	Skript ausführen	Datenbank wurde wiederhergestellt	Datenbank wurde wiederhergestellt	Fehlerfrei	-	

Abbildungsverzeichnis

Quellen

https://www.youtube.com/watch?v=iWTdJ1IYGtg

https://stackoverflow.com/questions/49533659/mongodb-projection-in-c-sharp

https://stackoverflow.com/questions/10169064/mongodb-authentication-with-connection-string

https://stackoverflow.com/questions/41615574/mongodb-server-has-startup-warnings-access-control-is-not-enabled-for-the-dat

www.mongodb.com/community/forums