PROJEKTDOKUMENTATION JETSTREAM SKISERVICE API

Alexander Ernst

Inhaltsverzeichnis

Ausgangslage 2
Anforderungen 2
Zeitplanung/PSP 3
Vorgehensweise 4
1. Informieren 4
1.1. Ausgangslage/Anforderungen 4
1.2. Informieren über Web API 4
2. Planen 4
2.1. Datenbank Aufbau 4
2.2. Web API Aufbau 4
2.3. Zeitplanung und PSP 4
3. Entscheiden 5
3.1. Für Versionierungsplattform/Tools entscheiden 5
3.2. Für Web API Aufbau entscheiden 5
3.3. Für Datenbankaufbau entscheiden 6
3.4. Für Zusatzanforderungen entscheiden 7
3.5. Für Web API Authentifikation entscheiden 7
4. Realisieren 8
4.1. Datenbank erstellen 8
4.2. DTOs für Zugriff erstellen 9
4.3. HTTP Methoden/Services erstellen 10
4.4. Zusatzfeatures erstellen 11
4.5. Authentifikation erstellen 12
5. Kontrollieren 12
5.1. Web API mit Postman testen 12
6. Auswerten 12
6.1. Reflexion/Fazit zu Projekt 12
6.2. Dokumentation fertigstellen/Präsentation erstellen 12

Ausgangslage

Die Firma Jetstream-Service führt als KMU in der Wintersaison Skiservice Arbeiten durch, will im Zuge der Digitalisierung die interne Verwaltung der Skiservice Aufträge komplett über ein Web und Datenbank basierten Anwendung abwickeln. Die bereits existierende Onlineanmeldung soll bestehen bleiben und mit den erforderlichen Funktionen für das Auftragsmanagement erweitert werden.
In der Hauptsaison sind bis zu 10 Mitarbeiter mit der Durchführung der Serverarbeiten beschäftigt. Diese sollen einen autorisierten passwortgeschützten Zugang zu den anstehenden Aufträgen erhalten und diese zur Abarbeitung übernehmen und ändern können.

Anforderungen

Das Auftragsmanagement muss folgende Funktionen zur Verfügung stellen:

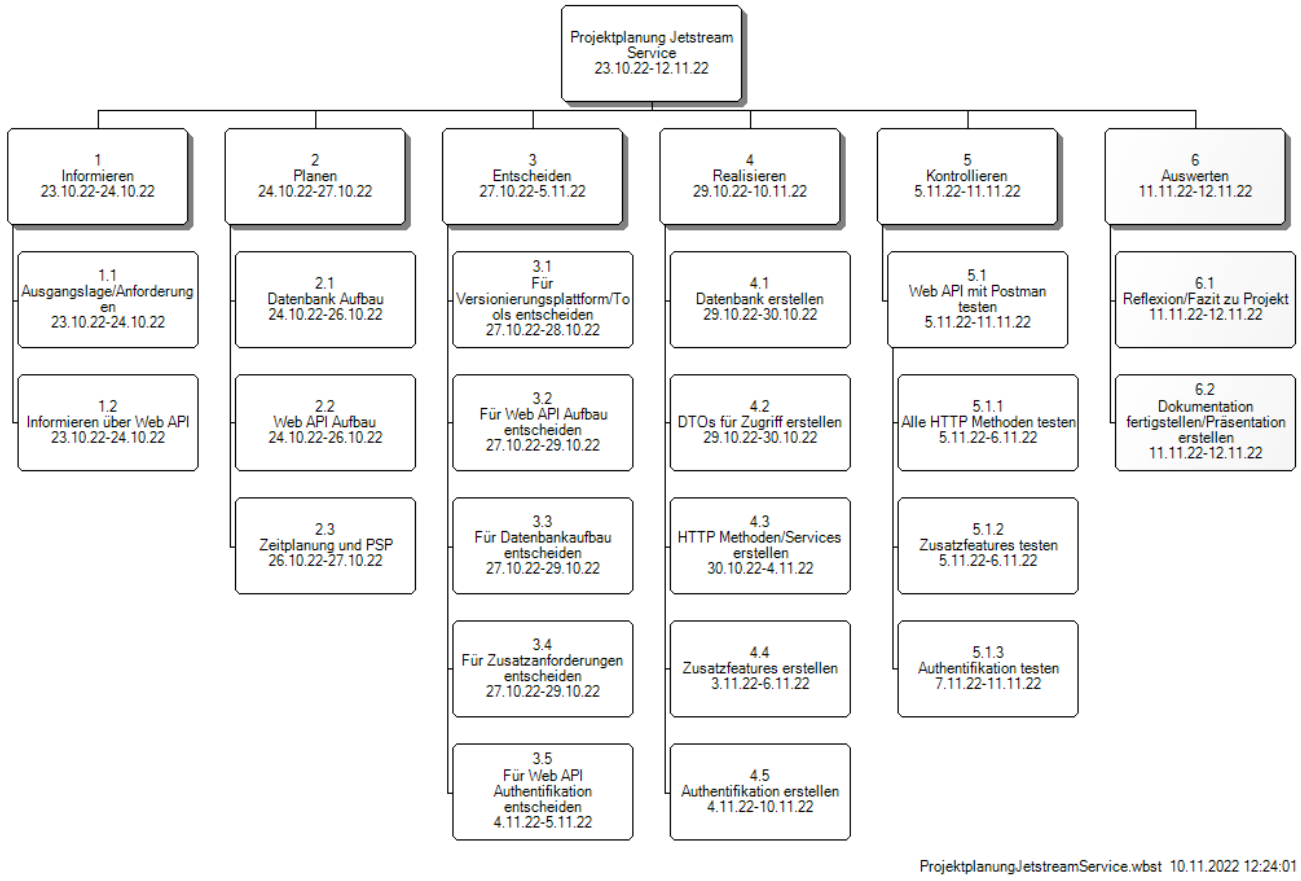
- Login mit Benutzername und Passwort
- Anstehende Serviceaufträge anzeigen (Liste)
- Bestehende Serviceaufträge mutieren. Dazu stehen folgende Stati zur Verfügung: Offen, InArbeit und abgeschlossen.
- Aufträge löschen (ggf. bei Stornierung)

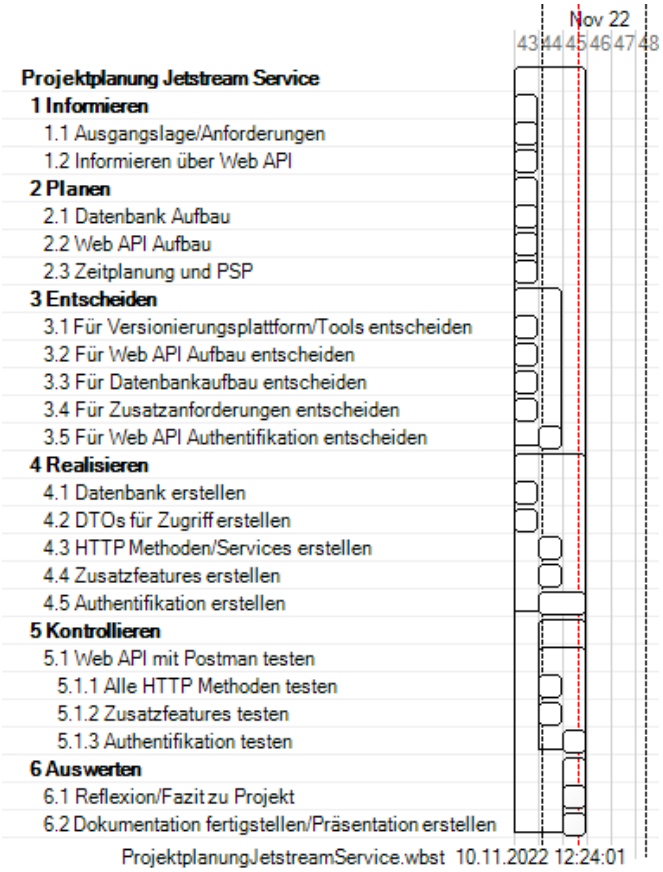
Nr.	Beschreibung
A1	Login Dialog mit Passwort für den autorisierten Zugang der Mitarbeiter (Datenänderungen).
A2	In der Datenbank müssen die Informationen des Serviceauftrags und die Login Daten der Mitarbeiter verwaltet sein.
A3	Erfasste Serviceaufträge abrufbar sein.
A4	Die erfassten Serviceaufträge müssen selektiv nach Priorität abrufbar sein.
A5	Mitarbeiter können eine Statusänderung eines Auftrages vornehmen.
A6	Mitarbeiter können Aufträge löschen (z.B. bei Stornierungen)
A7	Die aufgerufenen API-Endpoints müssen zwecks Fehlerlokalisierung protokolliert sein (DB oder Protokolldatei).
A8	Datenbankstruktur muss normalisiert in der 3.NF sein inkl. referenzieller Integrität
A9	Für die Web-API Applikation muss ein eigener Datenbankbenutzerzugang mit eingeschränkter Berechtigung (DML) zur Verfügung gestellt werden (Benutzer root bzw. sa ist verboten).
A10	Das Web-API muss vollständig nach Open-API (Swagger) dokumentiert sein.
A11	Das Softwareprojekt ist über ein Git-Repository zu verwalten.
A12	Ganzes Projektmanagement muss nach IPERKA dokumentiert sein

Nr.	Beschreibung
AO1	Die Mitarbeiter können zu einem Auftrag einen Freitext bzw. Kommentar hinterlegen
AO2	Ein Auftrag kann mit sämtlichen Datenfeldern geändert werden
AO3	Das Login des Mitarbeiters wird nach drei nachfolgenden Falschanmeldungen automatisch gesperrt.
AO4	Personalisierte Auftragsliste des eingeloggten Mitarbeiters. Der Mitarbeiter kann sich zusätzlich zur gesamten Auftragsliste nur die von ihm übernommenen Aufträge ansehen.
AO6	Eingeloggte Mitarbeiter können ein gesperrtes Login zurücksetzen.
AO7	Gelöscht Aufträge werden nicht aus der Datenbank entfernt, sondern nur als gelöscht markiert.

Zeitplanung/PSP

Mit WBSTool erstellt





Vorgehensweise

1. Informieren

1.1. Ausgangslage/Anforderungen

Als ersten Arbeitsschritt habe ich mich über die Anforderungen und die Ausgangslage dieses Projektes informiert. Hierzu habe ich die zur Verfügung gestellten Unterlagen verwendet.
Durch diese Recherche konnte ich die Anforderungen und das Grundkonzept der Web-APIs verstehen, und wusste ungefähr, was zu tun war.

1.2. Informieren über Web API

Als Nächstes habe ich mich über Technologien, die ich für dieses Projekt brauchen werde, informiert. Dies war aber auch ein laufender Prozess, dies heisst, dass ich mich in dem ganzen Laufe des Projektes weiter informieren/in das Themengebiet vertiefen musste.
Diese Recherche habe ich einerseits mit dem Unterrichtstoff durchgeführt. Andererseits habe ich aber auch mich im Internet weiter informiert.

2. Planen

2.1. Datenbank Aufbau

Hier musste ich den Datenbankaufbau planen, dies heisst, wie die Verbindungen zwischen den Tabellen aussehen soll, und ob ich einen Code First oder Database First Ansatz verwende.

2.2. Web API Aufbau

In diesem Schritt musste ich planen, wie ich, dass Web API aufbauen wollte, hierzu gehören die DTO Klassen, welche Methoden ich implementiere, und welche Zusatzfeatures ich noch hinzufüge.
Hier musste ich auch meine Authentifikationsart planen, hier gab es eigentlich 3 Optionen, JWT Token, API Key oder Basic Authentifikation.
Hier habe ich vor und Nachteile jeder Option aufgelistet:

Art	**Vorteil**	**Nachteil**
Basic Authentifikation	Einfach zu implementieren	Unsicher
API Authentifikation	Sicher	Ungeeignet für Anmeldung
JWT-Token	Sicher, geeignet für Anmeldung	Schwerer zu implementieren

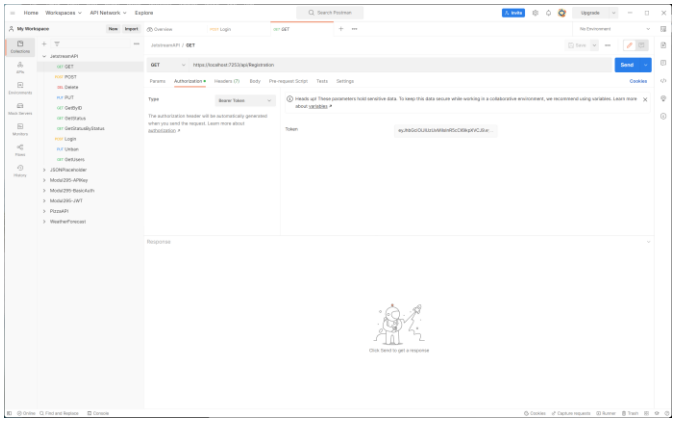
2.3. Zeitplanung und PSP

Als Letztes konnte ich eine Zeitplanung/ein PSP erstellen, da ich jetzt eine grobe Planung meines Projektes hatte.

3. Entscheiden

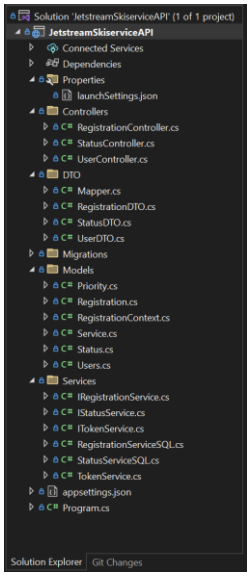
3.1. Für Versionierungsplattform/Tools entscheiden

Hier musste ich mich entscheiden, welche Versionierungsplattform/andere Tools ich verwende. Für das Versionieren des Codes habe ich GitHub verwendet, und für das Schreiben des Codes habe ich den IDE Visual Studio verwendet.
Als Testing Tool für die API habe ich Postman gebraucht, da dies eine grosse Funktionalität bietet und für einfaches exportieren von Tests erlaubt.
Zusätzlich habe ich mich auch noch entschieden einen Logger in das Projekt einzubauen, der Fehler in einer Logdatei loggt.
GitHub: https://github.com/alexanderternst/JetstreamSkiserviceAPI



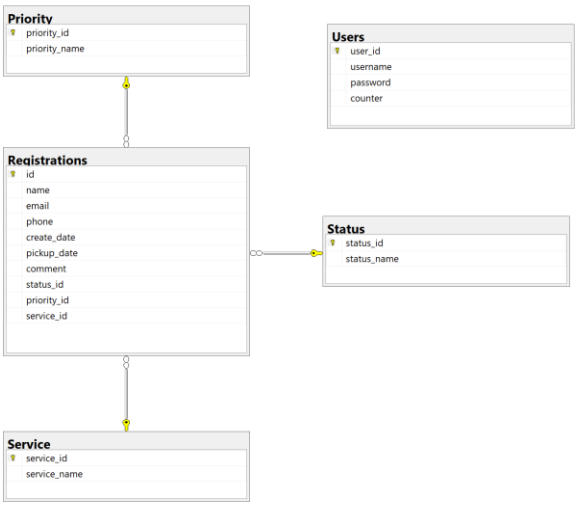
3.2. Für Web API Aufbau entscheiden

Hier habe ich mich entschieden, dass Projekt mit DTO (Data Transfer Object) Klassen umzusetzen, dies heisst, dass ich DTO Klassen benutze, um Daten zu lesen und zu schreiben.
Diese funktionieren so, dass man das, was man schreiben, lesen will, erst in ein DTO Objekt tut, und dieses dann ausgibt/schreibt.
Dies hat den Vorteil, dass, wenn sich die Namen von einer Property ändern, man diese nicht in der Datenbankklasse ändern muss.
Zusätzlich habe ich mich entschieden, die Logik des Programmes in Service Klassen zu verbauen und das ganze Projekt mit Dependency Injection umzusetzen. Dies hat den Vorteil der höheren Wiederverwendbarkeit und weniger Abhängigkeit.
In diesem Projekt habe ich drei Controller, einen für die Verwaltung von Registrationen, einen für das Auslesen von Registrationen nach Status, und einen für die Authentifikation.
Dementsprechend habe ich auch drei Service Klassen mit jeweils einem Interface.



3.3. Für Datenbankaufbau entscheiden

Als Erstes musste ich mir überlegen, ob ich die Datenbank mit einem Code- oder Database-First Prinzip erstelle. In diesem Projekt habe ich mich für die Codefirst Variante entschieden, da dies ein neues Themengebiet war, indem ich noch nicht so viel Erfahrung hatte und ich meine Erfahrung in diesem Bereich erhöhen wollte.
Beim Datenbankaufbau habe ich mich für eine Datenbank mit 4 Tabellen entschieden.
Eine Tabelle mit allen Registrationen, und 3 andere Tabellen (Status, Priority, Service) welche via. Foreign Key, mitden Registrationen verbunden sind. In diesen 3 Tabellen speichern wir Status, Priorität und Dienstleistung mit jeweils einer ID, auf die wir dann von den Registrationen zugreifen können.
Zusätzlich gibt es noch eine Tabelle mit Benutzern, welche sich durch die Authentifikation anmelden können (Mitarbeiter).



3.4. Für Zusatzanforderungen entscheiden

Hier habe ich mich entschieden, welche Zusatzanforderungen ich durchsetzen wollte.
Ich habe mich hier, für das Bannen von Benutzern nach 3 falschen versuchen (auch entsprechende Entbannung Funktion), und den Kommentar bei Registrationen entschieden. Zusätzlich habe ich auch noch eingebaut, dass man über Status auf Registrationen zugreifen kann.

3.5. Für Web API Authentifikation entscheiden

Hier habe ich mich für das JWT Token entschieden, da dies meiner Meinung nach für dieses Projekt am meisten Sinn ergibt.
Die Basic Authentifikation wäre zu unsicher und die API Key Authentifikation ist nicht für ein Benutzer-Login geeignet.
Ich habe mich hier entschieden die Benutzer in einer SQL Tabelle zu speichern, da dies ein guter Weg ist mehrere Benutzer zu speichern.

Jetzt musste ich mich noch entscheiden, auf welche Methoden Personen ohne Anmeldung Zugriff haben.
Ich habe es jetzt so durchgesetzt, dass man ohne JWT Token nur auf die POST Klasse zugreifen können, dies heisst normale Benutzer können Aufträge erstellen/ergeben, aber können keine Aufträge löschen/modifizieren/lesen.
Dies macht in diesem Projekt meiner Meinung nach Sinn, da dieses API für eine Registration gedacht ist, und ein Benutzer natürlich nicht andere Registrationen lesen sollte.

4. Realisieren

4.1. Datenbank erstellen

Hier habe ich angefangen, die Datenbank mit dem Code First Prinzip zu erstellen.
Als Erstes habe ich alle Klassen für die Datenbank erstellt (Registration, Status, Priority, Service, User) und eine Klasse namens RegistrationContext, welche die eigentliche Datenbank erstellt kreiert, der Connection String, der diese Klasse braucht, ist in ein JSON File ausgelagert.
Zusätzlich wird die Datenbankverbindung mit einem SQL User erstellt.

In den untigen Bildern sehen Sie den Aufbau dieser Klassen, einmal habe ich durch eine Eigenschaft (des Datentyp Status, Registration, Service) in der Klasse Registration eine Verbindung von Registrationen auf Stati, Prioritäten und Dienstleistungen.
Andererseits habe ich auch eine Eigenschaft in Form einer Liste (des Datentyp Registration) in den Klassen Status, Service und Priority.
Dies erlaubt uns eine Verbindung von Status auf Registrationen herzustellen, damit wir z.B. alle Registrationen mit Status offen sehen können.

```
using System.ComponentModel.DataAnnotations;

namespace JetstreamSkiserviceAPI.Models
{
    9 references | alex0417a, 3 days ago | 1 author, 8 changes
    public class Registration
    {
        [Key]
        3 references | alex0417a, 11 days ago | 1 author, 1 change
        public int id { get; set; }

        [StringLength(255)]
        4 references | alex0417a, 11 days ago | 1 author, 1 change
        public string name { get; set; }

        [StringLength(255)]
        4 references | alex0417a, 11 days ago | 1 author, 1 change
        public string email { get; set; }

        [StringLength(255)]
        4 references | alex0417a, 3 days ago | 1 author, 3 changes
        public string phone { get; set; }

        4 references | alex0417a, 11 days ago | 1 author, 1 change
        public DateTime create_date { get; set; }

        4 references | alex0417a, 11 days ago | 1 author, 1 change
        public DateTime pickup_date { get; set; }

        [StringLength(255)]
        4 references | alex0417a, 3 days ago | 1 author, 1 change
        public string comment { get; set; }

        3 references | alex0417a, 5 days ago | 1 author, 3 changes
        public Status Status { get; set; }
        4 references | alex0417a, 8 days ago | 1 author, 1 change
        public Priority Priority { get; set; }
        4 references | alex0417a, 8 days ago | 1 author, 1 change
        public Service Service { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace JetstreamSkiserviceAPI.Models
{
    4 references | alex0417a, 9 days ago | 1 author, 1 change
    public class Status
    {
        [Key]
        1 reference | alex0417a, 9 days ago | 1 author, 1 change
        public int status_id { get; set; }

        [StringLength(255)]
        5 references | alex0417a, 9 days ago | 1 author, 1 change
        public string status_name { get; set; }

        1 reference | alex0417a, 9 days ago | 1 author, 1 change
        public List<Registration> registrations { get; set; }
    }
}
```

```
using Microsoft.EntityFrameworkCore;

namespace JetstreamSkiserviceAPI.Models
{
    12 references | alex0417a, 3 days ago | 1 author, 9 changes
    public class RegistrationContext : DbContext
    {
        private readonly IConfiguration _config;
        0 references | alex0417a, 6 days ago | 1 author, 2 changes
        public RegistrationContext()
        {

        }

        0 references | alex0417a, 3 days ago | 1 author, 1 change
        public RegistrationContext(DbContextOptions<RegistrationContext> options, IConfiguration config)
            : base(options)
        {
            _config = config;
        }

        4 references | alex0417a, 11 days ago | 1 author, 1 change
        public DbSet<Registration> Registrations { get; set; }

        3 references | alex0417a, 9 days ago | 1 author, 1 change
        public DbSet<Status> Status { get; set; }

        2 references | alex0417a, 8 days ago | 1 author, 1 change
        public DbSet<Priority> Priority { get; set; }

        2 references | alex0417a, 8 days ago | 1 author, 1 change
        public DbSet<Service> Service { get; set; }

        4 references | alex0417a, 6 days ago | 1 author, 1 change
        public DbSet<Users> Users { get; set; }

        0 references | alex0417a, 3 days ago | 1 author, 6 changes
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            string t = _config.GetConnectionString("RegistrationDB");
            optionsBuilder.UseSqlServer($@"{t}");
        }
    }
}
```

Nachdem ich diese Klassen erstellt habe, habe ich mit diesen Befehlen in der Package Manger Console die Datenbank erstellt:

```
Add-Migration IntialCreate
Update-Database
```

Für das Einfügen von Status, Service, Priority und User habe ich einen SQL Skript, den Sie auf GitHub finden, verwendet.

4.2. DTOs für Zugriff erstellen

Als Nächstes habe ich DTO (Data Transfer Object) Klassen erstellt, über welche wir auf Daten zugreifen/und Daten schreiben.
Diese haben jetzt auch den grossen Vorteil, dass wen ich Registrationen mit ihrem entsprechenden Status ausgeben will, muss ich nicht alles ausgeben, sondern ich kann nur den Inhalt der DTO Klasse ausgeben, was die Ausgabe/Eingabe viel sauberer macht.

Wir haben drei DTO Klassen, eine für das Anmelden (UserDTO), eine für das Auslesen von Stati mit ihren Angehörigen Registrationen (StatusDTO) und eine für unsere Registrationen (RegistrationDTO).

```
using System.Text.Json.Serialization;

namespace JetstreamSkiserviceAPI.DTO
{
    24 references | alex0417a, 3 days ago | 1 author, 4 changes
    public class RegistrationDTO
    {
        [JsonPropertyName("registration_id")]
        7 references | alex0417a, 10 days ago | 1 author, 1 change
        public int id { get; set; }

        [JsonPropertyName("registration_name")]
        8 references | alex0417a, 10 days ago | 1 author, 1 change
        public string name { get; set; }

        [JsonPropertyName("registration_email")]
        8 references | alex0417a, 10 days ago | 1 author, 1 change
        public string email { get; set; }

        [JsonPropertyName("registration_phone")]
        8 references | alex0417a, 6 days ago | 1 author, 2 changes
        public string phone { get; set; }

        [JsonPropertyName("registration_create_date")]
        8 references | alex0417a, 10 days ago | 1 author, 1 change
        public DateTime create_date { get; set; }

        [JsonPropertyName("registration_pickup_date")]
        8 references | alex0417a, 10 days ago | 1 author, 1 change
        public DateTime pickup_date { get; set; }

        [JsonPropertyName("registration_status")]
        8 references | alex0417a, 10 days ago | 1 author, 1 change
        public string status { get; set; }

        [JsonPropertyName("registration_priority")]
        8 references | alex0417a, 10 days ago | 1 author, 1 change
        public string priority { get; set; }

        [JsonPropertyName("registration_service")]
        8 references | alex0417a, 10 days ago | 1 author, 1 change
        public string service { get; set; }

        [JsonPropertyName("registration_comment")]
        8 references | alex0417a, 3 days ago | 1 author, 1 change
        public string comment { get; set; }
    }
}
```

```
namespace JetstreamSkiserviceAPI.DTO
{
    11 references | alex0417a, 6 days ago | 1 author, 1 change
    public class StatusDTO
    {
        1 reference | alex0417a, 6 days ago | 1 author, 1 change
        public int status_id { get; set; }
        2 references | alex0417a, 6 days ago | 1 author, 1 change
        public string status_name { get; set; }
        1 reference | alex0417a, 6 days ago | 1 author, 1 change
        public List<RegistrationDTO> registration { get; set; } = new List<RegistrationDTO>();
    }
}
```

4.3. HTTP Methoden/Services erstellen

Als Nächstes habe ich die Controller und die Serviceklassen erstellt, in den Controllern passiert der Aufruf der HTTP Methoden und in dem Service befindet sich die ganze Logik, die Daten aus der SQL Datenbank liest/in die Datenbank schreibt.
Die Services sind zusätzlich noch mit DI (Dependency Injection) gebaut, dies heisst jede Serviceklasse hat ein Interface (erbt von einem Interface), über welches wir auf den Service zugreifen.
So schaffen wir eine immense Wiederverwendbarkeit und Abhängigkeit, dies heisst die Controller Klasse muss den Service gar nicht kennen, da diese durch das Interface komplett abgekoppelt sind.

Damit Dependency Injection funktioniert müssen wir im Programm.cs das Interface und die Klasse initialisieren und in der Controller Klasse das Interface im Konstruktor instanziieren (Wichtig ist hier das nur das Interface instanziiert wird, und nicht der konkrete Service).

Zusätzlich damit das JWT Token und die SQL Server Verbindung funktioniert müssen wir im Programm.cs auch Services hinzufügen, welche dann in entsprechenden Controllern instanziiert werden.

In diesem Programm haben wir drei Controller, einen für Registrationen, einen für Zugriff auf Registrationen über Status und einer für die Authentifikation.
Dementsprechend haben wir auch 3 Services mit entsprechenden Interfaces, auf die wir von den Controllern zugreifen.

Alle Services und Controller haben auch eine try-catch Fehlerbehandlung, sodass das Programm nicht einfach abstürzt und der Controller loggt Fehler mit einem Logger in einer log Datei.

Wenn man mit dem API Daten einfügen will, muss man bei Status, Priority und Service einfach den Namen des gewünschten Status, Priority oder Service eingeben.
Wichtig ist natürlich, dass dieser Name auch mit dem Namen in der Datenbank übereinstimmt.
Programm.cs (wo das Programm startet)

```
// SQL Verbindung
builder.Services.AddDbContext<RegistrationContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("RegistrationDB")));

// Seri Logger mit appsettings.json Konfiguration
var loggerFromSettings = new LoggerConfiguration()
    .ReadFrom.Configuration(builder.Configuration)
    .Enrich.FromLogContext()
    .CreateLogger();

builder.Logging.ClearProviders();
builder.Logging.AddSerilog(loggerFromSettings);

// configure DI for application services
builder.Services.AddScoped<IRegistrationService, RegistrationServiceSQL>();
builder.Services.AddScoped<IStatusService, StatusServiceSQL>();
builder.Services.AddScoped<ITokenService, TokenService>();

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "Jetstream Skiservice API", Version = "v1" });
});

// JWT
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
        };
    });
```

Service Registration (erbt von Interface/behält Logik des Programmes)

```
namespace JetstreamSkiserviceAPI.Services
{
    2 references | - changes | -authors, -changes
    public class RegistrationServiceSQL : IRegistrationService
    {
        private readonly RegistrationContext _dbContext;

        public List<Registration> registrations = new List<Registration>();

        0 references | - changes | -authors, -changes
        public RegistrationServiceSQL(RegistrationContext dbContext)
        {
            _dbContext = dbContext;
        }

        3 references | - changes | -authors, -changes
        public List<RegistrationDTO> GetAll()
        {
            try
            {
                registrations = _dbContext.Registrations.Include("Status").Include("Priority").Include("Service").ToList();

                List<RegistrationDTO> result = new List<RegistrationDTO>();
                registrations.ForEach(e => result.Add(new RegistrationDTO()
                {
                    id = e.id,
                    name = e.name,
                    email = e.email,
                    phone = e.phone,
                    create_date = e.create_date,
                    pickup_date = e.pickup_date,
                    status = e.Status.status_name,
                    priority = e.Priority.priority_name,
                    service = e.Service.service_name,
                    comment = e.comment,
                }));

                return result;
            }
            catch(Exception ex)
            {
                throw new Exception(ex.Message);
            }
        }
```

Controller Registration (instanziiert Interface/ruft Service durch Interface auf)

```
namespace JetstreamSkiserviceAPI.Controllers
{
    [ApiController]
    [Authorize]
    [Route("api/[controller]")]
    3 references | - changes | -authors, -changes
    public class RegistrationController : ControllerBase
    {
        private IRegistrationService _registrationService;
        private readonly ILogger<RegistrationController> _logger;

        0 references | - changes | -authors, -changes
        public RegistrationController(IRegistrationService registration, ILogger<RegistrationController> logger)
        {
            _registrationService = registration;
            _logger = logger;
        }

        [HttpGet]
        0 references | - changes | -authors, -changes
        public ActionResult<List<RegistrationDTO>> GetAll() => _registrationService.GetAll();

        // GET by Id action
```

4.4. Zusatzfeatures erstellen

Um das Zugreifen von Status auf Registrationen durchzusetzen, musste ich erstmals eine neue DTO Klasse (StatusDTO) erstellen, welche status_id, status_name und eine Liste des Typen Registrationen enthält erstellen. Dann konnte ich durch mehrere Schleifen die Stati mit den zugehörigen Registrationen auslesen.
Zusätzlich habe ich noch eingebaut, dass man Registrationen nach Status auslesen kann, dies heisst ich kann z.B. alle Registrationen mit Status offen auslesen.
Als Nächstes habe ich auch noch ein Feature umgesetzt, welches einen User nach drei falschen Passwortangaben bannt, umgesetzt. Natürlich gibt es hier aber auch eine entsprechende Methode, um den User zu entbannen.

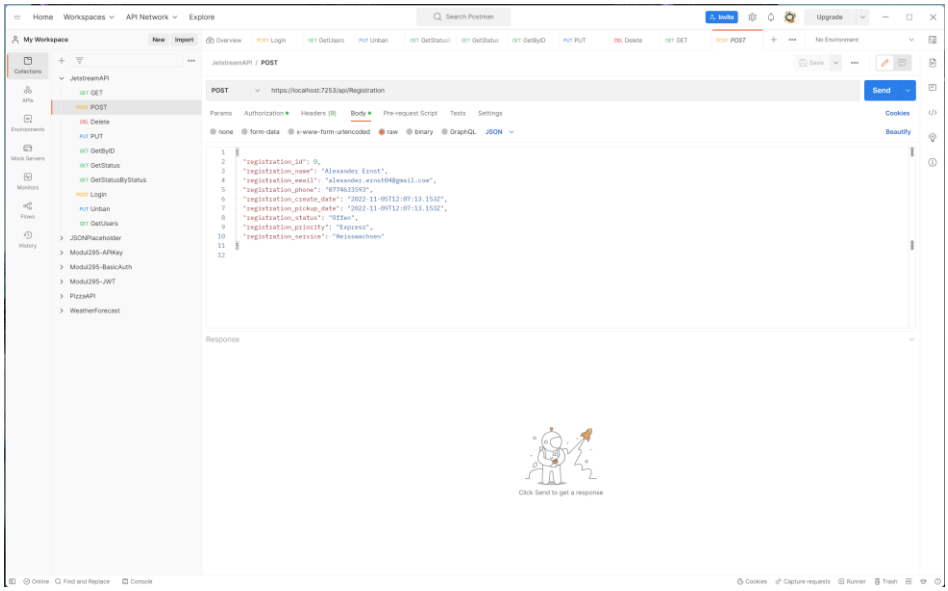
4.5. Authentifikation erstellen

Als letzten Schritt habe ich die Authentifikation mit einem JWT Token durchgeführt, Benutzer, die ein Token kriegen, sind in einer SQL Tabelle gespeichert, die ausgelesen wird.
Für das authentifizieren und kreieren des JWT Tokens habe ich einen eigenen User Controller und einen eigenen Service mit entsprechendem Interface.

5. Kontrollieren

5.1. Web API mit Postman testen

In dieser Aktivität habe ich alle HTTP Methoden inklusive Authentifikation mit Postman getestet, ich habe auch Szenarien getestet, in denen ich eine nichtexistierende ID angeben, usw.



6. Auswerten

6.1. Reflexion/Fazit zu Projekt

Ich denke, ich habe die Hauptanforderungen dieses Projektes geschafft, da das Web API mit Authentifikation und sogar Zusatzfeatures komplett lauffähig ist.
Ich habe durch dieses Projekt auch sehr viel über APIs und C# gelernt, und konnte mein Wissen um einiges erweitern.
Was meiner Meinung in diesem Projekt nicht so gut lief, war die Zeitplanung, ich habe in meinem Zeitplan meinen Aufwand um einiges unterschätzt.
Dies liegt daran, dass man oft mehrere Stunden an einem kleinen Problem arbeitet, was dazu führt, dass man nicht mit anderem weiterkommt.

Im Allgemeinen lief dieses Projekt aber schon sehr gut und ich habe viel Neues gelernt.

6.1.1. Was habe ich nicht geschafft/Verbesserungen

Dieses Projekt lief aber auch nicht perfekt, zwei Features, die ich gerne noch geschafft hätte, wäre die Verschlüsselung von dem Connection String und einen Mapper welches den Code etwas verschönern würde.
Da diese Features, aber nicht Teil der Anforderungen waren, habe ich mich auf anderes/wichtigeres konzentriert.

6.2. Dokumentation fertigstellen/Präsentation erstellen

Als aller letzten Arbeitsschritt habe ich die Dokumentation nochmals überarbeitet und die Präsentation der Projektarbeit erstellt.