

# Modul 321 Lerndokumentation Skybooker

Nevan Alberola



## Inhaltsverzeichnis

<b>1. Initialisieren</b>	<b>3</b>
1.1 Ausgangslage .....	3
1.2 Projektidee: SkyBooker – Buchungsplattform .....	3
1.3 Zielsetzung und Nutzen des Projekts.....	3
<b>2. Planen</b>	<b>4</b>
2.1 Projektstruktur und Planung.....	4
2.2 Zeitmanagement .....	4
2.3 Risiken und Gegenmassnahmen .....	4
<b>3. Entscheiden</b>	<b>5</b>
3.1 Technologiewahl .....	5
3.2 Nuget Pakete oder andere.....	5
3.3 Sources .....	5
<b>4. Realisieren</b>	<b>6</b>
4.1 Microservices aufbauen.....	6
4.2 Docker-Setup .....	6
4.3 Beispiel: Buchung mit RabbitMQ verschicken.....	6
4.4 Empfang in MessageService .....	8
4.5 API Gateway Routing (Ocelot).....	8
<b>5. Kontrollieren</b>	<b>9</b>
5.2 Validierung testen mit FluentValidation .....	9
5.3 Controller-Verhalten testen .....	10
<b>2 Auswerten</b>	<b>11</b>
2.1 Fazit.....	11
<b>3 Quellen</b>	<b>11</b>
3.1 Bild Verzeichnis.....	11

## 1. Initialisieren

### 1.1 Ausgangslage

In der heutigen digitalen Welt ist das Online-Buchen von Flügen, Hotels oder Aktivitäten zum Standard geworden. Dennoch sind viele bestehende Systeme entweder zu komplex, schwer zu bedienen oder technisch veraltet. Besonders kleine Anbieter haben es schwer, eine einfache, skalierbare und sichere Buchungsplattform zu finden. Genau hier setzt mein Projekt an.

### 1.2 Projektidee: SkyBooker – Buchungsplattform

SkyBooker ist eine Microservice-basierte Buchungsplattform, die eine modulare, leicht erweiterbare Lösung für Buchungsprozesse bereitstellt. Die Anwendung besteht aus mehreren Services (z. B. BookingService), die über ein API-Gateway (Ocelot) zentral erreichbar sind. Die Services kommunizieren untereinander über HTTP-Requests. Die Datenhaltung erfolgt sowohl in SQL Server als auch in MongoDB, abhängig vom jeweiligen Anwendungsfall.

### 1.3 Zielsetzung und Nutzen des Projekts

Das Hauptziel ist es, eine funktionale Buchungsplattform bereitzustellen, die:

- auf Microservice-Architektur basiert,
- containerisiert mit Docker läuft,
- sichere Authentifizierung via JWT unterstützt,
- mit moderner UI über Swagger dokumentiert ist.

#### Nutzen:

- Erlernung moderner Technologien (.NET 8, Ocelot, Docker)
- Praktische Anwendung von Clean Architecture Prinzipien
- Aufbau eines realitätsnahen Projekts für das Portfolio

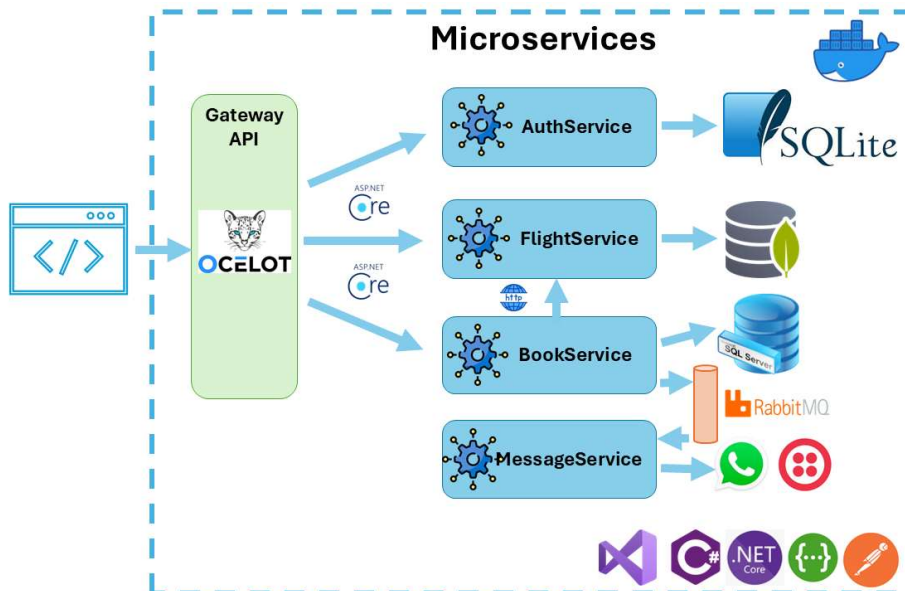


Bild 1: Aufgabe Übersicht, Erweiterungen Level 3

## 2. Planen

### 2.1 Projektstruktur und Planung

Für die Umsetzung des Projekts „SkyBooker“ wurde eine modulare Microservice-Architektur gewählt. Die einzelnen Services sind unabhängig voneinander entwickelt und laufen jeweils in einem eigenen Docker-Container. Folgende Hauptkomponenten wurden definiert:

- **BookingService** (Verwaltung der Buchungen)
- **ApiGateway** (Routing und Sicherheit)
- **SQL Server** (relationale Daten für Buchungen)
- **MongoDB** (optionale Erweiterung für NoSQL-Daten)
- **Docker Compose** (Orchestrierung)

### 2.2 Zeitmanagement

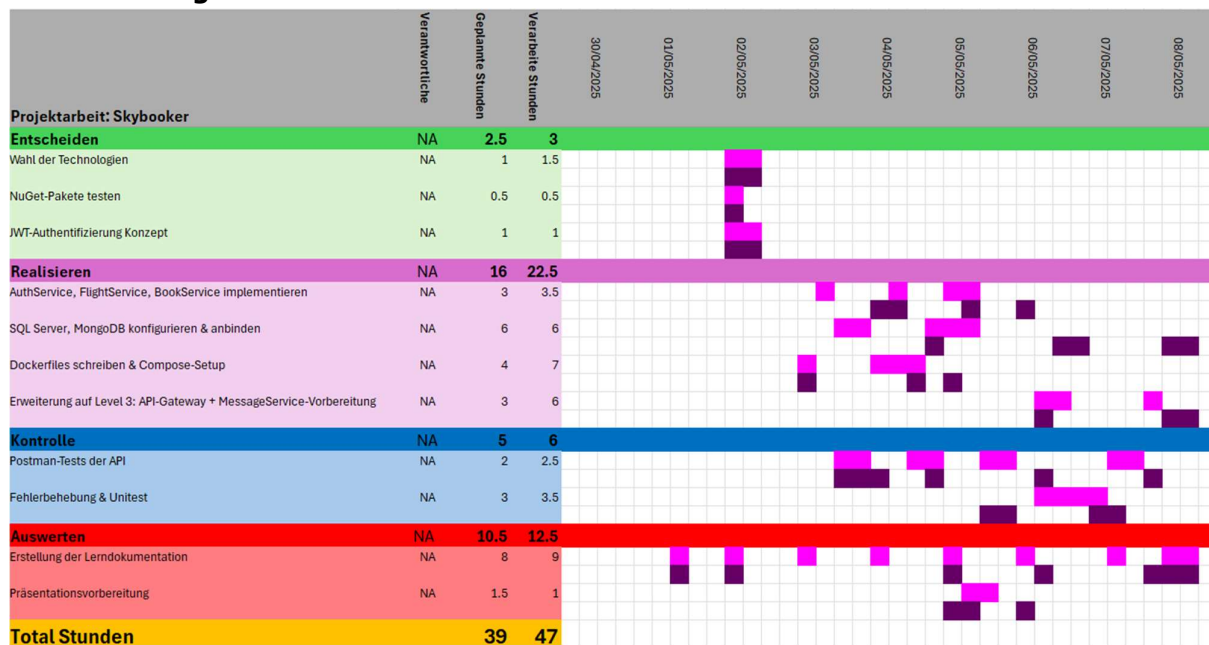


Bild 2: Skybooker Zeitmanagement

### 2.3 Risiken und Gegenmassnahmen

Risiko	Auswirkung	Gegenmassnahme
Komplexität von Docker	Verzögerungen	Frühes Testen mit kleinen Containern
API-Routing-Fehler	Dienste nicht erreichbar	Ocelot-Konfiguration schrittweise testen
Authentifizierungsprobleme	Zugriff auf Ressourcen blockiert	JWT-Debugging-Tools verwenden
Datenbankverbindungsprobleme	Datenverlust oder Fehler	Logs analysieren, Verbindungsstrings prüfen

### 3. Entscheiden

#### 3.1 Technologiewahl

Die Auswahl der Technologien basierte auf den Anforderungen an Modularität, Skalierbarkeit und Wartbarkeit des Projekts. Die Entscheidung fiel auf folgende Tools und Frameworks:

- **.NET 8 / C#**
- **Ocelot**
- **SQL Server**
- **MongoDB**
- **Docker**
- **Swagger**

#### 3.2 Nuget Pakete oder andere

- **BCrypt.Net-Next:** 4.0.3
- **Microsoft.AspNetCore.Authentication.JwtBearer:** 8.0.4
- **Microsoft.AspNetCore.OpenApi:** 8.0.4
- **Microsoft.EntityFrameworkCore.Design:** 9.0.4
- **Microsoft.EntityFrameworkCore.Sqlite:** 9.0.4
- **Microsoft.EntityFrameworkCore.SqlServer:** 9.0.4
- **Microsoft.Extensions.Http.Polly:** 9.0.4
- **MongoDB.Driver:** 3.3.0
- **Ocelot:** 24.0.0
- **RabbitMQ.Client:** 7.1.2 / 6.2.1
- **Swashbuckle.AspNetCore:** 8.1.1 / 6.5.0
- **FluentValidation.AspNetCore:** 11.3.0

#### 3.3 Sources

- [ChatGPT](#)
- [StackOverflow](#)

## 4. Realisieren

### 4.1 Microservices aufbauen

Das Projekt besteht aus vier zentralen Microservices:

- **AuthService** (mit SQLite)
- **FlightService** (mit MongoDB)
- **BookService** (mit SQL Server & RabbitMQ Publisher)
- **MessageService** (mit RabbitMQ Consumer & Twilio)

Diese Services wurden im Ordner Services/ implementiert und jeweils mit einem eigenen Dockerfile konfiguriert.

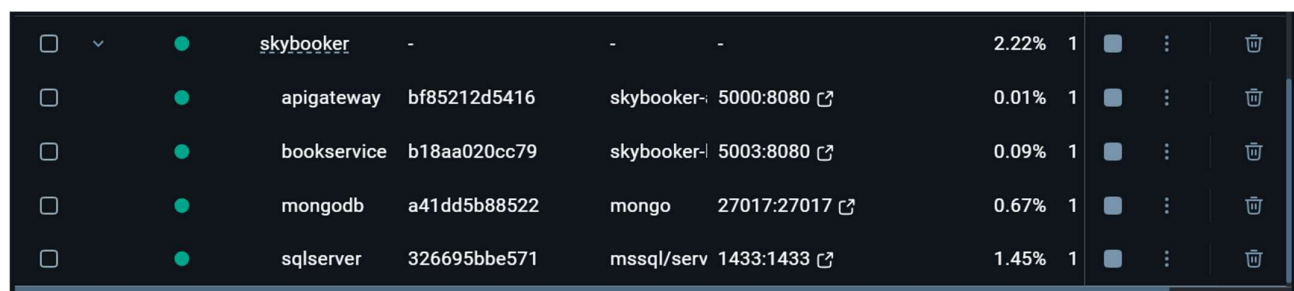
### 4.2 Docker-Setup

Für die Containerisierung der Services wurde folgender Befehl verwendet:

```
docker compose up -build
```

Die wichtigsten Ports aus der docker-compose.yml:

Service	Interner Port	Externer Port
API Gateway	8080	5000
BookService	8080	5003
MessageService	8080	5004
MongoDB	27017	27017
SQL Server	1433	1433
RabbitMQ Mgmt	15672	15672
RabbitMQ Port	5672	5672



<input type="checkbox"/>	▼	●	skybooker	-	-	-	2.22%	1			
<input type="checkbox"/>		●	apigateway	bf85212d5416	skybooker-	5000:8080 ↗	0.01%	1			
<input type="checkbox"/>		●	bookservice	b18aa020cc79	skybooker-	5003:8080 ↗	0.09%	1			
<input type="checkbox"/>		●	mongodb	a41dd5b88522	mongo	27017:27017 ↗	0.67%	1			
<input type="checkbox"/>		●	sqlserver	326695bbe571	mssql/serv	1433:1433 ↗	1.45%	1			

Bild 3: Docker..

### 4.3 Buchung mit RabbitMQ verschicken

#### Codeausschnitt aus BookingController:

```
string message = $"Reservation: {booking.PassengerFirstname}  
{booking.PassengerLastname}, Flight: {booking.FlightId}, Tickets:  
{booking.TicketCount}";  
_publisher.SendMessage(message);
```

#### Publisher in RabbitMqPublisher.cs:

```
var factory = new ConnectionFactory() { HostName = "rabbitmq", Port = 5672 };  
_connection = factory.CreateConnection();  
_channel = _connection.CreateModel();  
_channel.QueueDeclare(queue: "booking-queue", durable: false, exclusive: false,  
autoDelete: false, arguments: null);
```

## 4.4 Empfang in MessageService

### Konsument in RabbitMqConsumer.cs:

```
var consumer = new EventingBasicConsumer(channel);
consumer.Received += (model, ea) =>
{
    var body = ea.Body.ToArray();
    var message = Encoding.UTF8.GetString(body);
    Console.WriteLine($"[MessageService] Nachricht erhalten: {message}");
};
channel.BasicConsume(queue: "booking-queue", autoAck: true, consumer: consumer);
```

## 4.5 API Gateway Routing (Ocelot)

### Eintrag in ocelot.json:

```
{
  "DownstreamPathTemplate": "/api/booking",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "bookservice",
      "Port": 8080
    }
  ],
  "UpstreamPathTemplate": "/booking",
  "UpstreamHttpMethod": [ "GET", "POST" ]
}
```



## 5. Kontrollieren

### 5.2 Validierung testen mit FluentValidation

Die Klasse BookingValidator prüft:

- PassengerFirstname darf nicht leer sein
- TicketCount muss > 0 sein

**Testbeispiel:** (Datei: BookingValidatorTests.cs)

```
[Fact]
public void Valid_Booking_Should_Pass()
{
    var booking = new Booking
    {
        PassengerFirstname = "Anna",
        PassengerLastname = "Meier",
        FlightId = 1,
        TicketCount = 2
    };

    var validator = new BookingValidator();
    var result = validator.Validate(booking);

    Assert.True(result.IsValid);
}
```

**Test für ungültige Eingabe:**

```
[Fact]
public void Booking_With_Zero_Tickets_Should_Fail()
{
    var booking = new Booking
    {
        PassengerFirstname = "Anna",
        PassengerLastname = "Meier",
        FlightId = 1,
        TicketCount = 0
    };

    var validator = new BookingValidator();
    var result = validator.Validate(booking);

    Assert.False(result.IsValid);
    Assert.Contains(result.Errors, e => e.PropertyName == "TicketCount");
}
```

### 5.3 Controller-Verhalten testen

Mit Moq wurde ein Fake-DbContext simuliert, um den Controller ohne Datenbank zu testen.

#### POST mit gültiger Buchung

```
[Fact]
public async Task CreateBooking_ReturnsCreatedAtAction()
{
    var options = new DbContextOptionsBuilder<BookDbContext>()
        .UseInMemoryDatabase(databaseName: "TestDB")
        .Options;

    using var context = new BookDbContext(options);
    var controller = new BookingController(context);

    var booking = new Booking
    {
        PassengerFirstname = "Max",
        PassengerLastname = "Mustermann",
        FlightId = 2,
        TicketCount = 1
    };

    var result = await controller.CreateBooking(booking);

    var createdAtResult = Assert.IsType<CreatedAtActionResult>(result);
    var createdBooking = Assert.IsType<Booking>(createdAtResult.Value);
    Assert.Equal("Max", createdBooking.PassengerFirstname);
}
```

## 2 Auswerten

### 2.1 Fazit

Das Projekt war technisch sehr anspruchsvoll und oft frustrierend. Besonders die Einrichtung der Kommunikation zwischen den Services, das Debugging von JWT-Problemen und Docker-Konfigurationen führten zu vielen Rückschlägen. Dennoch konnte ich durch diese Herausforderungen viel lernen und meine Fähigkeiten deutlich erweitern.

## 3 Quellen

### 3.1 Bild Verzeichnis

<i>Bild 1: Aufgabe Übersicht, Erweiterungen Level 3 .....</i>	<b>Fehler! Textmarke nicht definiert.</b>
Bild 2: Skybooker Zeitmanagement.....	4
Bild 3: <i>Docker</i> .....	6