

Tanzania Eksperten

Rapport – Programmering 2 og Teknologi 2

UCL Erhvervsakademi og Professionshøjskole

DMVF241 – 3. semester

Forår 2025



Skrevet af Karina, Amjad, Thomas og Rene

Indholdsfortegnelse

Indledning	3
Problemformulering	3
Afgrænsning	3
Metode.....	4
Analyse	5
Systemarkitektur og teknologivalg	5
Backendudvikling og API	7
Databasesdesign.....	10
Deployment og Docker	11
Sikkerhed	11
Konklusion	12
Perspektivering	13
Videreudvikling	13
Litteraturliste	14
Bilag.....	15

Indledning

Denne rapport tager udgangspunkt i udviklingen af et nyt system til rejsebureauet *Tanzania Eksperten*. Projektet er udført i samarbejde mellem undervisere og product owner Peter Mwamoto fra virksomheden Tanzania Eksperten (herefter forkortet Tane).

Formålet med projektet er at skabe værdi for Peter og Tane ved at udvikle et system, der effektiviserer bureauets arbejdsgange samt administration af kunder og rejser.

Rapporten indledes med en præsentation af de anvendte metoder og begrundelserne for valget af disse. Derefter følger en analyse af kravspecifikationen, som er udleveret af Peter, og som har dannet grundlaget for udviklernes fokus og løsninger. Afslutningsvis præsenteres det endelige systemdesign og processen bag udviklingen af systemet.

Problemformulering

Den primær problemstilling for projektet var finde en måde at effektivisere kommunikation mellem Peter og hans klienter. Mere specifikt, gør det nemmere for:

1. At oprette rejseplaner for klienter.
2. Sende de rejseplaner, som pdf'er, til hans klienter for gennemgang.
3. Modificer derefter rejseplanerne efter klientens ønsker.
4. Sende Fakturaer til klienten for deres ønsket rejse.

Og endeligt

5. Skabe overblik over hans klienter og hvor lang de er i bookingprocessen.

Vores fokus for dette projekt vil derfor være at skabe et web app der kan udøve alle disse funktioner og gøre det simpelt at benytte for brugeren.

Det skal også være muligt at udvide brugen af systemet, så flere virksomheder kunne benytte sig af systemet. Derfor implementerer vi også et administratorsystem til at styre flere brugere og et login system til at facilitere det system.

Afgrænsning

Vi har brugt OOP (Objekt orienteret programmering) til at definere klasser, der har hvert sit ansvarsområde. Vi har opdelt projektets kode i moduler, som kan genbruges og manipuleres med uafhængigt af hinanden.

Vores afgrænsede domæner (bounded contexts) er at TANA.Domain indeholder domænemodeller og forretningslogik, vores TANA.Application indeholder applikationslogik og DTOer, vores TANA.Infrastructure indeholder tekniske tjenester,

som f.eks. pdf-generering. Vores TANA.Persistence indeholder dataadgang og repositories, og vores TANA.Web og TANA.API som indeholder præsentations- og API-laget.

Domænesproget i klasserne 'Rejse' 'RejseTur' 'Kunde' 'Bruger' 'Faktura' og 'Tur' er forretningsbegreber fra rejsebranchen.

Angående testning, så gør brug af interfaces og Application-laget det nemmere at skrive unit tests og mocke afhængigheder.

Begrænsninger

Kunder kan ikke interagere digitalt, de modtager kun pdf'er og der er ingen selvbetjeningsportal eller betalingsløsning eller bookingfunktionalitet.

Pdf'erne skal genereres manuelt af en medarbejder. Der er ingen automatisk afsendelse.

Der er ingen kortintegration, der vises ingen ruteplanlægning, eller geografisk visualisering af rejseplanerne.

Der er ingen prisberegninger, priserne skal indtastes manuelt, og der er ingen rabatter.

Vi har en monolitisk arkitektur, da selvom vi bruger clean architecture, så er hele projektet samlet i én applikation. Vi har altså ikke en microservice arkitektur. Vi bruger SMTP lokalt og derfor har vi heller ingen tredjepartsintegrationer. Vi har ikke implementeret integrationstest, kun en enkelt xUnit test med mocking i begyndelsen af projektet. PDF-genereringen sker lokalt, altså server-side, og gemmes som bilag som så kan sendes, men det er ikke understøttet af cloud.

Metode

Til udviklingen af vores system brugte vi et iterativt agilt system, nemlig SCRUM. Det var iterativt i det at vi havde et enkelt program som vi gradvist byggede op på gennem de flere sprints. Det var agilt i det at vi testede konstant vores system

Vores programmeringssprog har været C# og HTML i frameworket .Net 8. Vi har brugt Git til pull, push, merging og managing af branches, og vi har brugt MSSQL til database. Vi har brugt Docker med Docker Compose til oprette container for vores projekter og bruge Docker-Compose.Yaml til at deploy projektet.

Analyse

Systemarkitektur og teknologivalg

Vi har brugt entiteter i vores TANA.Domain layer for Tur, Rejse, Kunde, Bruger, RejseTur, Faktura, EmailSettings og Template.

For at hente data har vi brugt interface repositories som f.eks. ITurRepository og IRejseRepository.

Vi arbejdede med abstraktioner i domæne-laget og ikke implementeringer, hvilket er klassisk DDD.

Vi anvendte også Dependency Injections for at holde service-laget uafhængigt af persistence-laget ved brug af application services som ITurService og IKundeService.

Vi har holdt implementeringen fra TANA.Persistence adskilt fra vores interfaces. Vores persistence lag er vores infrastruktur. Det er implementering af tekniske detaljer, som f.eks. vores TurRepository som implementerer ITurRepository og bruger AppDbContext, altså vores Entity Framework.

Vi benyttede os af DTO'er, for afkobling og for at have kontrol over datatransport.

Vores TurController API afhænger kun af ITurService og ikke nogen repositories eller EF, som følger idéen af DDD, at API'er er bare en indgang til domænelogikken, og ikke hvor logikken selv bor.

I Tanzania Eksperten projekt valgte vi at bruge Clean Architecture til at lave vores system. Clean Architecture handler om at strukturere projektet i lag, hvor hvert lag har en specifik funktion (SRP). Det fundamentale principiel som Clean Architecture indebærer er at de indre lag gøres helt uafhængigt af de ydre lag. Det gør det nemmere at både skalere systemet ud til flere brugere, samt at udvide det med nye funktioner uden at det kræver store ændringer på systemets kernestruktur.

Efter Clean Architecture har vi opdelt vores projekt i 3 hovedlag.

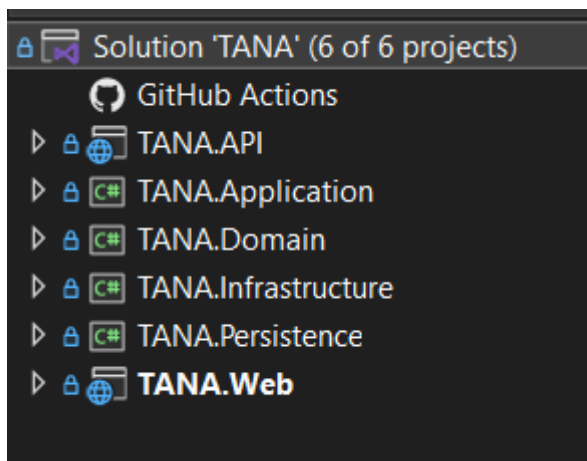
1- TANA.Domain er vores domain lag. Det fungerer som Hovedlaget og er derfor rygraden i vores systemet. Den er kerne laget af vores program og er dermed ikke afhængig af et andet lag. Laget indeholder alle vores grundlæggende klasser som vores system er bygget op af.

2- TANA.Application er applikationslaget, hvilket følger domænelaget. Det indeholder forretningslogik, kommandoer og forespørgsler. Dette lag betragtes som mellemlæddet mellem Domain og Persistence laget. Den indeholder alle de services som de ydre lagere gør brug af for at tilgå klasserne i Domain laget, samt interfaces.

3- TANA.Persistence er vores persistens lager og er ansvarlig for at styre og sikre gyldigheden af hver anmodning og er også ansvarlig for at håndtere Entity Framework. Derfor er det i dette lag hvor alle vores repositories lægger samt vores DbContext klasse. Laget er afhængigt både domain og af applikationslaget.

4- TANA.Infrastructure er ansvarlig for hentning af data fra databasen og er ansvarlig for eksterne tjenester såsom afsendelse af e-mail til kunden og betalingstjenester og afhænger af applikationslaget.

Clean arkitektur i vores projekt sikrer nem skalerbarhed i fremtiden, såsom at tilføje flere betalingsfunktioner eller sende sms'er til kunden via hans telefonnummer.



Vi valgte at bruge Docker til at lave en Docker Compose yml fil, dockerfiles, docker compose override yml fil og skrive Docker Compose Up –Build i vores powershell for at få oprettet vores containers og container images og porte hvor vi kører vores localhost projekt. Vi valgte at bruge .Net 8 i stedet for .Net 9 da .Net 9 stadig er ny og vi ikke har så meget kendskab til den endnu og .Net 8 projekter har adgang til flere anvendelige Libraries.

Til at oprette PDF's brugte vi QuestPDF, en Library der gør det muligt at nemt oprette og formatere PDF's i C#. QuestPDF er også gratis at bruge for mindre virksomheder og studerende, hvilket gjorde det mere attraktivt for os sammenlignet med de betalte alternativer.

Frontendudvikling og UI-design

Til front-end designet brugte vi Blazors framework for at udvikle webapplikationer. Herfra udnyttede vi Razor komponenter og injekt'ede vores services. Dette gør det muligt for brugeren at lave API kald direkte via webappen og dermed interager med databasen.

Siden det var specificeret i opgaven at front'end delen af opgaven var af mindre betydning end back'end i programmet, brugte vi mindre af vores tid på den. Desto mindre skabte vi et simpelt design der ville være nemt, selv for ikke IT-kyndige personer, at gøre brug af.

Vores UI-design, er baseret på simplicitet og gestaltlovene. Vi gør megen brug af popups, til opretning og redigering af data. Dette giver mindre visuelforvirring, da den originale side ikke fyldes med clutter. Hovedsageligt er lovene om nærhed og lighed de drivende faktorer bag UI-designet. På den måde kan vi lave en mere brugervenlig designflade, ved at brugere af systemet nemt forstår hvad knapper gør. Det tydeliggøres kun mere ved brug af forskellige farver, så brugeren nemt kan differentiere mellem funktioner i systemet.

For at implementere concurrency, altså samtidighed funktionalitet mellem flere bruger, gjorde vi brug af Async Tasks. Tasks gør det muligt for programmet at tage imod en liste af kommandoer og udføre dem asynkron i stedet for at skulle vente til første kommando er færdig før den udøver den næste.

Backendudvikling og API

Vi har valgt at lave en PdfController API til vores pdf-generering med en ViewPdf og en Generate for at oprette pdf'en og deres indhold.

Vi har valgt at lave en microservice API controller som hedder TurController, den er testet med GET request og POST request i Postman i <http://localhost:5228/api/tur> og den returnerer en 200 ok i begge requests. Ved at lave en microservice API har vi sørget for at den er loosely coupled. Den er selvstændig og simpel og indeholder kun det mest nødvendige som GetAll, Create og GetByIds. Når en microservice er loosely coupled betyder det at den kan deployes selvstændigt uden at have en effekt på andre ting i projektet. Den er fleksibel og hvis der er en fejl i microservicen, så har det ikke betydning for resten af projektet, og resten af projektet kan derfor stadig godt køre selvom microservicen fejler.

```
[ApiController]
[Route("api/tur")]
1 reference
public class TurController : ControllerBase
{
    private readonly ITurService _turService;

    0 references
    public TurController(ITurService turService)
    {
        _turService = turService;
    }

    [HttpGet]
    0 references
    public async Task<IActionResult> GetAll()
    {
        var ture = await _turService.GetAllTurAsync();
        return Ok(ture);
    }

    [HttpPost]
    0 references
    public async Task<IActionResult> Create([FromBody] TurDto dto)
    {
        await _turService.CreateTurAsync(dto.Navn, dto.Description, (int)dto.Pris, dto.Dage);
        return Ok();
    }

    [HttpPost("byids")]
    0 references
    public async Task<IActionResult> GetByIds([FromBody] List<int> ids)
    {
    }
}
```

POST http://localhost:5228/api/tur Send

Params Auth Headers (9) Body Scripts Settings Cookies Beautify

raw JSON

```
1 {
2   "navn": "Svømmetur i søen",
3   "description": "Tag med på tur til søen i Tanzania, og få en dykkert",
4   "pris": 200,
5   "dage": 1
6 }
```

Body 200 OK 313 ms 92 B

Raw Preview Visualize

1

http://localhost:5228/api/tur Save Share

GET http://localhost:5228/api/tur Send

Params Auth Headers (9) Body Scripts Settings Cookies

Query Params

	Key	Value	Description	Bulk Edit
	Key	Value	Description	

Body 200 OK 100 ms 466 B

{ } JSON Preview Visualize

```
14   "dage": 1
15 },
16 {
17   "id": 7,
18   "description": "Tag med på tur til søen i Tanzania, og få en dykkert",
19   "navn": "Svømmetur i søen",
20   "pris": 200,
21   "dage": 1
22 }
23 ]
```


Vi har brugt single responsibility til alle vores entiteter. De indeholder kun det mest nødvendige og hænger ikke sammen med hinanden (entiteterne). De opererer derfor ud fra single responsibility princippet.

Open/closed princippet bruges i projektet i vores klasser og funktioner. De er åbne for udvidelse, men lukkede for ændringer. Det vil sige at en ny funktion kan tilføjes uden at ændre på den allerede eksisterende kode.

Liskov substitution princippet bruger vi til når vores subklasser skal skiftes ud med vores grundklasser uden at ændre programmet. Vores subklasser opfører sig altså forudsigeligt når de bruges i stedet for vores grundklasser.

Vi har gjort så vores klienter ikke tvinges til at bruge metoder som vi ikke bruger med interface segregation princippet. Vi har altså lavet flere små interfaces i stedet for ét stort, for at undgå netop dette, og gøre så vi bruger mere specifikke metoder og funktioner.

I dependency inversion princippet siger man, at høj-level moduler ikke skal afhænge af lavt-level moduler, men at de begge skal afhænge af abstraktioner. Detaljer skal altså afhænge af abstraktioner og ikke omvendt.

Koden i projektet er forholdsvis isoleret, hvilket betyder at testbarheden er nem.

TDD står for Test Driven Development, hvor man tester først vha. XUnit tests, de punkter af sit program, som man senere ønsker at videreudvikle eller vælger at refaktorere, før man arbejder videre med sit projekt. Du skriver altså dine tests først. Dernæst laver man stub implementeringer, som er vores mocking, hvorefter vi så implementerer inkrementelle implementeringer, hvilket betyder at vi så har implementeret logik, altså vores klasser og interfaces, i dette tilfælde under vores Domain layer, hvor vi har gjort brug af Clean Architecture. Til sidst har vi refaktoreret koden og testene, for at det stemmer overens med hinanden og kommunikerer med hinanden (fra projekt til xunit test projekt).

```

public class UserServiceTests
{
    private readonly Mock<IUserRepository> _userRepositoryMock;
    private readonly IUserService _userService;

    0 references
    public UserServiceTests()
    {
        _userRepositoryMock = new Mock<IUserRepository>();
        _userService = new UserService(_userRepositoryMock.Object);
    }

    [Fact]
    0 references
    public async Task CreateUserAsync_ShouldCreateUser_WhenUsernameIsAvailable()
    {
        // Arrange
        var username = "testuser";
        var password = "password123";

        _userRepositoryMock.Setup(r => r.GetByUsernameAsync(username))
            .ReturnsAsync((User)null);

        _userRepositoryMock.Setup(r => r.AddAsync(It.IsAny<User>()))
            .ReturnsAsync((User u) => u);

        // Act
        var result = await _userService.CreateUserAsync(username, password);

        // Assert
        Assert.NotNull(result);
        Assert.Equal(username, result.Username);
    }
}

```

Dette er et eksempel fra vores tidligste stadie, nemlig Sprint 0, her er vores xUnit test med mocking, i hvertfald én af testene, som tjekker om når vi opretter en bruger, om brugeren så har et tilgængeligt brugernavn.

Vi har benyttet os af Swagger, ved at skrive `app.UseSwagger` og `app.UseSwaggerUI` i vores `program.cs` i vores TANA.API lag. Vores API'er som Swagger benytter sig af er vores `TurController` og `PdfController`.

Databasedesign

Vi valgte at bruge MSSQL til at oprette en database, hvor vi så lavede migrations i Entity Framework. Entityframework gør det nemt at oprette sammenhæng mellem klasserne i vores program og klasserne i databasen. Til dette oprettede vi en `DbContext` klasse der indeholdt alle de klasser som vores database skulle indeholde.

```

6 references
public DbSet<Kunde> Kunder { get; set; }
6 references
public DbSet<Rejse> Rejser { get; set; }
5 references
public DbSet<Tur> Turer { get; set; }
0 references
public DbSet<RejseTur> RejseTurer { get; set; }
0 references
public DbSet<Faktura> Fakturarer { get; set; }
0 references
public DbSet<Admin> Adminer { get; set; }
6 references
public DbSet<Bruger> Brugere { get; set; }
4 references
public DbSet<EmailSettings> EmailSettings { get; set; }
7 references
public DbSet<TemplateEntity> Templates { get; set; }
2 references
public DbSet<TemplateItemEntity> TemplateItems { get; set; }

```

Ved hjælp af migrations kunne vi så nemt genskabe databasen i hvert vores system.

Vi implementerede CRUD-operationer gennem repositories vi derefter kunde kalde i vores komponenter gennem en tilsvarende service. Dette gjorde det unødvendigt at bruge direkte SQL-kommandoer. Da det kun er muligt for en bruger at interagere med databasen på en måde vi har specificeret, er det ikke muligt begå SQL-injektioner, hvilket gør systemet mere sikker.

Deployment og Docker

Til deployment af vores system, har vi benyttet Docker som container-teknologi. Dette gør opsætningen lettere på tværs af forskellige maskiner, samt det sikrer at vores system kører og fungerer ensartet på alle systemer, uanset udviklingsmiljø.

Vi har oprettet en docker-compose fil, som henviser til de to dockerfiler vi har for hhv. TANA.Api og TANA.Web. De enkelte dockerfiler definerer hvordan containerne skal opbygges og køres, samt hvilke porte de skal mappes til. De henter alle nødvendige afhængigheder ned, kompilere koden, og kan så kommunikere på tværs af containere.

Med docker-compose, kan hele systemet nemt startes, enten via docker desktop, eller direkte fra udviklingsmiljøet med en command (docker compose up –build). Derfor kan vi både teste individuelle dele af systemet og den komplette løsning.

Sikkerhed

Under systemudviklingsfasen blev der lagt stor vægt på at beskytte følsomme data for både klienter og brugere, både under lagring i databasen og under transmission.

Der er truffet adskillige sikkerhedsforanstaltninger for at beskytte disse data, herunder anvendelse af OWASP Top 10-principperne, samt authorization/authentication, når brugere logger ind på systemet.

Eksempler på trusler, der er blevet adresseret, omfatter:

A01 – Broken Access Control:

Det er en af de mest alvorlige sikkerhedssårbarheder, og den er blevet undgået ved at begrænse brugernes adgang til bestemte funktioner eller følsomme kundedata.

For eksempel er det kun en administrator, der kan få adgang til siderne for brugeroprettelse og -sletning, mens en agent kan oprette og sende ture til sine tilknyttede kunder.

A03 – Injektion

For at beskytte felter mod SQL Injection-angreb blev EF CORE brugt i stedet for at bruge manuelt SQL-forespørgselssprog.

A05 – Security Misconfiguration:

Denne sårbarhed blev undgået gennem flere foranstaltninger, herunder brug af Docker til at sikre, at unødvendige porte især til databasen blev lukket.

Funktionen er også blevet brugt:

```
app.UseExceptionHandler("/Error", createScopeForErrors: true) i program.cs
```

At skjule følsomme fejlmeddelelser, mens systemet kører i et produktionsmiljø.

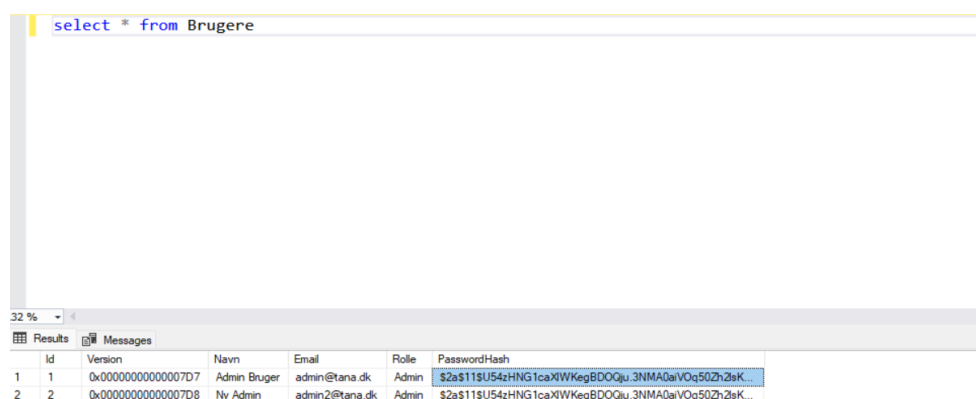
HSTS-protokollen er også aktiveret i program.cs for at tvinge browseren til at bruge den sikre HTTPS og dermed forhindre ethvert forsøg på adgang med HTTP.

Konklusion

Projektet har haft til formål at udvikle et digitalt system til Tanzania Eksperten, som skal effektivisere arbejdsgange for rejsebureauets ansatte. Gennem seks ugers udvikling har teamet implementeret en webapplikation med fokus på brugervenlighed, sikkerhed, skalerbarhed og modulært design.

Ved brug af teknologier som .NET8, Blazor, Entity Framework og Docker, samt principper som Clean Architecture og SOLID, har vi opbygget et fleksibelt og fremtidssikret system. Funktionaliteter som login, oprettelse af rejseplaner, PDF-generering og kundeadministration er blevet gennemført og testet.

For at sikre brugernes adgangskoder gemmer vi ikke adgangskoder som almindelig tekst i databasen. I stedet bruger vi BCrypt-hashing, så hver adgangskode konverteres til en unik hashværdi, før den gemmes. Derudover har vi implementeret rollestyring, HSTS og håndtering af OWASP-trusler som Broken Access Control og Security Misconfiguration. Alt i alt lever systemet op til de krav som blev stillet fra vores Product Owner og skaber værdi for både brugere og virksomheden.



Id	Version	Navn	Email	Rolle	PasswordHash
1	1	Admin Bruger	admin@tana.dk	Admin	\$2a\$11\$U54zHNG1caXlWKeqBDOQju.3NMA0aVOq50Zh2aK...
2	2	Ny Admin	admin2@tana.dk	Admin	\$2a\$11\$U54zHNG1caXlWKeqBDOQju.3NMA0aVOq50Zh2aK...

Perspektivering

Med den stigende afhængighed af digitale løsninger, som TANA, er der også et konstant behov for at innovere, der opfylder virksomheders behov, og sikrer driftshastighed og muliggør systemudvidelse ved at tilføje nye funktioner. Vi har som team taget dette i betragtning, da vi designede TANA systemet, så det kan modificeres og udvikles senere, hvis det er nødvendigt.

Microservice arkitektur og Docker containere bruges, hvilket gør det mere fleksibel, f.eks. er det muligt at tilføje nye funktioner uden at skulle re-designe hele systemet for også at undgå omkostninger. Det gør også det at opdele koden i lag det nemmer at ændre hvor som helst, da der er lavere coupling mellem de individuelle lag.

Måske kunne systemet på et tidspunkt også bruges af andre rejsebureauer, hvis de ændrer navn og logo. Ting som MobilePay-betalinger kan også tilføjes på grund af den stigende efterspørgsel og hastighed på betalinger, eller integration med andre API'er eller Azure-cloudløsninger.

Videreudvikling

Der er mange ting, vi gerne ville have gjort, hvis vi havde haft mere tid.

f.eks. kan man foretage en betaling direkte, så kunden modtager en faktura med et betalingslink fra MobilePay via e-mail sammen med rejseoplysningerne eller ved godkendelse af rejseplanen. Jeg tror, at denne tilføjelse vil gøre tingene lettere for både kunden og medarbejderen.

man kan også oprette en måde, hvor kunden kan modtage e-mail notifikationer, hvis der er ændringer i rejeseturen, rejsetiden eller andre vigtige oplysninger.

Tilføj kalenderintegration, så rejseplaner automatisk kan føjes til kundens Google eller Outlook kalender på deres enhed som en påmindelse til kunden om rejsetid.

Måske på længere sigt kan hele systemet implementeres i Azure.

Litteraturliste

- 1- OWASP Foundation – OWASP Top 10 sikkerhedsvejledning

<https://owasp.org/www-project-top-ten/>

- 2- QuestPDF – Dokumentation for PDF-generering

<https://www.questpdf.com>

- 3- Stack Overflow – Løsninger på specifikke tekniske udfordringer

<https://stackoverflow.com/questions>

- 4- OpenAI's ChatGPT som støtteværktøj til ideudvikling, sproglig formulering, stavekontrol.

<https://chatgpt.com>

- 5- OOP - Afgrænsning

[Object-Oriented Programming - C# | Microsoft Learn](#)

- 6- Bounded context - Afgrænsning

[What is Bounded Context? | Dremio](#)

7- Domain Driven Design

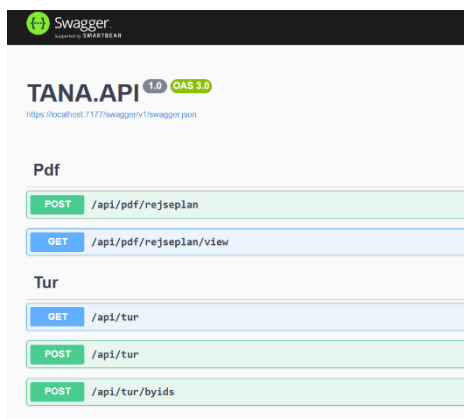
[Domain-Driven Design \(DDD\) | GeeksforGeeks](#)

8- SOLID – SOLID-principperne

[What are SOLID Principles? | Contabo Blog](#)

Bilag

Swagger-dokumentation



Docker Compose configuration

