Projektuppgift

DT071G – Programmering i C#.NET

Web API
ASP.NET Core Web API

Albin Rönnkvist

Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MITTUNIVERSITETET
Avdelningen för informationssystem och -teknologi

Författare: Albin Rönnkvist, alrn1700@student.miun.se
Utbildningsprogram: Webbutveckling, 120 hp
Huvudområde: Datateknik
Termin, år: HT, 2021

Sammanfattning

Jag har skapat ett Web API som hanterar back-end-funktionalitet för inlägg och användare, som kan användas till att skapa en blogg, ett forum eller liknande.

Web API:t är skapat med ramverket ASP.NET Core och tillvägagångssättet code-first med Entity Framework Core som skapar och ansluter till en SQL Server-databas. Säkerhet är implementerad med dolda miljövariabler, krypterade lösenord och JSON Web Tokens för autentisering och auktorisering av användare.

Koden är strukturerad med designmönster som MVC, Dependency Injection och Repository Pattern.

Nyckelord: C#, ASP.NET Core Web API, Entity Framework Core, SQL Server, Miljövariabler, Kryptografi, JWT, MVC, Dependency Injection, Repository Pattern

Innehållsförteckning

Terminologi

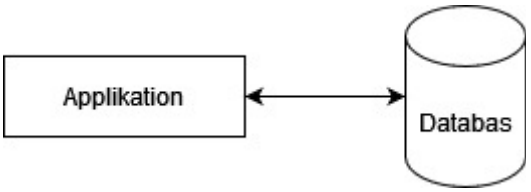
Akronymer/Förkortningar

DTO	Data Transfer Object
SDK	Software Development Kit
ORM	Object-Relational Mapping
MVC	Model-View-Controller
EF	Entity Framework
API	Application Programming Interface
IoT	Internet of Things
CRUD	Create, Read, Update & Delete
HTTP	HyperText Transfer Protocol
SQL	Structured Query Language
SSMS	SQL Server Management Studio
JSON	JavaScript Object Notation
JWT	JSON Web Token
DI	Dependency Injection
IoC	Inversion of Control

1 Introduktion

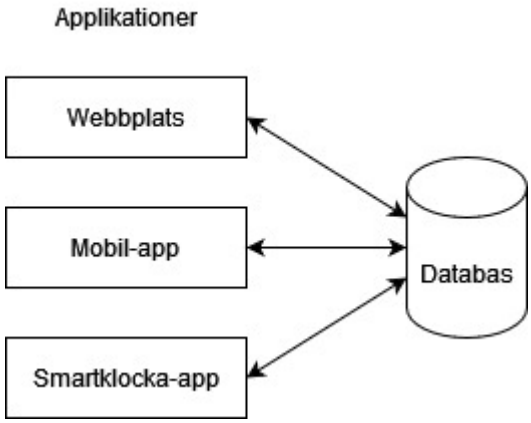
1.1 Bakgrund och problemmotivering

För att skapa en webbaserad produkt krävs oftast en applikation där besökare kan se produkten och interagera med den, samt en databas för att lagra produktens affärsdata. Så genom att kombinera en applikation med en databas får vi en fullt fungerande dynamisk applikation (se Figur 1).



Figur 1.

Om produkten växer och man vill stödja flera enheter som webbplatser, mobilapplikationer och IoTs måste man oftast skapa separata applikationer för varje enhet. Dessa applikationer kommer då direkt ansluta till en och samma databas (se Figur 2).



Figur 2.

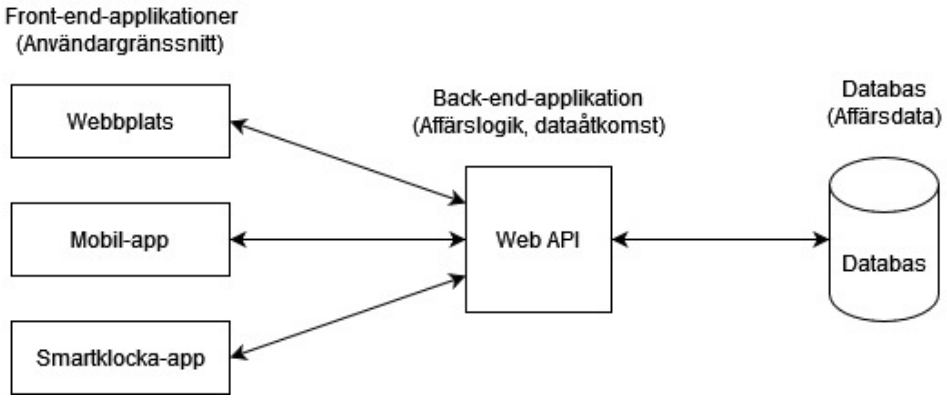
Vilket kan leda till följande problem [2]:

- **Duplicering av logik för varje applikation**: produkten ska ha affärslogik, det vill säga regler som bestämmer hur affärsdata kan skapas, lagras och ändras. I varje applikation måste man skapa samma logik för att ansluta till databasen. Det betyder att man repeterar kod och slösar bort både tid och pengar.

- **Risk för felaktig logik och kod**: eftersom logiken måste skrivas om i flera applikationer kan det hända att utvecklare gör misstag som leder till att olika applikationer kan ha annorlunda logik vilket i sin tur kan påverka affärsdatan.

- **Svårt att upprätthålla och skala**: denna struktur med direkt koppling mellan applikation och databas är svår att upprätthålla och skala. Om man vill uppdatera och förbättra affärslogiken måste man repetera ändringarna i varje kodbas eftersom logiken är skriven på olika ställen.

En lösning för detta är att använda Web API. Ett Web API är en fristående back-end-applikation som kan ansvara för all affärslogik och dataåtkomst. Det skapar en lös koppling mellan applikationerna och databasen där Web API:t är bron som kopplar de samman. Så en applikation pratar med Web API:t som i sin tur pratar med databasen (se Figur 3).



Figur 3.

Denna struktur leder till fördelar som [2]:

- **Ingen duplicering av kod och minskad risk för felaktig kod**: affärslogiken behöver bara skrivas en gång i Web API-applikationen. Den kan sedan konsumeras av flera olika applikationer samtidigt som återanvänder logiken utan att behöva skriva om koden. Det garanterar också att alla applikationer har samma logik och då kan det bara bli fel på ett ställe som man bara behöver ändra en gång.

- **Ökad flexibilitet**: teknikvärlden är fartfylld där nya enheter och front-end-lösningar dyker upp hela tiden. Med en återanvändbar back-end kan utvecklare fokusera på att skapa front-end-applikationer med valfria teknologier som stödjer olika enheter.

- **Ökad säkerhet**: själva koden som är skriven i Web API:t visas inte för applikationer som konsumerar den, de kan komma åt data och funktionalitet men de kan inte se hur detta implementeras. Applikationerna får inte heller en direkt koppling till databasen.

Sammanfattningsvis finns det många fördelar med att skapa ett Web API, framförallt i större projekt. Det kan spara både tid och pengar samt förbättra produkten.

I detta projekt ska jag skapa ett Web API som hanterar affärslogik och dataåtkomst. API:t ska kunna konsumeras av andra applikationer, via internet. API:t ska också jobba med persistent affärsdata i en databas.

1.2 Övergripande syfte

Projektets övegripande syfte är att utveckla ett Web API som jobbar med persistent data.

1.3 Avgränsningar

Projektet innefattar utveckling av ett Web API och delvis databashantering samt säkerhet. Rapporten tar inte upp andra användningsområden för API eller utveckling av användargränssnitt som konsumeras av API:t. Det kommer ske ständiga testningar av koden i projektet men med hänsyn till längden på rapporten kommer det exkluderas från rapporten.

1.4 Detaljerad problemformulering

1.4.1 Utvecklingsmiljö

Utvecklingsmiljön ska installeras och konfigureras för att kunna påbörja kodningen av applikationen.

1.4.2 Web API

API:t ska skapas och hantera inlägg samt användare som måste definieras med egenskaper och relationer.

API:t måste kunna nås med HTTP-anrop via en sökväg med olika slutpunkter som specificerar vilken funktionalitet klienten kan komma åt. Det måste också skickas tillbaka ett HTTP-svar med en statuskod och data, som kan skilja sig åt beroende på hur anropet behandlades av API:t.

Följande funktionalitet måste implementeras:

- **Inlägg**:
    - Ej autentiserad: läsa alla inlägg och läsa ett enda inlägg.
    - Autentiserad: skapa inlägg samt redigera och radera egna inlägg.
- **Användare**:
    - Ej autentiserad: läsa alla användare och läsa en användare. Registrera med unikt användarnamn och e-postadress samt ett lösenord som krypteras. Logga in.
    - Autentiserad: redigera egen användare, radera egen användare (och samtidigt radera användarens inlägg).

1.4.3 Databas

En databas behöver skapas och ansluta till API:t så att det kan jobba med persistent data.

Inlägg och användare ska kunna lagras databasen i en 1:N-relation, det vill säga att ett inlägg är skapat av en användare och en användare kan skapa många inlägg.

1.4.4 Säkerhet, autentisering och auktorisering

Viss funktionalitet måste säkras med autentisering och auktorisering. Därför behöver användare kunna lagras på ett säkert vis för att sedan autentiseras och auktoriseras.

2 Teori

2.1 C#

C# är ett objektorienterat och typ-säkert programmeringsspråk skapat av Microsoft. Utvecklare kan använda detta språket till att skapa många olika typer av program som körs på .NET-plattformen [29].

2.2 .NET

.NET Framework, .NET Core, Xamarin och .NET Standard är relaterade överlappande plattformar för utvecklare som används för att bygga applikationer och tjänster [1].

- **.NET Framework**:
  Byggdes för att fungera bäst på Windows-maskiner, praktiskt sett så används detta endast till att bygga Windows-applikationer. Det är en utvecklarplattform som innehåller en *CLI(common language runtime)* för att exekvera kod och en *BCL(Base Class Library)* som är ett stort bibliotek av klasser som kan användas till att bygga applikationer [1].

- **.NET Core (.NET 5.0+)**:
  Är en plattformsoberoende och "bantad" version av .NET Framework utan Windows-specifikt legacy-innehåll.
  .NET Core innehåller en plattformsoberoende implementation av CLR som kallas CoreCLR och ett bantat bibliotek av klasser som kallas CoreFX [1].

- **Xamarin**:
  Används för utveckling av mobilapplikationer.

- **.NET Standard**:
  En specifikation som representerar en samling av API:er som alla .NET-plattformer kan använda för att indikera vilken nivå av kompatibilitet de har.
  Eftersom de 3 ovanstående plattformarna kan vara lite olika att programmera i så underlättar den standarden för plattformsoberoende kompatibilitet och underlättar återanvändning av kod. Det skapar en röd tråd genom hela .NET-ekosystemet [1].

2.3 ASP.NET Core

ASP.NET Core är ett ramverk med öppen källkod, skapat av Microsoft, för att bygga moderna webbapplikationer och tjänster med .NET. Ramverket är plattformsoberoende och kan köras på Windows, Linux, macOS och Docker [3].

ASP.NET Core kan användas för att bygga Web API:s med .NET [4].

2.4 API & Web API

Ett API (Application Programming Interface) är ett gränssnitt som har en samling av metoder som tillåter andra system att få tillgång till funktionalitet eller data från en applikation, operativsystem eller andra tjänster [5].

Ett Web API är ett API på webben som kan nås via internet. Det är som en webbtjänst, fast det stödjer bara HTTP-protokollet. Dessa Web APIs kan konsumeras av en mängd olika klienter som webbläsare, mobilapplikationer, desktopapplikationer och IoTs [6].

2.5 HTTP

HTTP står för HyperText Transfer Protocol och är ett protokoll som beskriver hur webbaserade applikationer ska kommunicera och skicka data mellan varandra [7].

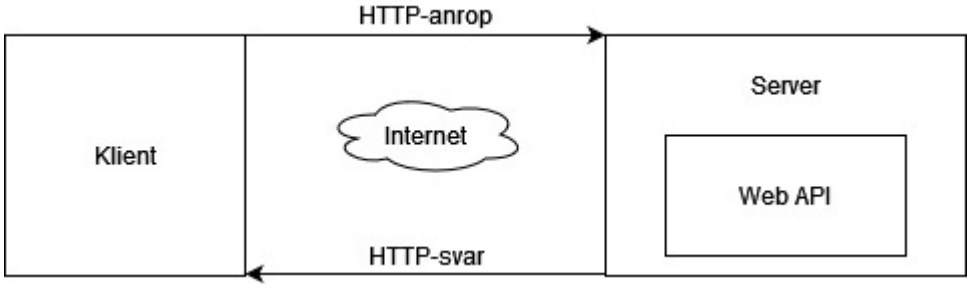
HTTP är ett klient-server protokoll (se Figur 4). I detta projekt är Web API:t en applikation som körs på en server. Den kommunicerar med klienter(webbplatser, mobilapplikationer osv) via anrop och svar på följande vis [8]:

1. Klienten skickar ett anrop till servern, via internet, som servern tar emot.

2. Servern kör sedan applikationen(Web API:t) som utför en metod som klienten efterfrågade. Servern har flera slutpunkter som utför olika metoder, dessa metoder kan vara saker som att läsa eller modifiera data i en databas.

3. När servern har utfört en metod så skickar den tillbaka ett svar till klienten, t.ex. resurser som klienten efterfrågade.

4. Klienten tar sedan emot dessa resurser och kan göra något med dem, t.ex. skriva ut data på en webbplats.



Figur 4.

2.5.1 Anrop (från klient)

Ett anrop består bland annat av följande element [8]:

- **En HTTP-metod**: ofta GET, POST, PUT, PATCH eller DELETE. Dessa metoder anger om vi vill läsa, skapa, uppdatera eller radera resurser.

- **Sökvägen till resursen**: http://example.com/api/posts. Det innehåller protokollet, domänen och en slutpunkt för en resurs. Den kan även innehålla en frågesträng med flera parametrar.

- **Valfria headers**: skickar med extra information till servern. T.ex. autentisering.

- **En valfri body**: innehåller resurser som ska skickas vidare till servern. Används t.ex. i POST-metoder för att skicka med data som ska skapas.

2.5.2 Svar (från server)

Svar innehåller bland annat följande element [8]:

- **En statuskod**: anger om anropet lyckades eller inte, och varför. Det finns följande grupper av statuskoder:
  - 1XX (100, 101, 102 osv): information.
  - 2XX: anropet lyckades.
  - 3XX: omdirigering.
  - 4XX: anropet misslyckades på grund av att klienten skickade ett felaktigt anrop.
  - 5XX: anropet misslyckades på grund av ett fel hos servern.
- **Ett statusmeddelande**: en kortare beskrivning av statuskoden.
- **Valfria headers**: lika som i ett anrop.
- **En valfri body**: resurser som ska skickas tillbaka till klienten.

2.6 MVC

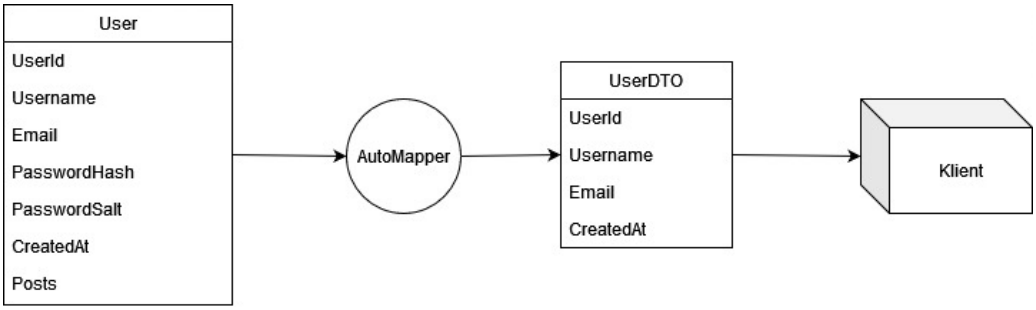
MVC är ett designmönster som används för att dela upp kod i [9]:

- Models: affärsobjekt. Beskriver hur ett objekt ska se ut med specifika egenskaper.
- Views: användargränssnitt. Det som användaren ser och kan interagera med.
- Controllers: applikationens logik, funktionalitet och beslut.

2.7 DTOs & AutoMapper

Ett DTO(Data Transfer Object) är ett objekt som definierar hur data ska skickas över nätverket.
Med DTOs kan man modifiera modeller och välja ut vilken data som ska visas för klienten. Då undviker man att skicka med hela objektet med alla egenskaper som visar hela databas-tabellen vilket kan vara osäkert och onödigt [10].

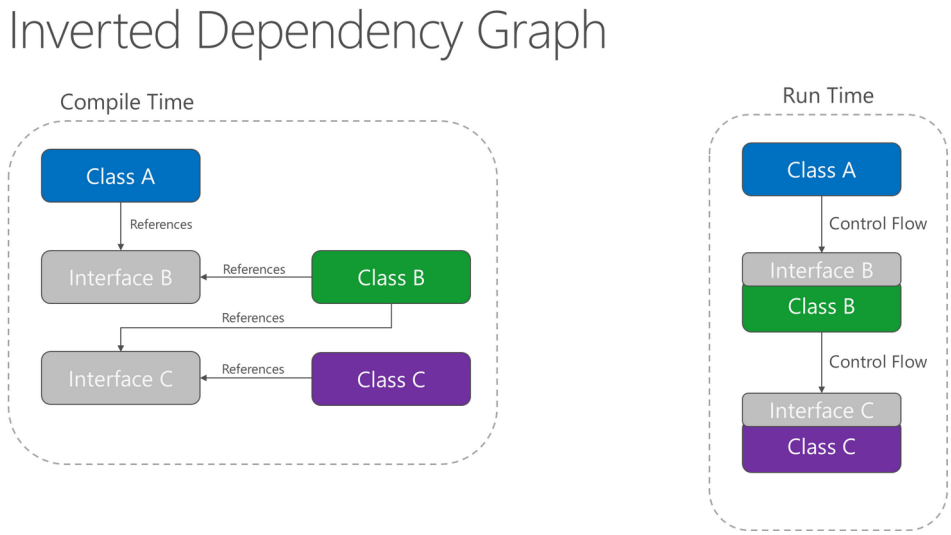
AutoMapper är en simpel återanvändbar komponent som automatiskt kopierar data från en objekt-typ till en annan (se Figur 5). Den fungerar som en bro mellan ursprungs-objektet och destinations-objektet där den mappar egenskapernas data i båda objekten [11].



Figur 5.

2.8 Dependency injection

En dependency är ett objekt som ett annat objekt är beroende av. Dependency Injection (DI) är ett designmönster som används för att uppnå Inversion of Control (IoC) mellan klasser och dess dependencies. Det innebär att klasser inte ska vara beroende av dependencies, utan klasser ska vara beroende av abstraktioner(t.ex. interfaces) av dependencies (se Figur 6).



Figur 6. (Källa [27])

I .NET kan detta mönster implementeras på följande vis [26]:

- Skapa ett interface som representerar en abstraktion av implementationen av en dependency.

- Registrera en dependency som en ”service” i en service container (en service är ett objekt som erbjuder en tjänst till andra objekt). .NET tillhandahåller en inbyggd service container, *IServiceContainer*. Services registreras i appens startpunkt, *Program.cs*, och läggs då till i *IServiceCollection* som är en samling av alla services som applikationen kan använda. Det kan ses som en samling med nyckel/värde-par där nyckeln är typen av ett objekt(ofta ett interface) som man vill hämta och värdet är klassen som implementerar interfacet [28].

- Injektera en service i konstruktorn i ”beroende-klassen” som är beroende av denna service. Ramverket kommer då skapa en instans av implementations-klassen i beroende-klassen.

Man behöver då inte specificera vilken implementations-klass som ska användas direkt i beroende-klassen, utan bara vilket interface. Implementations-klassen specificeras istället i projektets startpunkt.
Vill man ändra implementations-klass som en service implementerar så behöver man bara uppdatera konfigurationen i *Program.cs*, man behöver inte röra någon beroende-klass.

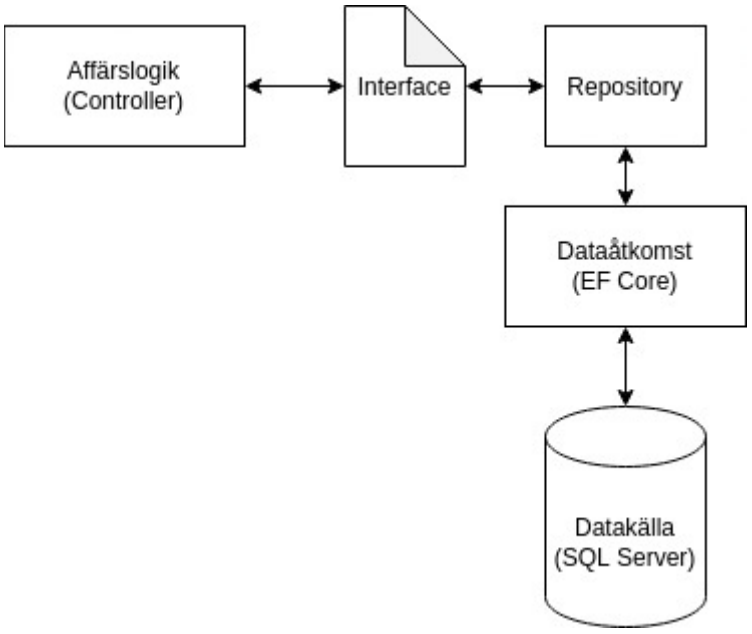
2.9 Repository pattern

Repository pattern är också ett designmönster. Med detta mönster skapar man en lös koppling mellan affärslogiken(controllers) och dataåtkomsten. På så vis vet inte en controller hur dataåtkomsten implementeras och vice versa (se Figur 7). Mönstret innehåller följande delar [12]:

- **Interfaces**: man skapar först ett interface. Interfacet är ett kontrakt som säger att alla klasser som implementerar detta kontrakt måste implementera alla members(metoder, egenskaper osv) som definieras i kontraktet.

- **Repositories**: sedan skapar man olika implementations-klasser, som kallas repositories, som implementerar ovanstående interface. Dessa klasser inkapslar all logik som krävs för att komma åt datakällor med hjälp av lagret för dataåtkomst.

- **Controllers och dependency injection**: i controllers injekteras sedan ett interface med dependency injection (som har registrerats som en service i applikationens startpunkt). Då skapar ramverket en instans av interfacets implementations-klass(ett repository i detta fall) som gör det möjligt att använda repository-objektet i controllern.



Figur 7.

Controllern får då tillgång till funktionalitet som beskrivs i interfacet, men controllern bryr sig inte om hur detta implementeras. Repositoriet bestämmer då fritt hur den ska komma åt data med valfritt lager för dataåtkomst från valfri datakälla. Med hjälp av dependency injection kan man enkelt växla mellan olika repositories genom att ändra konfigurationen i en service, där man ändrar interface-nyckelns värde till ett annat repository. På så vis har applikationen en lös koppling mellan controllers och dataåtkomsten som gör den flexibel för ändringar.

2.10 Microsoft SQL Server

Microsoft SQL Server är en relationsbaserad databashanterare med SQL som frågespråk. Systemet är byggt för att hantera och lagra data. Det är ett klient-server-baserat system, som betyder att det fungerar som en server, som innehåller många databaser, med ett flertal klienter som kan komma åt databaserna. Klienterna kan vara applikationer som körs på samma dator eller på en annan dator som får åtkomst över ett nätverk [13].

2.11 ORM, Entity Framework Core & code-first

ORM står för Object-Relational Mapping som är en teknik där objekt används för att koppla programmeringsspråket till databassystemet. En utvecklare behöver då bara jobba med affärsobjekt i koden medan ORM-verktyget genererar SQL som databasen kan förstå [14].

Entity Framework Core eller EF Core är Microsofts officiella plattform och kan användas som ett ORM-ramverk [15].

Code-first är det vanligaste tillvägagångssättet i EF Core. Detta betyder att man först skapar koden och sedan databasen på följande vis:

1. **Kod**: först skapar utvecklaren affärsobjekt(modeller) som t.ex. användare och inlägg samt en extra klass som EF Core behöver för att kommunicera med databasen.

2. **Databas**: baserat på dessa klasser skapar EF Core en databas med relevanta tabeller och relationer.

[16]

2.12 Asynkron programmering

I grunden har applikationer ett synkront flöde där kodens körs på en rak linje, en sak åt gången. Det vill säga ett en programsats måste fullföras innan nästa programsats kan startas. Om t.ex. en funktion är beroende av ett resultat från en annan funktion, så måste den vänta på att den andra funktionen körs klart och returnerar ett resultat. Detta kallas för blockerande kod och under denna väntetid hindras användaren från att utföra andra uppgifter samtidigt vilket kan upplevas som att applikationen har fryst [17].

Med ett asynkront programflöde kan flera uppgifter exekveras samtidigt som förhindrar blockering av kod. Detta flöde passar bra vid bland annat I/O-bundna anrop, som t.ex. läsa/skriva till en databas eller anrop till ett API. Det asynkrona flödet utnyttjar applikationens resurser fullt ut och kan göra bakgrundsuppgifter eller ta emot andra anrop samtidigt som de krävande uppgifterna utförs [18].

I C# och .NET görs detta med hjälp av Task-klassen och nyckelorden async / await [19].
En Task representerar en asynkron metod som kan returnera ett resultat. Nyckelorden async och await används för att skapa asynkrona metoder. Async specificerar att det är en asynkron metod. Await säger att den asynkrona metoden inte kan fortsätta förbi en punkt innan en asynkron process inuti metoden är färdig [19].

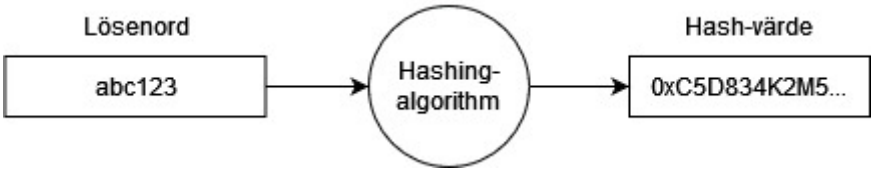
2.13 Kryptografi

Kryptografi är en teknik för att säkra information och kommunikation genom koder. För att på så vis förhindra obehöriga att få tillgång till informationen [20].

2.13.1 Hashing och Hashing-algoritmer

Hashing är en en-vägs-process för att röra om information till ett värde som är obegripligt för en människa. Tanken är att det ska vara omöjligt eller i alla fall extremt svårt att återskapa informationen från det nya värdet [21].

Hashing-algoritmer är funktioner som tar emot ett inmatat ursprungsvärde och utför matematiska operationer på värdet för att generera ett nytt värde, också kallat hash-värde (se Figur 8). Hash-värdet är obegripligt för människan och kan endast återskapas genom att mata in samma ursprungsvärde i algoritmen [21].

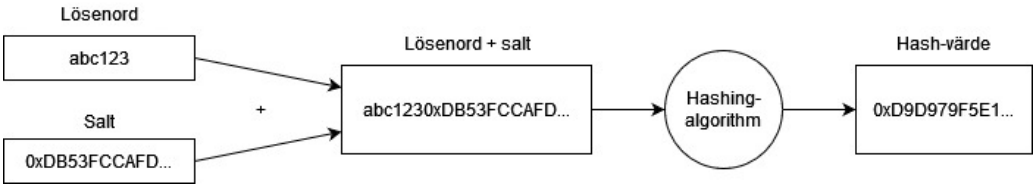


Figur 8.

Dessa algoritmer används därför ofta till att lagra lösenord i databaser. Vid registrering matar användaren in ett lösenord och algoritmen skapar ett nytt hash-värde som lagras i databasen. För att logga in så matar användaren in samma lösenord och då kommer algoritmen generera samma hash-värde som finns i databasen. Sedan kollar applikationen om de matchar och om de gör det så är användaren inloggad.
Om en hackare får tillgång till databasen så kan den inte läsa av lösenordet rakt upp och ned. Utan den måste ta reda på vilken algoritm som har använts och försöka mata in ett lösenord i algoritmen som matchar hash-värdet i databasen, det vill säga gissa sig fram. Det finns program som sköter detta automatiskt och även färdiga tabeller med vanliga lösenord och dess motsvarande hash-värden, som t.ex. rainbow-tabeller [22].
Om man använder en säker algoritm och ett säkert lösenord så är dessa program och tabeller extremt ineffektiva eftersom det finns så oerhört många kombinationer av lösenord och det tar väldigt lång tid att generera så många hash-värden. För säkra lösenord kan det ta över flera hundra miljoner år [23].

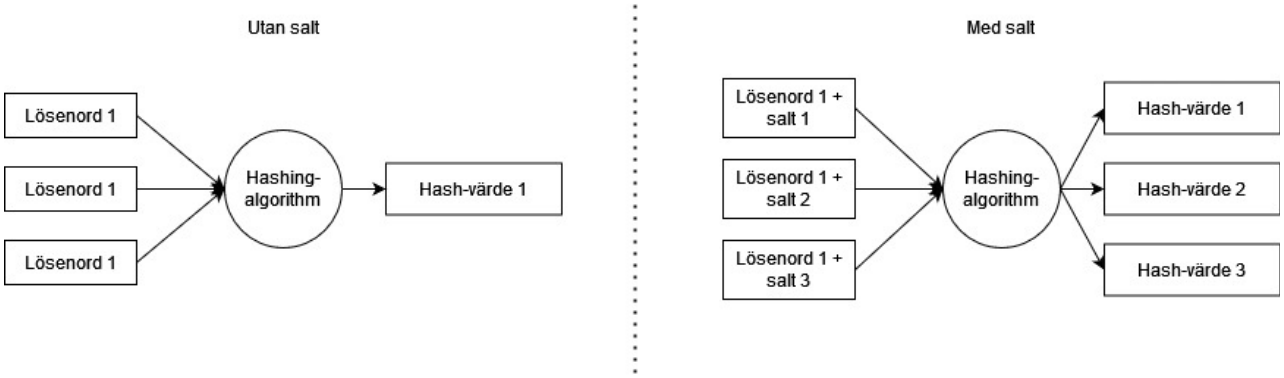
2.13.2 Salt

Vanliga lösenord är enkla att kolla upp i rainbow-tabeller. För att motverka detta kan man använda sig av salt-värden. Ett salt-värde är en sträng som konkateneras med lösenordet innan man matar in det i hashing-algoritmen (se Figur 9).



Figur 9.

Det genererade hash-värdet blir då annorlunda än vad det hade blivit om man bara matade in lösenordet. På så vis går det inte att kolla upp hash-värdet i en rainbow-tabell.
Om man dessutom genererar ett slumpmässigt salt-värde för varje användare så försvårar man det ännu mer genom att likadana lösenord inte får samma hash-värde i databasen. Så om en hackare lyckas knäcka ett lösenord så kan den inte matcha det med något annat lösenord i databasen (se Figur 10).



Figur 10.

[24]

2.13.3 Autentisering med bearer tokens

Bearer-autentisering, också kallat ”token-autentisering” är ett HTTP-autentisering-schema som involverar så kallade ”bearer tokens”. En bearer token är en samling information som lagras i en sträng och skickas med i anrop till en server för att autentisera användaren [30].
Bearer kan definieras som ett säkerhetsbevis där alla som har detta bevis får tillgång genom att vara ”bäraren” av beviset. Det vill säga ”ge bäraren av detta bevis tillgång” [31].

Vid autentisering kan det gå till på följande vis [34]:

1. Klienten skickar användarnamn och lösenord till servern. Sedan kollar servern att användarnamnet existerar i databasen och att det inmatade lösenordet matchar det lagrade lösenordet.

2. Om allt är korrekt genererar servern en bearer token med information baserat på användaren. Sedan skickas denna token tillbaka till klienten.

3. Sedan kan klienten använda denna token i Authorization-headern i HTTP-anrop som servern sedan validerar för att ge åtkomst till skyddade slutpunkter.

2.13.3.1 JSON Web Token

JSON Web Token, också kallat JWT, kan användas som en bearer token. Det är en öppen standard (RFC 7519) som definierar ett säkert sätt att skicka information/claims mellan två parter som JSON-objekt [25] [33].

En JWT består av tre delar [32] [34]:

1. **Header**: ett JSON-objekt som ofta innehåller vilken typ av token(som är en JWT) och hashing-algorithm som används vid signering. Objektet kodas sedan till en Base64Url-sträng.

2. **Payload**: ett JSON-objet som innehåller Claims som kan vara användarinformation som id, användarnamn och roll. Det kan också vara extra metadata som ett startdatum och slutdatum för en token.
   Detta objekt kodas också till Base64Url.

3. **Signature**: en signatur som säkerställer att en token inte har modifierats under dess livslängd. Signaturen skapas genom att man:

   1. först tar den Base64Url-kodade headern och den Base64Url-kodade payloaden och konkatenerar de med en punkt som separator.

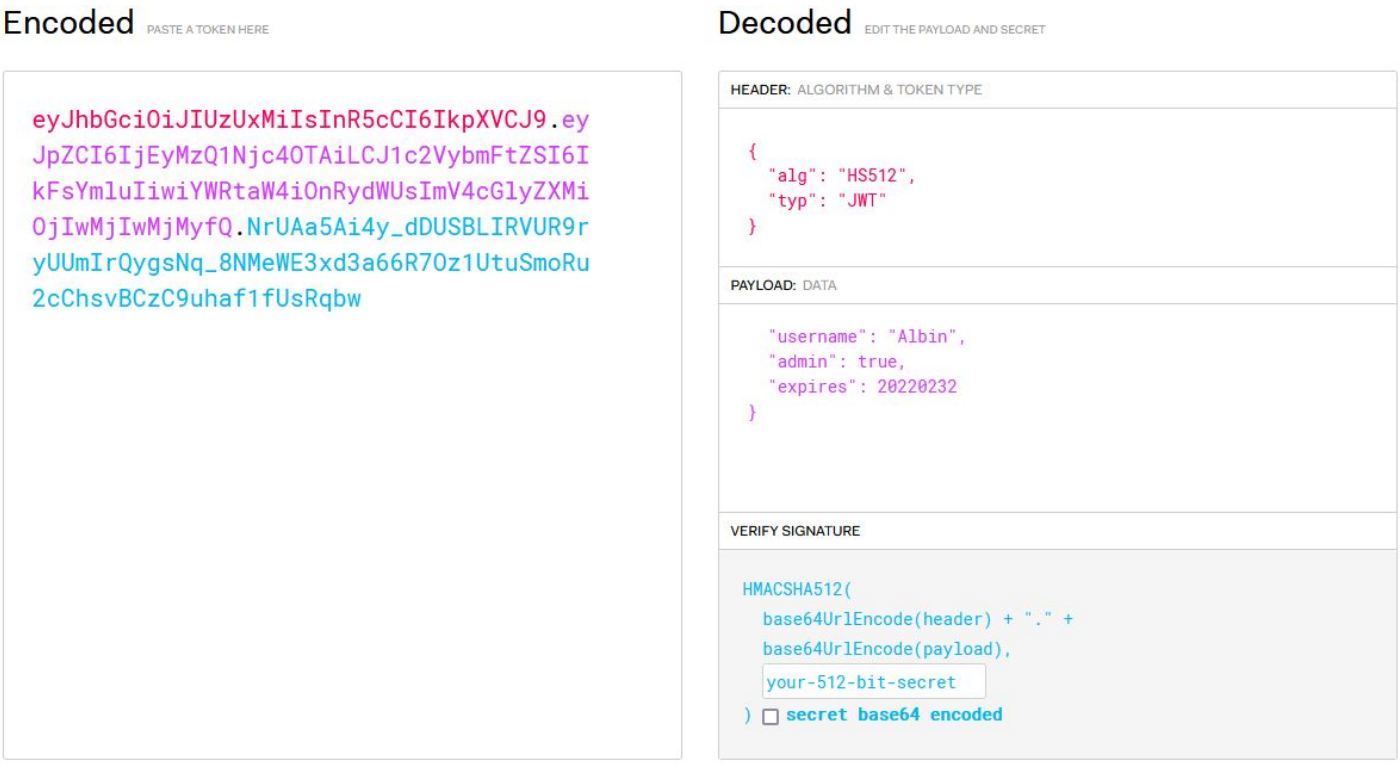
   2. Sedan läggs det till en hemlig nyckel. Den hemliga nyckeln ligger dolt på servern.

   3. Sedan hashas de ovanstående punkterna med hashing-algorithmen som anges i headern vilket skapar ett hash-värde.

   4. Hash-värdet kodas sedan till Base64Url-format.

   Så signaturen är ett Base64Url-kodat hash-värde genererat med konkatenerade samt kodade header och payload tillsammans med en hemlig nyckel.

Detta resulterar i 3 Base64Url-kodade strängar som separeras med punkter ”.” (se Figur 11). Strängen är så kompakt att den kan skickas via en URL eller HTTP-header.



Figur 11.

3 Metod

3.1 Utvecklingsmiljö

Windows kommer användas som operativsystem med Visual Studio Code(VS Code) som kod-editor. Även några tillägg kommer installeras i VS Code för att underlätta utvecklingen.

C# kommer användas som programmeringsspråk och .NET 6 SDK kommer installeras för att kunna bygga och köra koden.

Övriga ramverk, program, komponenter och så vidare kommer beskrivas där de implementeras i rapporten.

3.2 Web API

Web API:t kommer skapas i ramverket ASP.NET Core. Alla I/O-bundna anrop kommer skrivas på ett asynkront vis.

3.2.1 Struktur och innehåll

Koden kommer struktureras med designmönster som MVC, Dependency Injection och Repository Pattern.

Klass-diagram kommer skapas för att beskriva alla modeller. Det kommer också skapas tabeller för att beskriva adressen till API:t och alla dess slutpunkter.

3.2.2 Modeller, DTOs och Maps

Modeller kommer skapas som representerar alla affärsobjekt. DTOs kommer också skapas som är variationer av modellerna som klienten får tillgång till. Dessa objekt kommer automatiskt mappas(kopiera data mellan varandra) med AutoMapper-komponenten. Organisering och konfigurering av maps kommer implementeras med så kallade profiler. Dessa maps kommer sedan kunna användas i hela projektet där man behöver kopiera data mellan olika objekt-typer.

3.2.3 Controllers

Controllers kommer skapas med metoder, som motsvarar API:ts slutpunkter. De kommer ta emot HTTP-anrop från klienter, tolka och hantera anropen samt skicka tillbaka HTTP-svar.
Controllerns metoder kommer hantera anropen genom att validera modeller, fånga fel och returnera svar. Dessa anrop och svar kommer ibland innehålla data som måste skickas vidare till, eller hämtas från en datakälla.

3.2.4 Repository pattern

Repositories kommer skapas för att skapa en lös koppling mellan controllers och dataåtkomsten.
Interfaces ska skapas som beskriver vad ett repository måste implementera.
Repositories kommer implementera ett interface med alla dess metoder. Här kommer det finnas funktionalitet för dataåtkomst.
Dessa interfaces ska sedan injekteras i controllers med dependency injection.

3.3 Web API med databas

3.3.1 SQL Server och SQL Server Management Studio

SQL Server är valet av databashanterare och SQL Server Management Studio(SSMS) kommer användas för att jobba mot SQL Servern. T.ex. för att skapa användare, anslutningssträngar och få en översikt över databasens struktur.

3.3.2 Databasanslutning med EF Core

EF Core kommer användas som lager för dataåtkomst där ramverket kommer ansluta till en SQL Server databas och utföra operationer mot den. Denna anslutning kommer injekteras i repositories med dependency injection så att controllern kan jobba mot databasen via ett repository.

3.3.3 Code-first och migrationer med EF Core

EF Core med tillvägagångssättet code-first kommer generera databasen med dess tabeller och relationer utifrån modellerna. Migrationer kommer skapas och köras av EF Core som synkar databasen med modellerna.

3.4 Säkerhet, autentisering och auktorisering

Vissa hemliga värden i källkoden kommer lagras som miljövariabler lokalt på maskinen.

API:t kommer implementera säkerhet samt autentisering och auktorisering av användare. Slumpmässigt genererade salt-värden och en hashing-algoritm kommer användas för att lagra lösenord säkert i databasen. Det kommer också implementeras funktionalitet för att förhindra att två användare har samma användarnamn och e-postadress genom att se om det redan existerar i databasen.

JSON Web Tokens, också kallat JWT, kommer användas för att skicka information om autentiserade användare mellan klienten och servern på ett säkert vis för att auktorisera skyddade slutpunkter i API:t samt säkerställa att användare endast modifierar sin egen användarinformation och sina egna inlägg.

4 Konstruktion

4.1 Utvecklingsmiljö

Jag använder följande utvecklingsmiljö i projektet:

- **Operativsystem**: Windows.
- **Kod-editor**: Visual Studio Code med följande tillägg:
  - *ms-dotnettools.csharp*: lättviktiga utvecklingsverktyg för .NET Core, debugging med CoreCLR, intellisense, gå till definition, kontroll av syntax och fler funktioner.
  - *jchannon.csharpextensions*: underlättar och effektiviserar arbetsflödet med snippets för att t.ex. skapa en klass eller en konstruktor.
- **SDK**: *.NET 6.0 SDK* som innehåller komponenter för att kunna bygga en .NET-applikation.
- **Övrigt**: ASP.NET Core, EF Core, SQL Server, SSMS, AutoMapper, JwtBearer och andra komponenter. Jag går in mer specifikt på dessa delar senare i rapporten.

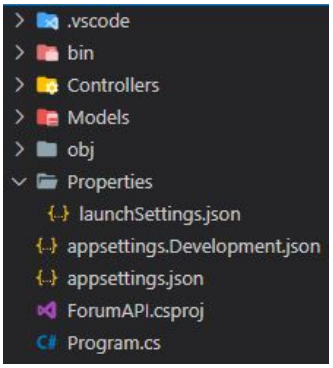
4.2 Web API

Jag har skapat ett nytt Web API-projekt med ASP.NET Core med följande kommando: **dotnet new webapi**.

4.2.1 Struktur och innehåll

4.2.1.1 Struktur / designmönster

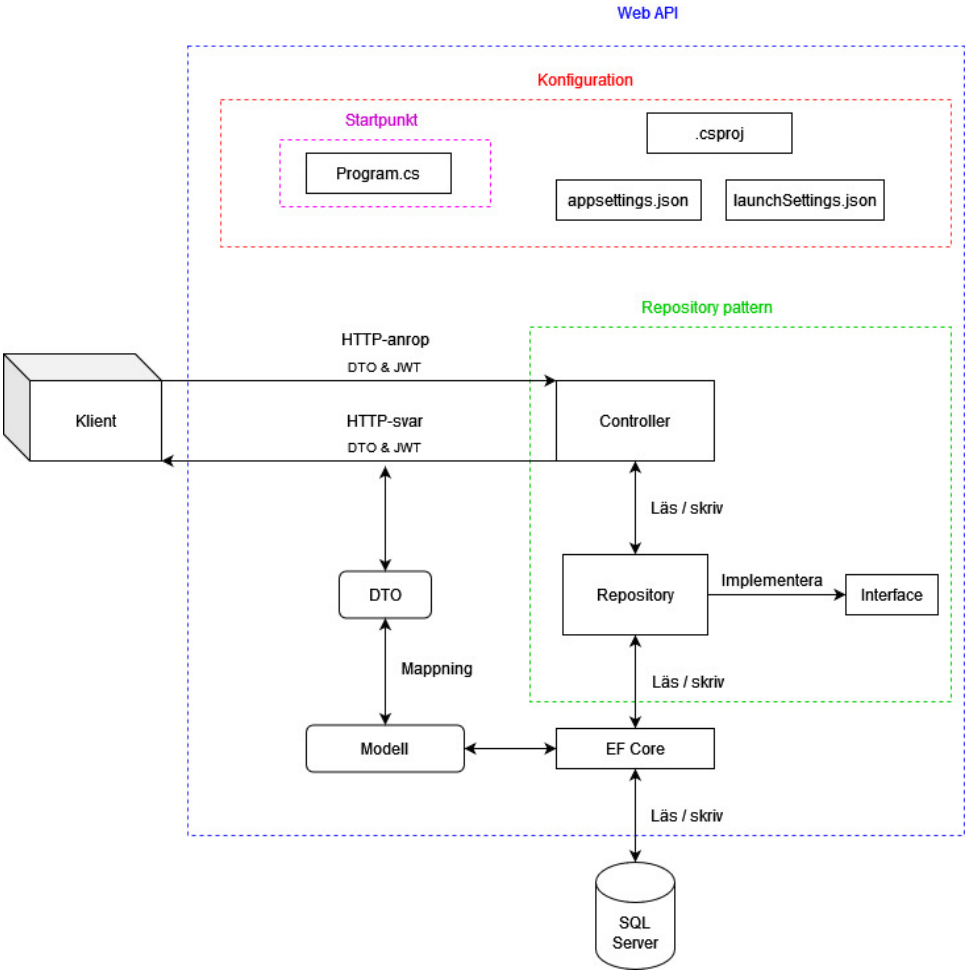
Ovanstående kommando genererar filerna och fil-strukturen som visas i Figur 12.



Figur 12.

- **Bin**- och **object**-mapparna innehåller den kompilerade koden.
- **launchSettings.json** innehåller konfiguration som används av ramverket när applikationen körs lokalt.
- **appSettings.json**-filerna innehåller konfiguration för applikationen där man kan lagra värden som ska kunna nås i hela projektet.
- **ForumAPI.csproj** innehåller konfiguration och information om projektet som dess filer, assemblies, id, version osv.
- **Program.cs** är startpunkten i projektet som exekverar applikationen. Klassen innehåller också konfiguration för att bland annat registrera services för dependency injection och en ”pipeline”, dvs ett mönster med middleware som bestämmer hur HTTP-anropen ska hanteras.

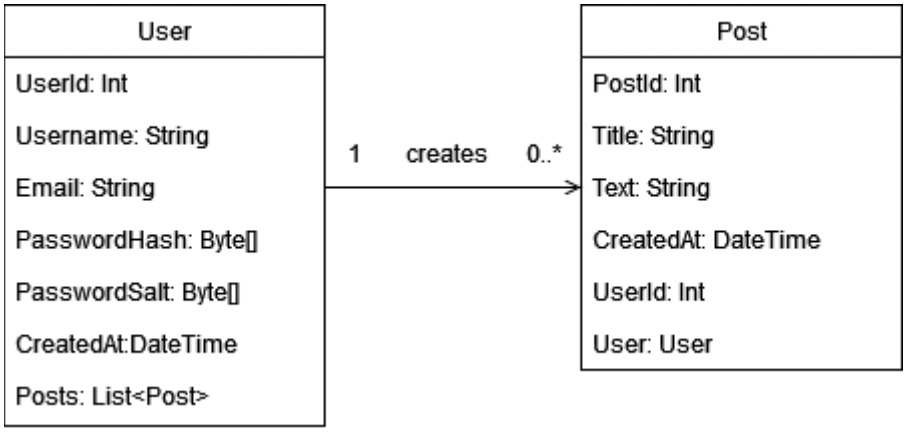
Redan här ser man också MVC-strukturen fast bara med *Models* och *Controllers*. Eftersom det är ett API jag skapar behövs inte *Views*, dvs användargränssnitt. Det är klienternas ansvar. Hela projektets struktur med designmönster beskrivs övergripande i Figur 13 nedan. Jag går in på detaljer senare i rapporten.



Figur 13.

4.2.1.2 Entiteter

I grunden tillhandahåller API:t funktionalitet för användare som skapar inlägg. API:t behöver därför inlägg(Post) och användare(User). De två entiteterna, som sedan ska bli modeller i koden, beskrivs i klassdiagrammet i Figur 14 nedan:



Figur 14.

Entiteterna har en ”en-till-många”-relation. Det vill säga att en användare kan skapa 0 eller fler inlägg och ett inlägg måste skapas av en enstaka användare. Beroende-entiteten Post är beroende av huvud-entiteten User, så om användaren raderas kommer också alla inlägg som användaren skapat att raderas.

Det vanligaste mönstret för att skapa relationer i EF Core(som jag kommer använda senare) är att definiera navigations-egenskaper i båda entiteterna och en egenskap som representerar en främmande-nyckel i beroende-entiteten [35]:

- **User.UserId** är primärnyckeln.
- **Post.UserId** är främmandenyckeln.
  Om beroende-entiteten innehåller en egenskap med ett namn som matchar det följande mönstret: ”<namn på huvud-entitet>Id”, så kommer den egenskapen att bli konfigurerad som en främmande-nyckel.
- **Post.User** är en referens-navigationsegenskap som innehåller en referens till en enstaka entitet.
- **User.Posts** är en samlings-navigationsegenskap som innehåller en referens till flera relaterade entiteter.

4.2.1.3 Slutpunkter

Klienter kan komma åt API:t och alla dess slutpunkter enligt följande struktur: **{protokoll}://{domän}/api/{slutpunkt}/{id}**.

Under lokal utveckling ser det ut såhär:
https://localhost:7177/api/{slutpunkt}/{id}.

T.ex: **https://localhost:7177/api/users/5** för att komma åt en specifik användare med id ”5”.

Funktionalitet för **användare** beskrivs i Tabell 1 nedan:

URI	Verb	Beskrivning	Statuskod - lyckat	Statuskod – ej lyckat
/api/users	GET	Läs alla användare	200 OK	400 Bad Request 404 Not Found
/api/users/{id}	GET	Läs en användare	200 OK	400 Bad Request 404 Not Found
/api/users/ register	POST	Registrera en användare	201 Created	400 Bad Request
/api/users/login	POST	Logga in en användare	201 Created	400 Bad Request 401 Unauthorized 404 Not Found
/api/users/{id}	PUT	Uppdatera en användare	204 No Content	400 Bad Request 401 Unauthorized 404 Not Found
/api/users/{id}	DELETE	Radera en användare och alla dess inlägg	204 No Content	-ll-

Tabell 1.

Funktionalitet för **inlägg** beskrivs i Tabell 2 nedan:

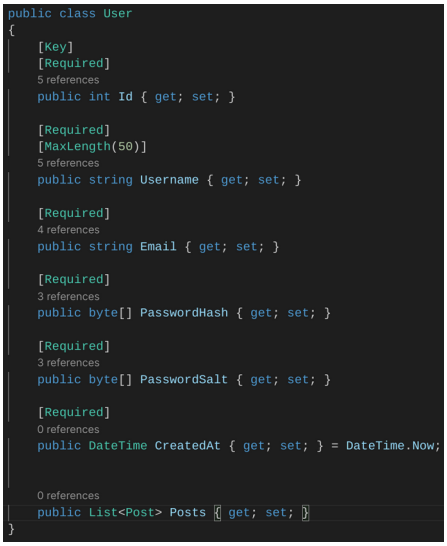
URI	Verb	Beskrivning	Statuskod - lyckat	Statuskod – ej lyckat
/api/posts	GET	Läs alla inlägg	200 OK	400 Bad Request 404 Not Found
/api/posts/{id}	GET	Läs ett inlägg	200 OK	400 Bad Request 404 Not Found
/api/posts	POST	Skapa ett inlägg	201 Created	400 Bad Request 401 Unauthorized
/api/posts/{id}	PUT	Uppdatera ett inlägg	204 No Content	400 Bad Request 401 Unauthorized 404 Not Found
/api/posts/{id}	DELETE	Radera ett inlägg	204 No Content	-ll-

Tabell 2.

4.2.2 Modeller, DTOs och maps

4.2.2.1 Modeller

Entiteterna som beskrevs ovan i sektion 4.2.1.2, kodas som klasser, även kallade modeller. Se exempel med *User*-modellen i Figur 15 nedan.

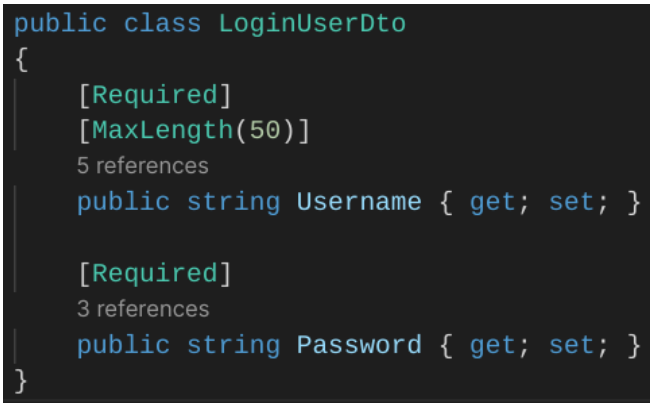


Figur 15.

Här används data-annotationer som bestämmer hur data ska lagras. Attributet ”*Key*” specificerar att egenskapen är en primärnyckel, ”*Required*” säger att egenskapen måste ha ett värde, dvs inte *null*. ”*MaxLength*” säger att egenskapen får ha ett värde med ett begränsat antal tecken [36].

4.2.2.2 DTOs

Jag har skapat DTOs för att dölja vissa information som skickas till klienten. Både för att inte exponera hela modellerna med känslig information och för att användaren inte ska behöva mata in eller ta emot irrelevant information. T.ex. en DTO som *LoginUserDto* (se Figur 16) som används för att logga in. Här behöver användaren bara mata in ett användarnamn och ett lösenord, allt annat är irrelevant. Därför skickar jag bara med de två egenskaperna till klienten.



Figur 16.

Ett annat exempel är *GetUserDto* (se Figur 17) som används för att läsa en användare, som alla kan komma åt. Här exkluderar jag bland annat lösenord och salt-värde som är en säkerhetsrisk om fel personer får tag i de.

```
public class GetUserDto
{
    1 reference
    public int Id { get; set; }

    0 references
    public string Username { get; set; }

    0 references
    public DateTime CreatedAt { get; set; }
}
```

Figur 17.

4.2.2.3 Maps

För att kunna kopiera data eller ”mappa” mellan modeller och DTOs använder jag NuGet-paketet *AutoMapper* som kan installeras med följande kommando: **dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection.**

Jag har skapat så kallade profiler för att organisera mina maps. Det har jag gjort genom att skapa en klass som ärver från *Profile*-klassen från *AutoMapper* där konfigurationen placeras i konstruktorn (se Figur 18). I konfigurationen använder jag *CreateMap<TSource, TDestination>()*-metoden där jag anger vilka objekt som ska kunna kopiera data mellan varandra och i vilken riktning [37].

```
public class PostProfile : Profile
{
    0 references
    public PostProfile()
    {
        CreateMap<Post, GetPostDto>();
        CreateMap<Post, UpdatePostDto>();
        CreateMap<CreatePostDto, Post>();
        CreateMap<UpdatePostDto, Post>();
    }
}
```

Figur 18.

Sedan har jag registrerat *AutoMapper* som en service i *Program.cs* (se Figur 19) för att kunna använda komponenten i applikationen med dependency injection.

```
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
```

Figur 19.

4.2.3 Controllers

Jag har skapat controllers för inlägg och användare med några metoder som motsvarar slutpunkter (se Tabell 1 och Tabell 2 i sektion 4.2.1.3 för struktur på slutpunkter).

I Figur 20 nedan visas grunden för att skapa en controller.

```
[ApiController]
[Route("api/posts")]
0 references
public class PostsController : ControllerBase
```

Figur 20.

Controllern ärver från *ControllerBase*-klassen som är en basklass för MVC-controllers utan stöd för *Views*. Jag anger också 2 attribut:

- **ApiController**: indikerar att controllern används för att svara på HTTP-anrop. Detta attribut är konfigurerat med funktioner och beteende som underlättar utvecklingen av ett API.

- **Route**: anger adressen till controllern som motsvarar en slutpunkt i API:t. Här anger jag namnet manuellt istället för att automatiskt använda filnamnet på controllern, som man också kan göra. Det är bra eftersom adressen alltid blir samma, även när man döper om controllern. Då behöver inte klienter uppdatera sina applikationer med en ny adress bara för att en utvecklare har bytt namn på controllern i koden.

I controllern skapar jag metoderna, också kallat actions, som innehåller slutpunktens funktionalitet (se exempel i Figur 21).

```
[HttpGet]
0 references
public async Task<ActionResult<IEnumerable<GetPostDto>>> GetAllPosts()
{
    return Ok();
}
```

Figur 21.

1. Först anger jag attributet **HttpGet** som specificerar att metoden stödjer HTTP GET-metoden som används för att hämta/läsa data.

2. Sen skapar jag en asynkron metod som returnerar ett resultat:

    - **async**: anger att metoden är asynkron.

    - **Task**: representerar den asynkrona metoden som kan returnera ett värde.

- **ActionResult**: representerar resultatet av metoden inom de vinklade paranteserna som är en lista med *GetPostDto*-objekt.

3. Till sist returnerar jag bara en statuskod utan något innehåll. Jag kommer fylla på controllern med funktionalitet senare i rapporten när jag kopplar till en databas.

4.2.4 Repository pattern

Nästa steg är att skapa ett lager mellan controllern och dataåtkomsten. För att uppnå det har jag använt mig av designmönstret Repository pattern. Mönstret delas in i interfaces, repositories och injektering i controllers.

4.2.4.1 Interfaces

Jag har skapat interfaces som beskriver vad ett repository måste implementera (se Figur 22).

```
public interface IPostRepository
{
    1 reference
    Task<IEnumerable<Post>> GetAllPostsAsync();
    3 references
    Task<Post> GetPostByIdAsync(int id);
    1 reference
    Task CreatePostAsync(Post post);
    1 reference
    void DeletePost(Post post);

    3 references
    Task<bool> SaveChangesAsync();
}
```

Figur 22.

4.2.4.2 Repositories

Sedan har jag skapat repositories som implementerar ett interface med alla dess metoder (se Figur 23).

```
public class PostRepository : IPostRepository
{
  0 references
  public Task<IEnumerable<Post>> GetAllPostsAsync()
  {
    throw new NotImplementedException();
  }

  0 references
  public Task<Post> GetPostByIdAsync(int id)
  {
    throw new NotImplementedException();
  }
```

Figur 23.

Just nu innehåller de ingen funktionalitet men senare i projektet när jag ansluter till en databas så kommer dessa repositories ha funktionalitet för att ta emot data och skicka vidare till antingen controllern eller databasen via ramverket som används för dataåtkomst.

4.2.4.3 Injektering i controllers

För att kunna injektera ett interface med en instans av ett repository i mina controllers så måste jag först registrera detta i *Program.cs* (se Figur 24).

```
builder.Services.AddScoped<IPostRepository, PostRepository>();
builder.Services.AddScoped<IUserRepository, UserRepository>();
```

Figur 24.

Varje gång en controller vill använda t.ex. *IPostRepository*-interfacet så kommer då *PostRepository*-klassen att implementeras.

Sen gör jag själva injekteringen i mina controllers (se Figur 25).

```
private readonly IPostRepository _postRepository;

0 references
public PostsController(IPostRepository postRepository)
{
    _postRepository = postRepository;
}
```

Figur 25.

Interfacet injekteras i konstruktorn och då skapar ramverket en instans av interfacets implementations-klass som sedan kan användas i controllern.

4.3 Web API med Databas

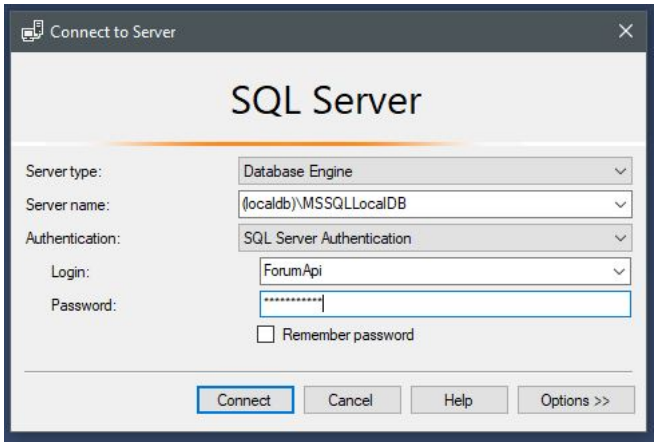
4.3.1 SQL Server och SQL Server Management Studio

Jag använder de två följande programmen för att skapa och jobba med en databas:

1. **Databashanterare**: SQL Server 2019.

2. **Databasverktyg:** SQL Server Management Studio 18.10 (SSMS).

I SSMS skapar jag en lokal databasanslutning och en användare som kan komma åt den anslutningen (se Figur 26).



Figur 26.

Jag lägger sedan in anslutningssträngen i *appsettings.json* (se Figur 27). Den kommer jag kunna nå i projektet när jag behöver den. Detta är känslig information och ska egentligen inte exponeras i kodbasen på detta vis, men det går jag igenom senare i rapporten under säkerhet.

```
"ConnectionStrings": {
  "DefaultConnection": "Data Source=(localdb)\\MSSQLLocalDB; Initial Catalog={namn_på_databas}; User ID={ditt_användarnamn}; Password={ditt_lösenord};"
}
```

Figur 27.

4.3.2 Databasanslutning med EF Core

Jag använder EF Core som ORM-ramverk. Först installerar jag nödvändiga NuGet-paket:

- **dotnet add package Microsoft.EntityFrameworkCore.Design**: verktyg som gör det möjligt att utföra EF Core-relaterade uppgifter i projektet.
- **dotnet add package Microsoft.EntityFrameworkCore.SqlServer**: gör det möjligt att använda EF Core med SQL Server (även Azure SQL Server).

Sen skapar jag en *DbContext*-klass som är en viktig del av EF Core (Se Figur 28). En instans av *DbContext* representerar en session med databasen som kan användas för utföra viktiga uppgifter som att hantera databasanslutningen, konfigurera modeller och relationer, spara data och ställa frågor till databasen [38].

```
public class DataContext : DbContext
{
    0 references
    public DataContext(DbContextOptions<DataContext> options) : base(options)
    {

    }

    0 references
    public DbSet<Post> Posts { get; set; }
    0 references
    public DbSet<User> Users { get; set; }
}
```

Figur 28.

1. Här skapar jag en klass som ärver av *DbContext*.

2. För att *DbContext*-klassen ska fungera behöver den en instans av *DbContextOptions*-klassen som hanterar information om konfiguration som t.ex. vilken anslutningssträng som ska användas, vilken databashanterare osv.
   Ett sätt att skicka med en instans av denna klassen är att skapa en konstruktor i klassen. Jag skickar då med objektet som en parameter i konstruktorn där jag också specificerar att objektet ska användas av *DataContext*-klassen.
   Nyckelordet *base* säger att jag vill skicka med *options*-parametern till basklassen som är *DbContext* i detta fall.

3. *DbSet*-klassen mappar till databasens tabeller med olika modeller och gör det möjligt att utföra CRUD-operationer mot databasen. Här anger jag samma modeller som jag har migrerat till databasen och ett namn på tabellen som jag kan använda för att komma åt den [39].

Till sist registrerar jag *DataContext*-klassen i *Program.cs* för att kunna använda den i resten av applikationen med dependency injection (se Figur 29).

```
builder.Services.AddDbContext<DataContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
```

Figur 29.

1. *AddDbContext* registrerar *DataContext*-klassen som en service som jag kan injektera i mina controllers.

2. *UseSqlServer()* specificerar att *DataContext* ska ansluta till en SQL Server-databas med hjälp av en anslutningssträng som skickas med som en parameter. Anslutningssträngen är den jag lade till i *appsettings.json* tidigare.

4.3.3 Code-first och migrationer med EF Core

Med den relativt simpla konfigurationen har jag skapat en databasanslutning som jag kan använda för att börja migrera koden för att skapa databasen. För att skapa

en migration skriver jag följande kommando:
dotnet ef migrations add InitialMigration

Detta genererar en ny mapp som heter ”Migrations” där man kan se tabeller och relationer som EF Core använder för att bygga databasen. Här dubbelkollar jag att allt ser ut som det ska med egenskaperna, primär- och främmande-nycklar, att id är unika och auto-inkrementeras, att en användares inlägg raderas om dess konto raderas osv.

När allt ser bra ut kan jag köra migrationen med följande kommando som skapar databasen utifrån mina migrationer:
dotnet ef database update

4.3.4 Uppdaterade repositories med funktionalitet

När databasanslutningen och databasen är på plats går jag tillbaka till mina repositories och uppdaterar de med funktionalitet.

Det första jag gör här är att injektera en instans av *DataContext* i konstruktorn (se Figur 30). Det är då som sagt objektet som är ansvarig för att jobba mot databasen, mitt context.

```
private readonly DataContext _context;

0 references
public PostRepository(DataContext context)
{
  _context = context;
}
```

Figur 30.

Objektet kan jag sedan använda där jag vill kommunicera med databasen, t.ex. när jag vill hämta en lista med alla inlägg (se Figur 31).

```
public async Task<IEnumerable<Post>> GetAllPostsAsync()
{
  var posts = await _context.Posts.ToListAsync();
  return posts;
}
```

Figur 31.

Här använder jag mitt context och säger att jag vill komma åt *Posts*-tabellen i databasen. Sedan använder jag *ToListAsync()*-metoden från mitt context som returnerar alla inlägg från *Posts*-tabellen som en lista. Till sist returnerar jag resultatet, som är en lista med inlägg, dvs *Post*-objekt.

4.3.5 Uppdaterade controllers med funktionalitet

Jag uppdaterar mina controllers så att de nu kan ta emot data och skapa/modifiera data i databasen med hjälp av injekterade repositories. Jag har även lagt till felhantering och annan affärslogik.

Se exempel i Figur 32 nedan där API:t hämtar en användare från databasen och returnerar den till klienten.

```
// GET api/users/{id}
[AllowAnonymous]
[HttpGet("{id}", Name="GetUserByIdAsync")]
1 reference
public async Task<ActionResult<GetUserDto>> GetUserByIdAsync(int id)
{
    var user = await _userRepository.GetUserByIdAsync(id);

    if(user == null)
    {
        return NotFound();
    }

    var res = _mapper.Map<GetUserDto>(user);

    return Ok(res);
}
```

Figur 32.

1. Först tar jag emot ett id från anropet som klienten specificerade.

2. Sedan anropar jag repositoriet och dess *GetUserByIdAsync()*-metod som hämtar en användare från databasen med id:t som klienten skickade med i anropet.
   Nyckelordet await säger att jag måste vänta på ett resultat från metoden innan jag kan gå vidare till nästa rad kod. Jag vill självklart inte göra en felhantering av användaren innan jag fått tillbaka ett resultat.

3. Sen kollar jag om jag fick tillbaka ett värde från databasen, dvs om en användare med det inmatade id:t existerar. Om den inte existerar så returnerar repositoriet värdet *null* och då returnerar jag *NotFound()* som producerar en *404*-statuskod, som betyder att servern inte kunde hitta resursen som efterfrågades.

4. Annars om databasen hittar en användare med det specifika id:t så kopierar jag data från *User*-objektet från databasen till ett *GetUserDto*-objekt som bara innehåller relevant information för klienten. Detta görs då automatiskt med en injekterad instans av *AutoMapper,* med en redan färdig profil, där *User*-objektet från databasen är ursprungsobjektet och *GetUserDto*-objektet är destinationsobjektet.

5. Till sist returnerar jag *Ok()* som producerar en *200*-statuskod, som betyder att anropet lyckades. Jag skickar även med *GetUserDto*-objektet i bodyn i HTTP-svaret som klienten kan läsa.

(Se Bilaga 1 för flödesschema).

4.4 Säkerhet, autentisering och auktorisering

För tillfället är alla slutpunkter med dess funktionalitet tillgängliga för alla klienter. Även viss känslig information lagras som vanlig text i kodbasen. För att lösa detta måste jag dölja den känsliga informationen och skydda mina slutpunkter med auktorisering.

4.4.1 Miljövariabler

Applikationen innehåller information som kan vara osäker i fel händer. T.ex. anslutningssträng till databasen, lösenord och andra hemliga värden. Denna information ska inte lagras synligt i källkoden.
Som lösning på detta problem använder jag verktyget *Secret Manager*. Detta verktyg lagrar dessa värden lokalt på maskinen i en dold JSON-fil som är separat från källkoden. Applikationen kan då nå ett värde och använda det, men det syns inte i källkoden [40].

Denna fil skapas med följande kommando: **dotnet user-secrets init**.
Då läggs det också till ett id för denna fil (*UserSecretsId)* i *.csproj*-filen som används för att identifiera filen och komma åt dess information.

På Windows kan denna fil nås genom sökvägen: ***C:\Users\{din_användare}\AppData\Roaming\Microsoft\UserSecrets\{UserSecretsId}***.
Man kan också nå värdena i filen med kommandot: **dotnet user-secrets list**.

I filen lagrar jag en anslutningssträng till databasen och en hemlig nyckel som jag använder senare för att signera och validera JWT-tokens (se Figur 33).

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=(localdb)\\MSSQLLocalDB; Initial Catalog={namn_på_databas}; User ID={användarnamn}; Password={lösenord};"
  },
  "TokenSettings": {
    "SigningKey": "{säker_token_minst_16_tecken}"
  }
}
```

Figur 33.

4.4.2 Säkra lösenord: salta, hasha och verifiera

Jag har skapat en klass som saltar, hashar och verifierar lösenord.

4.4.2.1 Salt

Den första metoden i klassen genererar en slumpmässig byte-array som representerar ett salt-värde (se Figur 34).

```
public static byte[] GenerateRandomSalt()
{
    byte[] salt = new byte[32];
    RandomNumberGenerator rng = RandomNumberGenerator.Create();
    rng.GetBytes(salt);

    return salt;
}
```

Figur 34.

4.4.2.2 Hasha lösenord

Den andra metoden genererar ett hash-värde (se Figur 35).

```
public static byte[] SaltAndHashPassword(string password, byte[] salt)
{
    SHA512 sha = SHA512.Create();

    string saltedPassword = password + salt;

    return sha.ComputeHash(Encoding.Unicode.GetBytes(saltedPassword));
}
```

Figur 35.

1. Metoden tar det inmatade lösenordet och ett salt-värde som input.

2. Sedan konkateneras input-värdena för att skapa ett nytt värde som är mer unikt än det ursprungliga inmatade lösenordet.

3. Det nya konkatenerade värdet skickas sedan in i en *SHA512*-hashing-algorithm som genererar ett hash-värde. Det är detta hash-värde som sedan ska lagras i databasen.

4.4.2.3 Verifiera lösenord

Till sist skapar jag en metod i klassen som verifierar ett lösenord (se Figur 36).

```
public static bool VerifyPasswordHash(string password, byte[] passwordHash, byte[] passwordSalt)
{
    var computedHash = SaltAndHashPassword(password, passwordSalt);

    for(int i = 0; i < computedHash.Length; i++)
    {
        if(computedHash[i] != passwordHash[i])
        {
            return false;
        }
    }
    return true;
}
```

Figur 36.

1. Metoden tar emot ett inmatad lösenord (från klienten), ett hash-värde (från databasen) och ett salt-värde (från databasen).

2. Det inmatade lösenordet och salt-värdet skickas sedan med som parameter i metoden i Figur 35 ovan som genererar ett hash-värde.

3. Det nya hash-värdet jämförs sedan med hash-värdet som skickades med i steg 1. De jämförs byte för byte i en for-loop där jag kollar om värdet vid varje index i byte-arrayerna matchar.

4. Om de inte matchar så returnerar jag *false*. Annars om de matchar så returnerar jag *true*, dvs att valideringen lyckades.

Dessa 3 metoder använder jag sedan i mina controllers när jag registrerar och loggar in användare.

4.4.3 Bearer-autentisering med JSON Web Tokens

För att möjliggöra bearer-autentisering med JWT i applikationen har jag gjort en del konfigurering:

- **Installera JWT Bearer namespace med följande kommando: dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer**. Detta namespace innehåller allt som behövs för att skapa bearer-autentisering med JWT.

- **Registrera autentiserings-middleware i *Program.cs (se Figur 37)*:**

  ```
  app.UseAuthentication();
  ```

  Figur 37.

  I ASP.NET Core hanteras autentisering av *IAuthenticationService*, som används av *authentication middleware* (se Figur 8). Denna service använder registrerade autentiserings-hanterare för att utföra autentiserings-relaterade metoder som att autentisera en användare eller meddela när en obehörig användare försöker komma åt en skyddad resurs. De registrerade autentisering-hanterarna och dess konfiguration kallas för scheman [41].

- **Registrera scheman i *Program.cs (se Figur 38)***:

```
var signingKey = System.Text.Encoding.UTF8.GetBytes(builder.Configuration.GetSection("TokenSettings:SigningKey").Value);
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(options => {
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(signingKey),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

Figur 38.

För att registrera ett schema anropar jag *AddAuthentication()* från *authentication service* och lägger till den schema-specifika förlängnings-metoden *AddJwtBearer()* som specificerar att jag vill använda JWT bearer-autentisering. *AddAuthentication()* tar också emot en parameter som är ett standard schema som kommer användas om inget annat anges. I detta fall ett standard JWT bearer-schema [41].

I *AddJwtBearer()* lägger jag till konfiguration för hur en JWT ska valideras [42]:

  - *ValidateIssuerSigningKey* = *true*: säger att API:t måste validera den privata säkerhetsnyckeln som signerade anropets JWT.

  - *IssuerSigningKey*: är den privata säkerhetsnyckeln som används för att validera signaturen i anropets JWT. Denna nyckel finns i min *secrets.json*-fil som jag skapade tidigare.

  - *ValidateIssuer* = *false:* säger att jag inte vill validera adressen till servern som genererade anropets JWT.

  - *ValidateAudience* = *false*: säger att jag inte vill validera adressen som skickar anropet.

- **Registrera *HttpContextAccessor* i *Program.cs (se Figur 50)*:**

```
builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
```

Figur 50.

ASP.NET Core når *HttpContext*-objektet genom *IHttpContextAccessor*-interfacet och dess implementations-klass HttpContextAccessor. Varje gång man skapar ett HTTP-anrop eller HTTP-svar så skapas ett *HttpContext*-objekt. Objektet innehåller information om det nuvarande anropet eller svaret. [43]. Detta objektet kommer användas för att läsa claims i en *JWT* som skickas med i ett anrop senare i rapporten. När vi registrerar det som en service så kan vi nå objektet överallt i projektet med depenedency injection.

Jag har skapat en klass med två metoder, en för att skapa en JWT (se Figur 39) och en för att läsa information/claims från en JWT (se Figur 40).

4.4.3.1 Skapa en JWT

```
public string CreateToken(User user)
{

    var claims = new List<Claim>(){
        new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
        new Claim(ClaimTypes.Name, user.Username),
        new Claim(ClaimTypes.Email, user.Email)
    };

    var key = new SymmetricSecurityKey(System.Text.Encoding.UTF8
        .GetBytes(_configuration.GetSection("TokenSettings:SigningKey").Value));
    var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);

    var tokenDescriptor = new SecurityTokenDescriptor
    {
        Subject = new ClaimsIdentity(claims),
        Expires = System.DateTime.Now.AddDays(1),
        SigningCredentials = credentials
    };
    var tokenHandler = new JwtSecurityTokenHandler();
    var token = tokenHandler.CreateToken(tokenDescriptor);

    return tokenHandler.WriteToken(token);
}
```

Figur 39.

1. Först tar jag emot ett *User*-objekt.

2. Sedan skapar jag claims som används för att lagra information om användaren i en token. Informationen hämtas från *User*-objektet.

3. Jag skapar en symmetrisk säkerhetsnyckel som kommer användas för att signera en token. Här använder jag den dolda nyckeln från *secrets.json* som hämtas från konfigurations-klassen med dependency injection. *SymmetricSecurityKey()* skapar den symmetriska säkerhetsnyckeln. *SigningCredentials()* representerar säkerhetsnyckeln och den algoritm som används för att signera denna token.

4. *SecurityTokenDescriptor()* innehåller all information som behövs för att skapa en JWT med claims, slutdatum och signering.

5. *JwtSecurityTokenHandler()* skapar en JWT med ovanstående information med hjälp av *CreateToken()*-metoden.

6. Till sist serialiseras denna token till ett kompakt format med *WriteToken()*-metoden.

4.4.3.2 Läs claims från JWT

För att läsa information/claims i en JWT använder jag mig av *HttpContext*-objektet (se Figur 40).

```
public int GetUserId() {
    try {
        checked {
            var userId = int.Parse(_httpContextAccessor.HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier));
            return userId;
        }
    }
    catch
    {
        return -1;
    }
}
```

Figur 40.

Här hämtar jag ett id-värde som ligger i ett *NameIdentifier*-claim i *HttpContext*-objektet och returnerar det. Om objektet inte innehåller några claims av typen *NameIdentifier* så returnerar jag värdet ”-1” som en användare inte kan ha.

4.4.4 Registrera användare

Funktionalitet för registrering av användare implementeras i controllern (se Figur 41).

```
// POST api/users/register
[AllowAnonymous]
[HttpPost("register")]
0 references
public async Task<ActionResult<RegisterUserDto>> RegisterUserAsync(RegisterUserDto req)
{
    // Check if username and email already exists
    var usernameExists = await _userRepository.UsernameExistsAsync(req.Username);
    var emailExists = await _userRepository.EmailExistsAsync(req.Email);

    // If they already exist
    if(usernameExists || emailExists)
    {
        // 409 conflict
        return Conflict();
    }

    // Map req-object to a full user-object
    var newUser = _mapper.Map<User>(req);

    // Generate salt and hashed password and add to newUser-object
    byte[] salt = SecurePassword.GenerateRandomSalt();
    newUser.PasswordSalt = salt;
    newUser.PasswordHash = SecurePassword.SaltAndHashPassword(req.Password, salt);

    // Register and save to database
    await _userRepository.RegisterUserAsync(newUser);
    await _userRepository.SaveChangesAsync();

    // Map object to return as response
    var res = _mapper.Map<GetUserDto>(newUser);

    // 201 created
    return CreatedAtRoute(nameof(GetUserByIdAsync), new {Id = res.Id}, res);
}
```

Figur 41.

1. Först kollar jag om användarnamn och e-postadress redan finns i databasen med metoder från *UserRepository*. Om de gör det så returnerar

jag statuskoden *409* som betyder att det inmatade värdet skapar en konflikt i databasen som inte är tillåten enligt uppsatta regler.

2. Om varken användarnamn eller e-postadress existerar så mappar jag *RegisterUserDto*-objektet till ett fullt *User*-objekt. Jag genererar ett salt-värde och ett hash-värde med hjälp av metoderna jag skapade i sektion 4.4.2, som sedan läggs till som värde i objektets motsvarande egenskaper.

3. Jag skickar sedan med *User*-objektet i *RegisterUserAsync()*-metoden från repositoriet som ger objektet en ”skapa”-status som EF Core, eller mer specifikt en instans av *DbContext*-klassen(ett context) spårar.

4. Sedan anropar jag *SaveChanges()*-metoden som ligger i repositoriet med hjälp av *SaveChangesAsync()*.
   När ett objekt modifieras, skapas eller raderas så får den en status som spåras av instansen av *DbContext*-klassen *i repositoriet.* När *SaveChanges()*-metoden i mitt context anropas så kommer databasen att uppdateras med ändringarna som har spårats. Om denna metod ej anropas så kommer inga ändringar att göras i databasen [44].

5. Sedan kopierar jag data från det nya *User*-objektet som skapades till ett *GetUserDto*-objekt som är säkert att skicka tillbaka till klienten.

6. Till sist returnerar jag statuskoden *201* som betyder att något skapades och specificerar vad som har skapats samt via vilken slutpunkt användaren kan nå resursen.

4.4.5 Autentisera användare (logga in)

Även inloggning av användare sker i controllern (se Figur 42).

```
// POST api/users/login
[AllowAnonymous]
[HttpPost("login")]
0 references
public async Task<ActionResult<LoginUserDto>> LoginUserAsync(LoginUserDto req)
{
    // Check if user exists in database
    var user = await _userRepository.LoginUserAsync(req.Username);

    // Wrong username
    if(user == null)
    {
        // 401 unauthorized
        return Unauthorized();
    }

    var verifiedPassword = SecurePassword.VerifyPasswordHash(req.Password, user.PasswordHash, user.PasswordSalt);

    // Wrong password
    if(!verifiedPassword)
    {
        // 401 unauthorized
        return Unauthorized();
    }

    // Create token
    var token = _token.CreateToken(user);

    return Ok(token); // return token
}
```

Figur 42.

1. Metoden tar emot ett *LoginUserDto*-objekt som innehåller användarnamn och lösenord.

2. Jag hämtar sedan en användare från databasen som matchar det inmatade användarnamnet. Om användaren inte finns i databasen så får jag tillbaka värdet *null* från repositoriet, och då returnerar jag statuskoden *401* som betyder att användaren inte är auktoriserad.

3. Annars om en användare hittades så får jag tillbaka ett *User*-objekt. Då verifierar jag lösenordet med *VerifyPasswordHash()*-metoden från sektion 4.4.2.3. Här skickar jag med det inmatade lösenordet från anropet och hash-värdet samt salt-värdet från *User*-objektet som kom från databasen. Om lösenorden inte matchar returnerar jag återigen statuskoden *401*.

4. Annars om användarnamnet existerar i databasen och lösenordet är korrekt så skickar jag med *User*-objektet till *CreateToken()*-metoden som jag skapade i sektion 4.4.3.1.

5. Till sist skickar jag tillbaka den token som genererades, till klienten, tillsammans med statuskoden *200* som betyder att anropet lyckades.

(Se Bilaga 2 för flödesschema).

4.4.6 Auktorisera användare

När en användare har loggat in får den som sagt en token av API:t. Genom att ha denna token eller ”bära den” så får användaren tillgång till skyddade slutpunkter i API:t så länge som denna token inte modifieras eller slutar att gälla. Denna token skickas med i Authorize-headern i varje HTTP-anrop som kan läsas och verifieras av API:t.

Jag har lagt till *Authorize*-attributet i mina controllers som specificerar att controllern kräver autentisering Alla slutpunkter i controllern kommer då, som standard, kräva att besökare är autentiserade för åtkomst till funktionaliteten. För de slutpunkterna som inte kräver autentisering lägger jag till attributet *AllowAnonymous*. Detta måste anges där alla typer av besökare ska få tillgång, för att motverka *Authorize*-attributet jag lade till för hela controllern.

För att auktorisera användare som ska modifiera ett inlägg eller en användare så jämför jag id:t från en JWT med id:t i databasen (se Figur 43). Id:t från en JWT hämtas med metoden i sektion 4.4.3.2. På detta vis försäkrar jag mig om att användaren bara modifierar saker som de skapat själv.

```
var currentUserId = _token.GetUserId();
if(user.Id != currentUserId)
{
    return Unauthorized();
}
```

Figur 43.

5 Resultat

Ett Web API har skapats som jobbar med persistent data i en databas.

En utvecklingsmiljö har installerats och konfigurerats för att koda API:t.

API:t hanterar inlägg och användare som har beskrivits som entiteter med egenskaper och relationer. API:t har en sökväg och olika slutpunkter som kan nås med HTTP-anrop för att komma åt funktionalitet. API:t skickar tillbaka HTTP-svar med en statuskod och data, som skiljer sig åt beroende på hur anropet behandlades av API:t. Följande funktionalitet har implementeras:

- **Inlägg**:
    - Ej autentiserad: läsa alla och läsa ett enda.
    - Autentiserad: skapa, redigera och radera egna inlägg.
- **Användare**:
    - Ej autentiserad:
        - Läsa alla användare och läsa en användare.
        - Registrera med unikt användarnamn och unik email samt ett lösenord som krypteras.
        - Logga in.
    - Autentiserad:
        - Läsa all information om egen användare.
        - Redigera egen användare(API:t kollar om användarnamn och email är unika).
        - Radera egen användare (och samtidigt radera användarens inlägg).

En databas har skapats och anslutits till API:t så att det kan jobba med persistent data.

Användare har lagrats på ett säkert vis med krypterade lösenord och användare kan autentiseras samt auktoriseras för att få tillgång till skyddade slutpunkter och funktionalitet.

6 Slutsatser

I det stora hela har arbetet gått bra, jag har inte stött på några större problem under själva utvecklingen. Jag gjorde dock misstaget att lägga till miljövariabler för sent i projektet så att hemlig information låg synligt i källkoden på Github.

API:t kan förbättras med bland annat mer felhantering, validering av objekt, användarroller, versioner, beskrivning av HTTP-svar osv.
Det skulle också kunna publiceras till en molnbaserad plattform som Azure, så att alla kan använda och testa det.

Utöver själva kodandet var det en hel del nya koncept och designmönster som tog ett tag att sätta sig in i och förstå. Jag läste om många olika designmönster och ”best practices”, utöver de i rapporten, som jag tyvärr inte hann implementera. Det skulle jag vilja fördjupa mig ännu mer inom för att skapa ett bättre API med mer strukturerad och återanvändbar kod.

Källförteckning

[1] Mark J. Price. C# 8.0 and .NET Core 3.0, 4:e upplaga. Packt; Oktober 2019.

[2] Dot Net Tutorials, ”ASP.NET Core Web API Tutorials” https://dotnettutorials.net/course/asp-net-core-web-api-tutorials/ Hämtad 2021-12-13.

[3] Microsoft, ”ASP.NET”, https://dotnet.microsoft.com/en-us/apps/aspnet Hämtad 2021-12-13.

[4] Microsoft, ”ASP.NET Web APIs”, https://dotnet.microsoft.com/en-us/apps/aspnet/apis Hämtad 2021-12-13.

[5] Tutorials Teacher, ”What is Web API?”, https://www.tutorialsteacher.com/webapi/what-is-web-api Hämtad 2021-12-13.

[6] Mozilla, ”Introduction to web APIs”, https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction Hämtad 2021-12-13.

[7] Dot Net Tutorials, ”HTTP (HyperText Transport Protocol)”, https://dotnettutorials.net/lesson/hypertext-transport-protocol/ Hämtad 2021-12-13.

[8] Mozilla, ”An overview of HTTP”, https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview Hämtad 2021-12-13.

[9] Microsoft, ”ASP.NET MVC Pattern”, https://dotnet.microsoft.com/en-us/apps/aspnet/mvc Hämtad 2021-12-13.

[10] Microsoft, ”Create Data Transfer Objects (DTOs)”, https://docs.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5 Publicerad 2020-02-19. Hämtad 2021-12-13.

[11] AutoMapper, ”Getting Started Guide”, https://docs.automapper.org/en/latest/Getting-started.html Hämtad 2021-12-13

[12] Microsoft, ”Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application (9 of 10)”, https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-

mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application Publicerad 2021-03-05. Hämtad 2021-12-13.

[13] Infotec, ”What is Microsoft SQL Server and What is it Used For?”, https://www.infotectraining.com/blog/what-is-microsoft-sql-server-and-what-is-it-used-for/ Publicerad 2017-07. Hämtad 2021-12-14.

[14] Educba, ”What is ORM?”, https://www.educba.com/what-is-orm/ Hämtad 2021-12-14.

[15] Microsoft, ”Entity Framework Core”, https://docs.microsoft.com/en-us/ef/core/ Publicerad 2021-05-25. Hämtad 2021-12-14.

[16] C# corner, ”Understanding Entity Framework Core And Code First Migrations In EF Core”, https://www.c-sharpcorner.com/article/understanding-entity-framework-core-and-code-first-migrations-in-ef-core/ Publicerad 2018-11-17. Hämtad 2021-12-14.

[17] Webbling, ”Introduktion till asynkron programmering”, https://webbling.se/index.php/Introduktion_till_asynkron_programmering Hämtad 2021-12-15.

[18] Medium, ”Asynchronous Programming : Why, When and Why Not!”, https://medium.com/@rajatsikder/asynchronous-programming-use-cases-86727de31992 Publicerad 2018-07-02. Hämtad 2021-12-15.

[19] Microsoft, ”Asynchronous programming with async and await”, https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ Publicerad 2021-10-27. Hämtad 2021-12-15.

[20] Kaspersky, ”Cryptography Definition”, https://www.kaspersky.com/resource-center/definitions/what-is-cryptography Hämtad 2021-12-16.

[21] JScrambler, ”Hashing Algorithms”, https://blog.jscrambler.com/hashing-algorithms/ Publicerad 2020-08-12. Hämtad 2021-12-16.

[22] Ionos, ”What are rainbow tables?”, https://www.ionos.com/digitalguide/server/security/rainbow-tables/ Publicerad 2019-04-10. Hämtad 2021-12-16.

[23] BetterBuys, ”Estimating Password-Cracking Times”, https://www.betterbuys.com/estimating-password-cracking-times/ Hämtad 2021-12-16.

[24] auth0, ”Adding Salt to Hashing: A Better Way to Store Passwords”, https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/ Publicerad 2021-02-25. Hämtad 2021-12-16.

[25] JWT, ”Introduction to JSON Web Tokens”, https://jwt.io/introduction Hämtad 2021-12-16.

[26] Microsoft, ”Dependency Injection in .NET”, https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection. Publicerad 2022-01-05. Hämtad 2022-01-15.

[27] Microsoft, ”Dependency inversion”, https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/architectural-principles#dependency-inversion. Publicerad 2021-12-16. Hämtad 2022-01-15.

[28] Microsoft, ”Dependency injection with .NET Core”, https://docs.microsoft.com/en-us/archive/msdn-magazine/2016/june/essential-net-dependency-injection-with-net-core. Publicerad 2019-01-31. Hämtad 2022-01-15.

[29] Microsoft, ”A tour of the C# language”, https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. Publicerad 2021-11-30. Hämtad 2021-12-15.

[30] DevOps School, ”What is Bearer token and How it works?”, https://www.devopsschool.com/blog/what-is-bearer-token-and-how-it-works/. Publicerad 2021-05-07. Hämtad 2021-12-20.

[31] Swagger, ”Bearer Authentication”, https://swagger.io/docs/specification/authentication/bearer-authentication/. Hämtad 2021-12-20.

[32] Medium, ”Understanding about web Authentication methods”, https://medium.com/@itsmeakhil/understanding-about-web-authentication-methods-ccae46734108. Publicerad 2020-08-09. Hämtad 2021-12-20.

[33] Datatracker, ”JSON Web Token (JWT)”, https://datatracker.ietf.org/doc/html/rfc7519. Hämtad 2021-12-20.

[34] C# corner, ”How To Use JWT Authentication With Web API”, https://www.c-sharpcorner.com/article/how-to-use-jwt-authentication-with-web-api/. Publicerad 2019-02-28. Hämtad 2021-12-20.

[35] Microsoft, ”Relationships”, https://docs.microsoft.com/en-us/ef/core/modeling/relationships?tabs=fluent-api%2Cfluent-api-simple-key%2Csimple-key. Publicerad 2021-12-18. Hämtad 2022-01-05.

[36] Entity Framework Core, ”Data Annotations”, https://entityframeworkcore.com/model-data-annotations. Hämtad 2022-01-05.

[37] AutoMapper, ”Configuration”, https://docs.automapper.org/en/stable/Configuration.html. Hämtad 2022-01-05.

[38] Entity Framework Tutorial, ”Entity Framework Core: DbContext”, https://www.entityframeworktutorial.net/efcore/entity-framework-core-dbcontext.aspx. Hämtad 2022-01-05.

[39] Dot Net Tutorials, ”DbSet in Entity Framework”, https://dotnettutorials.net/lesson/dbset-in-entity-framework/. Hämtad 2022-01-05.

[40] Microsoft, ”Safe storage of app secrets in development in ASP.NET Core”, https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets?view=aspnetcore-6.0&tabs=windows. Publicerad 2021-22-04. Hämtad 2022-01-10.

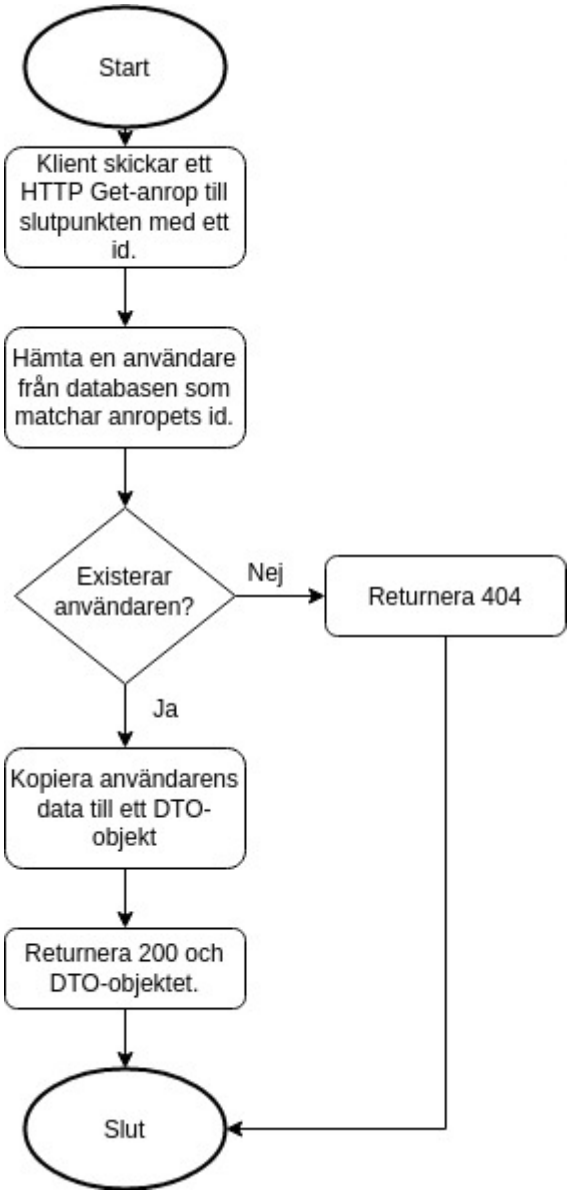
[41] Microsoft, ”Overview of ASP.NET Core authentication”, https://docs.microsoft.com/en-us/aspnet/core/security/authentication/?view=aspnetcore-6.0. Publicerad 2021-05-25. Hämtad 2022-01-05.

[42] Microsoft Devblogs, ”JWT Validation and Authorization in ASP.NET Core”, https://devblogs.microsoft.com/dotnet/jwt-validation-and-authorization-in-asp-net-core/. Publicerad 2017-04-06. Hämtad 2022-01-05.

[43] Microsoft, ”HttpContext Class”, https://docs.microsoft.com/en-us/dotnet/api/system.web.httpcontext?view=netframework-4.8. Hämtad 2022-01-05.

[44] Microsoft, ”Change Tracking in EF Core”, https://docs.microsoft.com/en-us/ef/core/change-tracking/. Publicerad 2021-03-11. Hämtad 2022-01-14.

Bilaga 1: Flödesschema, hämta användare med id



Bilaga 2: Flödesschema, logga in