Reflektioner inlämningsuppgift: Filmstudion , 8 Yhp

REST

Innan jag börjar förklara vill jag bara tala om att i slutet av denna pdf så finns en grundskiss (mindmap) på hela strukturen, som jag tog fram innan hela programmeringsfasen inleddes. Den visar tydligt designmönstret i programkoden.
Mitt mönster är uppbyggt utifrån sex huvudgrupper; Domain, Persistence, Services, Controllers, Transfer och Mapping.
Domain innehåller programmets entiteter, entiteternas interface, en klass i mappen ”Helpers” som tillhandahåller en text som krypteringen baseras på, samt fyra klasser i mappen ”Authorization” som jobbar med autentiseringen av användare. Detta är den inre delen i programstrukturcirklen, domänlogiken.
För att hantera och spara innehåll för entiteterna och få en tydlig struktur finns mappen ”Persistence”. Här finns klassen som jobbar med databasen, ”AppDbContext”, för att spara information. Även i detta fallet, viss ”seed” för att lägga till testinnehåll.
Vi har en mapp med alla ”Repositories” för entiteterna, där det sparade innehållet från databasen hämtas in. Detta görs genom en abstract klass ”BaseRepository” som ärvs av alla andra repositories. På så vis behöver inte ”_context” skapas i alla repositories, utan ärvs (polymorfism). I repositorierna så finns det några grundmetoder, t.ex. att göra olika listor av entiteterna och/inte inkludera fält med andra klasser, fält som inte sparats i databasen. Det kan ha halkat in någon metod som egentligen skulle kunna ligga i service, men av tidsbrist har jag inte kunnat omstrukturera ”refactor” koden som jag skulle önskat. Grundmetoderna som ska användas av servicedelen skapas i repositoriernas interface, som då uppfylls av repositorierna. För att inte repositorierna ska bli för stora och avancerade med många metoder, samt att skilja repositorierna från service till controllers så skapas mellandelen ”Service”. Här finns serviceklasser som liknar repositorier-klasser, men har namn ”Service” istället för ”Repository” och som innehåller fler metoder för controllers. Det gör det tydligt i programkoden, vilken del som tillhandahåller metoder åt controllers. På detta sättet kapslar jag in koden ytterligare, vilket gör att ändring i koden underlättas, då en ändring påverkar en mindre del i kodstrukturen (encapsulation).
För att uppfylla åtkomstpunkterna i kraven så har jag skapat ”Controllers” med namn som dels motsvarar den entity-klass som ska kommas åt, men också som i detta fallet en controller

som heter MyStudioController.cs, eftersom en åtkomstpunkt ska vara api/mystudio. T.ex. för åtkomstpunkt api/films, (Film) så skapas controllern FilmsController.cs.
I varje controllerklass så talar jag först om att det är en api-kontroller [ApiController], sedan grundrouten för controllern [Route("api/[controller]")]. Varje ytterligare åtkomstpunkt med denna grundroute definieras i HttpAttributet. För att t.ex. registrera en filmstudio anges i HttpPost-attributet ”register”, [HttpPost("register")], eller för att lämna tillbaka en film, [HttpPost("return")]. I detta senaste fallet behövs dock två query parametrar (filmID och studioId), för att det ska funka. Består dock endpointen av en url parameter ligger det i attributet mellan ”måsvingar”, ex. [HttpPatch("{filmId}")].
Överföringen av data är tillståndslös och följer arkitekturmönstret REST. Resurserna i koden innehåller de delar av CRUD som kravspecifikationen har.
Genom att skapa controllers med namn som är andra delen i ”pathen” av endpointen (/api/ första, eftersom det är ett API) och Action-metoderna definierar den tredje delen, så blir det ett bra och lättförståeligt mönster enligt OOP.

Implementation

De interna modellerna (entities) som finns är Film, FilmCopy, FilmStudio och Users. Modellerna utgår från substantiven i kravspecifikationen (OOP) men också interfacen i kravspecifikationen. Dessa modeller utgör grunden för API:et, då det är dessa resurser som API:et ska hantera och distribuera. Modeller kan innehålla känslig information, som t.ex. password i User-klassen. Eftersom att vi kanske inte vill visa all denna informationen för api-användaren så har jag skapat nya transfer-klasser, som har som uppgift att transportera den data vi vill visa, från den interna klassen till användaren av API:et. För att tydliggöra detta i kodens designmönster, så ligger de tillsammans med deras interfaces i mappen ”Transfer”. Filen är också döpt till klassen de ska transportera information från, uppgift, om det tillhör en request eller response samt ”Data”, eftersom det är just det klassen transporterar. Ex. UserAuthenticateRequestData.cs, där klassen utgår från User, det handlar om att autensiera, det är en förfrågan (request) samt data-transport. Tydligt plats i strukturen och namn på mapp och fil enligt OOP.

Lista -- interna klassen => synlig klass:

FilmStudio.cs => FilmStudioNoCityResponceData.cs -- *innehåller **inte;** City och lista med FilmCopies.*

Film.cs => *IFilm interfacet används vid responce. Behöver inte någon transferklass enligt kravspecifikation.*

User.cs => UserRegisterResponceData.cs -- *innehåller **inte;** FilmStudioID, FilmStudios och Password.*

UserAuthenticateResponceData.cs -- *innehåller **inte;** Password.*

FilmCopy => *FilmCopy ligger bara med som en lista i Film om den inkluderas. Ingen infomation i denna klassen behöver döljas, därför behöver den inte någon transferklass enligt kravspecifikation.*

Det är alltså utifrån dessa transper-klasser som data skickas ut samt tas emot i API:et.
För att lyckas med detta använder jag Automapper för att överföra data mellan klasserna. Automapper behöver en instruktion (mappning) mellan vilka klasser som data ska överföras. Denna mappning görs i klasser som ligger i mappen ”Mapping”. Klasserna är döpta efter den klass den ska ”mappa” från, samt ”Mapping”, ex. UserMapping.cs. Här definierar också namnen vart klassen utgår från samt uppgift. De ligger också samlade i en egen mapp som är döpt efter uppgiften, vilket ger en klar bild och är enligt OOP.

Säkerhet

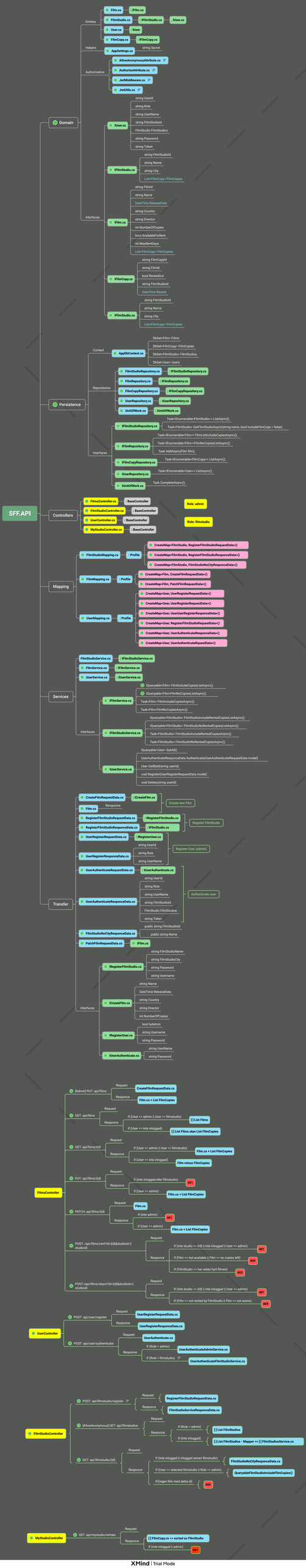
Denna del går ihop lite med föregående avsnitt. Begränsningen av data som skickas till användaren görs genom de transfermodeller som angetts i föregående avsnitt. För att undvika att data som inte api-användaren ska ha åtkomst till, överförs bara de fält som ska skickas från entity-klassen till en ny transfer-klass. Ett exempel är endpoint /api/users/authenticate. Här vill jag att användaren bara ska behöva skicka sitt användarnamn samt lösenord, och därför innehåller UserAuthenticateRequestData.cs bara fälten UserName och Password. Detta är alltså en transferklass som begränsar informationen eller talar om vilken information som ska skickas med i anropet. När API:et sedan svarar och skickar data till användaren, vill jag inte att användaren ska få all information i klassen User, speciellt inte nu när den ärver av den inbyggda klassen IdentiyUser, då skull enormt mycket information skickas tillbaka till användaren. Därför har jag skapat en transferklass som heter UserAuthenticateResponceData.cs och bara innehåller de fält som jag vill att användaren ska

få tillbaka i förfrågan. I detta fall Id, Role, UserName, Token, FilmstudioID och klassen Filmstudio med dess information. De två senare fälten blir ”null” om det inte är en filmstudio som autentiserar sig. På detta sättet säkerställer jag att ingen känslig data ofrivilligt skickas tillbaka till användaren.
Det finns tillfällen då en entity-klass inte innehåller någon känslig data eller att jag vill att användaren ska få all data. Då behöver jag inte skapa någon transferklass, utan använder original-modellen. Exempel är endpoint: /api/films/{id} där ett anrop (autentiserad admin) ska returnera ett json-objekt som innehåller allt som interfacet IFilm innehåller.

Det finns flera säkerhetsåtgärder när det gäller skapade användare i API:et. Dels krypteras lösenordet med BCryptNet, när en användare skapas, vilket gör att även om lösenordet skulle skickas med i ett svar på ett anrop, så kan det inte användas. Den andra säkerhetsåtgärden är att man måste använda token för att göra anrop till API:et. Token är satt till sju dagars giltighetstid och kan ändras ifall man vill öka säkerheten. Token skapas med en nyckel som finns i appsettings.js. Nyckeln kan vara vilken text som helst och bör vara svår att lista ut.
Vid ett autentiserings request, så anges användarnamn och lösenord. Som responce får användaren tillbaka token samt sina användaruppgifter, förutom lösenordet.
Anrop är också skyddade med roller. Har användaren en roll som ”admin” får användaren viss information och är användaren en ”filmstudio” så får den en annan information. Det kan också handla om tillträde till en viss endpoint. T.ex. ett anrop till api/films/ med PUT, kräver både Authentication och Authorization vilket gör att bara user med rollen ”admin” får tillträde.
För att få fram denna information i ett anrop med token som authorization, så avkodas token och UserId:t i token används för att hämta användaren ur databasen. Användaren sparas sedan undan i HttpContext.Items["User"] och kan användas för roll-autentisering i controllers.
Jag valde att inte använda IdentityCore, utan gjorde egna authorization-klasser.
På platsen Domain/Authorization ligger dessa klasser sparade. Där finns en klass som skapar token, samt avkrypterar token för autentisering och auktorisation (JwtUtils.cs). Två klasser som skapar attributen [Authorize] och [AllowAnonymous] (AllowAnonymousAttribute.cs och AuthorizeAttribute.cs), och en klass som används som middleware (JwtMiddleware.cs).

I klientgränssnittet fungerar det så att vid inloggning skriver användaren in sitt användarnamn och lösenord. När knappen ”logga in” klickas på, skickas med fetch ett anrop till api/users/authenticate med användarnamn och password som ett json-objekt i bodyn.

Via responsen sparas den returnerade token, filmStudioId samt userName undan i localstorage. Variabler i javascriptfilen hänvisar sedan till de undansparade värdena.
Alla övriga anrop till API:et görs efter det med de undansparade värdena. Som ”Authorization”, ”Bearer {tokenvariabeln}” i headern, och där det behövs filmstudions id som url-parameter i pathen.
För att användaren ska uppleva det som en inloggning, försvinner också inloggningsfälten samt att filmstudions namn visas tillsammans med utloggningsknappen. En knapp för att lista alla hyrda filmer visas också när filmstudion har autentiserat sig, d.v.s. loggat in.
Vid utloggning rensas localstorage, namnet på filmstudion tas bort, knappen ”lånade filmer” tas bort, inloggningsfält för username och password samt inloggningsknappen visas åter igen.



Sveriges Förenade Filmstudios

Användarnamn

Lösenord

Logga in

Filmbibliotek **Lånade filmer**

Die Hard

Låna

Die Hard -2

Ej tillgänglig

Die Hard

Återlämna