Arkitekturdokument

Version 0.3 PUM02 2025-04-23

Projektidentitet

Medlemmar

Namn	Ansvarsområde	Mail
Alice Almgren	Teamledare	alial202@student.liu.se
Anton Taber	Testledare	antta671@student.liu.se
Axel Berg	Arkitekt	axebe390@student.liu.se
Isabel Neubauer	Dokumentansvarig	isane541@student.liu.se
Jakob Tormalm	Analysansvarig	jakto054@student.liu.se
Samuel Tuvstedt	Backendutvecklingsledare	samtu593@student.liu.se
Samuel Åkesson	Konfigurationsansvarig	samak519@student.liu.se
Simon Gunnarsson	Frontendutvecklingsledare	simgu061@student.liu.se
Stina Åström	Kvalitetssamordnare	stias606@student.liu.se

Kund:
Digitaliseringsavdelningen, Linköpings universitet

Handledare:
Eric Ekström

Examinator / kursansvarig:
Lena Buffoni

Dokumenthistorik

Version	Datum	Ändringar	Granskad av	Utfärdat av
0.3	2025-04-23	Lägg till EmploymentRate till User		S.Åk
0.2	2025-04-10	'Entra-ID' i Figur 1. Öka kontrast i Figur 3. PascalCase för datatyper i Avsnitt 3.2. Byt från 'Assignment' till 'Task'.	A.B	S.Åk, A.B
0.1	2025-03-11	Skapad.	S.G	A.B, S.Åk, S.Ås

Innehåll

1. Referenser

[1] A. Almgren *m.fl.*, ”Kravspecifikation”, technical report, feb. 2025.

[2] ”Google JSON Style Guide”. Tillgänglig vid: https://google.github.io/styleguide/jsoncstyleguide.xml?showone=Property_Name_Format#Property_Name_Guidelines. [Åtkomstdatum: 11 mars 2025]

[3] A. Almgren *m.fl.*, ”Kvalitetsplan”, technical report, feb. 2025.

[4] Svelte, ”Svelte Introduction”, 2024, *Svelte*. Tillgänglig vid: https://svelte.dev/docs/svelte/overview. [Åtkomstdatum: 26 februari 2025]

[5] Microsoft, ”Entity Framework Core”, 11 december 2024, *Microsoft*. Tillgänglig vid: https://learn.microsoft.com/en-us/ef/core/. [Åtkomstdatum: 26 februari 2025]

[6] ”OpenAPI Specification”. Tillgänglig vid: https://swagger.io/specification. [Åtkomstdatum: 11 april 2025]

[7] ”Redoc Github”. Tillgänglig vid: https://github.com/Redocly/redoc. [Åtkomstdatum: 11 april 2025]

[8] Prettier, ”What is Prettier?”, *Prettier*. Tillgänglig vid: https://prettier.io/docs/. [Åtkomstdatum: 14 februari 2025]

[9] CSharpier, ”CSharpier”, 11 februari 2025, *CSharpier*. Tillgänglig vid: https://csharpier.com/. [Åtkomstdatum: 14 februari 2025]

[10] typescript-eslint, ”typescript-eslint”, *typescript-eslint*. Tillgänglig vid: https://typescript-eslint.io/getting-started. [Åtkomstdatum: 17 februari 2025]

[11] K. S. Dániel Varró, ”Software Architecture”, 12 februari 2025. Tillgänglig vid: https://www.ida.liu.se/~TDDC88/theory/05architecture.pdf. [Åtkomstdatum: 11 september 2024]

[12] K. S. Dániel Varró, ”Design Patterns and UML Modeling Practice”, 12 februari 2025. Tillgänglig vid: https://www.ida.liu.se/~TDDC88/theory/06DesignPatterns-UMLPractice.pdf. [Åtkomstdatum: 05 september 2023]

2. Inledning

2.1. Syfte

I detta dokument beskrivs och motiveras tidsredovisningssystemets design- och arkitekturval för att hjälpa nuvarande och framtida utvecklare att förstå systemet. Arkitekturen grundas i stor utsträckning på de krav som specificerats av kunden, de resulterande begränsningarna och den förväntade kvalitetsnivån för produkten.

Dokumentet fungerar som stöd och vägledning för utvecklarna av produkten genom att ge en övergripande beskrivning och motivering av systemets arkitektur på en hög nivå. Dokumentet förväntas även vara hjälpsamt till dem som ska underhålla systemet genom att ge en tydlig översikt och förklaring av de bakomliggande besluten gällande arkitekturen.

2.2. Arkitekturella mål och filosofi

De arkitektuella målen baseras på att systemet ska vara enkelt att underhålla och robust för långsiktig användning. Systemet förväntas vara i drift under en längre tid vilket är viktigt att ha i åtanke vid designen av arkitekturen.

Baserat på detta har nedanstående designmål definierats:

- **Underhållsvänligt:** Systemet ska vara enkelt att underhålla, vilket innebär välstrukturerad kommenterad kod, bra dokumentation och en arkitektur som underlättar vidareutveckling.
- **Robust och långsiktigt hållbart:** Eftersom systemet förväntas vara i drift under en längre tid, ska arkitekturen vara stabil och pålitlig.
- **Integration med LiU:s SSO:** Systemet ska stödja integration med LiU:s SSO.
- **Skalbarhet och prestanda:** Arkitekturen ska möjliggöra hantering av stora mängder data över tid utan att försämra systemets prestanda.
- **Modularitet och vidareutveckling:** Arkitekturen ska vara modulär så att kunden enkelt ska kunna vidareutveckla och anpassa det efter framtida behov.

2.3. Antaganden och beroenden

Systemets arkitekturella val begränsas av de krav av programmeringsspråk och ramverk som har bestämts i *kravspecifikationen* [1]. Användargränssnittet begränsas även av de tillgänglighetskrav som efterfrågats av kunden.

Systemet ska även integreras med LiU:s SSO vilket utgör ytterligare ett beroende eftersom det kräver att tjänsten är tillgänglig och fungerar som förväntat.

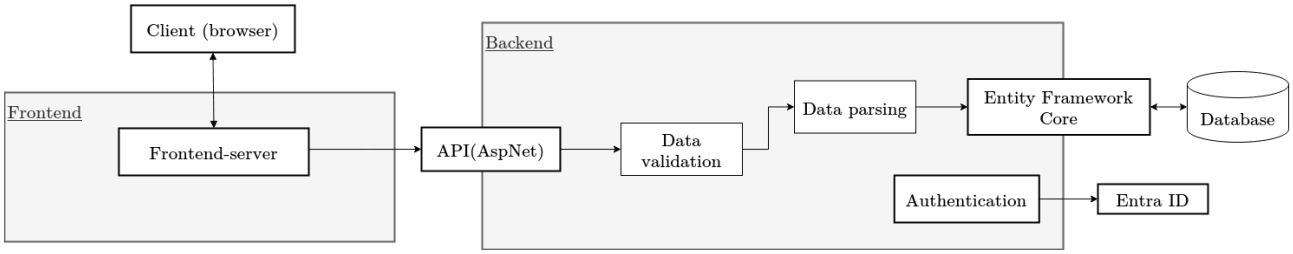
Dessutom påverkas implementationen och tidsåtgången av projektgruppens kompetens och förmåga att lära sig de programmeringsspråk och ramverk som ska användas.

3. Systemet

Detta kapitel beskriver övergripande implementationsdetaljer kring produkten.

3.1. Översikt på systemet

Figur 1 illustrerar hur systemet är uppbyggd. Till vänster syns den externa webbläsaren och dess kontakt med frontend-servern. Till höger syns backend-servern och dess kontakt med frontend, databas och extern Entra-ID applikation för autentisering.



Figur 1: Blockschema över systemet

Frontend-serverns uppgift är att generera och skicka webbsidor till webbläsaren samt hantera all kommunikation mellan webbläsaren och systemet. Den data som används för att generera webbsidorna hämtas från backend-serverns API, se Avsnitt 7.2.

Backend-serverns uppgift är att ta emot förfrågningar från frontend-servern, generera lämpliga objekt från databas-data och utföra all logik. Ibland kommer frontend göra förfrågningar efter objekt som perfekt överensstämmer med tabeller i databasen, men ibland kommer mer tunna objekt (se Avsnitt 3.2.2) genereras från de typerna.

3.2. Datatyper

En del data återkommer ofta tillsammans och grupperas och namnges och defineras därför som systemets *datatyper*.

Dessa datatyper återkommer som objekt i frontend, backend och data.

En uppdelning mellan två kategorier av datatyper har identifierats: *tunna* datatyper och *hårda datatyper*.

3.2.1. Hårda datatyper

Hårda datatyper kännetecknas av att de har en direkt motsvarig tabell i databasen, som en konsekvens av *Entity Framework*. För det mesta kommer frontenden inte vara medveten om dess existens, utan får arbeta med *tunna datatyper*.

En lista över alla hårda datatyper:

- User : Information om en viss användare.
- Favorites : Vilka Tasks som en viss användare vill ska synas högst upp i listorna.
- TimeReport : En tidsredovisning. Är kopplad till både en Task och en User.
- Task : En uppgift. Är kopplad till grupp och aktivitet. Har namn, start- och slutdatum.

3.2.2. Tunna datatyper

Tunna datatyper kännetecknas av att frontend ofta använder dem för att bygga upp dokumentet. De är ofta väldigt specifika och sammanhangsberoende.

Exempel på tunna datatyper:

- TaskWithoutUser : En Task som inte har information om medlemmar, eftersom användare redan är inloggad och vet att den är medlem.
- TimeReportView : En lista av TimeReport för en viss tidsperiod. Behövs till exempel för att rita upp en månadsvy i frontend utan onödig data.

3.3. Dataserialisering

Alla datatyper måste kunna konverteras till strängformat för att kunna skickas genom nätverksförfrågningar. I både backend- och frontendkoden finns stöd för automatisk serialisering, i form av JSON. Enligt Google JSON Style Guide [2] ska properties vara i camelCase, och det är även så C# automatiskt JSON-serialiserar objekt.

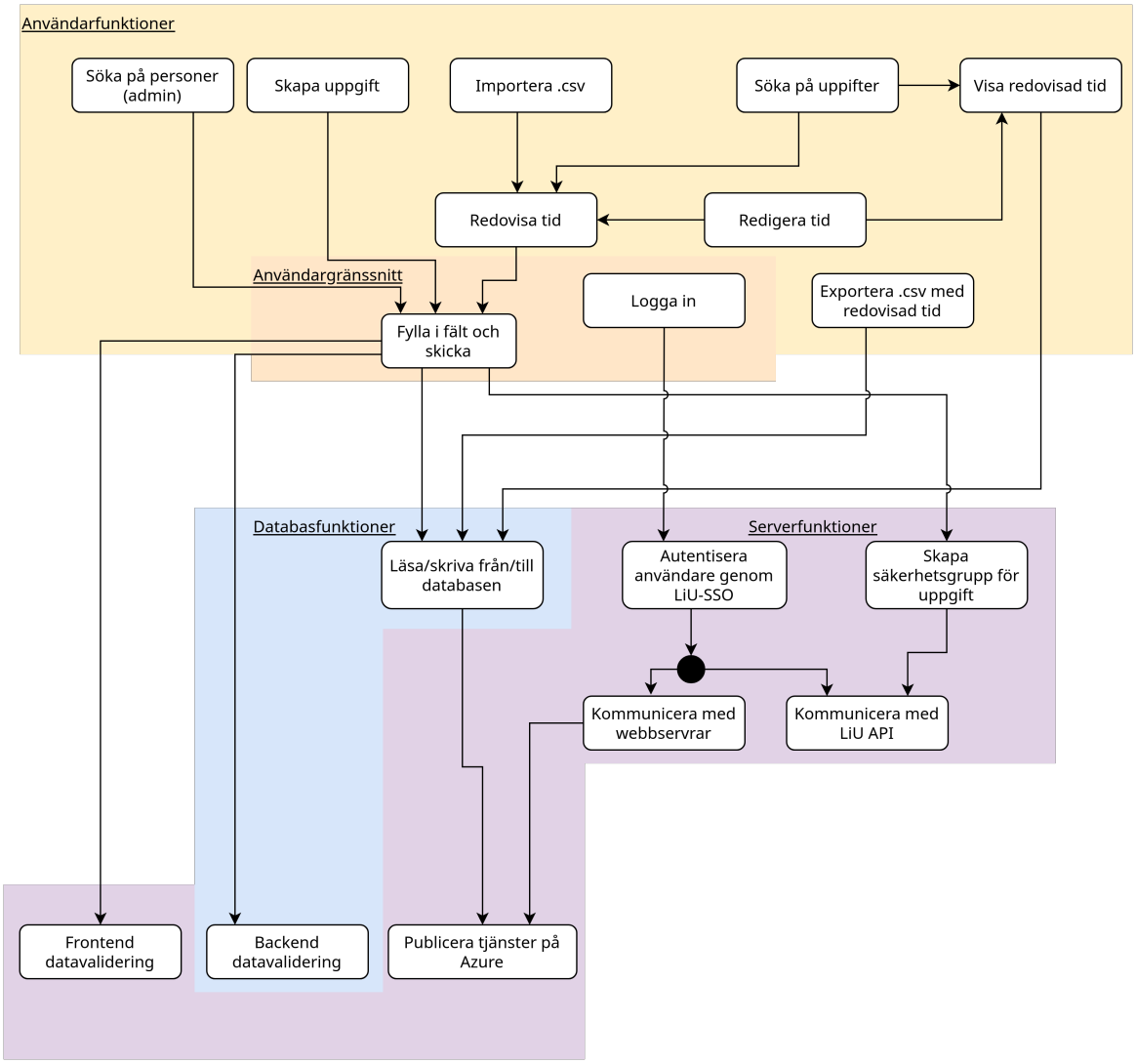
Ett speciallfall för en del tunna datatyper som härstammar från time_report är att de måste kunna konverteras till .csv-format. Tabell 1 visar formatet systemet förväntar sig under import/export av .csv filer.

Tabell 1: .csv-representation av en del tunna datatyper som beror på TimeReport

	Jan 1	Jan 2	...	Jan 31
Lisam-DevGroup-TimeTracker	0	0	0	0
LiuApp-DevGroup-TimeTracker	0	0	0	0

3.4. Systemanatomi

Baserat på användningsfallen som definierats i *kravspecifikationen* [1] har en systemanatomi skapats. Den syns i Figur 2.



Figur 2: Systemanatomin

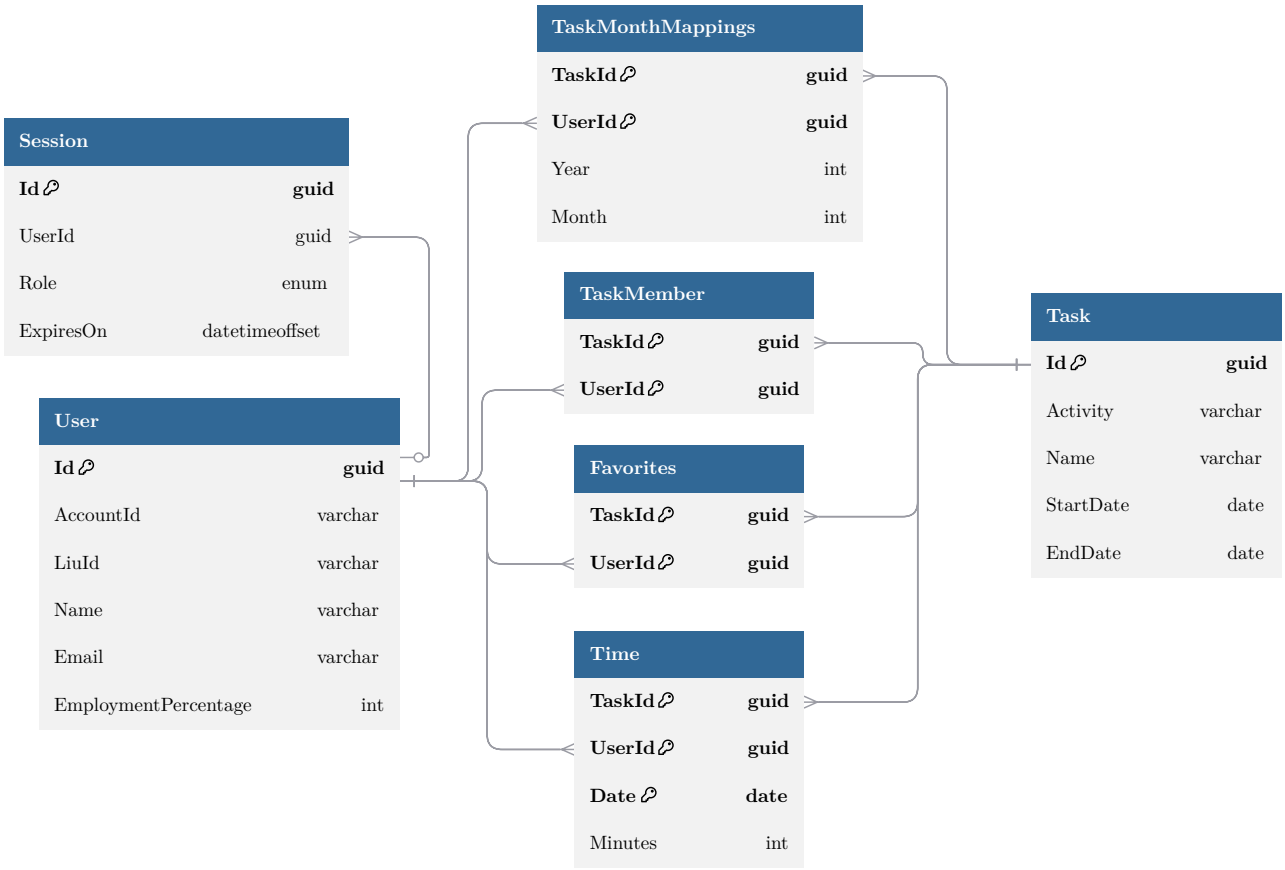
Systemanatomin visar vilka funktionaliteter som är beroende av andra funktionaliteter.

Högst upp syns funktionalitet som slutanvändaren är intresserad av. Varje lager nedanför är funktionalitet som krävs för att uppnå allt ovanför. Man kan läsa det som att varje funktionalitets höjd överensstämmer med dess abstraktionsnivå.

3.5. Struktur på databas

Alla hårda datatyper som definieras i Avsnitt 3.2.1 har, som tidigare nämnt, en motsvarande tabell i databasen. Många av tabellerna har relationer till varandra, vilket illustreras genom pilar mellan kolumner.

En översikt över databasen kan ses i Figur 3.



Figur 3: Diagram över databasen

4. Arkitekturpåverkande krav

I Tabell 2 listas de krav som påverkar arkitekturen på något sätt och bör därmed has i åtanke vid designbeslut. Samtliga är hämtade från *kravspecifikationen* [1]. Kraven är listade tillsammans med en kort beskrivning samt en motivering till varför det är relevant för arkitekturen.

Tabell 2: Lista över arkitekturpåverkande krav

Krav-ID	Beskrivning	Motivering
FR1, FR2, FR4, FR9, FR10, FR12	Beskriver inloggningsprocessen samt hur systemet ska anpassas baserat på användarens grupp.	Kräver integration med LiU:s SSO och hantering av grupper.
FR3, FR20	Anger att endast LiU-ID ska lagras som personuppgift, och ingen data får raderas från databasen, även om en användare tas bort från en uppgift.	Begränsar databasdesignen i vilken data som ska sparas och kräver att systemet säkerställer att tidsredovisningsdata bevaras.
FR17, FR18	Systemet ska kunna exportera och importera data i .CSV-format.	Kräver stöd för att skriva och läsa .CSV-filer.
FR16, FR22, FR23	Systemet ska ha stöd för filtrering och sökning på uppdrag och personer.	Kräver att databasen har stöd för sökningar och filtreringar.
NFR1, NFR2	Systemet ska implementeras med C#, ASP.NET, Svelte och Typescript.	Begränsar valen för implementeringen av frontend och backend.
NFR4, NFR8	Systemet ska följa LiU:s grafiska profil och specifika tillgänglighetskrav	Påverkar designen av front-end.

5. Val och begränsningar

I detta avsnitt beskrivs några designbeslut, begränsningar och motiveringen till dessa.

5.1. Systemövergripande

- Användare av tidsredovisningssystemet autentiseras genom LiU:s SSO. Det har valts eftersom kunden använder det systemet för att hantera tillgång till systemet.
- Systemet har delats upp i två servrar: en för frontend och en för backend. Det har gjorts för att uppnå modularitet, vilket var ett mål enligt *kvalitetsplan* [3]. Om servern hade implementerats i en enda kodbas är det större risk att det blir otydligt vilken det av systemet som gör vad, var man ska leta för att hitta ett beskrivningen för ett visst beteende och att ett komplicerat nät av beroenden skapas. Uppdelningen gör så att varje server kan jobba isolerat och har en tydlig uppgift.

5.2. Frontend

- Användargränssnittet ska vara byggd i ett frontend-ramverk. Detta för att ett gränssnitt med ren HTML+CSS+JS kräver för mycket repetition av kod (boilerplate).
- Frontend ska vara baserat på och utvecklat med Svelte [4].

5.3. Backend

- I enlighet med kundens vilja ska backend-servern köras i ASP.NET och driftsättas på Azure.
- Databasen driftsätts med Microsoft SQL i enlighet med kundens vilja.
- Uppkopplingen till databasen kommer ske genom Entity Framework Core [5] som gör att databasens tabeller och anrop kan definieras i C#-kod direkt i backend. Mer om Entity Framework Core står i Avsnitt 7.1.
- För sessioner har vi valt att använda vår egna lösning som sparar sessioner genererar ett GUID som id och sparar den i databasen. Anledningen är att den förbyggda lösningen, `Microsoft.AspNetCore.Session`, gör antagandet att backend-servern får förfrågningar direkt från klienten och skickar därför en `Set-Cookie`-header. Eftersom all trafik skickas genom frontend-servern med vår arkitektur innebar det att vi behövde skapa en ful lösningen för att få den förbyggda att fungera. Dessutom sparar den förbyggda lösningen alla sessioner i minnet och vi ville spara dem i databasen. Vi gjorde det valet innan vi hade fått tillgång till LiU-SSO och inte visste hur länge en LiU-SSO-session är giltig. Med tanke på att de bara är giltiga i en och en halv timme spelar det ingen större roll om man skulle förlora alla sessioner vid server omstart och att spara dem i minnet skulle vara ok, om inte bättre.

5.4. Utvecklingsmiljö

- Vid utveckling av C#-kod kommer vi använda oss utav OpenAPI [6] och Redoc [7] för att generera dokumentation till API. Den kan då användas som referens av både frontend och backend för vilka ändpunkter som finns, vilken data den förväntar sig och vilken data den skickar.

- Vid utveckling i Svelte ska formatteringsverktyget *Prettier* [8] användas för att få ett snyggt och unisont utseende på koden. På samma vis kommer CSharpier [9] användas för att formatera C#-koden.
- Vid utveckling i Svelte ska *ESLint* [10] användas för att hitta potentiella problem i koden.

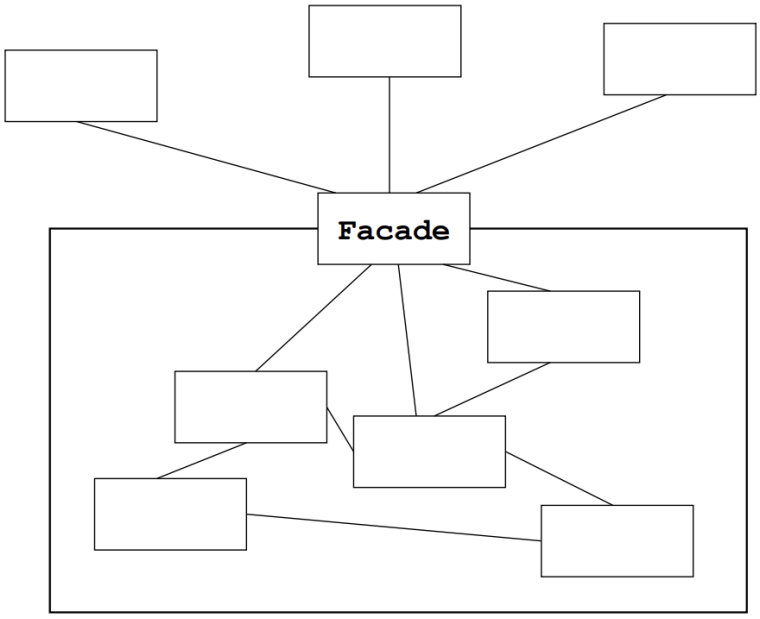
6. Designmönster

6.1. *Two-tier, Thin-client*

Systemet följer en klient-server modell som delar upp systemet i två nivåer där den största belastningen ligger på servern:

- **Klient:** Detta representerar användargränssnittet och hanterar all interaktion med användaren såsom inmatning och presentation av data.
- **Server:** Ansvarar både för affärslogiken och datahantering. Detta inkluderar lagring, databasoperationer och hämtning av data men även att förmedla data mellan klient och servern. Här behandlas också användarförfrågningar och beräkningar utförs [11].

6.2. Fasad



Figur 4: Illustration av designmönstret fasad [12]

Fasad är ett designmönster som tillhandahåller ett enhetligt gränssnitt till en uppsättning gränssnitt i ett delsystem. Mönstret illustreras i Figur 4, där *Facade* är den engelska benämningen på fasad. Det fungerar som ett högre-nivå gränssnitt med syftet att förenkla användandet av delsystemet. Fördelarna med fasad är att det minskar systemets komplexitet, gör systemet mer återanvändbart och enklare att anpassa. Dessutom främjar det svag koppling, vilket är önskvärt eftersom det möjliggör förändringar, gör koden blir mer lättförståelig och testbar genom att isolera fel [12].

Detta förenklar kommunikationen mellan systemet frontend och backend, eftersom frontend kan göra anrop till fasaden utan att behöva känna till den underliggande strukturen.

Genom att använda designmönstret fasad uppfylls kundens krav på ett system som är lätt att underhålla.

7. Abstraktioner

De två huvudsakliga abstraktionerna som systemet kommer implementera ligger mellan backend-servern och databasen samt mellan backend-server och frontend-servern.

7.1. Entity Framework Core

Mellan backend-servern och databasen används verktyget *Entity Framework Core* [5]. Det förenklar och abstraherar kommunikation mellan backend-servern och databasen.

Verktyget kopplar en klass i C#-koden på backend-servern till en tabell i databasen och innehåller även funktionaliteten att uppdatera databasens tabeller efter en ändring har gjorts i koden.

Fördelen med att använda ett sådant verktyg är att all data som hämtas från databasen garanterat har rätt typ. Det förenklar för utvecklarna att skriva mer koncis och säker kod. Eftersom databasens struktur definieras i koden kan den användas som referens av utvecklarna utan att behöva öppna ett annat program, den kan även av samma anledning enkelt återskapas för en ny miljö, exempelvis en testningsmiljö.

7.2. API

Mellan backend-servern och frontend-servern kommer en API finnas som gränssnitt. Målet med API:n är att abstrahera bort så mycket som möjligt från frontend-servern så att den enkelt kan hämta alla data den behöver för att generera en webbsida med ideellt ett anrop. Alternativet är att istället behöva göra ett stort antal anrop till API:n för att sedan filtrera alla data som hämtats. Det separerar även ansvaret mellan backend-servern och frontend-servern där backend-serverns jobb är att hantera data och frontend-serverns jobb att generera webbsidor. API:n kommer använda sig utav designmöstret fasad (Avsnitt 6.2) genom att ge frontend-servern ett simpelt gränssnitt för att hämta data som gömmer komplexiteten av att samla ihop den data.