CHALMERS

Administrativ dashboard i Optimizely CMS

Säkerhetsanalys av API-kommunikation och åtkomstkontroll

Examensarbete inom högskoleingenjörsprogrammet i datateknik

Nuha Alshami

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA

Göteborg, Sverige 2026
www.chalmers.se

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2026
www.chalmers.se

Administrativ dashboard i Optimizely CMS

Säkerhetsanalys av API-kommunikation och åtkomstkontroll

Nuha Alshami

CHALMERS

Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
Göteborg 2026

Administrativ dashboard i Optimizely CMS
Säkerhetsanalys av API-kommunikation och åtkomstkontroll

Handledare:
Marcus Andersson, Precio Fishbone
Henrik Jansson Valter, Chalmers tekniska högskola
Examinator: John J. Camilleri, Chalmers tekniska högskola

Examensarbete 2026
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg
Telefon +46 31 772 1000

Administrativ dashboard i Optimizely CMS
Säkerhetsanalys av API-kommunikation och åtkomstkontroll
NUHA ALSHAMI
Institutionen för Data- och informationsteknik
Chalmers Tekniska Högskola

Abstract

The following thesis aims to investigate security aspects of API and web applications. Those include broken access control and secure communication between systems. Furthermore, the work consists of developing an Optimizely based web application that uses a backend-API for a lot of its logic. Several security mechanisms were then implemented in different layers. The web application has security for authentication, session handling and role-based access control which together work to make administrative resources restricted. The communication between the systems is protected with a server-side proxy which stops the API-key from being exposed in the browser. The other layers of security are left to the backend-API that handles API-key validation, project membership authorization, input validation, rate limiting, audit logging and supplementary defenses with CORS and Content Security Policy.

The implementation was tested from an attacker perspective that utilizes browser DevTools, terminal for requests and local traffic inspection with Fiddler. The results show that the implemented solution mitigates the risk of unauthorized access and follows central security principles even if it still has limitations. For example, the use of a shared symmetric secret between the systems and the solution was only tested in a local development environment.

Förord

Detta arbete utfördes inom högskoleingenjörsprogrammet i datateknik vid Chalmers tekniska högskola i samarbete med Precio Fishbone AB. Bakomliggande syfte är det ökade behovet av säkra administrativa webbapplikationer där en central risk i säkerhetssynpunkt är risken för bruten åtkomstkontroll och osäker kommunikation. Jag vill ge stort tack till Precio Fishbone AB som gett mig möjligheten att utföra examensarbetet hos dem. Jag också tacka min handledare Henrik Jansson Valter och examinator John J. Camilleri hos universitetet som gett vägledning och värdefull återkoppling under arbetets gång.

Nuha Alshami, Göteborg, Maj 2026

Beteckningar

Nedan listas beteckningar som används inom denna avhandling i alfabetisk ordning:

API – Application Programming Interface

ASP.NET Core – Ramverk för utveckling av webbapplikationer och API:er i .NET

BFF – Backend-for-Frontend, ett mellanliggande lager mellan frontend och bakomliggande tjänster

CMS – Content Management System

CORS – Cross-Origin Resource Sharing

CSP – Content Security Policy

CSRF – Cross-Site Request Forgery

HTTP – Hypertext Transfer Protocol

HTTPS – Hypertext Transfer Protocol Secure

IDOR – Insecure Direct Object Reference

IP – Internet Protocol

JWT – JSON Web Token

OWASP – Open Worldwide Application Security Project

RBAC – Role-Based Access Control

REST – Representational State Transfer

SQL – Structured Query Language

XSS – Cross-Site Scripting

Innehåll

1. Inledning

Företag och organisationer använder webbapplikationer i allt större utsträckning såväl internt som externt. Systemen används ofta för administrativa uppgifter där säkerhet blir avgörande vid hantering och bearbetning av data. Vid säkerhetsincidenter blir ofta känslig information kompromitterad genom till exempel obehörig åtkomst till databasen eller manipulation av webbapplikationens funktioner genom API:et.

Detta arbete undersöker säkerhetsaspekter i administrativa webbapplikationer med fokus på säker kommunikation mellan webbapplikationen och ett backend-API samt rollbaserad åtkomstkontroll.

1.1 Bakgrund

Precio Fishbone AB är ett svenskt konsult- och mjukvaruutvecklingsföretag som utvecklar digitala lösningar för kunder inom privat och offentlig sektor. Med cirka 170 specialister på kontor i Sverige, Storbritannien och Vietnam byggs Microsoftbaserade lösningar som utvecklar operativa fördelar och långsiktig affärsnytta [1].

Företaget har ett partnerskap med Optimizely för att erbjuda kunder effektiva och anpassningsbara digitala plattformar [2]. Optimizely är ett CMS (content management system) vilket är ett system som hanterar och publicerar olika typer av information [3]. Plattformen Optimizely möjliggör bland annat effektiv innehållsproduktion genom återanvändbara mallar, anpassningsbara komponenter och redaktörgränssnitt som inte kräver programmeringskunskaper [4].

Webbapplikationer baseras till stor del på CMS plattformar som Optimizely. De erbjuder bland annat lösningar för användaradministration, innehållshantering och rollbaserad åtkomstkontroll. Plattformarna är utformade i syfte att göra webbutveckling snabbare och effektivare [3].

Plattformen möjliggör även att utvecklare kan skapa webbapplikationer som bara finns tillgänglig för administratörer. Sådana system skiljer sig från publika webbplatser genom att ge användare särskilda roller som får möjligheten att skapa, ändra och radera information. Dessa webbapplikationer kan skapas på ett säkert sätt genom att kräva rätt behörighetsnivåer och säker kommunikation med backend. Kopplingen kan ske mellan webbplatsen och ett backend-API som i detta projekt.

Bristande konfiguration på åtkomstkontroll medför stor risk att obehöriga individer får tillgång till funktionalitet som är avsedd för administratörer [5]. CMS plattformars inbyggda rollbaserade åtkomstkontroll som inte är upp till standard medför stor säkerhetsrisk av denna anledning.

Säkerhetsincidenter sker också ofta på grund av otillräckligt skydd av API kommunikation med webbapplikationen [6]. I ett verkligt scenario skulle obehöriga användare kunna manipulera funktionaliteter och data. Med detta till bakgrund finns det behov av att

undersöka hur Optimizely hanterar rollbaserad åtkomstkontroll och hur kommunikationen mellan backend-API och en webbapplikation kan ske på ett säkert sätt.

1.2 Syfte

Syftet med examensarbetet är att undersöka hur rollbaserad åtkomstkontroll i Optimizely CMS kan användas samt hur säker kommunikation mellan webbapplikation och ett backend-API kan utvecklas för att minska risken för obehörig åtkomst. Arbetet syftar också till att analysera till vilken utsträckning lösningen uppfyller etablerade säkerhetsprinciper som l*east privilege* och *OWASP:s* rekommendationer.

1.3 Mål

Examensarbetets mål är:

- Utveckla en administrativ webbapplikation i .NET och Optimizely CMS med rollbaserad åtkomstkontroll och autentisering.
- Implementera ett backend-API i .NET med säkerhetsmekanismer för kommunikation med webbapplikationen.
- Undersöka hur den utvecklade lösningen förhåller sig till säkerhets principer.

Examensarbetet utgår från följande frågeställningar:

1. Hur säkerställer den utvecklade lösningen i Optimizely att administrativa resurser endast nås av behöriga användare, samt att åtkomst upphör efter utloggning?

2. Hur kan kommunikationen mellan en Optimizely baserad administrativ webbapplikation och ett backend-API skyddas mot obehöriga anrop?

3. I vilken utsträckning uppfyller den utvecklade lösningen etablerade säkerhetsprinciper, som least privilege och OWASP:s rekommendationer?

1.4 Avgränsningar

Arbetet är fokuserat på säkerheten kring autentisering, auktorisation samt API kommunikation i en administrativ webbapplikation. Utvecklingen sker i lokal miljö och dess tester inkluderar inte produktion aspekter. Säkerhetsanalysen omfattar inte professionell penetrationstestning eller externa hotaktörer, i stället begränsas det till lokala attacker via inspektionsverktyg som Fiddler samt anrop från terminalen för att simulera maskinnära attacker. API och Optimizely projekt använder samma lokala databas med klartext lagring som inte testas under hög belastning. Projektet i helhet är lämpligt för demonstrering men skulle behöva utvecklas innan en potentiell deployment i produktion med verkliga användare på grund av utvecklingsmiljöns begränsningar. En annan viktig avgränsning är att rapporten behandlar säkerhetsaspekter som implementeras i lösningen. Övrig funktionalitet som införts i webbapplikationen och CMS gränssnittet förklaras därmed endast i den utsträckning som krävs för förståelse av säkerhetsanalysen.

2. Metod

Följande kapitel förklarar metoden som planerades innan genomförandet av arbetet, här redogörs vilka planer, verktyg och tekniker som valdes för att uppnå de satta målen.

2.1 Övergripande metodansats

Examensarbetet är utformat som ett praktiskt utvecklingsprojekt där fokus ligger på att utveckla och analysera en teknisk lösning ur ett säkerhetsperspektiv. Arbetet bygger på studie om design och implementation som sedan analyseras för dess brister och styrkor i relation till säkerhet. Beslutet att utesluta en empirisk studie ur detta projekt gjordes för att fokus ska ligga på implementation, testning ur ett attackperspektiv samt analys där säkerhetslösningar granskas i relation till etablerade säkerhets principer.

2.2 Förstudie och kunskapsinhämtning

Före arbetets början planerades det in en förstudie till inhämtning av nödvändig teknisk och teoretisk information. Detta gjordes för att uppnå en förståelse av plattformspecifika och generella säkerhetsmekanismer. Där förstudien infattade en genomgång av hur rollbaserad åtkomstkontroll fungerar hos plattformen Optimizely från deras officiella dokumentation. Ytterligare planerades inhämtning av OWASP Top 10 säkerhetsrekommendationer och brister vad det gäller åtkomstkontroll.

2.3 Planerad utvecklingsmodell, verktyg och tekniker

Det planerade arbetet delades upp i mindre steg enligt en iterativ utvecklingsmodell där olika funktionaliteter och säkerhetsmekanismer utvecklades successivt. Utvecklingsmodellvalet går ut på att dela upp arbetet i mindre upprepade cyklar precis som iterationer där varje iteration är en full utvecklingsprocess med planering, design, kodning och testning. Detta leder till en fungerande delprodukt som utökas tills uppnådd fullt fungerande system i sin helhet. I kontrast till ett arbetssätt där utveckling sker på allt i systemet innan testning upptäcks problem mycket tidigare i processen med den iterativa utvecklingsmodellen.

De verktyg och tekniker som planeras användas är Optimizely CMS för utveckling av den administrativa webbapplikationen som ska ske i programmeringsspråk C#, CSS och html. Backend-API:t ska implementeras med C# i plattformen .NET. Design på API:t ska bygga på en REST baserad arkitektur. Det ska även ha en API-nyckelautentisering för att skydda åtkomsten.

Utvecklingsprocessen planeras att utföras i en lokal utvecklingsmiljö med hjälp av Visual Studio Code texteditor för kod.

3. Teknisk bakgrund

Detta kapitel redogör för den tekniska bakgrund som underlättar förståelsen av rapporten.

3.1 Rollbaserad åtkomstkontroll (RBAC)

RBAC är en etablerad modell för åtkomstkontroll som syftar till att förstärka gränssnittet mellan olika roller ur en säkerhetsaspekt. Principen är att behörigheter inte ges till individuella användare, i stället kopplas de till roller. Användaren kan tilldelas en eller flera roller och inom rollen/rollerna finns det behörigheter till specifika operationer och resurser. Inom administrativa webbapplikationer finns exempelvis roller som administratör och redaktör vilka har olika nivåer av åtkomst i systemet [7].

Fördelen med RBAC är att rollseparationen motsvarar den som finns i många organisationer. I jämförelse med individuella åtkomstlistor medför detta minskad komplexitet och fel risk. Enligt NIST bidrar RBAC till att den administrativa kostnaden minskar, effektivare behörighetsstyrning och den följer säkerhetspolicy bättre [7].

Enligt OWASP:s Top 10 identifieras en bruten åtkomstkontroll att vara den vanligaste orsaken till säkerhetsbrister i moderna applikationer. Typiska åtkomstkontroll brister är [5]:

- Brott mot minsta privilegieprincipen
    - Åtkomst ges i större bredd än nödvändigt
- Förbigående av åtkomstkontroller
    - Kan göras genom att manipulera URL:er, det interna tillståndet av applikationen eller API-anrop
- Otillåten åtkomst till andra användares resurser (IDOR)
    - Via ändring eller gissning kan man hitta unika identifierare som sedan kan användas för att ändra och läsa andra användares data.
- Otillräcklig åtkomstkontroll på API-anrop.
    - HTTP metoder POST, PUT, DELETE (skapa, uppdatera, ta bort) har otillräcklig auktorisation
- Privilegieförhöjning
    - En användare kan få mer behörighet än dess roll tillger genom att använda administratörfunktionaliteter utan tillhörande roll.
- Metadata manipulation
    - Cookies, åtkomsttoken (JSON webb token, JWT) eller andra metadata kan manipuleras eller återanvändas för att ha obehörig åtkomst.
- CORS misskonfiguration
    - På grund av bristande CORS begränsningar kan otillåtna källor göra API-anrop
- Force browsing till skyddade sidor
    - Genom att ange URL:er kan man nå autentiserade sidor utan behörighet.

3.2 Motåtgärder mot bruten åtkomstkontroll

Med dessa säkerhetsbrister som kan existera i ett system finns även metoder för att motverka obehörig åtkomst. För effektivitet av åtkomstkontroll krävs att den implementeras med en

betrodd serverbaserad kod. Om API:t är serverlöst bör åtgärden vara att angriparen inte ska kunna manipulera kontroller eller dess metadata. Ytterligare implementerbara egenskaper [8]:

- ”deny by default”, konceptet att neka åtkomst för resurser utan beviljad behörighet med undantag för publika resurser.
- I syfte av att minska felkonfiguration risken bör åtkomstkontroll mekanismer implementeras en gång och därefter återanvändas inom systemet. I samband med åtgärden bör användning av Cross origin Resource Sharing (CORS) minimeras.
- Åtkomstkontroller bör byggas med separation baserat på dataägarskap i stället för att ge behörighet för en användare att skapa, uppdatera, läsa och ta bort resurser av en viss typ.
- Domänmodeller bör verkställa unika affärsregler och begränsningar för applikationen
- Inaktivera kataloglistning på webbservern och se till att känsliga filer som versionshanterings metadata (.git) eller reservfiler inte finns i webbens rotkatalog.
- Vid upprepade misslyckanden av att någon försöker få åtkomst bör administratörer notifieras
- Åtkomst till API och controller har hastighetsbegränsningar (rate limiting) för att minska skada av automatiserade attacker
- Vid användning av sessionsbaserad autentisering bör sessionsidentifierare göras ogiltiga på servern efter utloggning. För stateless JWT tokens bör livstiden vara förkortad för att angreppsarean ska minimeras. Om livslängden måste vara längre kan refresh tokens samt OAuth standarder för återkallande av åtkomst.
- Använda väletablerade ramverk, verktyg eller designmönster som ger en simpel och förklarande åtkomstkontroll.

3.3 Åtkomstkontroll och behörighetsmodell i Optimizely CMS

Optimizely CMS använder rollbaserad behörighet (RBAC) för administrativa gränssnitt och funktioner samt åtkomsträttigheter på innehåll i redigeringsmiljön. Innehåll kan allokeras rättigheter som skiljer sig åt mellan användare eller grupp. Rättighetsnivåer som finns är begränsningar av vem som får se, skapa, redigera och publicera innehåll. Dessa rättigheter tilldelas i syfte av att antingen kontrollera åtkomsten till enskilda innehållsobjekt och/eller för att vilka funktionaliteter som blir tillgängliga i användargränssnittet [9].

Roller används för att tilldela eller begränsa åtkomst till CMS administration. Optimizely har virtual roles (till exempel CmsAdmins, CmsEditors) som kan mappas till en eller fler grupper/roller och därmed ge en mer flexibel behörighetsmodell där medlemskap kan beräknas på ett dynamiskt sätt. Dessa roller används för att styra åtkomst utan att varje enskild sida ska ha regler som är specialkonfigurerade [10].

3.4 API-autentisering och API-nycklar

En API-nyckel genereras slumpmässigt till en alfanumerisk sträng som hålls hemligt och skickas mellan klient och API. Vid anrop från klienten till API skickas nyckeln via en HTTP header vanligtvis. Därefter verifierar API server att nyckeln är korrekt innan åtkoms till API sker. Denna metod gör att API-nyckel begränsar åtkomst samt för att se vilka system som använder API:t [11].

Från en kryptografisk synpunkt kan användningen av samma API-nyckel analyseras med teorin för symmetriska system. Enligt Padhiar och Mori identifieras symmetrisk kryptografi av att en och samma nyckel används för att autentisera och verifiera eller kryptera och dekryptera. Symmetriska lösningar är simpla att implementera och har en låg beräkningskostnad [12]. Med det till fördel blir de lämpliga för kontrollerade miljöer och interna system.

Den delade hemliga nyckeln framför en säkerhetsrisk för läckage till angripare av systemet. För att hålla API:t säkert kan man rotera/byta nyckel varje 90–180 dagar och ta bort nycklar som inte längre används.

Ett annat kryptografiskt alternativ på typ av system är asymmetriska system. Detta system har nyckelpar av publik och privat karaktär för varje system/användare. Den privata nyckeln hos hemligt medan vem som helst kan se den publika. Asymmetriska system har i kontrast till symmetriska högre säkerhet då de har mekanismer som nyckelseparation, token utgång och möjlighet till återkallning av åtkomst men har kostnaden av ökad komplexitet i administration och implementering [11], [12].

API nycklar är bra för att verifiera applikation identifikation och autentisering men den brister när de brister när det kommer till användaridentifiering och autentisering. Därmed brister säkerhetsaspekten när endast anropande applikationen identifieras och autentiseras snarare än den bakomliggande användaren. På grund av detta är det bra att ha med API nycklar i området API säkerhet men att det ska finns fler sätt att autentisera och validera.

3.5 Cookiebaserad autentisering och sessionshantering

Cookiebaserad autentisering betyder att användaren kopplas till en session av servern efter inloggning. För att sessionen ska identifieras används ett sessionsidentifierande värde eller en cookie som webbläsaren skickar med i efterföljande anrop. På det sättet behåller en redan autentiserad användare sin åtkomst utan ny inloggning. Dock medför detta också en säkerhetsrisk där en stulen eller återspelad sessionscookie kan ge angripare åtkomst till den aktiva sessionen [13].

3.6 CSRF och anti-forgery tokens

Cross-Site Request Forgery (CSRF) sker när webbapplikationen tar emot och behandlar en förfalskad begäran som skapats av en angripare. För att motverka detta kan anti-forgery tokens användas. Dessa genereras för en begäran eller ett formulär och kontrolleras sedan efter mottagande i servern. Detta gör det svårare för en angripare att skicka falska anrop [14].

3.7 CORS

Cross-Origin Resource Sharing (CORS) är en mekanism som begränsar vilka origins som får åtkomst till vissa resurser. Vid anrop till CORS skyddad tjänst inkluderas information om begärans origin. Servern jämför sedan anropets origin med de tillåtna, vid match svarar servern med korrekt CORS header annars blockeras åtkomsten av webbläsaren [15].

3.8 CSP och nonce

Content Security Policy (CSP) bestämmer vilka resurser en sida får ta emot och exekvera. Det fungerar som ett extra skydd mot innehållsinjektionsattacker såsom XSS. När det gäller nonce baserad CSP ger servern ett nonce (”number used once”) värde i headern som gör att all skriptkod som inte matchar värdet i policyn blockeras. Det genererade nonce värdet ska vara nytt för varje svar och även slumpmässigt. Denna säkerhetsmekanism gör exekvering av injicerad kod svårare [16].

3.9 Rate limiting

Rate limiting används för att begränsa antal anrop en klient får göra under en viss tidsperiod [17].

3.10 Audit logging

Audit logging är när händelser relevanta till säkerhet sparas för att man ska kunna spåra misstänkt aktivitet. Det är en viktig del av incidentutredning, nätverks- och systemadministration. Information som sparas inkluderar bland annat autentiseringsförsök och användaraktiviteter. Vidare används mekanismen för uppföljning av misstänkt aktivitet snarare än ett direkt säkerhetsskydd i realtid [18].

3.11 Server-side proxy / backend-for-frontend

Backend-for-frontend (BFF) är ett lager i backend som sitter mellan frontend och bakomliggande tjänster för anpassning av funktionalitet och data till en specifik form som tjänsten kräver. Det är ett arkitekturmönster där backend tjänster anpassar integrationer, dataorkestrering och API anpassning utefter behov som frontend har. Vidare leder detta till en centralisering av anropslogik, datatransformering och därmed anpassning utefter plattformen på serversidan [19]. Inom denna rapport används server-side proxy som samlingsbegrepp för denna typ av mellanliggande lager på serversidan.

4. Genomförande

Följande kapitel beskriver genomförandet av projektets tekniska lösning och beskriver endast säkerhetsaspekter av projektets implementation.

4.1 Miljökonfiguration och verktyg

Den tekniska lösningen omfattade en Optimizely baserad webbapplikation och ett fristående backend-API. De utvecklades på visual studio code och kördes på olika portar med kommunikation över HTTPS. Utvecklingen av webbapplikationen skedde i .NET 8 med ptimizelys CMS och kördes sedan på en oanvänd port. Sedan utvecklades också API:t i .NET 10 och kördes på en annan port. Konfiguration för kommunikation mellan systemen skedde genom att ha API:ts bas URL i webbapplikationens konfigurationsfil. Vidare sparades data med databasen som var en lokal SQL Server Express instans. Den delades mellan webbapplikationer och API:t som skapade sina tabeller vid första uppstart respektive via Entity Framework Core migrationer. Implementation av frontend skedde med react och typescript som byggdes med vite och sedan renderades i Optimizely med razor views.

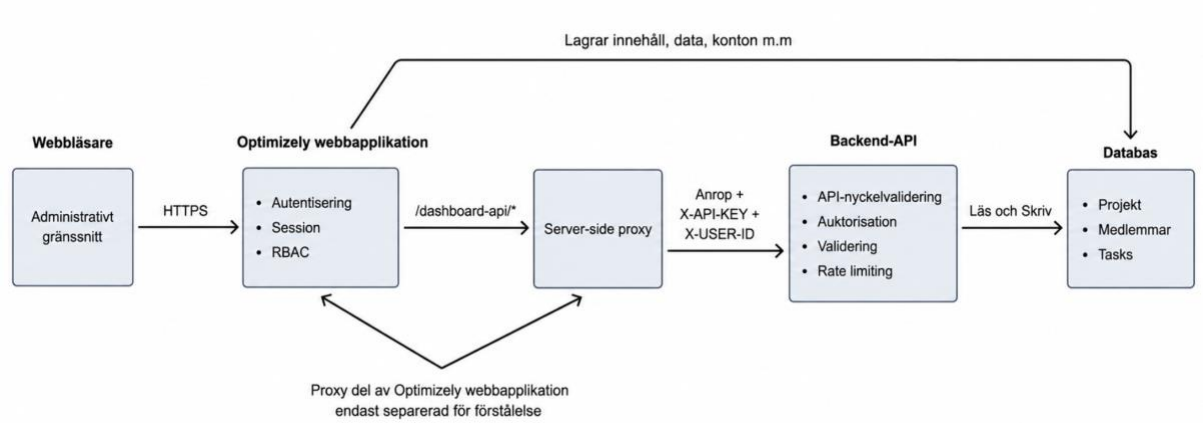
4.2 Övergripande systemarkitektur

Den tekniska lösningen som utvecklas består av en administrativ webbapplikation byggd i Optimizely och ASP.NET Core samt ett privat backend-API också byggt i ASP.NET Core. Arkitekturen mellan dessa två delar är uppbyggd så att säkerhetsfunktioner delas i flera lager.

Den administrativa webbapplikationen hanterar projekt, projektmedlemmar och tasks men får även ansvar över användarautentisering, sessionshantering, rollbaserad åtkomstkontroll samt data presentering. I kontrast till webbapplikationen är backend-API:t inte använt av slutanvändaren direkt, i stället fungerar det som en intern tjänst för sidan. API:t har egna ansvarsområden vilka inkluderar affärslogik, validering av inkommande data, persistent lagring i databas samt en användarspecifik auktorisation i förhållande till projektmedlemskap.

Under första implementationen kopplades webbapplikationen direkt till API:t vilket visade på en säkerhetsrisk då API-nyckeln skickades med i headern och syntes i webbläsarens DevTools. Denna implementation reviderades sedan så att frontend kommunicerade med en server-side proxy i Optimizely applikationen. Lösningen gjorde att webbläsaren inte längre exponerade API-nyckeln, den begränsas till serversidan utan synlighet i klientmiljön.

Flera säkerhetslager samverkar alltså genom webbapplikationens hantering av autentisering, rollstyrning och skydd av administrativa objekt, API:t har i stället i uppgift att införa API-nyckelvalidering, användarspecifik auktorisation, rate limiting och inputvalidering. Vidare kopplas dessa två delar ihop på ett säkert sätt med ett proxy server-side lager som möjliggör same-origin kommunikation samt injicering av API-nyckel från serverkonfigurationen (se Fig. 1).



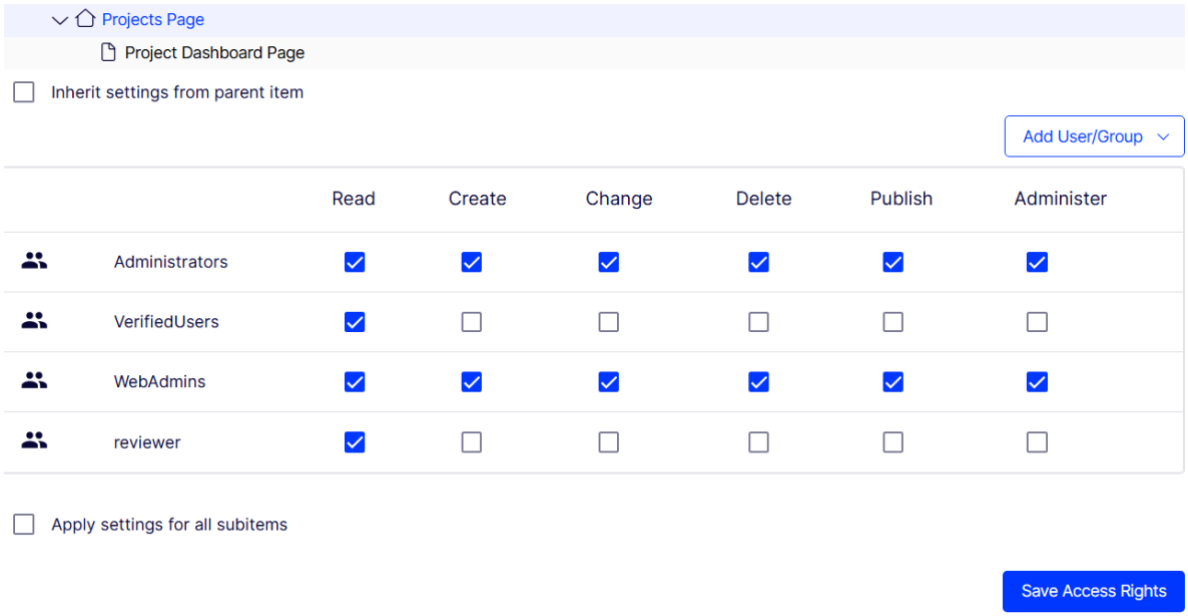
Figur 1: Översiktligt diagram av arkitekturen som visar användarens kommunikation med Optimizelys webbapplikation som lagrar information till databasen vidare finns även kopplingen vidare till server-side proxyns vidarebefordring av anrop med korrekt nyckel och användar-id till backend-API:t som sedan läser och skriver till databasen

4.3 Autentisering och behörighetsstyrning i webbapplikationen

Webbapplikationen hanterar administrativa resurser genom Optimizelys och ASP.NET cores inbyggda stöd för auktorisation och autentisering. Innan användare får tillgång till administrativ funktionalitet kräver webbapplikationen att man autentiserar sig genom Optimizelys Identity-baserade lösning.

Attributet [Authorize] användes i alla controlleractions som hanterar administrativa resurser. Detta gör att ASP.NET Core utför en kontroll av autentiserad och giltig session som måste finnas innan en administrativ åtgärd får utföras. Vid fall av försök att nå en skyddad resurs utan tillhörande åtkomst för kontot omdirigeras användaren till sidan för inloggning.

Ytterligare implementerades även rollbaserad åtkomstkontroll för separering mellan olika typer av användare. För testning skapades endast två typer av användare, en med åtkomst till webbapplikationens funktionaliteter som dashboard för att hantera projekt och tasks samt en med all möjlig åtkomst till CMS funktionaliteter utöver sidans normala funktionaliteter. Första kontot som skapades hade automatiskt åtkomst till allt för att kunna sätta åtkomsten för andra konton via CMS gränssnittet som man kom åt via URL tillägget /episerver/cms. De olika CMS funktioner som man kan tillge konton är att skapa, ändra, radera, publicera och administrera och om ett konto bara ska ha åtkomst till sidans användargränssnitt utan CMS gränssnittet bockas endast rutan för läsbehörighet. För att kunna ändra åtkomst för sitt eget och andras konton behöver rutan “Administer” vara fylld (se Fig. 2).



Figur 2: Bild tagen från den tekniska lösningens CMS gränssnitt där olika användargrupper tilldelas åtkomst till CMS funktionaliteter samt en ruta (read) för åtkomst till webbapplikationens gränssnitt.

Utöver den generella autentiseringen via Identity.Application implementerades även projektmedlemskaps autentisering. Ett exempel är SignalR-hubben för realtidskommunikation som även kräver att användaren är medlem i det projektet som anslutningen avser. Det görs med att varje användare har en identifierare som skickas via headern till API:t från webbapplikationen och därefter använder controllern värdet för att kontrollera om användaren är en medlem i projektet. Denna säkerhetsmekanism motverkar att användare ska få insyn till projekt de inte tillhör.

4.4 Sessionshantering och säker utloggning

Webbapplikationen använder Identity.Application för en cookie baserad autentisering. Autentisering cookien som identifierar användaren skapas vid en lyckad inloggning men under utveckling upptäcktes ett problem med utloggningsflödet. Efter utloggning kunde man fortfarande via korrekt URL nå samma inloggnings session. Detta tydde på att sessionen inte avslutades korrekt utan i stället omdirigerades användaren bara visuellt till inloggningssidan. Lösningen var att en utloggnings controller implementerades som anropar SignOutAsync(”Identity.Application”). Den gjorde att autentiseringscookien upphävdes vid utloggning och sessionen avslutades korrekt, därefter omdirigeras användaren till inloggningssidan precis som tidigare.

4.5 Server-side proxy mellan frontend och backend-API

I den första implementeringen kommunicerade frontend direkt med API:t via HTTP förfrågningar med nyckeln inkluderad i headern. Den medförda API-nyckeln blev då synlig i webbläsarens DevTools. Trots att HTTPS krypterar kommunikationen påverkar det inte när förfrågan ankommit till frontend och därmed upptäcktes en stor säkerhetsrisk.

Server-side proxyn som löste problemet infördes i Optimizely applikationen under sökväg /dashboard-api/projects. Den fungerar som en mellanhand mellan klient och API, frontend skickar då anrop till webbapplikationens same-origin-endpoint. Proxyn kontrollerar sedan att

förfrågan kom från en autentiserad användare och därefter hämtar API-nyckeln från datorns miljövariabler. Vidare hämtas sessionens aktuella användares identifierare och proxyn skickar vidare anropet till API:t. Det anropet har med sig två viktiga headers, X-API-Key som har API-nyckeln samt X-USER-ID som innehåller identifierare för användaren som är inloggad (se Fig. 3). Efter förbättrad implementering verifierades det att API-nyckeln inte längre syntes i webbläsarens DevTools.

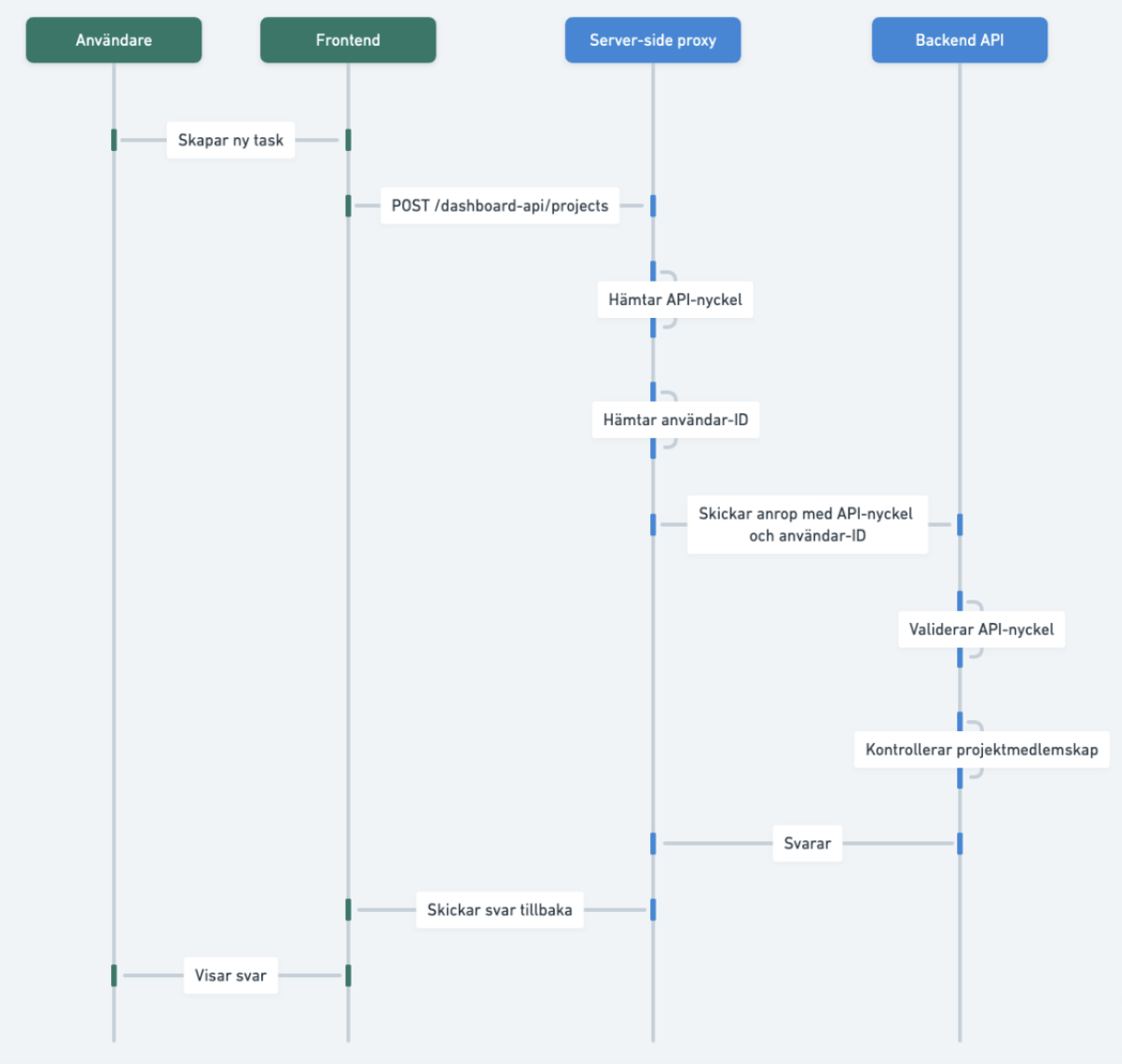
4.6 API-autentisering med API-nyckel

API:t skyddas främst av en API-nyckel som fungerar som en gemensam hemlighet mellan Optimizely applikationen och backend-API:t, där den via proxyn skickas med i HTTP headern X-API-Key. Efter rate limiting i API:ts behandlingskedja (se Fig. 4) implementerades ett centralt middleware som granskar de anrop som skickas till API:t och försöker läsa nyckeln från förfrågans header och jämför sedan den med den korrekta nyckeln som hämtas från datorns miljövariabler. Denna jämförelse görs med metoden CryptographicOperations.FixedTimeEquals() för att minska risken för timing attacker där angripare utnyttjar små skillnader i svarstid för att hitta korrekt nyckel. Vid saknad eller inkorrekt nyckel i headern svarar API:t med HTTP statuskod 401 (Unauthorized) och det sker ett stopp i behandlingskedjan innan resten av API:t kan nås och vid korrekt nyckel skickas förfrågan vidare till API-controllrar.

För att minska risk för versionshantering (exempelvis det hamnar på github), exponering (exempelvis via kodgranskning) lagras nyckeln i datorns miljövariabler i stället för direkt i koden.

4.7 Auktorisation med användarkontext och projektmedlemskap

För att användare ska få tillgång till deras projekts resurser implementeras en auktorisationsmodell med användarautentisering och projektmedlemskap. Optimizely applikationens proxy läser användarens identifierare för sessionen och skickar vidare värdet i X-USER-ID headern. Därefter använder API:t identifieraren för att avgöra åtkomsten till resurser, detta sker vid varje anrop innan användaren får se eller ändra en specifik grupps resurser, se Fig. 4 för projektmedlemskap kontrollens position i API behandlingskedjan vid inkommande anrop. Kontrollen implementerades genom en metod som verifierar medlemskap utifrån projekt- och användar-ID och vid fall av försökt intrång till projekt utan medlemskap via exempelvis korrekt URL skickas en HTTP statuskod 403 (Forbidden).



Figur 3: Sekvensdiagram som beskriver relationerna i ordning för skapandet av en ny task, hur det relaterar till proxyn i Optimizely servern samt därefter backend-API:t. Förklarar relationer mellan avsnitt 4.4-4.6.

4.8 Inputvalidering av inkommande data

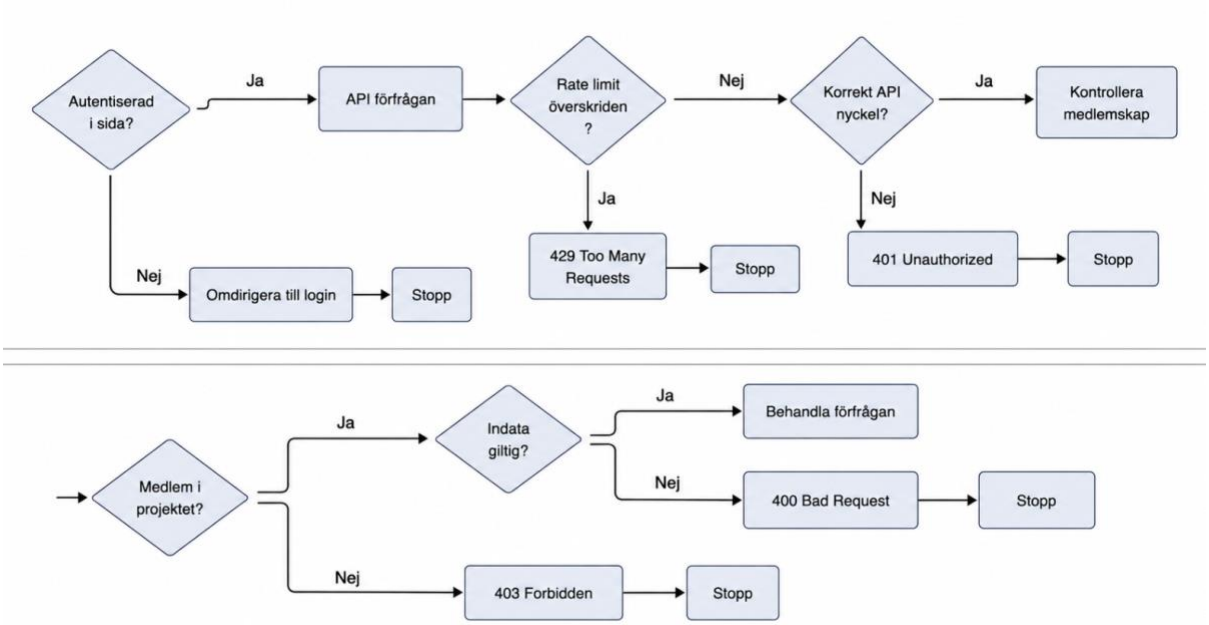
Inputvalidering implementerades sist i API behandlingskedjan (se figur 4) i backend-API:t för skydd från användarfel och ogiltiga eller skadliga indata med byggda valideringsregler via biblioteket FluentValidation. Valideringsregler togs fram för bland annat projekt, tasks, projektmedlemmar. Dessa objekt är med reglerna begränsade till exempelvis att textfält befinner sig inom längdgränser som är rimliga, att obligatoriska fält finns, enum värden är giltiga samt att numeriska värden ligger inom ett intervall som är tillåtet. Vidare finns även validering för giltigt e-postadress format i relevanta fält.

När ett skrivande anrop skickas till API sker valideringen efter anropets tillhörande JSON deserialiserats men innan databasen kan läsas eller manipuleras. Om valideringen inte lyckas svarar API:t med HTTP statuskod 400 (Bad Request) med tillhörande information om vilka fält som hade felaktigt input.

4.9 Rate limiting

En annan säkerhetsmekanism som implementerades i API:t var rate limiting som begränsar antal anrop en klient får göra under en bestämd tid i syfte att motverka överbelastning och vissa typer av automatiserade attacker. Den infördes i form av en middleware som placeras först i API behandlingskedjan (se figur 4) vilket medför även anrop med API nyckelautentisering också ingår i begränsningen. Begränsningskonfigurationen i systemet sattes till femtio API-anrop per minut för en IP adress. Vid första överskridning av gränsen skapas en fördröjning av 60 sekunder där API-anrop blockeras från IP adressen. När tiden passerat, om gränsen överskrids igen ökar fördröjningen till 120 sekunder osv. Vid varje gränsöverskridande svarar API:t med en HTTP statuskod 429 (Too Many Requests).

För att beskriva hur åtkomstkontrollen sker i behandlingskedjan när API:t får en förfrågan från webbapplikationen används följande flödesschema:



Figur 4: Flödesschemat sammanfattar säkerhetskontroller i behandlingskedjan från autentisering i webbapplikationen till rate limiting, API-nyckel validering, projektmedlemskap, inputvalidering innan API-anropet behandlas. Figuren är uppdelad i två för klarhet, efter ”Kontrollera medlemskap” ska pilen rikta mot ”Medlem i projektet?”.

4.10 Audit logging

I mål av att göra systemet spårbart och skapa möjligheten för säkerhetsrelaterad uppföljning vid misstänkt aktivitet implementerades audit logging i backend-API:t. Där viktiga operationer loggas i systemet med relevant metadata. Operationer det innefattar är bland annat att skapa, uppdatera och ta bort projekt likaså tasks samt även autentiseringsförsök som misslyckats. Den data som de loggade händelserna relaterar till och sparas omfattar associerad användare, typ av operation, IP adress involverad, när det skett samt tillstånd av lyckande eller misslyckande med tillhörande orsak. Audit logging skyddar säkerheten i systemet via spårbarhet och skapandet av möjligheten till analys för misstänkt aktivitet i efterhand.

4.11 CORS och CSP

Backend-API:t har en CORS-policy som begränsar anrop så de bara kan komma från specifika origins som behövs för utvecklingsmiljön. Det innefattar Optimizely applikationen och en lokal frontendmiljö. Policyn gör då att anrop till API:t från alla andra origins blockeras. Säkerhetsmekanismen fungerar som ett kompletterande lager för otillåtna origin anrop då det begränsas till origins på nätverket, direkta anrop med verktyg som curl tillåts fortfarande.

Bortsett från CORS implementerades säkerhetsheadern Content-Security-Policy (CSP) för att motverka cross-site scripting (XSS) som fungerar genom att servern genererar en unik token vid varje förfrågan. Den placeras sedan i CSP headern och på tillåtna script taggar för att blockera injicerad skadlig kod. Token som används i CSP sammanhanget är en nonce (“number used once”). CSP skyddar inte kommunikationen till API:t direkt men den medför till bättre allmänt skydd av API:t då sidan är starkt bunden till API kopplingen. Om webbapplikationens säkerhet komprometteras ökar risken för intrång till API:t indirekt.

4.12 Testning utifrån ett angriparperspektiv

En del av projektet är att göra testning från ett angriparperspektiv för att hitta potentiella brister. Testningen fokuserade på försök att kringgå, observera eller manipulera skydden som systemet har implementerat. De verktyg som användes var powershells curl.exe för simulering av HTTP anrop direkt på datorn. Även verktyget Devtools för inspektering av anrop, headers och nätverkstrafik i webbläsaren. Testning av sessionshantering skedde med inkognitofönster för att ingen påverkan ska ske av tidigare cookies. Sedan användes webbfelsökningsverktyget Fiddler Classic för analys av HTTPS trafik i lokal miljö.

Verktygen valdes för att kontrollera exponering i webbläsaren, direkta API anrop och inspektera lokal trafik mellan systemen på serversidan. Testningen avgränsades till manuella kontroller med verktygen och omfattade inte penetrationstestning, automatiserad sårbarhetsskanning eller produktionsmiljö.

Totalt omfattade testsviten 18 manuella testfall som sammanfattas i Tabell 1. Testerna konfirmerade främst säkerhetsmekanismernas funktionalitet. De täckte autentisering, auktorisation, inputvalidering, rate limiting, nyckel exponering, sessionshantering, CORS och CSP.

Tabell 1. Sammanfattning av testsvitens omfattning

Testkategori	**Antal testfall**	**Verktyg**	**Förväntat resultat**	**Täckning**
API-nyckel saknas/felaktig	2	curl	401 Unauthorized	Grundläggande API-autentisering
Korrekt API-nyckel men fel projektmedlemskap	1	curl	403 Forbidden	Projektbaserad auktorisation

Testkategori	Antal testfall	Verktyg	Förväntat resultat	Täckning
Inputvalidering	3	curl	400 Bad Request	Ogiltig JSON, saknade fält samt felaktiga enum- och längdvärden
Rate limiting	3	curl	429 Too Many Requests	Gräns på 50 anrop/minut och ökande fördröjning
API-nyckelexponering i frontend	2	DevTools	Nyckel synlig före proxy, ej synlig efter proxy	Klientexponering av API-nyckel
Sessionshantering efter utloggning	3	Inkognitoläge i webbläsare	Omdirigering till inloggningssida	Manipulation av URL efter utloggning
CORS	1	Testsida	Blockering	API anrop från otillåten origin
CSP	3	DevTools och razor testfil	Blockering	Scriptinjektionsattacker

API-anrop till API endpoints med saknad eller inkorrekt API-nyckel gav svar HTTP 401 (Unauthorized) som verifierade att middleware blockar anrop utan korrekt nyckel innan ankomst till resten av API:t. Vid anrop till projekt med korrekt nyckel men saknad medlemskap svarade API:t med HTTP 403 (Forbidden) vilket visar på att auktorisationsmodellen med användarspecifik autentisering fungerade korrekt.

Inputvalidering testades med JSON som är felaktig eller ofullständig gav HTTP 400 (Bad Request), samma svar dök även upp vid användning av fältvärden som inte är tillåtna. Ytterligare gjordes även testning för rate limiting genom att från samma IP skicka många anrop på en kort tid som överskred den satta gränsen av femtio anrop per minut och fick HTTP 429 (Too Many Requests) samt fördröjning på 60 sekunder där API-anrop inte utfördes. Upprepade testet efter fördröjningen och konfirmerande den ökande fördröjningen vid varje begränsningsöverskridande.

Som förklarat i avsnitt 4.4 upptäcktes problemet med API-nyckelexponeringen i webbläsaren som sedan löstes och DevTools användes både för att upptäcka problemet och verifiera att det var löst. Efter implementeringen av server-side proxy lösningen var endast same-origin anrop synliga i DevTools, utan API-nyckel exponerad.

Med Fiddler inspekterades lokal HTTPS trafik mellan webbapplikationens proxy och backend-API:t. Det visade att nyckeln var synlig i server till server kommunikationen men inte i klientens egen trafik. Detta medför att trafiken skyddas från passiv nätverkslyssning med HTTPS protokollet men i maskinnära miljö kan innehåll fortfarande inspekteras.

Testning av sessionshantering skedde med utloggning och sedan manipulering av URL för att nå projektsidan igen men det resulterade i omdirigering till inloggningssidan. Samma test gjordes på webbapplikationens alla skyddade resurser med samma resulterande beteende. Det bekräftade korrekt avslutad session.

En testsida togs fram för testning av CORS-policyn via anrop mot API:t från en sida som inte är inkluderad i tillåtna origin. Anropet blockerades trots medföljd korrekt API-nyckel vilket visar på korrekt implementation av policyn som förhindrar obehöriga origins från att nå API:t, även om nyckeln är korrekt.

CSP implementeringen testades också genom försök att injicera och exekvera script som inte har korrekt nonce i en razor fil som renderas på webbapplikationen. Testningen utfördes med tre olika injektionsmetoder vilka inkluderar inline JavaScript (<script>alert(1)</script>), event handler skript (onclick="alert(1)") och CSS style skript (test). Dessa lades i en fil som renderas på webbapplikationens sidor. Blockeringen observerades i DevTools Console för inline styles och javascript. Event handler skriptets blockering skedde först vid knapptryck.

5. Resultat och analys

Detta kapitel presenterar arbetets resultat genom att besvara de tre frågeställningarna samt tillhörande analys i relation till etablerade säkerhets principer och rekommendationer.

5.1 Hur säkerställer den utvecklade lösningen i Optimizely att administrativa resurser endast nås av behöriga användare, samt att åtkomst upphör efter utloggning?

Den utvecklade lösningen säkerställer att administrativa resurser endast är tillgängliga för behöriga användare genom en kombination av autentisering och rollbaserad åtkomstkontroll. Den rollbaserade åtkomstkontrollen om används fungerar genom att koppla behörigheter till utsedda roller. Behörigheterna som tilldelas innefattar administrativa funktionaliteter samt möjlighet att se och ändra resurser. Vid otillräcklig behörighet nekas åtkomst till skyddade resurser vilket inkluderar åtkomst till CMS gränssnittets olika delar. Rollbaserad åtkomstkontroll är något som redan fanns i Optimizelys CMS gränssnitt. Dock för webbapplikationens autentisering användes ASP.NET Cores [Authorize] attributet i controllers och vid handlingar som berör skyddad funktionalitet vilket verkställer krav av autentisering innan åtkomst.

En controller för utloggning implementerades med SignOutAsync(“Identity.Application”) metoden så autentiseringscookien skulle ogiltigförklaras på serversidan vid utloggning (se avsnitt 4.3). Detta gjordes för att avsluta sessionen ordentligt så ingen åtkomst till administratör webbapplikationen ska finnas kvar via URL manipulation. Testning konfirmerade sessionens korrekta avslutning och efter omdirigering till inloggningssidan fanns inte möjlighet att komma åt sessionen utan ytterligare en inloggning.

5.2 Hur kan kommunikationen mellan en Optimizely-baserad administrativ webbapplikation och ett backend-API skyddas mot obehöriga anrop?

En flerlagersarkitektur har implementerats för skydd av Optimizely applikationen och backend-API:ts mellanliggande kommunikation. Den innefattar en kombination av API autentisering, användarspecifik auktorisation samt en minskad hemlighets exponering i klientmiljön.

Mellan klient och backend-API:t finns en server-side proxy (se avsnitt 4.4) som tar emot anropet från frontend, lägger på unikt användar-id som sessionen tillhör och lägger även på API-nyckeln. Denna lösning gör att hemligheterna behålls i serversidan av webbapplikationen utan risk för exponering i frontend. Proxyn minskar attackarean genom att begränsa nyckeln till servermiljön men en lyckad attack kan trots det nå korrekt nyckel via miljövariabel avläsning.

API:ts autentisering sker i en centraliserad middleware (se avsnitt 4.5). Anrop som ankommer till API:t måste genomgå nyckelvalidering i middleware innan åtkomst till andra delar som controller eller affärslogik. API-nyckeln som ankommit via proxyn jämförs med en lagrad korrekt nyckel med CryptographicOperations.FixedTimeEquals(). Metoden gör alltså

att jämförelsen tar lika lång tid varje gång för att skapa en minimerad risk att timing attacker sker, där en angripare använder små svarstids skillnader för att dra slutsatser om korrekt nyckel. Vid avsaknad eller fel nyckel svarar API:t med HTTP 401 (Unauthorized).

Ytterligare sker också användarspecifik auktorisation. Användar-id som proxyn skickat med i anropet används av API:t (se avsnitt 4.6) för att kontrollera projektmedlemskap innan data från specifika projekt kan synas eller ändras. Vid försök att nå projekt utan tillhörande medlemskap svarar API:t med HTTP 403 (Forbidden).

Innan exekvering av API:ts affärslogik sker säkerhetsmekanismen att validera inkommande data (se avsnitt 4.7), som motverkar att skadliga eller ogiltiga data kommer till sensitiva delar av systemet. I syfte att motverka en användare från att belasta API:t med många automatiserade anrop implementerades även rate limiting med satt gräns på femtio anrop/min per IP adress med ökande fördröjningar där API-anrop inte lyckades (se avsnitt 4.8) vilket försvårar brute force attacker där det görs automatiserade API-anrop med olika nycklar för att nå korrekt nyckel.

5.3 I vilken utsträckning uppfyller den utvecklade lösningen etablerade säkerhets principer, som minsta privilegieprincipen och OWASP:s rekommendationer kring åtkomstkontroll?

Den utvecklade lösningen uppfyller minsta privilegieprincipen på flera nivåer av systemet. Först begränsas behörigheter via roller i Optimizely och användare blir endast tilldelade de roller som krävs för att de ska kunna göra sitt jobb. I CMS miljö begränsas funktionaliteter i olika nivåer till roller och de rollerna tilldelas sedan bara användare med behov och tillåtelse. Normala användare som har registrerats av en redaktör i CMS gränssnittet har tillgång till webbapplikationens resurser exkluderat CMS gränssnitts åtkomst.

Nästa nivå är begränsningen av åtkomst via projektmedlemskap som skapar åtkomstkontroll på ytterligare en nivå. Åtkomst till specifika projekt kan endast åtkommas vid tillhörande medlemskap. På tredje nivån begränsas vissa känsliga uppgifters åtkomst som API-nyckeln och användar-id till webbapplikationens server vilket minskar attackarea. Sammanfattningsvis bidrar detta till att användare och systemdelar tilldelas endast åtkomst baserat på behov för dess funktion vilket följer principen minsta privilegieprincipen.

Lösningen följer OWASP:s rekommendationer för att motverka bruten åtkomstkontroll. De rekommenderar att följa minsta privilegieprincip, ovan beskrivs hur det relaterar till den tekniska lösningen. Försök att kringgå åtkomstkontroll via manipulation av URL eller direkta anrop görs svårare med backend-API:ts säkerhetskontroller samt proxy lösningen. Medlemskapsverifieringen i API:t gör att åtkomst till andra användares resurser i projekt nekas utan korrekt projektmedlemskap. API-anrops otillräckliga auktorisation motverkas via införandet av kontroll för både applikations- samt användaridentitet.

Testningen indikerar att CORS-policyn fungerar som den ska i den lokala miljön. En testsida som inte ingår i CORS-policy för tillåtna origin anropade API med korrekt nyckel som misslyckades med en CORS blockering. Korrekt sessionshantering är implementerad i webbapplikationen för att minska risk för åtkomst efter utloggning.

Deny by default principen följs via middleware i API som nekar alla anrop utan korrekt API-nyckel, säkerhetsheaders för minskning av angreppsytor till frontend och rate limiting där alla API-anrop nekas i en period efter begränsöverskridningen.

Placeringen av säkerhetsmekanismerna som implementerats i olika lager av API:t och webbapplikationen följer principen om defence in depth där flera oberoende skyddslager minimerar den risk som finns för enskilda svaghet.

Vidare finns även begränsningar, API-nyckeln är en symmetrisk modell med samma hemlighet mellan båda system. Det medför större attackarea då nyckeln kan hittas från båda system. En asymetrisk lösning hade minimerat risken med nyckelseparation. Ytterliggare en begränsning är att CORS-policyn bara gäller webbläsarbaserade anrop till API, direkta anrop via verktyg som curl.exe motverkas inte.

6. Slutsats

Detta kapitel presenterar arbetets slutsats, en reflektion kring svagheter och styrkor samt vad som potentiellt kan utvecklas med arbetet.

6.1 Slutsatsen

Inom examensarbetets omfattning har det utförts en undersökning av hur en administrativ webbapplikation i Optimizely kan kopplas till ett .NET utvecklat backend-API på ett sätt som stärker säkerhet i sammanhang av åtkomstkontroll och API-kommunikation.

Åtkommet resultat indikerar att Optimizelys rollbaserade åtkomstkontroll tillsammans med korrekt sessionshantering skyddar administrativa resurser och säkerställer upphörande av åtkomst efter utloggning. Det visar också att kommunikationen mellan webbapplikationen och backend-API:t skyddas väl med flera skyddsmekanismer. Detta inkluderar en kombination av server-side proxy, API-nyckelautentisering, auktorisering via projektmedlemskap, inputvalidering, audit logging, rate limiting och korrekt sessionshantering. I kombination skyddar dessa mekanismer mot obehöriga anrop.

Den utvecklade lösningen förhåller sig i hög grad till etablerade säkerhetsprinciper som least privilege samt flera OWASP rekommendationer mot bruten åtkomstkontroll. Däremot finns det begränsningar när det kommer till bland annat att autentiseringen som sker system till system har en symmetrisk delad hemlighet.

Arbetets övergripande slutsats är därför att den utvecklade lösningen är uppbyggd med en säkerhetsmotiverad modell som fungerar för administrativa webbapplikationer med Optimizely CMS.

6.2 Styrkor i den utvecklade lösningen

En styrka i den utvecklade lösningen är att flera oberoende kontroller fungerar i kombination för tillgång till API:t. Anropet måste komma från tillåten origin, en användare med rätt medlemskap i projektet, inkludera giltig och korrekt formulerade data, API-nyckel samt anropen måste även förhålla sig till satt anropsfrekvens. Det skapar ett system med skyddsmekanismer i flera lager som tillsammans jobbar för ett mer robust skydd.

En annan styrka är att systemet inte bara arbetar förebyggande mot attacker. Implementationen av audit loggning gör att systemet får en mekanism för att i efterhand analysera misstänksam aktivitet. Det betyder att den utvecklade lösningen har förebyggande och uppföljande egenskaper vid attackscenarion med obehöriga anrop till API:t.

Ytterligare en styrka är designvalet som gjordes om skapanden av konton. Möjligheten att skapa nya konton är begränsat till en administratör med rätt åtkomst i CMS gränssnittet. Detta val medför en minskning på attackarean då användaren först granskas av någon med tillräcklig behörighet innan kontot skapas. När inte användaren kan skapa sitt eget konto minskas sannolikheten för höjning av behörighetsnivå oauktoriserat.

6.3 Svagheter och begränsningar

Bortsätt från arbetets styrkor finns även tydliga begränsningar. Främst att kommunikationen som sker mellan Optimizely webbapplikationen och backend-API:t är baserat på en symmetrisk modell. De delar alltså en gemensam API-nyckel vilket gör implementationen enkel men också riskabel ur en säkerhetsaspekt. Det ökar attackarean då nyckeln finns i båda systemen. En mer avancerad autentisering som exempelvis ett asymmetriskt system hade ökat säkerhet men också utgjort en högre komplexitet och längre utvecklingstid, därför bör en övervägning göras. Då backend-API;t i den utvecklade lösningen hanterar administrativa resurser som vid läckage skulle göra skada är säkerhetsdesignförslaget väl grundat.

En annan svaghet är att allt arbete skett i en lokal utvecklingsmiljö. Det gjorde att den säkerhetsmässiga analysen hade hårda begränsningar. Bara attacker från samma maskin kunde simuleras men normalt sätt sker de från andra. Vidare innefattade testningen inte en professionell penetrationstestning eller full säkerhetsgranskning. De tester som gjordes verifierade främst säkerhetsmekanismernas funktionalitet där exempelvis projektmedlemskapets kontroll innan åtkomst. Testerna var därmed inte heltäckande och omfattande då de bland annat inte inkluderade belastningstestning, statisk kodanalys, automatiserad sårbarhetsskanning. Den minimerade testningen inom detta arbete skedde av orsaker som tidsbrist samt begränsningen till en lokal miljö. En slutsats som kan dras med detta till grund är att den utvecklade lösningen inte kan generaliseras till att vara tillräckligt skyddad utanför detta arbetes avgränsningar.

Något annat som också kan uppfattas som en svaghet är att server-side proxyn är en central del av säkerhetsmodellen. Viktig information som API-nyckel, användarens identifierare och anropet som kommer från frontend med projekt-id passerar proxyn innan ankomst till backend-API:t. Vid en lyckad attack på proxyn eller servern där proxyn befinner sig så skulle en angripare kunna få API-nyckel, användar-id samt projekt där användaren är medlem som behövs för att passera både autentisering och auktorisation kontrollen i API:t. I det scenariot hade angriparen kunnat skicka ett förfalskat anrop till API:t som ser legimt ut och få åtkomst till känslig data via API:t. CORS skyddar från att anrop skickas från andra webbplatser än de tillåtna men i testningen sågs att skyddet påverkade inte API anrop med curl terminalverktyget. Därmed blir attackscenariot realistiskt och proxyn måste skyddas väl. Ett alternativ för att minska risken hade varit att lägga på anti-forgery token på alla tillståndsändrande händelser för att motverka förfalskade anrop (CSRF).

6.4 Reflektion kring säkerhetsmekanismernas olika nivåer

Säkerhetsmekanismerna kan klassas utifrån funktion i säkerhetsmodellen. Autentisering, rollbaserad åtkomstkontroll, API-nyckelvalidering och projektmedlemskap hör till ett förebyggande lager som förhindrar obehöriga anrop innan systemet påverkas. För att minska skadeinverkan av skadliga, felaktiga eller automatiserade anrop finns mekanismer i ett annat lager, inputvalidering och rate limiting. Sedan efter flera förebyggande skydd av olika typer av angrepp finns också en mekanism implementerad för möjligheten till uppföljning vid misstänksam aktivitet. Audit loggning fungerar då i ett tredje lager för att analysera efter ett angrepp skett för att motverka framtida attacker. Dessa tre lager av säkerhetsmodellen gör den mer heltäckande för olika typer av angrepp.

Den utvecklade lösningen har olika mekanismer som ibland överlappar i skydd mot vissa angrepp och det skapar defence in depth där man inte ska förlita sig på en säkerhetsmekanism för att skydda mot viss typ av angrepp. För bruten åtkomstkontroll finns flera mekanismer som API-nyckel och om nyckeln skulle läcka så kontrolleras också projektmedlemskap med användaridentifieraren. Vidare om båda skulle läcka så finns CORS implementerat för att se till så förfrågan till API inte kan komma från otillåten webbplats, men möjligheten att skicka anrop med terminalverktyg curl finns fortfarande. Sedan så kan andra mekanismer kopplas på som förstärkande skydd på de primära skyddsmekanismerna. Rate limiting skyddar mot automatiserade attacker som kan användas för att lista ut API-nyckel, inputvalidering finns för att validera inkommande data vid fall av lyckat obehörigt anrop samt audit loggning för att analysera efter ett angrepp. Vid fall att en av dessa mekanismer skulle fallera så finns de andra kvar och skyddar, därmed behöver en angripare ta sig förbi flera lager av säkerhet och därmed minskas sannolikheten för intrång och även konsekvenser av intrång med inputvalidering och audit loggning.

6.5 Samhälleliga, etiska och ekologiska aspekter

Ur samhälleliga aspekten så utgör arbetet en minskad risk för manipulation av interna system då administrativa systems säkerhet undersöks och därmed identifieras säkerhetsbrister. Ytterligare en samhällelig betydelse är att förtroende för digitala system ökar när åtkomstkontroll och säker API kommunikation hanteras ansvarsfullt. Vidare bidrar den ansvarsfulla hanteringen av administrativa funktionaliteter och minimerade risken för obehörig åtkomst till en etisk fördel. Ur ett ekologiskt perspektiv finns begränsningar i detta arbete då det ej utförts i produktionsmiljö och därmed ingen miljöpåverkan undersökts. Inom avgränsningarna kan endast resursanvändning vid utveckling påpekas. Detta avser testning och drift av systemet som till exempel elförbrukning för datorer.

6.6 Metodreflektion

Den planerade metodens iterativa karaktär visade sig vara effektiv. Problem som exempelvis API-nyckel exponeringen i webbläsarmiljön upptäcktes efter testad implementation som sedan reviderades. Den iterativa utvecklingsmodellen bidrog till att utveckla en nerskalad version som sedan utvecklades vidare utefter uppkomna problem och idéer. Nackdelen var dock att planen inte blev tillräckligt utförlig. Utvecklings formatet byggde på att skala upp men det gjorde att första planen inte fullt ut räknade med framtida utveckling. Exempelvis hårdkodades ToDo, InProgress och Done i koden vilket skulle göra det omständligt att lägga till fler tillstånd som tasks kan befinna sig i.

Utvecklingsmodellen skapade även avvikelse från den planerade tidfördelningen för systemets delar (se appendix A). Om man jämför med den verkliga tidfördelningen (se appendix B) märks det tydligt att delarna tog längre tid än planerat. Under iterationerna i arbetet hittades problem och uppstod idéer som löstes respektive implementerades. Vid en mer utförlig planering hade många problem undvikits samt idéer planerats utförligare för att effektivisera arbetet.

Om arbetet hade genomförts igen hade en mer utförlig planering varit fördelaktig. Vidare hade införandet av en tydligt formulerad hotmodell i planeringsfasen också förstärkt metoden. Inom detta arbete var hotmodellen formulerad i samband med införandet av varje säkerhetsmekanism och därav blev testningen mer en funktionalitetsbekräftelse än en

angriparsimulering. Införandet av en tydligt formulerad hotmodell i planeringsfasen hade gjort testningen mer genomtänkt och heltäckande.

6.7 Vidareutveckling

Arbetets omfattning visar på flera utvecklingsområden. En är att ersätta eller komplettera den symmetriska modellen för API-nyckeln med en asymmetrisk modell eller användarspecifika tokens. Sedan finns också möjligheten att vidareutveckla behörighetsmodellen för projekt så att olika medlemmar kan ha olika behörigheter inom projektet. Exempelvis genom att bara ägaren kan bjuda in fler medlemmar och ta bort projektet. Vid möjlighet att införa lösningen i en produktionsmiljö hade fler säkerhetstester kunnat ske över nätverk från andra maskiner. Detta hade skapat en realistisk analys av systemets säkerhet med verkliga angreppsscenarion. Vidare hade en mer heltäckande testning i allmänhet gjort arbetet mer generaliserbart säkert.

Vid vidareutveckling hade också audit loggningen kunnat utvecklas till att notifiera lämpliga personer vid misstänksam aktivitet vilket hade möjliggjort försvar i realtid av attacken där exempelvis ett konto kan avaktiveras tillfälligt vid upptäckt misstänksam aktivitet.

Litteraturförteckning

[1] Precio Fishbone AB, ”Vi är Precio Fishbone”. [Online]. Tillgänglig: https://www.preciofishbone.se/vilka-vi-ar/vi-ar-precio-fishbone/. [Hämtad: 23 jan. 2026]

[2] Precio Fishbone AB, ”Optimizely”. [Online]. Tillgänglig: https://www.preciofishbone.se/vad-vi-gor/produkter/optimizely/. [Hämtad: 23 jan. 2026].

[3] CMS.se, ”Vad är ett CMS?”. [Online]. Tillgänglig: https://cms.se/cms/. [Hämtad: 23 jan. 2026].

[4] Optimizely, ”Content Management”. [Online]. Tillgänglig: https://www.optimizely.com/sv/products/content-management/. [Hämtad: 23 jan. 2026].

[5] OWASP Foundation, “A01:2021 – Broken Access Control”. [Online]. Tillgänglig: https://owasp.org/Top10/A01_2021-Broken_Access_Control/. [Hämtad: 23 jan. 2026].

[6] OWASP Foundation, “OWASP API Security Top 10”. [Online]. Tillgänglig: https://owasp.org/API-Security/. [Hämtad: 23 jan. 2026].

[7] D. F. Ferraiolo and D. R. Kuhn, “Role-Based Access Controls”, in Proceedings of the 15th National Computer Security Conference, Baltimore, MD, USA, Oct. 1992, pp. 554–563.

[8] OWASP Foundation, *“A01:2025 – Broken Access Control,”* OWASP Top 10, 2025. [Online].
Tillgänglig: https://owasp.org/Top10/2025/A01_2025-Broken_Access_Control/.
[Hämtad: 2 feb. 2026].

[9] Optimizely, “Access rights,” *Optimizely Content Cloud for Administrators (Webhelp)*. [Online]. Tillgänglig: https://webhelp.optimizely.com/latest/en/content-cloud-for-administrators/access-rights.htm. [Hämtad: 9 mar. 2026].

[10] Optimizely, “Virtual roles,” *Optimizely Developer Documentation - Content Management System*. [Online]. Tillgänglig: https://docs.developers.optimizely.com/content-management-system/docs/virtual-roles. [Hämtad: 9 mar. 2026].

[11] G. Jackson och M. Goodwin, *“What is an API key?”* IBM Think, IBM Corporation. [Online].
Tillgänglig: https://www.ibm.com/think/topics/api-key.
[Hämtad: 2 feb. 2026].

[12] S. A. Padhiar and K. H. Mori, “A Comparative Study on Symmetric and Asymmetric Key Encryption Techniques,” in *A Comparative Study on Symmetric and Asymmetric Key Encryption Techniques*, IGI Global, 2021, ch. 8. doi: 10.4018/978-1-7998-6988-7.ch008. [Online]. Tillgänglig:
https://www.researchgate.net/publication/354859313_A_Comparative_Study_on_Symmetric_and_Asymmetric_Key_Encryption_Techniques.
[Hämtad: 2 feb. 2026].

[13] P. Kotturu, S. P. Kammula, S. S. Bunga and P. S. Atluri, “Prevention of Session Hijacking and Authentication Providing to the Session Cookie,” *International Journal of Recent Technology and Engineering (IJRTE)*, vol. 8, no. 4, pp. 9685–9690, Nov. 2019. doi: 10.35940/ijrte.D9979.118419. [Online]. Tillgänglig: https://www.researchgate.net/publication/364088769_Prevention_of_Session_Hijacking_and_Authentication_Providingtothe_Session_Cookie. [Hämtad: 25 apr. 2026].

[14] B. Latha, M. Indumathi, C. Chitra and B. Gopinath, “Restriction of Forgery Attacks using AntiForgery Token in Machine Learning,” in *Proc. International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*, Chennai, India, 2021. doi: 10.1109/ICAECA52838.2021.9675710. [Online]. Tillgänglig: https://www.researchgate.net/publication/357310656_Restriction_of_Forgery_Attacks_using_AntiForgery_Token_in_Machine_Learning. [Hämtad: 25 apr. 2026].

[15] S. Subramanian and S. R. Vadivel, “Web Security: Same Origin Policy and its Exemptions,” *IJISET - International Journal of Innovative Science, Engineering & Technology*, vol. 2, no. 2, 2015. [Online]. Tillgänglig: https://www.researchgate.net/publication/383851994_Web_Security_Same_Origin_Policy_and_its_Exemptions. [Hämtad: 25 apr. 2026].

[16] L. G. Petkova, “Content Security Policy Validation,” presented at *Engineering. Technologies. Education. Security. 2019*, Veliko Tarnovo, Bulgaria, Jun. 2019. [Online]. Tillgänglig: https://www.researchgate.net/publication/340236439_CONTENT_SECURITY_POLICY_VALIDATION. [Hämtad: 25 apr. 2026].

[17] M. Manoharan, “API Rate Limiting Mechanisms in SaaS Applications: A Systematic Analysis of DDoS Protection Strategies,” *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 10, no. 6, pp. 1787–1798, Dec. 2024. doi: 10.32628/CSEIT241061223. [Online]. Tillgänglig: https://www.researchgate.net/publication/387234390_API_Rate_Limiting_Mechanisms_in_SaaS_Applications_A_Systematic_Analysis_of_DDoS_Protection_Strategies. [Hämtad: 25 apr. 2026].

[18] O. Söderström and E. Moradian, “Secure Audit Log Management,” *Procedia Computer Science*, vol. 22, pp. 1249–1258, 2013. [Online]. Tillgänglig: https://www.researchgate.net/publication/275067687_Secure_Audit_Log_Management. [Hämtad: 30 apr. 2026].

[19] L. Mada, “Understanding Backend for Frontend Architecture: Exploring Backend for Frontend (BFF) Architecture Across Platforms and its Dynamic Adaptations,” *Global Journal of Computer Science and Technology: C Software & Data Engineering*, vol. 25, no. 1, 2025. [Online]. Tillgänglig: https://www.researchgate.net/publication/396809894_Understanding_Backend_for_Frontend_Architecture_Exploring_Backend_for_Frontend_BFF_Architecture_Across_Platforms_and_its_Dynamic_Adaptations. [Hämtad: 30 apr. 2026].

Appendix

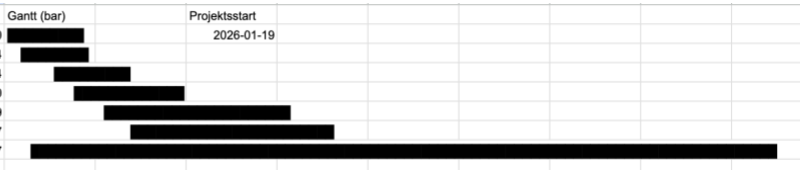
(A)

Aktivitet	Startdatum	Slutdatum	Dagar	Start-offset	Gantt (bar)	Projektsstart
Planering + litter	2026-01-19	2026-01-27	9	0		2026-01-19
Få upp Optimize	2026-01-23	2026-01-30	8	4		
Planera kod arch	2026-02-02	2026-02-10	9	14		
Bygga API	2026-02-08	2026-02-16	9	20		
Bygga webbsida	2026-02-17	2026-02-24	8	29		
Testning och säk	2026-02-25	2026-03-11	15	37		
Rapportskrivning	2026-01-26	2026-04-01	66	7		

Figur 5: Gantt schema för arbetets plan

(B)

Aktivitet	Startdatum	Slutdatum	Dagar	Start-offset	Gantt (bar)	Projektsstart
Planering + litteratur	2026-01-19	2026-01-27	9	0		2026-01-19
Få upp Optimizely	2026-01-23	2026-01-30	8	4		
Planera kod architektur	2026-02-02	2026-02-10	9	14		
Bygga API	2026-02-08	2026-02-20	13	20		
Bygga webbsidan	2026-02-17	2026-03-10	22	29		
Testning och säkerhetsanalys	2026-02-25	2026-03-20	24	37		
Rapportskrivning	2026-01-26	2026-04-23	88	7		



Figur 6: Gantt schema för verklig arbetsprocess

GÖTEBORGS
UNIVERSITET

CHALMERS