



Prestanda i Vue-komponenter

Ett experiment med lazy loading och eager loading mot
Wagtail CMS

Peter Landbris

URI: [urn:nbn:se:bth-27984](https://nbn-resolving.org/urn:nbn:se:bth-27984)

This thesis is submitted to the Faculty of Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Higher Education Diploma in Software Engineering. The thesis is equivalent to 10 weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author:

Peter Landbris

E-mail: pela23@student.bth.se

University advisor:

Lecturer Philip Berglund

Department of Computer Science

Faculty of Computer Science
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

In modern web development, optimized performance and user experience are crucial to meet user expectations. This study explores how different data loading strategies affect the performance of Vue components when communicating with a Wagtail CMS. The objectives are to investigate and compare the performance between two different strategies for content loading in Vue components against Wagtail's standardized API: eager loading and lazy loading. By analyzing these strategies, the study intends to provide guidance for developers in choosing the optimal loading strategy based on usage context. Through the design science methodology, two implementations, eager and lazy loading, of the same functionality in Vue.js against Wagtail CMS are designed and evaluated. Systematic measurements of initial loading time, filtering time, number of network requests, and transferred data volume are performed under standardized test conditions. The results show that lazy loading provides 40.8% faster initial page loading, while eager loading offers approximately 400 times faster filtering response. After two user interactions, eager loading becomes 21.6% faster overall. In conclusion, the choice between eager loading and lazy loading should be based on the application's usage context. For applications where users perform few interactions per session, lazy loading is recommended, while eager loading is superior for applications with intensive user interaction.

Keywords: Vue.js, lazy loading, eager loading, Wagtail CMS

Sammanfattning

I modern webbutveckling är optimerad prestanda och användarvänlighet avgörande för att möta användarnas förväntningar. Denna studie utforskar hur olika strategier för datainladdning påverkar prestandan i Vue-komponenter när de kommunicerar med ett Wagtail CMS. Syftet är att undersöka och jämföra prestandan mellan två olika strategier för innehållsladdning i Vue-komponenter mot Wagtails standardiserade API: eager loading och lazy loading. Genom att analysera dessa strategier ämnar studien ge vägledning för utvecklare i valet av optimal laddningsstrategi baserat på användningskontext. Genom design science-metodiken utformas och utvärderas två implementationer, eager och lazy loading, av samma funktionalitet i Vue.js mot Wagtail CMS. Systematiska mätningar av initial laddningstid, filtreringstid, antal nätverksanrop och överförd datamängd utförs under standardiserade testförhållanden. Resultaten visar att lazy loading ger 40,8% snabbare initial sidladdning, medan eager loading erbjuder cirka 400 gånger snabbare filtreringsrespons. Efter två användarinteraktioner blir eager loading 21,6% snabbare totalt. Slutsatsen är att valet mellan eager loading och lazy loading bör baseras på applikationens användningskontext. För applikationer där användare utför få interaktioner per session rekommenderas lazy loading, medan eager loading är överlägset för applikationer med intensiv användarinteraktion.

Nyckelord: Vue.js, lazy loading, eager loading, Wagtail CMS

Acknowledgments

A special thanks to Tristan Bridge, my supervisor at GRIDH.

Innehållsförteckning

Abstract	i
Sammanfattning	ii
Acknowledgments	iii
1 Introduktion	1
1.1 Bakgrund	1
1.2 Studiens omfattning	2
1.3 Översikt	2
2 Litteraturstudie	3
2.1 Laddningsstrategier och deras prestandapåverkan	4
2.2 Implementering av laddningsstrategier i olika miljöer	4
2.3 Metodologiska ansatser för utvärdering av laddningsstrategier	5
2.4 Sammanfattning och forskningsgap	6
3 Metod	7
3.1 Forskningsdesign	7
3.1.1 Design science-metodiken	7
3.2 Precisering av forskningsfrågor	8
3.3 Implementation av laddningsstrategier	9
3.3.1 Teknisk miljö	9
3.3.2 Eager loading-implementation	9
3.3.3 Lazy loading-implementation	10
3.3.4 Växlingsmekanism	11
3.4 Testmiljö och mätparametrar	12
3.4.1 Testmiljö	12
3.4.2 Mätparametrar	12
3.4.3 Datainsamlingsprocess	13
3.5 Testscenarier	14
3.5.1 Initial sidladdning	14
3.5.2 Filtrering efter titel	14
3.6 Analysmetoder	14
3.6.1 Kvantitativ analys	15
3.6.2 Visualisering	15

3.6.3	Kvalitativ analys	15
3.7	Metodimplementation	15
3.7.1	Kodinstrumentation	15
3.7.2	Strukturerad datainsamling	16
3.7.3	Praktiskt genomförande	16
3.8	Metodkritik	16
3.8.1	Validitet	16
3.8.2	Reliabilitet	17
3.8.3	Begränsningar	18
4	Resultat och analys	19
4.1	Sammanfattning av resultat	19
4.1.1	Initial laddningstid	19
4.1.2	Filtreringstid	19
4.1.3	Nätverksanrop	20
4.1.4	Dataöverföring	20
4.2	Initial laddningstid	20
4.3	Filtreringstid	21
4.4	Nätverksanvändning	22
4.5	Kombinerad prestandaanalys	23
4.6	Kod- och implementationskomplexitet	24
4.7	Sammanfattning av resultat i relation till forskningsfrågorna	25
4.7.1	RQ1: Strukturering av Vue-komponenter för innehållsladdning	25
4.7.2	RQ2: Prestandapåverkan vid val av laddningsstrategi	25
4.8	Resultat i design science-kontext	26
5	Diskussion	27
5.1	Metoddiskussion	27
5.1.1	Val av forskningsdesign	27
5.1.2	Val av testmiljö och datainsamling	28
5.1.3	Val av mätparametrar	28
5.2	Resultatdiskussion	28
5.2.1	RQ1: Strukturering av Vue-komponenter för innehållsladdning	28
5.2.2	RQ2: Prestandapåverkan vid val av laddningsstrategi	29
5.2.3	Jämförelse med tidigare forskning	30
5.3	Implikationer för praktiken	31
5.4	Begränsningar och hot mot validitet	31
5.4.1	Intern validitet	31
5.4.2	Extern validitet	32
5.4.3	Reliabilitet	32
5.5	Etiska överväganden	32
6	Slutsatser och framtida arbete	33
6.1	Övergripande bidrag	33
6.2	Slutsatser från prestandajämförelsen	34
6.2.1	Initial laddningstid och användarupplevelse	34
6.2.2	Interaktionsresponsivitet och datafiltrering	34

6.2.3	Nätverkseffektivitet och bandbreddsanvändning	35
6.2.4	Total användarupplevelse över tid	35
6.3	Resultat ur ett bredare perspektiv	35
6.3.1	Balansen mellan teknisk optimering och användarupplevelse .	35
6.3.2	Alternativa forskningsfrågor	36
6.3.3	Studiens plats i ett bredare sammanhang	36
6.3.4	Alternativa slutsatser	36
6.4	Obesvarade aspekter och framtida arbete	37
6.4.1	Obesvarade aspekter	37
6.4.2	Framtida arbete	37
6.5	Avslutande reflektion	38

Referenser		39
-------------------	--	-----------

1.1 Bakgrund

I modern webbutveckling är optimerad prestanda och användarvänlighet avgörande för att möta användarnas förväntningar och behov. Webbapplikationer blir alltmer komplexa samtidigt som användarnas krav på snabba och responsiva gränssnitt fortsätter att öka. Vid utveckling av webbapplikationer är det därför centralt att samtliga delsystem optimeras för att säkerställa systemets övergripande prestanda.

Vue.js [1] har etablerat sig som ett populärt JavaScript-ramverk för uppbyggnad av användargränssnitt sedan lanseringen 2014. Dess komponentbaserade arkitektur, reaktiva datamönster och flexibilitet gör ramverket väl lämpat för utveckling av moderna webbapplikationer. En utmaning inom utveckling av webbapplikationer är dock att balansera prestandaoptimering med funktionalitet, exempelvis när applikationer hanterar datamängder från content management systems (CMS).

Wagtail [2], ett CMS byggt på Django-ramverket, används i allt större utsträckning som headless CMS i kombination med moderna JavaScript-ramverk som Vue.js. Genom Wagtails standardiserade API kan utvecklare bygga dynamiska klientapplikationer som använder CMS-innehåll utan att vara bundna till traditionella serverrenderade lösningar. Denna arkitektur innebär dock vissa utmaningar kring hur innehåll ska laddas effektivt för att optimera både prestanda och användarvänlighet.

Ett av de mer centrala besluten vid implementering av datahantering i Vue-komponenter är valet mellan eager loading och lazy loading. **Eager loading** innebär att all data hämtas vid sidladdning, vilket resulterar i längre initial laddningstid men snabbare interaktioner därefter. **Lazy loading** innebär istället att data hämtas först när den behövs, vilket ger snabbare initial laddning men kan leda till fördröjningar vid senare interaktioner. I denna studie undersöks dessa strategier genom implementationer i Vue med hjälp av onMounted för initial dataladdning samt watchers för att hantera dynamiska uppdateringar av innehåll.

Vid Göteborgs universitet har forskningsinfrastrukturen GRIDH (Göteborgs forskningsinfrastruktur för digitala humaniora) fått ett uppdrag att utveckla en digital portal till tidskriften Adoranten. Tidskriften, som har utgivits årligen sedan 1970-talet och innehåller artiklar om hållristningar och förhistorisk konst, saknar i dagsläget en effektiv digital sökbar portal. Projektet involverar utveckling av en sökbar portal i Vue med Wagtail CMS som backend, där optimering av prestanda vid innehållsladdning och filtrering av artikeldata kommer undersökas vidare i denna studie.

Turcotte et al. har visat att implementering av lazy loading kan öka responsiviteten i webbapplikationer avsevärt [3]. Samtidigt har studier av van Zyl et al. [4] och

Priya et al. [5] påvisat att valet mellan lazy och eager loading måste anpassas efter användningskontext, där lazy loading visar fördelar vid enstaka förfrågningar medan eager loading presterar bättre när samma data används upprepade gånger.

1.2 Studiens omfattning

Denna studie fokuserar på att undersöka och jämföra prestandan mellan två olika strategier för innehållsladdning i Vue-komponenter mot Wagtails standardiserade API: eager loading och lazy loading. Eager loading-strategin implementeras genom att hämta alla artiklar via Wagtails API med efterföljande filtrering i Vue. Lazy loading-strategin implementeras genom direkt filtrering via Wagtails sök-API.

Forskningsproblematiken i denna studie centreras kring avvägningen mellan initial laddningstid och interaktionsresponsivitet. Trots att tidigare forskning undersökt laddningsstrategier i olika kontexter, saknas avgränsade empiriska studier som specifikt jämför dessa strategier i moderna komponentbaserade JavaScript-ramverk mot headless CMS. Denna kunskapslucka blir särskilt relevant då ett ökande antal organisationer tycks migrera från traditionella CMS-lösningar till headless-arkitekturer.

Studien jämför prestandan mellan dessa strategier genom mätningar av konkreta prestandafaktorer som tid för initial laddning, tid för filtrering, antal API-anrop och överförd datamängd. Dessa mätningar utförs under testscenariet; initial sidladdning och filtrering på titel.

För att isolera effekterna av implementationerna i Vue-komponenter används Wagtails API i standardutförande utan modifieringar. Studien inriktar sig på frontend-implementation och jämförelse mellan eager och lazy loading, medan avancerade optimeringar som virtuell scroll utesluts från huvudstudien.

Studien avgränsas till att endast mäta objektiva prestandafaktorer och utesluter därmed användarupplevelse och subjektiva bedömningar. Fokus ligger på att generera empiriska resultat som kan vägleda utvecklare i valet mellan eager och lazy loading för Vue-komponenter mot CMS-baserat innehåll.

1.3 Översikt

Rapporten är strukturerad i sex huvudkapitel. Efter denna introduktion följer kapitel 2, "Litteraturstudie", som presenterar relevant forskning och teoribildning inom området och sätter studiens frågeställningar i kontext. Kapitel 3, "Metod", beskriver den metodologiska ansatsen, preciserar forskningsfrågorna och redogör för studiens genomförande. Här beskrivs även studiens utvärderingsramverk och datainhämtningsprocesser.

I kapitel 4, "Resultat och analys", presenteras studiens empiriska fynd tillsammans med datavisualisering och analys. Kapitel 5, "Diskussion", innehåller en kritisk reflektion över både de tillämpade metoderna och de erhållna resultaten i relation till forskningsfrågorna. Avslutningsvis summerar kapitel 6, "Slutsatser och framtida arbete", studiens viktigaste bidrag samt identifierar områden för fortsatt forskning inom prestandaoptimering av Vue-komponenter mot CMS-system.

Som nämnts ovan finns det begränsat med empiriska studier för denna specifika kontext. Sökningarna efter relevant litteratur utfördes i databaserna Scopus, Google Scholar och Blekinge tekniska högskolas samlade söktjänst Summon@BTH. Sökningarna strukturerades kring relevanta nyckeltermerna, som kombinerades på olika sätt. Nedan följer de mest relevanta sökkombinationerna:

- "lazy AND loading"(69 träffar i Scopus med begränsning till "Computer science")
- "eager AND loading"(10 träffar i Scopus med begränsning till "Computer science")
- Åagtail CMS"(24 träffar i Google Scholar, ej relevanta)
- "Vue.js AND performance"(29 träffar i Google Scholar, ej relevanta)
- "Vue.js AND lazy AND eager AND loading"(78 träffar i Google Scholar, ej relevanta)

Ett urvalskriterium var att litteraturen huvudsakligen skulle vara publicerad de senaste fem åren, för att säkerställa relevans inom den snabbt utvecklande webb-utvecklingsdomänen. Detta resulterade i väldigt få träffar, därav utökades tidsavgränsningen till senaste femton åren. För att identifiera relevant litteratur användes följande kriterier:

- Publikationer som undersöker lazy loading och eager loading
- Studier som utvärderar prestandaskillnader mellan olika laddningsstrategier
- Artiklar som relaterar till Vue.js, CMS-system eller liknande teknologier

Litteratursökningen resulterade i flera relevanta studier som behandlar olika aspekter av lazy loading och eager loading samt deras påverkan på prestandan i webb-utveckling. I de kommande avsnitten kategoriseras och analyseras dessa tidigare forskningsstudier för att ge ett teoretiskt ramverk för denna studie. Två referenser görs även till Llanos [6], som hittades genom att följa referenser i van Zyl [4]. Denna artikel analyseras dock inte i den följande genomgången.

2.1 Laddningsstrategier och deras prestandapåverkan

En central fråga inom webbutveckling är hur innehåll och data ska laddas för att optimera användarupplevelsen. Laddningsstrategier kan grovt delas in i två huvudsakliga kategorier: eager loading och lazy loading. Båda strategierna har sina fördelar och nackdelar beroende på användningskontext.

Turcotte et al. [3] har presenterat "Lazifier", ett verktyg som automatiskt identifierar JavaScript-paket som kan laddas med lazy loading-teknik. Deras studie visar att strategisk implementering av lazy loading kan ge betydande prestandaförbättringar. I deras utvärdering av tio open-source webbapplikationer kunde initial applikationsstorlek reduceras med i genomsnitt 36,2% och laddningstiden förbättras med 29,7%. Studien är relevant för vårt arbete eftersom den visar hur automatiserad analys kan identifiera potentiella områden för lazy loading i befintlig kod, och att detta också kan förbättra prestandan i Vue-komponenter.

För att förstå de tekniska förutsättningarna för olika laddningsstrategier är van Zyl et al. [4] studie av optimeringsmetoder för Object Relational Mapping (ORM) verktyg särskilt värdefull. De undersöker hur lazy loading, eager loading och caching påverkar prestandan för ORM-verktyget Hibernate. Deras resultat visar att lazy loading ger bättre prestanda för frågeoperationer, medan eager loading kan vara mer fördelaktigt för objekt som används upprepade gånger. Studien visar att valet av laddningsstrategi bör vara kontextberoende.

Mjelde och Opdahl [7] har genomfört en omfattande undersökning av olika tekniker för att minska laddningstider på webbplatser för olika enhetstyper. De visar att lazy loading är särskilt fördelaktigt för mobila enheter med begränsad bandbredd. Deras studie lyfter dock fram en viktig avvägning: medan lazy loading minskar den initiala laddningstiden kan det påverka användarupplevelsen negativt om innehåll laddas med fördröjning när användaren scrollar. Något som inte undersöks vidare i föreliggande studie men som bör tas i åtanke.

Priya et al. [5] studie om optimering av webbapplikationer med eager loading ger ytterligare insikter. De undersöker hur eager loading kan implementeras tillsammans med nyckel-värde databaser för att förbättra prestandan i webbapplikationer. Deras experiment visar dramatiska prestandaförbättringar, där bearbetningstiden för vissa datauppsättningar minskade med upp till 96% när eager loading användes istället för lazy loading. Studien fokuserar främst på backend-prestanda i isolering.

2.2 Implementering av laddningsstrategier i olika miljöer

Laddningsstrategier implementeras på olika sätt beroende på teknisk miljö och applikationskontext. Flera studier belyser hur dessa strategier kan implementeras i specifika tekniska miljöer, vilket ger värdefulla insikter för vår implementation i Vue mot Wagtail CMS.

Tuaycharoen et al. [8] beskriver implementationen av lazy loading i ett webbaserat schemalaggningsystem. Deras studie visar att implementering av lazy loading i

en realtidsapplikation resulterade i cirka 50% snabbare sidresponstider jämfört med traditionell laddning. De beskriver även en arkitektur som tydligt separerar frontend från backend, vilket liknar den arkitektur vi ämnar använda med Vue-komponenter mot Wagtail API. Studien ger konkreta exempel på hur lazy loading kan implementeras på ett sätt som stödjer realtidsuppdateringar av data.

Kargin et al. [9] presenterar Lazy ETL för vetenskapliga datalager, en metod för att reducera initial laddning till enbart metadata, medan faktisk data hämtas vid behov under frågeexekvering. Trots att studien fokuserar på ETL-processer (Extract, Transform, Load) snarare än webbutveckling, är principen liknande: att undvika att ladda mer data än nödvändigt i ett första skede.

Zagarese et al. [10] diskuterar avancerade laddningsstrategier för dataintensiva webbtjänster. Deras studie introducerar en teknik kallad dynamic object offloadings som kan betraktas som en vidareutveckling av traditionell lazy loading. Genom att analysera användningsmönster kan systemet dynamiskt avgöra vilka objekt som ska laddas i förväg och vilka som ska laddas vid behov. Denna adaptiva strategi är särskilt intressant som en potentiell framtida utveckling av vår implementation, även om den ligger utanför huvudfokus för denna studie.

Vue.js egen dokumentation [11] om prestandaoptimering diskuterar både lazy loading och eager loading i kontexten av Vue-komponenter. Dokumentationen rekommenderar lazy loading för större komponenter och resurser som inte omedelbart behövs vid sidladdning, men betonar också vikten av att balansera lazy loading mot användarupplevelsen. Denna officiella dokumentation utgör en viktig referenspunkt för best practices inom Vue.js-utveckling.

2.3 Metodologiska ansatser för utvärdering av laddningsstrategier

För att utvärdera olika laddningsstrategier krävs en strukturerad metodologisk ansats. Flera tidigare studier har använt olika metoder för att mäta och jämföra prestandan av olika implementationer.

Keuning et al. [12] demonstrerar en framgångsrik tillämpning av design science för utveckling av ett handledningssystem för kodrefaktorisering. Även om deras studie inte direkt berör laddningsstrategier, erbjuder den ett värdefullt exempel på hur design science kan tillämpas för utveckling och utvärdering av mjukvarusystem. Deras arbete stödjer vårt val av design science som metodologiskt ramverk för denna studie, särskilt genom att visa hur de tre faserna av problemundersökning, behandlingsdesign och behandlingsvalidering kan strukturera forskningsprocessen.

Wieringa [13] beskriver design science som design och undersökning av artefakter i en kontext", vilket är direkt tillämpbart på vår studie av laddningsstrategier i kontexten av Vue och Wagtail CMS. Wieringas guidelines ger ett ramverk för att systematiskt analysera hur våra implementationer påverkar prestandan och hur vi kan utvärdera deras effektivitet genom konkreta mätningar.

2.4 Sammanfattning och forskningsgap

Litteraturen visar att valet mellan lazy loading och eager loading kan ha betydande inverkan på prestandan i webbutveckling. Turcotte et al. [3] och Mjelde & Opdahl [7] ger empiriska bevis för hur lazy loading kan minska den initiala sidladdningstiden, medan van Zyl et al. [4] och Priya et al. [5] visar att eager loading kan vara fördelaktigt under vissa omständigheter, särskilt när samma data används upprepade gånger.

Trots den omfattande forskningen om laddningsstrategier finns det enligt den litteraturstudie som utförts och beskrivits ovan få studier som specifikt undersöker hur dessa strategier kan implementeras och optimeras i Vue-komponenter mot headless CMS-system som Wagtail. Denna studie ämnar utforska detta område genom att undersöka hur Vue-komponenter kan struktureras för effektiv innehållsladdning, och specifikt hur valet mellan eager loading och lazy loading påverkar prestandan i denna avgränsade kontext.

Tabell 2.1 sammanfattar tidigare forskning och hur föreliggande studie relaterar till den.

Tabell 2.1: Sammanfattning av tidigare forskning och relation till föreliggande studie

Studie	Fokus	Relation till vår studie
Turcotte et al. [3]	Automatisk identifiering av lazy loading-kandidater	Ger empiriskt stöd för prestandaförbättringar med lazy loading
van Zyl et al. [4]	Optimering av ORM-verktyg med olika laddningsstrategier	Kontextberoende val mellan eager och lazy loading
Priya et al. [5]	Eager loading i webbapplikationer	Kvantitativa jämförelser mellan strategierna i backend-isolering
Tuaycharoen et al. [8]	Lazy loading i realtidsapplikationer	Implementation av lazy loading i webbmiljö
Kargin et al. [9]	Lazy ETL för vetenskapliga data	Principer för behovsbaserad datahämtning
Zagarese et al. [10]	Avancerade laddningsstrategier	Dynamisk objekthantering
Mjelde & Opdahl [7]	Tekniker för reducerad laddningstid	Användningskontext för olika tekniker
Vue.js [11]	Best practices för Vue.js	Referenspunkt för implementation
Keuning et al. [12]	Design science-tillämpning	Metodologiskt ramverk

Detta kapitel beskriver den metodologiska ansatsen för studien, forskningsfrågorna och studiens genomförande. Inledningsvis presenteras studiens forskningsdesign baserad på design science-metodiken. Därefter preciseras forskningsfrågorna följt av en detaljerad beskrivning av implementationerna för eager loading och lazy loading. Vidare redogörs för testmiljön, mätparametrar och testscenarier. Avslutningsvis diskuteras metodens validitet och reliabilitet.

3.1 Forskningsdesign

Studien tillämpar design science-metodologin enligt Wieringa [13], då detta ramverk är särskilt lämpligt för studier som syftar till att utveckla och utvärdera tekniska artefakter i en specifik kontext. Alternativa metoder som fallstudier eller enkätundersökningar övervägdes men valdes bort då de inte ger samma objektiva prestandamätningar som krävs för att besvara RQ2. Design science betonar betydelsen av både artefakt-design och systematisk utvärdering, vilket överensstämmer med studiens mål att implementera och utvärdera två olika strategier för innehållsladdning.

3.1.1 Design science-metodiken

Design science-metodiken strukturerar forskningsprocessen i tre distinkta designcykler [13]:

1. **Problemundersökning** - En fas där problemområdet kartläggs genom litteraturstudier och analys av befintliga tekniska lösningar.
2. **Behandlingsdesign** - Den fas där potentiella lösningar utformas baserat på insikterna från problemundersökningen.
3. **Behandlingsvalidering** - En fas där de utvecklade lösningarna utvärderas genom empiriska tester.

I denna studie tillämpas dessa designcykler på följande sätt:

Problemundersökning

Studien inleddes med en litteraturstudie för att kartlägga tidigare forskning kring lazy loading och eager loading i webbutveckling. Ett praktiskt exempel i form av Adoranten-projektet användes för att konkretisera problemområdet och dess relevans.

Behandlingsdesign

Baserat på insikterna från problemundersökningen designades två distinkta implementationer för innehållsladdning i Vue.js mot Wagtail CMS:

1. **Eager loading-implementation:** En lösning där all artikeldata hämtas vid initial sidladdning med efterföljande klientsidig filtrering.
2. **Lazy loading-implementation:** En lösning där endast den data som behövs hämtas genom direkta API-anrop med filtrering på serversidan.

Dessa implementationer utformades specifikt för att möjliggöra systematiska jämförelser av deras prestandaegenskaper under olika användningsscenarier.

Behandlingsvalidering

För att validera och jämföra de två implementationerna utvecklades ett ramverk för prestandamätning. Detta ramverk möjliggör insamling av objektiva mätvärden som tidsåtgång för initial laddning och filtrering under standardiserade testförhållanden.

Studien följer särskilt Wieringas [13] guideline "G5 design argument", som består av två delar:

1. **Causation part:** Hur valet av laddningsstrategi påverkar mätbara prestandafaktorer.
2. **Valuation part:** Utvärdering av strategiernas effektivitet genom konkreta mätningar.

Genom att strukturera studien enligt dessa principer säkerställs att både orsaksamband och värderingsaspekter inkluderas, vilket bidrar till såväl teoretisk som praktisk relevans av forskningsresultaten.

3.2 Precisering av forskningsfrågor

Studiens forskningsdesign utgår från följande två forskningsfrågor:

RQ1: "Hur kan Vue-komponenter struktureras för att hantera innehållsladdning med eager loading respektive lazy loading mot Wagtails standardiserade API?"

RQ2: "Hur påverkar valet mellan eager och lazy loading i Vue.js prestandan vid innehållsladdning mot Wagtails standardiserade API?"

För att besvara dessa forskningsfrågor har studien operationaliserats i följande underfrågor:

För **RQ1**:

- Vilka skillnader i kodstruktur krävs för implementering av eager loading jämfört med lazy loading i Vue-komponenter?
- Hur kan Vue.js reaktiva system utnyttjas för effektiv datahantering i respektive strategi?

För **RQ2**:

- Hur påverkas tiden för initial sidladdning av valet mellan eager och lazy loading?
- Hur påverkas tiden för filtrering av valet mellan eager och lazy loading?
- Hur varierar prestandaskillnaderna mellan strategierna beroende på datavolym (antal träffar vid filtrering)?

3.3 Implementation av laddningsstrategier

För att undersöka prestandaskillnader mellan eager loading och lazy loading implementerades två versioner av samma funktionalitet i Vue.js mot Wagtail CMS API. Båda implementationerna inkluderades i samma komponent med en växlingsmekanism för att säkerställa identiska testförutsättningar.

Testdata bestod av 32 utgåvor innehållande totalt 200 artiklar. Denna datamängd bedömdes som tillräckligt omfattande för att visa på meningsfulla prestandaskillnader mellan strategierna.

3.3.1 Teknisk miljö

Implementationerna utvecklades i följande tekniska miljö:

- **Frontend:** Vue.js (version 3)
- **Backend:** Wagtail CMS (version 5.1)
- **API:** Wagtails standardiserade REST API (v2)
- **Nätverkskommunikation:** HTTP/HTTPS
- **Komponentstruktur:** Composition API med `<script setup>`

3.3.2 Eager loading-implementation

Eager loading-implementationen kännetecknas av följande egenskaper:

1. **Initial datahämtning:** Alla artiklar hämtas via ett enda API-anrop vid komponentens montering.
2. **Lokal datalagring:** Hämtad data lagras i en reaktiv referens (`allArticles`).
3. **Klientsidig filtrering:** Filtrering sker lokalt genom JavaScript-filtrering av den hämtade datan.

Kärnan i eager loading-implementationen utgörs av följande kodfragment:

```
onMounted(async () => {
  // Fetch all articles
  try {
    const response = await fetch('https://shfa.dh.gu.se/
      wagtail/api/v2/pages/?type=journal.ArticlePage&limit
      =200')
    if (!response.ok) {
      throw new Error('network response was not ok')
    }
    const data = await response.json()

    // Store in all articles
    allArticles.value = data.items || []
    results.value = allArticles.value
  } catch (error) {
    console.error('Error fetching ArticlePages:', error)
  }
})

// Filter local articles array when searchTerm changes
watch(searchTerm, async (newValue) => {
  // Use original eager loading implementation
  if (!newValue) {
    results.value = allArticles.value
  } else {
    const start = performance.now()
    results.value = allArticles.value.filter(item =>
      item.title.toLowerCase().includes(newValue.
        toLowerCase()))
    const end = performance.now()
    console.log('Eager load filtering took ${end - start}.
      toFixed(2)} ms')
  }
})
```

Denna implementation följer principen för eager loading genom att hämta all data vid initial laddning och därefter filtrera lokalt. Prestandamätningar implementeras genom `performance.now()` för att mäta tidsåtgången för filtreringsoperationen.

3.3.3 Lazy loading-implementation

Lazy loading-implementationen har följande egenskaper:

1. **Behovsbaserad datahämtning:** Data hämtas endast när den efterfrågas genom ett sökvillkor.
2. **Serversidefiltrering:** Filtrering sker på serversidan genom Wagtails sök-API.

3. Minimal datalagring: Endast nödvändig data lagras i klienten.

Kärnan i lazy loading-implementationen utgörs av följande kodfragment:

```
watch(searchTerm, async (newValue) => {
  if (useLazyLoading.value) {
    // Lazy loading implementation
    const start = performance.now()

    try {
      const response = await fetch('https://shfa.dh.gu.se/wagtail/api/v2/pages/?type=journal.ArticlePage&search=${encodeURIComponent(newValue)}')
      if (!response.ok) throw new Error('Network response was not ok')
      const data = await response.json()
      results.value = data.items || []

      const end = performance.now()
      console.log('Lazy load search took ${Math.floor(end - start)} ms')
    } catch (error) {
      console.error('Error in lazy loading search:', error)
      results.value = []
    }
  }
})
```

Denna implementation följer principen för lazy loading genom att hämta data först när den behövs, baserat på användarens sökterm. Även här används `performance.now()` för att mäta tidsåtgången för hela operationen, inklusive nätverkskommunikation och databehandling.

Det är viktigt att notera att både eager loading- och lazy loading-implementationerna är designade för att hantera både utgåvor (`IssuePage`) och de mer flertaliga artiklarna (`ArticlePage`). För lazy loading resulterar detta i två separata nätverksanrop vid varje sökning - ett för varje innehållstyp. Detta bidrar till fler nätverksanrop men möjliggör sökning i alla relevanta innehållstyper.

3.3.4 Växlingsmekanism

För att möjliggöra systematisk jämförelse mellan de två implementationerna implementerades en växlingsmekanism genom den reaktiva referensen `useLazyLoading`:

```
const useLazyLoading = ref(false) // Toggle between eager
and lazy loading implementations
```

Genom att ändra värdet på denna referens kan testaren växla mellan eager loading och lazy loading utan att ändra koden, vilket säkerställer konsistenta testförhållanden.

3.4 Testmiljö och mätparametrar

För att säkerställa tillförlitliga och jämförbara mätresultat utformades en standardiserad testmiljö med väldefinierade mätparametrar.

3.4.1 Testmiljö

Testerna utfördes i en miljö som återspeglar verkliga användningsscenarioer, med en lokal frontend-applikation (Vue.js) som kommunicerade med en server-deployad backend (Wagtail CMS). Vue.js-applikationen exekveras helt i användarens webbläsare, där all JavaScript-kod och filtrering (för eager loading) sker lokalt på klientens dator. Denna uppsättning representerar väl hur moderna webbapplikationer vanligtvis opererar, där klienten kommunicerar med en server över ett nätverk.

Testerna utfördes under följande förhållanden:

- **Hårdvara:** HP ZBook 14 G2 med Intel Core i7-5500U CPU @ 2.40GHz, 16GB RAM
- **Operativsystem:** Microsoft Windows 10 Pro (version 10.0.19045)
- **Webbläsare:** Google Chrome (version 135.0.7049.85)
- **Nätverksförhållanden:** Fast bredbandsuppkoppling (100 Mbps) för att minimera nätverksvariationer
- **Testverktyg:** Inbyggda prestandaverktyg i webbläsaren (Chrome DevTools) samt Vue-komponentens egna tidtagningsfunktioner

För att minimera påverkan från externa faktorer utfördes alla tester:

- Med samma webbläsare och hårdvara
- I inkognitoläge i Chrome, med en ny inkognitosession för varje testomgång
- Med identisk uppsättning bakgrundsprocesser
- Under en begränsad tid på dygnet för att minimera påverkan från serverbelastning

3.4.2 Mätparametrar

Valet av mätparametrar baserades på deras direkta koppling till forskningsfråga RQ2 om prestandapåverkan. Tidsmätningar för laddning och filtrering valdes för att fånga användarupplevd prestanda, medan nätverksanrop och dataöverföring inkluderades för att kvantifiera resursanvändningen. Dessa parametrar är etablerade inom fältet och används av Turcotte et al. [3] och Priya et al. [5] i liknande studier. Parametrarna mättes både vid initial sidladdning och vid efterföljande filtreringar, där eager loading typiskt visar högre värden initialt men minimala värden eller nollvärden vid interaktion, medan lazy loading visar det omvända mönstret.

Följande mätparametrar användes för att kvantifiera prestanda:

1. Tid för initial laddning (ms)

- Definition: Tiden för komponentens montering (lazy och eager loading), inkluderat att alla artiklar har hämtats (eager loading).
- Mätmetod: `performance.now()` vid start och slut av laddningsoperationen.
- Relevans: Mäter användarens upplevda väntetid vid första besöket på sidan.

2. Tid för filtrering (ms)

- Definition: Tiden från det att användaren ändrar söktermen till dess att filtrerade resultat har renderats.
- Mätmetod: `performance.now()` vid start och slut av filtreringsoperationen.
- Relevans: Mäter responsiviteten vid användarinteraktion med sökfunktionen.

3. Nätverksanrop (antal)

- Definition: Antalet HTTP-förfrågningar som görs mot API:et.
- Mätmetod: Chrome DevTools Network-panel.
- Relevans: Indikerar systemets belastning på nätverket och servern.

4. Överförd datamängd (kB)

- Definition: Totala mängden data som överförs via nätverket.
- Mätmetod: Chrome DevTools Network-panel.
- Relevans: Indikerar bandbreddsanvändning och ger insikt i hur de olika strategierna påverkar nätverkstrafiken över tid.

3.4.3 Datainsamlingsprocess

Datainsamlingen genomfördes genom följande process:

1. Förberedelse:

- Rensa webbläsarens cache
- Öppna Chrome DevTools Network-panel
- Aktivera prestandaregistrering

2. Initial laddningstest:

- Ladda komponenten, registrera tidsstämpel (lazy loading)
- Registrera tidsstämpel när alla artiklar är laddade (eager loading)

3. Filtreringstest:

- Mata in sökterm i sökfältet
- Registrera tidsstämpel när filtrerade resultat har renderats (eager och lazy loading)

4. Dokumentation:

- Dokumentera nätverksanrop och överförd datamängd från Chrome Dev-Tools
- Spara rådata för senare analys

Varje test utfördes tio gånger för att säkerställa statistisk tillförlitlighet, och medelvärden beräknades för alla mätparametrar.

3.5 Testscenarier

För att undersöka prestandaskillnader mellan eager loading och lazy loading under olika förhållanden utformades två huvudsakliga testscenarier.

3.5.1 Initial sidladdning

Detta scenario mäter prestandan vid första laddningen av komponenten, innan någon användare interagerat med sökfunktionen. Scenariot testar hypotesen att eager loading medför längre initial laddningstid jämfört med lazy loading på grund av den större datamängden som hämtas initialt.

3.5.2 Filtrering efter titel

Detta scenario mäter prestandan vid filtrering av artiklar baserat på en sökterm som matchar artikeltitlar. Scenariot testar hypotesen att eager loading ger snabbare filtreringsrespons jämfört med lazy loading efter initial laddning, eftersom ingen ytterligare nätverkskommunikation krävs vid filtrering.

Filtreringsscenarierna genomfördes med specifika söktermer som resulterade i olika antal träffar:

- "Rock"(13 träffar)
- "Ärt"(6 träffar)
- "USA"(1 träff)

Detta urval av söktermer möjliggör analys av hur antalet resultat påverkar prestandaskillnaderna mellan eager loading och lazy loading-strategierna.

3.6 Analysmetoder

För att analysera insamlad data och kvantifiera prestandaskillnader mellan eager loading och lazy loading användes följande metoder:

3.6.1 Kvantitativ analys

- **Medelvärdesberäkning:** Beräkning av genomsnittlig tidsåtgång för varje operation.
- **Procentuell jämförelse:** Prestandaskillnaden mellan implementationerna uttryckt i procent.

3.6.2 Visualisering

- **Stapeldiagram:** Jämförelse av initial laddningstid och filtreringstid mellan eager loading och lazy loading.
- **Linjediagram:** Visualisering av hur total prestanda, dataöverföring och nätverksanvändning utvecklas över användarinteraktioner.

3.6.3 Kvalitativ analys

- **Jämförelse av kodstruktur:** Analys av skillnader i implementationsstruktur.

3.7 Metodimplementation

3.7.1 Kodinstrumentation

För att möjliggöra noggrann mätning instrumenterades koden med prestandamätningens funktionalitet. Detta innebär:

1. **Tidtagning:** Implementering av `performance.now()` för att mäta exekveringstiden för kritiska operationer.

```
const start = performance.now()
// Operationen som ska mätas
const end = performance.now()
console.log(`Operation took ${end - start}.toFixed(2)} ms`)
```

2. **Loggning:** Implementering av konsollloggning för att dokumentera mätresultat och systemtillstånd.

```
console.log(`Eager load initial rendering took ${end - start} ms`)
```

```
console.log(`Lazy load initial rendering took ${end - start} ms`)
```

```
console.log(`Eager load filtering took ${end - start} ms`)
```

```
console.log(`Lazy load search took ${end - start} ms`)
```

3.7.2 Strukturerad datainsamling

För att säkerställa konsistens i testförfarandet följdes en manuell testprocess med tydliga steg. Detta genom systematisk insamling av mätvärden i fördefinierade Excel-tabeller, med separata kolumner för:

- Initial laddningstid
- Filtreringstid för varje sökterm
- Antal nätverksanrop (både initialt och vid sökning för lazy loading)
- Initial datamängd (för båda strategierna)
- Sökning datamängd (som är 0 för eager loading, men varierar för lazy loading)

3.7.3 Praktiskt genomförande

Testerna genomfördes enligt följande process:

1. **Konfiguration:** Ställ in testmiljön och aktivera lämpliga utvecklarverktyg
2. **Nätverksvariationer:** För att minimera effekten av nätverksvariationer genomförs testerna vid samma tid på dygnet, och varje test upprepas 10 gånger för att beräkna ett tillförlitligt medelvärde.
3. **Baseline-mätning:** Genomför mätningar för eager loading-implementationen
4. **Experimentell mätning:** Genomför mätningar för lazy loading-implementationen
5. **Dataanalys:** Analysera insamlade data och beräkna relevanta statistiska mått
6. **Validering:** Upprepa mätningar vid behov för att validera resultat

3.8 Metodkritik

Denna sektion diskuterar potentiella begränsningar och svagheter i den valda metodologin.

3.8.1 Validitet

Intern validitet:

- **Oönskade variabler:** Trots ansträngningar att minimera externa påverkansfaktorer kan nätverkslatens och serverbelastning variera mellan tester.
- **Instrumenteringseffekter:** Själva mätinstrumenteringen kan påverka prestandan marginellt.

Extern validitet:

- **Begränsad datamängd:** Testerna utfördes på en begränsad datamängd från Adoranten-projektet, vilket kan påverka generaliseringsförmågan till större system.
- **Specifik teknisk miljö:** Resultaten är specifika för Vue.js och Wagtail CMS, vilket kan begränsa applicerbarheten på andra tekniska miljöer.

3.8.2 Reliabilitet

För att höja studiens reliabilitet vidtogs följande åtgärder:

- **Standardiserad testmiljö:** Alla tester utfördes under identiska förhållanden.
- **Upprepad mätning:** Varje test upprepades för att minimera slumpmässiga variationer.
- **Dokumentation:** Detaljerad dokumentation av testmiljö och förfarande för att möjliggöra replikering.
- **Ny inkognitosession:** Varje testomgång utfördes i en ny inkognitosession för att eliminera webbläsarspecifik caching.

Trots dessa åtgärder finns det flera faktorer som kan ha påverkat testernas tillförlitlighet och som bör beaktas vid tolkning av resultaten, två enligt forskningslitteraturen förekommande är:

- **Bakgrundsprocesser och CPU-schemaläggning:** Operativsystemets hantering av samtidiga processer kan påverka CPU-tillgängligheten mellan testkörningar, vilket Llanos [6] identifierar som en betydande faktor i prestanda-testning.
- **Nätverksslatens-variation:** Zagarese et al. [10] demonstrerar i sin studie att förhållandet mellan datastorlek och nätverksgenomströmning har en betydande inverkan på webbtjänsters prestanda. Även med en fast bredbandsuppkoppling kan nätverksslatensnivåer variera över tid på grund av faktorer utanför vår kontroll, vilket kan påverka mätningar av nätverksberoende operationer som API-anrop mellan Vue-komponenter och servern.

För att hantera dessa potentiella hot mot reliabiliteten beräknades medelvärden över upprepade testkörningar, vilket minskar effekten av slumpmässiga variationer. Dock kvarstår vissa systematiska faktorer som kan påverka den absoluta noggrannheten i mätvärdena, även om den relativa jämförelsen mellan strategierna fortfarande är giltig.

3.8.3 Begränsningar

Metodologin har följande begränsningar:

- **Begränsad användarinteraktion:** Testerna simulerar endast grundläggande användarinteraktion och tar inte hänsyn till mer komplexa användarscenarier.
- **Fokus på mätbara faktorer:** Studien fokuserar på objektiva mätbara faktorer och utelämnar subjektiva aspekter av användarupplevelsen.
- **Begränsad teknisk variation:** Testerna omfattar endast två implementationsstrategier och utelämnar andra möjliga optimeringar.

I detta kapitel presenteras resultaten från prestandamätningarna för eager loading och lazy loading i Vue-komponenter mot Wagtail CMS API. Inledningsvis ges en sammanfattning av de viktigaste resultaten följt av detaljerade mätdata för respektive prestandaindikator. Kapitlet avslutas med en analys som relaterar resultaten till studiens båda forskningsfrågor.

4.1 Sammanfattning av resultat

Studien jämförde prestandan mellan eager loading och lazy loading genom systematiska mätningar över 10 testomgångar. Nedan följer en sammanfattning av de viktigaste resultaten:

4.1.1 Initial laddningstid

- **Eager loading:** 458,3 ms i genomsnitt
- **Lazy loading:** 271,4 ms i genomsnitt
- **Förbättring med lazy loading:** 40,8% snabbare initial laddning

4.1.2 Filtreringstid

- **Eager loading:**
 - Rock-sökning: 0,44 ms i genomsnitt
 - Art-sökning: 0,38 ms i genomsnitt
 - USA-sökning: 0,36 ms i genomsnitt
- **Lazy loading:**
 - Rock-sökning: 155,58 ms i genomsnitt
 - Art-sökning: 160,92 ms i genomsnitt
 - USA-sökning: 154,9 ms i genomsnitt
- **Prestandaskillnad:** Eager loading ger cirka 400 gånger snabbare filtrering

4.1.3 Nätverksanrop

- **Eager loading:**
 - Initialt: 47 anrop
 - Vid filtrering: 0 ytterligare anrop
- **Lazy loading:**
 - Initialt: 46 anrop
 - Vid filtrering: 2 ytterligare anrop per sökning

4.1.4 Dataöverföring

- **Eager loading:**
 - Initial datamängd: 1194 KB
 - Datamängd vid filtrering: 0 KB
- **Lazy loading:**
 - Initial datamängd: 1082 KB (9,4% mindre än eager loading)
 - Datamängd vid filtrering:
 - * Rock-sökning (13 träffar): 10,3 KB
 - * Art-sökning (6 träffar): 5,0 KB i genomsnitt
 - * USA-sökning (1 träff): 1,64 KB i genomsnitt

4.2 Initial laddningstid

Initial laddningstid avser tiden från att komponenten monteras tills att den är redo för användarinteraktion.

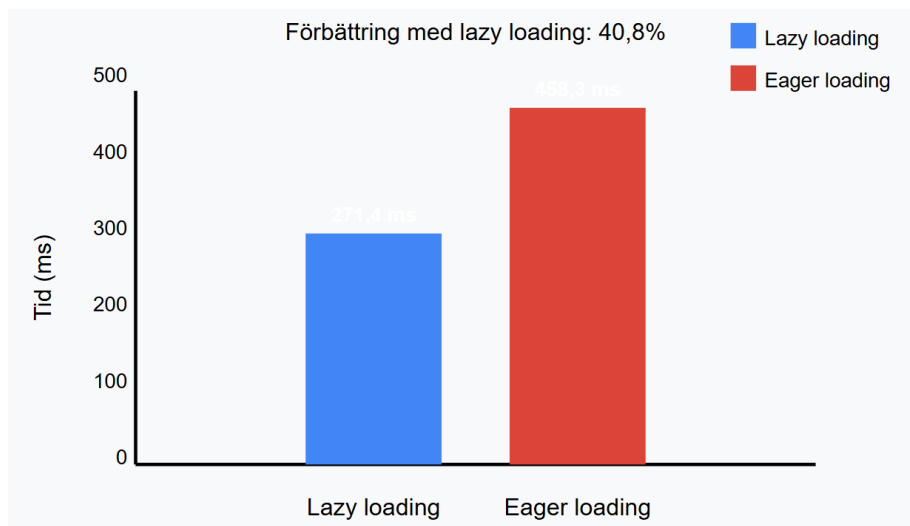
Tabell 4.1: Initial laddningstid (ms) över 10 testomgångar

Implementation	Medelvärde	Standardavvikelse
Eager loading	458,3	1,8
Lazy loading	271,4	1,3

Mätningarna visar en betydande prestandafördel för lazy loading vid initial sidladdning. Lazy loading-implementationen laddade sidan i genomsnitt 40,8% snabbare än eager loading-implementationen. Den låga standardavvikelsen för båda implementationerna indikerar konsistenta mätresultat över testomgångarna.

Denna prestandaskillnad kan förklaras genom att eager loading både renderar sidan, hämtar utgåvor, och hämtar alla 200 artiklar (som jämförelsen ligger till grund för) vid initial sidladdning. Lazy loading däremot renderar enbart sidan och hämtar utgåvor, vilket resulterar i kortare initial laddningstid.

Figur 4.1 visar tydligt lazy loadings fördelar vid initial sidladdning jämfört med eager loading.



Figur 4.1: Jämförelse av initial laddningstid

4.3 Filtreringstid

Filtreringstid avser tiden från att användaren initierar en sökning tills att filtrerade resultat visas. För att undersöka hur antalet träffar påverkar filtreringstiden testades tre söktermer: "Rock"(13 träffar), "Ärt"(6 träffar) och "USA"(1 träff).

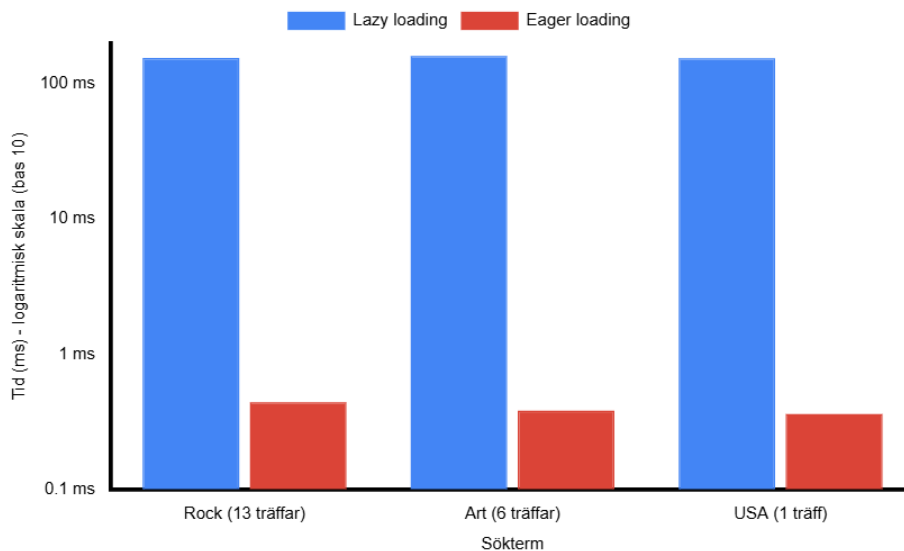
Tabell 4.2: Filtreringstid (ms) för olika söktermer

Sökterm	Eager loading	Lazy loading	Skillnad (faktor)
"Rock"	0,44	155,58	~354x
"Ärt"	0,38	160,92	~424x
"USA"	0,36	154,9	~430x

Resultaten visar en dramatisk prestandaskillnad mellan implementationerna vid filtrering. Eager loading filtrerade innehåll på under 0,5 ms i samtliga fall, medan lazy loading krävde cirka 155-161 ms. Detta innebär att eager loading var cirka 400 gånger snabbare vid filtrering.

För eager loading observerades att filtreringstiden minskade med färre sökträffar (0,44 ms för 13 träffar vs 0,36 ms för 1 träff), vilket indikerar en linjär komplexitet för filtreringsalgoritmen. För lazy loading var mönstret mindre tydligt; söktermen "Ärt"(6 träffar) resulterade faktiskt i något längre filtreringstid (160,92 ms) än "Rock"(13 träffar, 155,58 ms), vilket tyder på att nätverkskommunikationen är den dominerande faktorn snarare än resultatmängdens storlek vid en datamängd på 200 artiklar.

Figur 4.2 förtydligar skillnaden i filtreringstid för olika söktermer mellan strategierna. Då skillnaden är extremt stor används en logaritmisk skala (bas 10) för y-axeln.



Figur 4.2: Jämförelse av filtreringstid för olika söktermer

4.4 Nätverksanvändning

Analysen av nätverksanvändning visar mönster som är centrala för att förstå långsiktig effektivitet för respektive strategi.

Implementation	Initiala nätverksanrop	Dataöverföring initial (KB)	Nätverksanrop per sökning	Dataöverföring per sökning (KB)
Eager loading	47	1194	0	0
Lazy loading	46	1082	2	1,64-10,3 (beroende på resultat)

Vid initial laddning gjorde eager loading 47 nätverksanrop jämfört med lazy loading som gjorde 46 anrop. Den mest markanta skillnaden syns vid filtrering:

- **Eager loading** krävde inga ytterligare nätverksanrop eller dataöverföring.
- **Lazy loading** krävde konsekvent 2 anrop per sökning (då den likt eager loading filtrerar i både utgåvor och artiklar). Implementationen överförde mellan 1,64-10,3 KB beroende på antalet träffar, där fler träffar alltså innebar större överföring.

För ett scenario med initial laddning följt av sammanlagt tre sökningar blir den totala dataöverföringen:

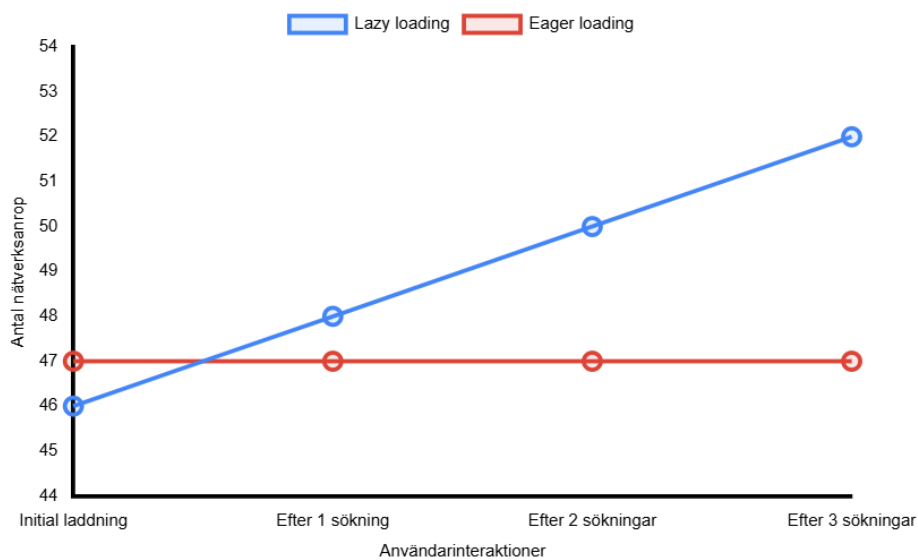
- Eager loading: 1194 KB total
- Lazy loading: 1082 KB + 10,3 KB + 5,0 KB + 1,64 KB = 1098,94 KB total

Detta visar att lazy loading fortfarande är mer dataeffektivt (8% mindre total dataöverföring) efter tre sökningar, men denna fördel minskar för varje sökning.

Figur 4.3 och 4.4 illustrerar hur lazy loading använder färre resurser initialt men ackumulerar fler anrop och dataöverföring över tid.



Figur 4.3: Total dataöverföring över användarinteraktioner



Figur 4.4: Antal nätverksanrop över användarinteraktioner

4.5 Kombinerad prestandaanalys

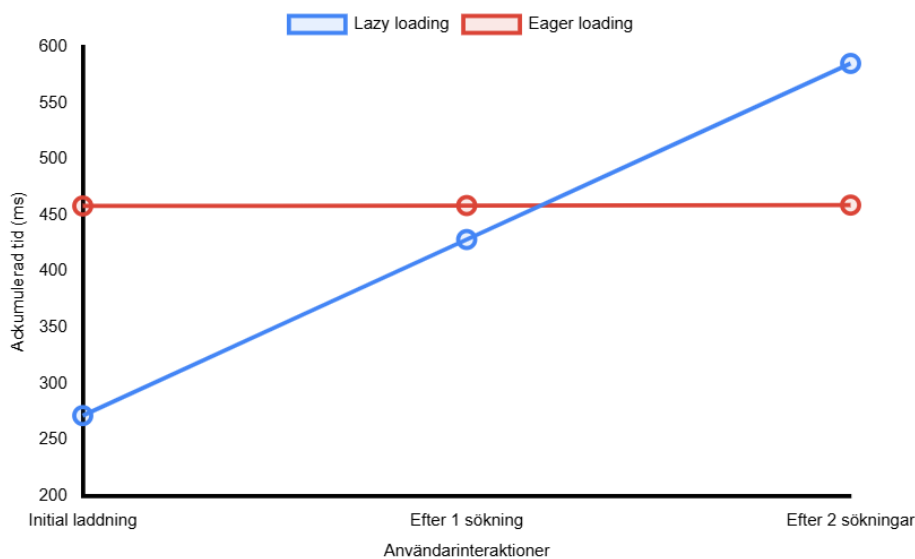
För att förstå den totala prestandapåverkan för användare är det viktigt att analysera den kombinerade genomsnittliga tiden för initial laddning och filtrering.

För ett scenario där en användare besöker sidan och utför en sökning är lazy loading cirka 30,3 ms (6,6%) snabbare totalt sett. Denna fördel försvinner dock vid upprepade sökningar. Efter två sökningar är eager loading redan 126,3 ms (21,6%) snabbare totalt sett.

Tabell 4.3: Kombinerad genomsnittlig tid för initial laddning och filtrering

Scenario	Eager loading (ms)	Lazy loading (ms)	Skillnad (ms)
Initial laddning	458,3	271,4	186,9
Första sökningen (Rock")	0,4	157,0	-156,6
Total tid (första sökningen)	458,7	428,4	30,3
Andra sökningen (Ärt")	0,4	157,0	-156,6
Total tid (två sökningar)	459,1	585,4	-126,3

Figur 4.5 illustrerar tydligt brytpunkten där eager loading blir mer effektivt än lazy loading efter den andra användarinteraktionen.



Figur 4.5: Total prestanda över användarinteraktioner

4.6 Kod- och implementationskomplexitet

För att besvara RQ1, "Hur kan Vue-komponenter struktureras för att hantera innehållsladdning med eager loading respektive lazy loading mot Wagtails standardiserade API?", analyseras de strukturella skillnaderna mellan implementationerna.

Tabell 4.4: Strukturella skillnader mellan implementationerna

Strukturell aspekt	Eager loading	Lazy loading
Datahämtning	Större initial datahämtning i onMounted	Minimal initial datahämtning
Datalagring	Lagrar all data i reaktiva referenser	Minimal datalagring, hämtar vid behov
Filtrering	JavaScript-filtrering lokalt med filter()	API-anrop med sökparameter
Reaktivitet	watch triggas lokal filtrering	watch triggas API-anrop
Felhantering	Enklare (inga nätverksfel vid filtrering)	Mer komplex (behöver hantera nätverksfel)

Båda implementationerna använder Vue.js reaktiva system, men på fundamentalt olika sätt:

1. **Eager loading** utnyttjar reaktiva referenser för att lagra och manipulera data lokalt.
2. **Lazy loading** använder reaktiva triggers för att initiera asynkrona API-anrop.

4.7 Sammanfattning av resultat i relation till forskningsfrågorna

Baserat på resultaten kan studiens forskningsfrågor nu besvaras på ett empiriskt grundat sätt.

4.7.1 RQ1: Strukturering av Vue-komponenter för innehållsladdning

För RQ1, "Hur kan Vue-komponenter struktureras för att hantera innehållsladdning med eager loading respektive lazy loading mot Wagtails standardiserade API?", visar resultaten att:

1. **Eager loading-komponenter** kan struktureras för:
 - Omfattande initial datahämtning i onMounted-hooken
 - Effektiv lagring av all data i reaktiva referenser (t.ex. allArticles och allIssues)
 - Lokal tidseffektiv JavaScript-filtrering med metoder som filter() och includes()
2. **Lazy loading-komponenter** kan struktureras för:
 - Minimal initial datahämtning
 - Reaktiva watch-funktioner som utlöser behovsbaserade API-anrop
 - Effektiv hantering av asynkrona processer

4.7.2 RQ2: Prestandapåverkan vid val av laddningsstrategi

För RQ2, "Hur påverkar valet mellan eager och lazy loading i Vue.js prestandan vid innehållsladdning mot Wagtails standardiserade API?", visar resultaten tydligt att:

1. **Initial laddning:** Lazy loading ger 40,8% snabbare initial sidladdning (271,4 ms vs 458,3 ms).
2. **Filtrering:** Eager loading ger cirka 400 gånger snabbare filtreringsrespons (under 0,5 ms vs cirka 155-161 ms).

3. **Nätverksanvändning:** Lazy loading använder 9,4% mindre data vid initial laddning, men kräver ytterligare nätverksanrop vid varje filtrering.
4. **Total effektivitet:** För användare som utför en sökning är lazy loading fortfarande 6,6% snabbare totalt sett, men efter två sökningar blir eager loading 21,6% snabbare.

Den djupare betydelsen av dessa resultat och hur de relaterar till tidigare forskning kommer att diskuteras i nästföljande kapitel.

4.8 Resultat i design science-kontext

I enlighet med design science-metodiken har denna studie undersökt två olika artefakter (eager och lazy loading-implementationer) i en specifik kontext (Vue-komponenter mot Wagtail CMS). Resultaten relaterar tydligt till Wieringas [13] guideline G5 design argument genom att belysa:

1. **Causation part:** Hur implementationerna påverkar mätbara prestandafaktorer.
2. **Valuation part:** Kvantifiering av prestandakonsekvenserna för olika användningsscenarier.

Genom denna design science-ansats har studien kunnat generera empiriskt grundad kunskap om de konkreta prestandakonsekvenserna av olika designval inom frontend-implementationer mot CMS-system.

I detta kapitel diskuteras först den tillämpade metoden och dess påverkan på studiens genomförande och validitet. Därefter följer en diskussion av resultaten i relation till forskningsfrågorna. Kapitlet avslutas med en diskussion om resultatens implikationer i ett bredare perspektiv.

5.1 Metoddiskussion

Denna studie använde en kvantitativ experimentell metod inom ramen för design science för att jämföra prestandan mellan eager loading och lazy loading i Vue-komponenter mot Wagtail CMS. Design science-ansatsen, som beskrivs av Wieringa [13], fokuserar på design och undersökning av artefakter i en kontext". I denna studie representerade implementationerna av eager loading och lazy loading de studerade artefakterna, medan Vue.js-komponenter mot Wagtail CMS utgjorde kontexten.

Design science-metodiken med dess tre cykler (problemundersökning, behandlingsdesign och behandlingsvalidering) visade sig vara lämplig för denna studie då den gav ett strukturerat ramverk för att systematiskt utforska och utvärdera laddningsstrategier. Genom att tillämpa Wieringas [13] guideline G5 design argument" kunde studien tydligt separera kausala samband (hur implementationerna påverkade prestandafaktorer) från värderingsaspekter (hur effektiva strategierna var i olika scenarier).

5.1.1 Val av forskningsdesign

Valet av en experimentell ansats med kontrollerade testmiljöer gav möjlighet att isolera och kvantifiera effekterna av olika laddningsstrategier. Denna metodik var särskilt lämplig för att besvara studiens forskningsfrågor som krävde objektiva mätvärden för prestandajämförelser. Alternativa metoder, som fallstudier i verkliga applikationer, skulle ha kunnat ge insikter om hur strategierna fungerar i praktiken men med lägre kontroll över externa faktorer.

Studiens forskningsdesign med fokus på specifika prestandaparametrar (tid för initial laddning, tid för filtrering, nätverksanrop och överförd datamängd) gav konkreta och jämförbara mätvärden. Detta tillvägagångssätt ligger i linje med Turcotte et al. [3] som använder liknande parametrar för att utvärdera laddningsstrategier i webbutveckling.

5.1.2 Val av testmiljö och datainsamling

Valet att använda en standardiserad testmiljö med Chrome DevTools för mätningar av nätverksanrop och överförd datamängd bidrog till studiens reliabilitet. I vår metodik har vi, i likhet med van Zyl et al. [4], noga dokumenterat testmiljön, inklusive använd hårdvara, programvaruversioner och nätverkskonfiguration, för att säkerställa tillförlitlighet och reproducerbarhet i resultaten.

Det finns dock metodologiska begränsningar att beakta. Testscenarierna genomfördes i en kontrollerad nätverksmiljö vilket kan skilja sig från verkliga produktionsmiljöer där variationer i nätverksbelastning, geografisk distribution och latens kan påverka resultaten. Mjelde och Opdahl [7] diskuterar laddningstidsreducerande tekniker för olika enhetstyper, och påpekar att effektiviteten av olika laddningsstrategier kan vara beroende av nätverksförhållanden.

En metodologisk styrka var implementationen av jämförbart mätningsförfarande för båda strategierna, vilket möjliggjorde direkt jämförelse. Genom att använda `performance.now()` för tidtagning och Chrome DevTools för nätverksmätningar säkerställdes att samma metodik användes för båda strategierna, vilket ökar resultatens jämförbarhet.

5.1.3 Val av mätparametrar

De valda mätparametrarna (tid för initial laddning, tid för filtrering, nätverksanrop och överförd datamängd) var lämpliga för att utvärdera prestandan hos olika laddningsstrategier. Turcotte et al. [3] använde liknande parametrar i sin studie av lazy loading i JavaScript-applikationer.

Valet att fokusera på specifika testscenarier (initial laddning och filtrering) var motiverat utifrån vanliga användningsmönster och tidigare forskning, där exempelvis Priya et al. [5] använder liknande scenarier för att utvärdera prestandaskillnader mellan olika laddningsstrategier. Dock kan begränsningen till två huvudsakliga scenarier ha utelämnat andra relevanta användningsfall, som mer komplexa filtreringar eller kombinerade operationer.

En potentiell brist är att studien inte inkluderade användarupplevelserelaterade mätningar som upplevd responsivitet. Mjelde och Opdahl [7] undersöker olika tekniker för att minska laddningstider på webbplatser, och diskuterar detta med hänsyn till användarupplevelse. Det hade kunnat vara värdefullt att komplettera de kvantitativa måtten med kvalitativa användarupplevelsemätningar.

5.2 Resultatdiskussion

I detta avsnitt diskuteras resultaten i relation till forskningsfrågorna och tidigare forskning.

5.2.1 RQ1: Strukturering av Vue-komponenter för innehållsladdning

Den första forskningsfrågan, "Hur kan Vue-komponenter struktureras för att hantera innehållsladdning med eager loading respektive lazy loading mot Wagtails standardi-

serade API?", besvarades genom implementationerna som beskrivs i metodkapitlet.

Som visades i kapitel 4.7.1 kan Vue-komponenter effektivt struktureras för båda laddningsstrategierna genom att utnyttja ramverkets reaktiva system. För eager loading-komponenter var strukturen centrerad kring omfattande initial datahämtning i `onMounted`-hooken och lokal JavaScript-filtrering, medan lazy loading-komponenter fokuserade på behovsbaserade API-anrop utlösta genom reaktiva watch-funktioner.

En viktig insikt från struktureringsarbetet är att Vue.js reaktiva system erbjuder naturliga mekanismer för båda strategierna. Som beskrivs i Vue.js officiella dokumentation [11], har ramverket inbyggt stöd för reaktiva uppdateringar vid dataförändringar, vilket gör att både eager loading (med lokala datauppdateringar) och lazy loading (med asynkrona API-anrop) kan implementeras effektivt utan behov av externa bibliotek. Detta kan ses som en fördel jämfört med vissa äldre JavaScript-ramverk som kan kräva ytterligare hantering för liknande funktionalitet.

Resultaten belyser också en viktig designövervägande: valet mellan att placera filtreringslogik på klienten (eager loading) eller servern (lazy loading). Detta val har betydande konsekvenser för kodkomplexitet och prestanda. I eager loading-implementationen hanterades filtreringen med JavaScript-metoder som `filter()` och `includes()`, medan lazy loading-implementationen delegerade filtreringen till serversidesökfunktioner genom Wagtails API-parametrar. Denna distinktion är central för utvecklare som överväger olika arkitekturella mönster.

5.2.2 RQ2: Prestandapåverkan vid val av laddningsstrategi

Den andra forskningsfrågan, "Hur påverkar valet mellan eager och lazy loading i Vue.js prestandan vid innehållsladdning mot Wagtails standardiserade API?", besvarades genom de experimentella mätningarna.

Initial laddningstid

Som visades i kapitel 4.7.2 gav lazy loading en signifikant bättre prestanda vid initial sidladdning, med en genomsnittlig förbättring på 40,8% jämfört med eager loading. Detta resultat var förväntat och stämmer väl överens med tidigare forskning, som Turcotte et al. [3], som visat att lazy loading kan minska initial laddningstid avsevärt. I deras studie rapporterar de att the code transformations proposed by Lazifier resulted in an average initial application size reduction of 36.2%, which caused applications to speed up initial load time by 29.7% on average."

Den observerade skillnaden kan förklaras av den fundamentala skillnaden i strategiernas natur: eager loading hämtar all potentiellt användbar data omedelbart, medan lazy loading endast hämtar minimal data för initial rendering.

Filtreringstid

Ett anmärkningsvärt resultat är den dramatiska skillnaden i filtreringstid, där eager loading visade sig vara cirka 400 gånger snabbare än lazy loading. Detta resultat är mer extremt än vad som observerades i Priya et al. [5], där deras mätningar av databashämtningar visade att eager loading minskade bearbetningstiden med cirka 96% vid eager loading. Den större skillnaden i denna studie kan förklaras av att Priyas implementation behöll viss databasinteraktion även vid eager loading, medan vår eager loading-implementation utförde filtrering helt i klientens minne.

Filtreringsoperationen i eager loading-implementationen går alltså att implementera helt lokalt utan nätverkskommunikation, medan lazy loading-implementationen

involverar fullständig nätverkskommunikation med servern för varje filtrering. Denna observation belyser en viktig avvägning: medan lazy loading optimerar initial laddningstid, kan den avsevärt försämra responsiviteten vid användarinteraktioner som kräver datafiltrering.

En intressant observation är att filtreringstiden för eager loading minskade något med färre sökträffar (från 0,44 ms för 13 träffar till 0,36 ms för 1 träff), vilket indikerar en linjär komplexitet för filtreringsalgoritmen. För lazy loading var detta mönster mindre tydligt, vilket tyder på att nätverkskommunikationen dominerar över den faktiska filtreringsoperationen på serversidan.

Nätverksanvändning

Resultaten visar att lazy loading använder 9,4% mindre data vid initial laddning men kräver ytterligare nätverksanrop vid varje filtrering. Denna skillnad bör ses i samband med renderingprocessen, där eager loading både renderar sidan och hämtar all data vid initial laddning, medan lazy loading delar upp hämtningen av data över tid.

En viktig observation är att den totala dataöverföringen för lazy loading förblir lägre än för eager loading efter tre sökningar, men denna fördel minskar med varje ytterligare sökning. Detta resultat är särskilt relevant för applikationer där användningsmönstret påverkar val av optimal strategi. För applikationer där användare typiskt gör få sökningar per session, kan lazy loading vara mer fördelaktigt för bandbreddsanvändning, medan applikationer med intensiva sökmönster kan gynnas av eager loading på lång sikt.

5.2.3 Jämförelse med tidigare forskning

Resultaten från denna studie bekräftar generellt tidigare forskningsfynd, men med några viktiga nyanser. Turcotte et al. [3] rapporterade att lazy loading kunde minska initial applikationsstorlek med i genomsnitt 36,2% och laddningstiden med 29,7%, vilket är i linje med vår observation av 40,8% snabbare initial laddning med lazy loading.

Van Zyl et al. [4] har studerat optimeringsmetoder för Object Relational Mapping (ORM) verktyg, vilket visar att olika laddningsstrategier kan ha varierande prestanda beroende på användningskontext. Deras resultat indikerar att lazy loading ger bättre prestanda för vissa operationer, medan eager loading kan vara fördelaktigt för andra, vilket överensstämmer med vår observation av överlägsen filtreringsprestanda med eager loading.

Mjelde och Opdahl [7] har undersökt olika tekniker för att minska laddningstider på webbplatser för olika enheter, vilket kompletterar våra observationer om lägre dataöverföring vid initial laddning med lazy loading. Dock belyser våra resultat att denna fördel minskar med upprepad interaktion, ett mönster som inte tydligt behandlades i deras studie.

Resultaten från denna studie bygger på och utvidgar befintlig kunskap genom att tillhandahålla detaljerade empiriska data om prestandaavvägningarna mellan eager loading och lazy loading i den specifika kontexten av Vue-komponenter mot Wagtail CMS.

5.3 Implikationer för praktiken

Resultaten från denna studie har flera praktiska implikationer för webbutvecklare som arbetar med Vue.js och CMS-system som Wagtail.

För det första ger studien riktlinjer för när respektive laddningsstrategi är lämplig. För applikationer där snabb initial laddning är prioriterad och där användarinteraktioner är begränsade, kan utifrån studien lazy loading rekommenderas. Exempel på sådana applikationer kan vara informativa webbplatser med begränsad användarinteraktion eller första visningen av en komplex applikation där upplevd snabbhet är kritisk.

För applikationer där användarinteraktion är intensiv, särskilt med upprepade filtreringar eller sökningar efter sidladdning, rekommenderas eager loading. Exempel kan vara applikationer för dataanalys där användare förväntas göra flera filtreringar och sökningar under en session.

Studien belyser också vikten av att överväga användningsmönster vid val av laddningsstrategi. Genom att analysera typiska användarsessioner, särskilt antalet förväntade filtreringar eller sökningar per session, kan utvecklare göra mer informerade beslut om lämplig strategi. För applikationer med varierade användningsmönster kan en hybrid lösning övervägas, där viss data laddas eager medan annan laddas lazy.

Resultatens tydliga indikation på att eager loading ger överlägsen filtreringsprestanda men sämre initial laddningstid, och vice versa för lazy loading, underbygger vikten av att prioritera mellan initial laddningstid och interaktionsresponsivitet. Denna avvägning är central i modern webbutveckling och bör göras med hänsyn till applikationens specifika krav och användarnas förväntningar.

5.4 Begränsningar och hot mot validitet

Studien har flera begränsningar som bör beaktas vid tolkning av resultaten.

5.4.1 Intern validitet

Den interna validiteten, eller trovärdigheten av kausala samband inom studien, kan påverkas av flera faktorer:

Implementationsspecifika faktorer: De specifika implementationerna av eager loading och lazy loading kan ha påverkat resultaten. Alternativa implementationsstrategier för samma koncept kan potentiellt ge annorlunda resultat.

Nätverksförhållanden: Trots att studien använde en realistisk klient-server-modell, kan de specifika nätverksförhållandena under testerna ha påverkat resultaten. I verkliga produktionsmiljöer kan varierande nätverksförhållanden leda till större variationer i prestanda, särskilt för lazy loading som är mer beroende av nätverkskommunikation.

Mätinstrumentering: Själva mätinstrumenteringen, särskilt användningen av `performance.now()` för tidtagning, kan teoretiskt påverka prestandan marginellt. Denna påverkan bör dock vara konsekvent mellan implementationerna.

5.4.2 Extern validitet

Den externa validiteten, eller möjligheten att generalisera resultaten till andra kontexter, begränsas av:

Specifik teknikstack: Studien fokuserade specifikt på Vue.js komponenter mot Wagtail CMS. Resultaten kan inte direkt generaliseras till andra JavaScript-ramverk eller CMS-system utan vidare undersökning.

Begränsad datamängd: Testdatat bestod av 200 artiklar, vilket är en relativt liten datamängd jämfört med vissa produktionssystem. För system med signifikant större datamängder kan prestandaskillnaderna vara ännu mer uttalade.

Förenklade testscenarier: Testscenarierna fokuserade på grundläggande laddning och filtrering, där filtrering utfördes simultant över både utgåvor och artiklar. Trots denna flerdimensionella filtrering var testscenarierna relativt standardiserade.

5.4.3 Reliabilitet

Reliabiliteten, eller möjligheten att reproducera resultaten, stärktes genom:

Standardiserade testförhållanden: Alla tester utfördes under identiska förhållanden med samma hårdvara, webbläsare och nätverkskonfiguration.

Multipla testomgångar: Varje test upprepades tio gånger för att minska slumpmässiga variationer.

Dokumenterad metodologi: Testmetodologin dokumenterades i detalj för att möjliggöra replikering.

Trots dessa åtgärder kan vissa faktorer ha påverkat reliabiliteten, såsom variationer i nödvändiga bakgrundsprocesser och oförutsägbara systemhändelser. Även med vår standardiserade testmiljö och användning av nya inkognitofönster för varje test, kan exempelvis bakgrundsprocesser i operativsystemet som Llanos [6] demonstrerar ha påverkat CPU-tillgängligheten mellan testomgångar. Zagarese et al. [10] belyser hur variationer i nätverkslatens kan påverka mätningar av nätverksberoende operationer. Detta understryker vikten av upprepade mätningar och användning av medelvärden, vilket var en central del av vår metodik.

5.5 Etiska överväganden

Denna studie involverade inga mänskliga försökspersoner eller känsliga data, vilket minimerar de etiska implikationerna. Dock bör följande etiska aspekter beaktas:

Resursanvändning: Valet mellan eager loading och lazy loading påverkar resursanvändning, vilket har implikationer för energiförbrukning och miljöpåverkan.

Tillgänglighet: Laddningsstrategier kan påverka tillgängligheten för användare med begränsad bandbredd eller med olika typer av funktionsnedsättningar. Studien undersökte inte specifikt hur strategierna påverkar tillgänglighet, men detta är en viktig aspekt att beakta vid implementation i verkliga system.

Dessa etiska överväganden bör tas i beaktande vid tillämpning av studiens resultat i produktionsmiljöer.

Kapitel 6

Slutsatser och framtida arbete

Detta kapitel presenterar studiens huvudsakliga slutsatser baserat på de empiriska resultaten. Först sammanfattas studiens bidrag till kunskapsområdet, följt av de viktigaste slutsatserna från jämförelsen mellan eager loading och lazy loading i Vue-komponenter mot Wagtail CMS. Därefter diskuteras resultaten ur ett bredare perspektiv, med fokus på aspekter som inte var uppenbara vid projektets början samt hur andra forskare tolkat liknande frågeställningar. Kapitlet avslutas med obesvarade frågor och förslag på framtida forskningsriktningar.

6.1 Övergripande bidrag

Denna studie har genom en systematisk design science-ansats utforskat hur olika laddningsstrategier påverkar prestandan i Vue-komponenter mot Wagtail CMS. Genom att utveckla och utvärdera två distinkta implementationer, eager loading och lazy loading, har studien bidragit med empiriskt grundad kunskap om prestandaimplikationerna av dessa designval. Design science-metodiken, som beskrivs av Wieringa [13] som design och undersökning av artefakter i en kontext", har gett ett strukturerat ramverk för att undersöka dessa implementationer i den specifika kontexten av Vue.js och Wagtail CMS.

Studiens huvudsakliga bidrag är:

1. **Empirisk kunskap om prestandaavvägningar:** Studien har kvantifierat exakta prestandaskillnader mellan eager loading och lazy loading under olika användningsscenarier, vilket ger utvecklare ett solitt beslutsunderlag.
2. **Implementationsmönster för Vue-komponenter:** Genom att konkretisera hur Vue-komponenter kan struktureras för respektive laddningsstrategi mot Wagtail CMS, bidrar studien med praktiskt tillämpbara mönster för webbutvecklare.
3. **Kontextbaserade designrekommendationer:** Studien visar att den optimala strategin är starkt beroende av användningskontexten, vilket understryker vikten av att basera designbeslut på förväntade användningsmönster.

Dessa bidrag adresserar direkt forskningsfrågorna om hur Vue-komponenter kan struktureras för olika laddningsstrategier (RQ1) och hur dessa strategier påverkar prestandan (RQ2).

6.2 Slutsatser från prestandajämförelsen

Baserat på de empiriska resultaten från studiens systematiska prestandamätningar kan följande huvudslutsatser dras:

6.2.1 Initial laddningstid och användarupplevelse

Den empiriska undersökningen visar tydligt att lazy loading ger betydligt snabbare initial sidladdning, med en genomsnittlig förbättring på 40,8% jämfört med eager loading (271,4 ms vs 458,3 ms). Denna signifikanta skillnad bekräftar den grundläggande principen bakom lazy loading: att minimera den initiala datamängden för snabbare första rendering.

Resultaten överensstämmer med tidigare forskning av Turcotte et al. [3] som rapporterade en genomsnittlig förbättring av initial laddningstid på 29,7% med lazy loading, medan vår studie visar en något större förbättring på 40,8% i vårt specifika användningsfall med Vue och Wagtail. Denna skillnad kan förklaras av olika implementationskontexter och teknologiska förutsättningar.

För applikationer där användarens första intryck är kritiskt, exempelvis innehållstunga webbplatser där användare snabbt vill få en överblick, erbjuder lazy loading en betydande fördel. Detta bekräftar ett av de fundamentala designprinciperna inom modern webbutveckling: att optimera för perceived performance där användarens upplevelse av hastighet prioriteras.

6.2.2 Interaktionsresponsivitet och datafiltrering

Ett anmärkningsvärt resultat är den dramatiska skillnaden i filtreringstid mellan strategierna. Eager loading visade sig vara cirka 400 gånger snabbare vid filtrering (under 0,5 ms jämfört med cirka 155-161 ms för lazy loading). Denna extrema skillnad understryker eager loadings styrka vid användarinteraktioner efter initial laddning.

Eager loading-implementationen kunde utföra filtrering helt lokalt utan nätverkskommunikation, medan lazy loading-implementationen krävde fullständig nätverkskommunikation med servern för varje filtrering. Detta belyser en kritisk avvägning: medan lazy loading optimerar initial laddningstid, kan den avsevärt försämra responsiviteten vid användarinteraktioner.

Ett intressant mönster framträdde också när vi undersökte filtreringstider i relation till resultatmängdens storlek. För eager loading minskade filtreringstiden med färre sökträffar (0,44 ms för 13 träffar jämfört med 0,36 ms för 1 träff), vilket indikerar en linjär komplexitet för filtreringsalgoritmen. För lazy loading var detta mönster mindre tydligt, vilket tyder på att nätverkskommunikationen dominerar över själva filtreringsoperationen.

Denna insikt är särskilt värdefull för utveckling av datatunga applikationer där användare förväntas göra flera filtreringar och sökningar under en session. För sådana applikationer kan eager loading vara överlägset trots den längre initiala laddningstiden.

6.2.3 Nätverkseffektivitet och bandbreddsanvändning

Analysen av nätverksanvändning visar att lazy loading använder 9,4% mindre data vid initial laddning (1082 KB jämfört med 1194 KB), men kräver ytterligare nätverksanrop vid varje filtrering. Detta mönster belyser en fundamental skillnad i hur de två strategierna fördelar nätverksbelastningen över tid.

För särskilda användningskontexter, som mobilapplikationer där bandbredd kan vara begränsad eller kostsam, blir dessa insikter särskilt relevanta. Vår studie visar att den totala dataöverföringen för lazy loading förblir lägre än för eager loading efter tre sökningar (1098,94 KB jämfört med 1194 KB), men denna fördel minskar med varje ytterligare sökning. Detta ger en tydlig indikation på när respektive strategi är att föredra baserat på förväntade användningsmönster.

6.2.4 Total användarupplevelse över tid

Den kombinerade analysen av initial laddning och filtreringstid ger en nyanserad bild av strategiernas effektivitet över en hel användarsession. För en användare som besöker sidan och utför en sökning är lazy loading cirka 30,3 ms (6,6%) snabbare totalt. Efter två sökningar blir dock eager loading redan 126,3 ms (21,6%) snabbare totalt.

Denna brytpunkt där eager loading blir den effektivare strategin utgör ett konkret beslutskriterium för utvecklare. För applikationer där användare typiskt utför flera interaktioner per session framstår eager loading som det optimala valet trots den initiala prestandaförsämringen.

Detta resultat överensstämmer med van Zyl et al. [4] som menar att valet mellan lazy loading och eager loading måste vara kontextberoende. Vår studie bidrar med en mer precis kvantifiering av exakt när denna brytpunkt inträffar, för Vue-komponenter mot Wagtail CMS.

6.3 Resultat ur ett bredare perspektiv

När vi betraktar studiens resultat ur ett bredare perspektiv framträder aspekter som inte var uppenbara vid projektets planeringsstadium. Ett sådant perspektiv handlar om den accelererande komplexiteten i moderna webbapplikationer och hur detta påverkar designbeslut kring datahantering.

6.3.1 Balansen mellan teknisk optimering och användarupplevelse

Resultaten från vår studie pekar på en fundamental spänning i modern webbutveckling: konflikten mellan teknisk optimering och användarnära upplevelsefaktorer. När vi inledningsvis designade studien låg fokus primärt på objektiva prestandamätningar. Genom forskningsprocessen har vi dock insett att den optimala laddningsstrategin inte kan avgöras enbart genom att mäta millisekunder. Den subjektiva användarupplevelsen, framför allt den upplevda responsiviteten, är lika central.

Detta synliggör behovet av en mer holistisk syn på prestandaoptimering där både kvantitativa mätningar och kvalitativa användarupplevelsefaktorer vägs in. Trots att eager loading visar sig vara betydligt snabbare vid interaktioner, kan den upplevda tröskeln vid initial laddning påverka användarens benägenhet att fortsätta använda applikationen överhuvudtaget.

6.3.2 Alternativa forskningsfrågor

Genom arbetet med källmaterialet har flera intressanta alternativa forskningsfrågor identifierats som inte rymdes inom ramen för denna studie, eller tydligt tycks ha utforskats i andra studier:

1. **Hur bör användarnära faktorer påverka valet av laddningsstrategi?** - Turcotte et al. [3] undersöker tekniska aspekter av lazy loading, men berör bara ytligt hur användarupplevelsefaktorer bör vägas in. En fördjupad studie kring detta skulle kunna ge värdefulla insikter.
2. **Vilka hybridstrategier kan kombinera fördelarna från både eager och lazy loading?** - Van Zyl et al. [4] indikerar att kontextberoende strategival är att föredra, men utforskar inte specifika implementationer av dynamiskt växlande strategier.

6.3.3 Studiens plats i ett bredare sammanhang

Vår studie bidrar till ett bredare forskningsområde kring optimering av webbapplikationer i en tid när gränssnittsarkitekturer blir alltmer komplexa. Andra forskare har placerat detta område i varierande kontexter:

Priya et al. [5] placerar laddningsstrategier i kontexten av databasoptimering och undersöker dem primärt utifrån databasextrahering i backend. Vår studie kompletterar detta perspektiv genom att undersöka strategierna mer utifrån ett frontendperspektiv inom komponentdesign.

Mjelde och Opdahl [7] sätter laddningsstrategier i samband med olika enhetstyper och nätverksförhållanden, vilket breddar perspektivet bortom ren prestandaoptimering till att inkludera adaptiva strategier baserade på användarens kontext. Användarkontext är också något som diskuteras i föreliggande studie, men enhetstyper har inte undersökts.

6.3.4 Alternativa slutsatser

Andra forskare har dragit delvis annorlunda slutsatser i relaterade studier, vilket ger nya perspektiv på våra forskningsfrågor:

Van Zyl et al. [4] betonar caching som en central optimeringsteknik, medan vår studie fokuserar mer på de ursprungliga laddningsstrategierna. Detta pekar på potentialen i att kombinera olika optimeringsmetoder för att ytterligare förbättra prestandan.

Mjelde och Opdahl [7] lyfter fram att lazy loading är särskilt fördelaktigt för mobila enheter, vilket går utöver föreliggande studies mer generella slutsatser. Detta antyder att beaktande av enhetstyp bör vara en faktor vid val av laddningsstrategi.

Medan vår studie visar en brytpunkt där eager loading blir mer effektivt efter ett fåtal interaktioner, drar Priya et al. [5] slutsatsen att eager loading generellt är att föredra för alla databashämtningsscenarier. Detta är rimligt då de uteslutande undersöker backend-prestanda. Denna kontrast belyser dock vikten av att beakta hela kedjan av användande i en specifik kontext.

6.4 Obesvarade aspekter och framtida arbete

Trots studiens omfattande resultat kvarstår flera obesvarade aspekter som förtjänar vidare utforskning:

6.4.1 Obesvarade aspekter

1. **Användarupplevelsens betydelse** - Studien fokuserade på objektiva prestandamätningar men behandlade inte den subjektiva användarupplevelsen. Hur påverkas användarnas upplevda responstid av de olika strategierna?
2. **Skalbarhet vid större datamängder** - Experimentet utfördes med en begränsad datamängd. Hur förändras prestandaförhållandet mellan strategierna när datamängden växer?
3. **Påverkan på serversidans resurser** - Studien fokuserade på klientsidans prestanda men undersökte inte hur de olika strategierna påverkar serverbelastning.
4. **Prestanda under varierade nätverksförhållanden** - Testerna utfördes under optimala nätverksförhållanden. Hur presterar strategierna under varierande latens och bandbredd?

6.4.2 Framtida arbete

För att adressera dessa obesvarade aspekter och vidareutveckla forskningsområdet föreslås följande riktningar för framtida arbete:

1. **Användarcentrerade studier** - Genomföra användarstudier som mäter den upplevda prestandan av olika laddningsstrategier, med fokus på hur användare subjektivt upplever responsivitet och väntetider.
2. **Dynamiska system** - Utveckla och utvärdera laddningssystem som dynamiskt växlar mellan eager och lazy loading baserat på användningsmönster, nätverksförhållanden och enhetstyp.
3. **Energieffektivitet och resursförbrukning** - Analysera hur valet av laddningsstrategi påverkar energiförbrukning på klientens enhet, vilket blir allt viktigare i mobila sammanhang och för hållbar mjukvaruutveckling.

Dessa forskningsriktningar skulle bygga vidare på grunderna som lagts i denna studie och samtidigt bredda förståelsen för hur laddningsstrategier kan optimeras i en växande och successivt mer komplex webbutvecklingsmiljö.

6.5 Avslutande reflektion

Denna studie har genom design science-metodiken bidragit med empiriskt grundad kunskap om prestandaimplikationerna av olika laddningsstrategier i Vue-komponenter. Resultaten visar tydligt att valet mellan eager loading och lazy loading har väsentlig inverkan på både initial laddningstid och interaktionsresponsivitet, med en identifierad brytpunkt där eager loading blir fördelaktigt efter ett fåtal användarinteraktioner.

Denna insikt understryker vikten av att webbutvecklare gör medvetna designval baserade på förväntade användningsmönster snarare än att följa generella best practices utan hänsyn till kontexten. För applikationer där användare tenderar att utföra få interaktioner per session är lazy loading att föredra, medan eager loading är överlägset för applikationer med intensiv användarinteraktion.

Genom att placera våra resultat i relation till tidigare forskning framträder också behovet av ett mer holistiskt perspektiv på prestandaoptimering, där tekniska mätningar kombineras med användarcentrerade analyser.

Studien avslutar sig själv där den började, med ambitionen att bidra till effektivare Vue-komponenter i CMS-baserade applikationer, men öppnar samtidigt för nya forskningsriktningar som kan fördjupa förståelsen för hur moderna webbapplikationer kan optimeras för att möta dagens höga prestandakrav.

Referenser

- [1] Vue.js guide: Introduction. <https://vuejs.org/guide/introduction.html>, 2023. Accessed: 25 March 2025.
- [2] Wagtail Documentation. <https://docs.wagtail.org/en/stable/>, 2023. Accessed: 25 March 2025.
- [3] A. Turcotte, S. Gokhale, and F. Tip. Increasing the responsiveness of web applications by introducing lazy loading. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 459–470, 2023.
- [4] P. van Zyl, D. G. Kourie, L. Coetzee, and A. Boake. The influence of optimisations on the performance of an object relational mapping tool. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, pages 150–159, 2009.
- [5] M. D. Priya, B. Sabarinathan, and M. Surya. Optimization of web applications using eager loading. In *International Conference on Intelligent Computing and Applications*, pages 711–722, 2021.
- [6] Diego R. Llanos. Tpc-c-uva: an open-source tpc-c implementation for global performance measurement of computer systems. *SIGMOD Record*, 35(4):6–15, 2006.
- [7] E. Mjelde and A.L. Opdahl. Load-time reduction techniques for device-agnostic web sites. *Communications of the ACM*, 62(4):328–346, 2019.
- [8] N. Tuaycharoen, V. Prodpran, and B. Srithong. Classschedule: a web-based application for school class scheduling with real-time lazy loading. In *2018 5th International Conference on Business and Industrial Research (ICBIR)*, pages 210–214, 2018.
- [9] Y. Kargin, M. Ivanova, Y. Zhang, S. Manegold, and M. Kersten. Lazy etl in action: Etl technology dates scientific data. *Proceedings of the VLDB Endowment*, 6(12):1286–1289, 2013.
- [10] Q. Zagarese, G. Canfora, E. Zimeo, and F. Baude. Enabling advanced loading strategies for data intensive web services. In *2012 IEEE 19th International Conference on Web Services*, pages 480–487, 2012.
- [11] Vue.js guide: Performance optimization. <https://vuejs.org/guide/best-practices/performance.html>, 2023. Accessed: 25 March 2025.
- [12] H. Keuning, B. Heeren, and J. Jeuring. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*, pages 562–568, 2021.

- [13] R. Wieringa. Design science as nested problem solving. In *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, 2009.

