Examensarbete

HÖGSKOLAN I SKÖVDE

ANALYS AV LADDNINGSTIDER FÖR RAMVERKEN: REACT, ANGULAR OCH INFERNO

En prestandajämförelse med fokus på laddningstider för e-handelssidor.

ANALYSIS OF LOADING TIMES FOR THE FRAMEWORKS: REACT, ANGULAR AND INFERNO

A performance comparison with a focus on loading times for e-commerce websites.

Examensarbete för kandidatexamen med huvudområde Informationsteknologi

Grundnivå 30 Högskolepoäng
Vårtermin 2024

Wilmer Lundquist

Handledare: Johan Bjurén
Examinator: Henrik Gustavsson

Sammanfattning

Denna rapport bygger på ett experiment som syftar till att undersöka och jämföra laddningstiderna för tre olika ramverk, React, Angular och Inferno, på en e-handelssida. Målet är att identifiera vilket ramverk som presterar kortast laddningstiderna, vilket kan vara av särskild vikt för e-handelsföretag. Tidigare studier, såsom Zhou, Giyane & Nyasha (2013), har visat att majoriteten av internetanvändarna tenderar att överge en webbplats om den inte laddar inom genomsnittet av 8 sekunder. Därmed är det avgörande för e-handelsplatser att optimera sina laddningstider för att bibehålla användarnas engagemang och minska att de lämnar webbplatsen.

Genom att utveckla tre e-handelssidor med varsitt ramverk, React, Angular och Inferno kommer flera mätningar att genomföras för att ta reda på vilket av ramverken som presterar de lägsta laddningstiderna över en varierande datamängd. Resultatet från mätningarna kommer att analyseras och visualiseras i form av olika grafer.

Nyckelord: Ramverk, React, Angular, Inferno, laddningstider

Innehållsförteckning

1 Bakgrund

När det kommer till webbapplikationer idag är kravet på laddningstider stort. Användare förväntar sig snabb respons och en smidig användarupplevelse. Det är särskilt viktigt inom områden som e-handel, där laddningstiden är en av de mest avgörande faktorerna för dess användbarhet. De flesta internetanvändare kommer helt enkelt att välja att stänga ner webbplatsen om den inte lyckas ladda inom genomsnittet av 8 sekunder (Zhou, Giyane & Nyasha, 2013). Långsamma laddningstider kan leda till förlorade kunder och därmed påverka konverteringsgraden negativt för företaget (Zhou, Giyane & Nyasha, 2013). Vilket understryker behovet av att välja ett ramverk som erbjuder snabba laddningstider.

Idag används ofta JavaScript till webbapplikationer, som är ett mångsidigt programmeringsspråk. Javascript möjliggör interaktivitet och dynamik på webbsidor, vilket kan bidra till en förbättrad användarupplevelse. Genom att använda JavaScript kan utvecklare skapa dynamiskt innehåll och interaktiva element, vilket kan ge användare en responsiv och interaktiv användarupplevelse. Den ökande popularitеten av JavaScript har lett till en mängd olika JavaScript-ramverk som syftar till att hjälpa utvecklare att ta itu med programmeringsuppgifter. Därefter har antalet JavaScript-ramverk stigit snabbt upp till tusentals versioner (Pano, Graziotin & Abrahamsson, 2018). Varje ramverk bär med sig sina unika designfilosofier, syntaxregler samt varierande laddningstider. Ramverken uppdateras ständigt med nya varianter och versioner, vilket enligt Nakajima, Matsumoto & Kusumoto (2019) kan bidra till en risk att den tillgängliga informationen om Javascript-ramverk blir föråldrad i takt med att både ramverken och webbläsarna uppdateras och förnyas. Mångfalden av ramverk gör valet komplicerat för utvecklare, då det är viktigt att hitta det som passar deras specifika projekt. Speciellt med tanke på att den föråldrade informationen kan utgöra ett hinder för utvecklare att förstå och använda ramverkens egenskaper effektivt.

1.2 Ramverk

Ett ramverk inom webbutveckling är en fördefinierad struktur eller plattform som ger utvecklare en ram eller grund att bygga sina webbapplikationer på. Det fungerar som en samling av verktyg, bibliotek och regler som underlättar och standardiserar processen att skapa skalbara och dynamiska applikationer för webben. Ramverk används för att snabba upp utvecklingen, främja bästa praxis och möjliggöra en enhetlig struktur över olika delar av webbapplikationen. Ramverken erbjuder också en uppsättning tekniker, såsom kompilering, rendering, layout och målning, för att optimera prestandan och användarupplevelsen på webbapplikationerna (P. Singh, M. Srivastava, M. Kansal, A. P. Singh, A. Chauhan and A. Gaur, 2023). I dagens mångfald av ramverk genereras innehållet på olika sätt, vilket kan påverka laddningstiderna av webbapplikationer. Till exempel skapar vissa ramverk som React en Virtual Document Object Model (DOM) för att effektivt hantera och optimera uppdateringar av användargränssnittet (Roldán, 2023). Medan andra ramverk som Angular använder tekniker som Change Detection för att övervaka förändringar i tillståndet och synkronisera användargränssnittet direkt med den verkliga DOM:en (Uluca, 2024).

1.2.1 React

React är ett JavaScript-bibliotek som släpptes 2013 av Facebooks (Meta) utvecklingsteam. React utvecklades för att underlätta skapandet av användargränssnitt genom att främja en komponentbaserad arkitektur. React är skrivet i JavaScript och använder sig av en deklarativ programmeringsmodell. React tillhandahåller ett effektivt sätt att bygga användargränssnitt (UI) genom att tillåta utvecklare att beskriva hur UI ska se ut baserat på dess nuvarande tillstånd. React är en av de mest populära biblioteken för att skapa användargränssnitt. Det är väl känt för att vara snabbt, tack vare sitt sätt att arbeta med Virtual DOM, vilket möjliggör uppdateringar av användargränssnittet genom att minimera antalet verkliga DOM-uppdateringar (Roldán, 2023). Utvecklare kan dra nytta av ett stort utbud av tillägg och verktyg som hjälper till att förenkla utvecklingsprocessen. React kännetecknas också av dess stora community och stöd från företag som Meta.

```
import React, { Component } from 'react';

class MyComponent extends Component {
  render() {
    return (
      <div>
        <h1>React</h1>
        <p>Paragraph</p>
      </div>
    );
  }
}

export default MyComponent;
```

Figur 1. Kodexempel i React för att skapa en rubrik och en paragraf.

1.2.2 Angular

Angular skapades av ett utvecklingsteam på Google och släpptes 2010, denna version av Angular var skapad med JavaScript. Det skapades för att separera logik och vy enligt Model-View-Controller(MVC) stil mönstret och byggdes på tanken om deklarativ programmering. Angular är ett ramverk utvecklat av Google och är en omskrivning av AngularJS. Angular annonserades 2014 och släpptes i en tidig version 2016. Angular är skrivet i TypeScript och har nu ett större fokus på objektorienterad programmering. En viktig funktion i Angular är dess inbyggda Change Detection, som övervakar asynkrona händelser och triggas när det finns en förändring som kan påverka användargränssnittet. Change Detection möjliggör en automatisk uppdatering av användargränssnittet när data eller tillstånd ändras i applikationen. Angular erbjuder en omfattande plattform för webbutveckling. Plattformen inkluderar ett komponentbaserat ramverk för att bygga skalbara webbapplikationer, en samling välintegrerade bibliotek som täcker en mängd olika funktioner, inklusive routing, formulärhantering, klient och server kommunikation. Dessutom finns det en uppsättning utvecklarverktyg som hjälper till att utveckla, bygga, testa och uppdatera kod. Angular refereras till som ett plattformsoberoende utvecklingskit av dess utvecklare (Uluca, 2024).

```
<!DOCTYPE html>
<html lang="en" ng-app="myApp">
<head>
  <meta charset="UTF-8">
  <title>AngularJS Component</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"
></script>
</head>
<body>

<div ng-controller="MyController">
  <h1>AngularJS</h1>
  <p>Paragraph in an AngularJS component.</p>
</div>

<script>
  var app = angular.module('myApp', []);
  app.controller('MyController', function($scope) {
    // Controller logic here
  });
</script>

</body>
</html>
```

Figur 2. Kodexempel i Angular för att skapa en rubrik och en paragraf.

1.2.3 Inferno

Inferno är ett JavaScript-bibliotek som delar många likheter med React och erbjuder en plattform för att utveckla användargränssnitt. Inferno använder precis som React en virtual DOM för att göra uppdateringar av användargränssnittet. Genom att skapa en virtuell representation av DOM-trädet och jämföra med det verkliga DOM-trädet kan den identifiera och tillämpa nödvändiga ändringar utan att behöva manipulera den verkliga DOM:en direkt. Vilket resulterar i en minskad mängd verkliga DOM-manipulationer (Roldán, 2023). Inferno, precis som React, förespråkar en väg för dataflöde där data rör sig enbart i en riktning. Genom ett enkelspårigt dataflöde går det enkelt att spåra och hantera dataflödet i en applikation, vilket resulterar i ökad förutsägbarhet och lättare felsökning. Inferno använder sig likt React av JSX (JavaScript XML) för att definiera användargränssnittets struktur och utseende inom JavaScript-koden. JSX är ett syntaktiskt socker för att skriva HTML-liknande element och JavaScript-kod i React. Det gör det lättare att skapa komponenter genom att blanda HTML och JavaScript i samma fil. Det går att använda Javascript-uttryck och logik direkt i JSX för att dynamiskt generera innehåll baserat på variabler eller villkor (Roldán, 2023).

```
import Inferno from 'inferno';
import Component from 'inferno-component';

class MyComponent extends Component {
  render() {
    return (
      <div>
        <h1>Inferno</h1>
        <p>paragraph in an Inferno component.</p>
      </div>
    );
```

```
  }
}

export default MyComponent;
```

Figur 3. Kodexempel i Inferno för att skapa en rubrik och en paragraf.

1.2.4 Vue

Vue.js är ett JavaScript-ramverk som först släpptes 2014 och har sedan dess blivit ett populärt ramverk bland utvecklare. Ramverket används för att bygga och leverera webbapplikationer till användare, och möjliggör skapandet av reaktiva användargränssnitt för frontend-applikationer. Vue erbjuder en organiserad mekanism för att strukturera och bygga webbapplikationer samt fungerar som en omvandlare som översätter Vue-kod till vanliga webbteknologier som HTML, CSS och JavaScript. Det följer även MVVM-mönstret, där ViewModel binder data mellan View och Model och möjliggör direkt kommunikation mellan view och model, vilket främjar komponentens reaktivitet. Vue har blivit ett populärt val för många utvecklare. Med sin omfattande dokumentation och användbara verktyg är Vue en plattform som passar för att bygga moderna och prestanda optimerade webbapplikationer (Shavin, 2023).

1.3 Virtual DOM vs Change Detection

Konceptet av Virtual DOM är att skapa en virtuell kopia av webbsidans struktur i minnet, separerad från den faktiska webbläsar-DOM:en. Virtual DOM är en teknik som React och Inferno använder sig av för att hantera och optimera uppdateringar av användargränssnittet. Genom att först uppdatera denna virtuella representation och sedan jämföra förändringarna med den verkliga DOM:en kan ramverken selektivt uppdatera endast de nödvändiga delarna av webbsidan. Av den orsaken minskar onödiga operationer på den faktiska DOM:en (Roldán, 2023). Change detection är en teknik som används av Angular och andra liknande ramverk för att övervaka användarinteraktioner och andra händelser för att avgöra om applikationen behöver reagera på förändringar. Det innebär att kontinuerligt jämföra det aktuella tillståndet för DOM-elementen med det tidigare tillståndet och vid behov uppdatera användargränssnittet (Uluca, 2024). Skillnaden mellan Virtual DOM och Change detection är att Change Detection övervakar förändringar i applikationens tillstånd och uppdaterar användargränssnittet direkt när förändringar upptäcks. Medan Virtual DOM använder en virtuell representation av DOM-trädet för att jämföra och uppdatera endast de delar av användargränssnittet som har ändrats.

1.4 Lämpliga ramverk

Enligt Pano, Graziotin och Abrahamsson (2018) är faktorer som prestanda, storlek, användarvänlighet, inlärningsbarhet och gemenskapens storlek av central betydelse när utvecklare beslutar vilket ramverk de ska använda till sitt projekt. Faktorerna har spelat en betydande roll i valet av ramverk i denna studie. Faktorn gemenskapens storlek har varit kopplad till två mer välkända ramverk, React och Angular, medan andra faktorer som prestanda, användarvänlighet och inlärningsbarhet också har vägts in. Vidare kommer studien att utvidgas med inkluderingen av ett nyare ramverk med en mindre gemenskaps storlek, Inferno.js, där fördelarna snarare ligger i prestanda och storlek - två avgörande faktorer enligt Pano, Graziotin och Abrahamsson (2018).

I en liknande undersökning där React, Angular och vue.js undersöktes för att bedöma deras prestanda. Bidrog denna studie också till valet av ramverk och en breddad förståelse för hur jämförelsen av ramverk kan gå till i denna studie, där en jämförelse mellan React, Angular och Inferno ska genomföras. Där valet av Inferno liknar den tidigare studiens val av Vue, där det förklarades att Vue är ett nyare ramverk som fångat rampljuset på grund av dess snabba prestanda (P. Singh, M. Srivastava, M. Kansal, A. P. Singh, A. Chauhan and A. Gaur 2023).

2 Problemformulering

Dagens utvecklare står inför flera utmaningar för att skapa applikationer med korta laddningstider, där några viktiga problem kommer att beskrivas i denna del. Den ökande populariteten av JavaScript har lett till en mängd olika JavaScript-ramverk som syftar till att hjälpa utvecklare att ta itu med programmeringsuppgifter. Därefter har antalet JavaScript-ramverk stigit snabbt upp till tusentals versioner (Pano, Graziotin & Abrahamsson 2018).

Det första problemet uppstår på grund av det breda utbudet av tillgängliga ramverk. Med det ökande antalet ramverk, finner webbutvecklare ofta det svårt att välja det mest lämpliga ramverket att använda. För webbutvecklare är det avgörande att välja ett ramverk som matchar deras behov och som erbjuder kod av hög kvalitet och prestanda (Mariano, 2017). Därför syftar denna studie till att bidra med dokumentation samt jämförelser mellan olika ramverk i form av laddningstider på e-handelssidor, för att utvecklare ska kunna få information för att kunna värdera och fatta beslut om vilket ramverk som är mest lämpligt för en e-handelssida med krav på laddningstider.

Det andra problemet är att tidigare information om ramverk kan bli föråldrade. Antalet ramverk och deras olika versioner ökar kontinuerligt och snabbt. Det har tidigare genomförts jämförelser i prestanda mellan olika ramverk i både akademiska kretsar och på webben. Det finns gott om information om ramverk tillgängligt på webben, men dock riskerar sådan statisk information att bli föråldrad med både ramverken och webbläsares uppdateringar (Nakajima, Matsumoto & Kusumoto, 2019). Föråldrad information om ramverken kan utgöra ett hinder för att förstå ramverkens egenskaper. På grund av detta är det viktigt att hålla information kring ramverken, inklusive prestandatester kontinuerligt uppdaterade (Nakajima, Matsumoto & Kusumoto, 2019).

Ett ytterligare problem är vikten av en god prestanda, särskilt när det kommer till laddningstider på e-handelssidor. Som tidigare nämnt kommer de flesta internetanvändare att välja att stänga ner webbplatsen om den inte lyckas ladda inom genomsnittet av 8 sekunder (Zhou, Giyane & Nyasha, 2013). Enligt Vihervaara & Alapahuluoma (2018) har flera studier betonat vikten av att organisationer verifierar laddningstiderna för sina webbsidor. Vilket är särskilt viktigt för e-handelssektorn eftersom snabba webbplatser uppnår högre lönsamhet. Laddningstiden har en mätbar effekt på flera olika affärs metriker, som varumärkesuppfattning, konvertering, intäkter, övergivna varukorgar och sidvisningar (Vihervaara & Alapahuluoma, 2018). Därför är prestandan i form av laddningstider av stort intresse och också ett problem som organisationer och företag behöver ta itu med.

Ett annat problem är att användarna till e-handelssidor snabbt växer dag för dag på grund av den enklare internetåtkomsten, där prestanda hos webbapplikationer spelar en nyckelroll för att tillfredsställa slutanvändarna (Amjad et al., 2021). Den ökande användarbasen på e-handelssidor understryker ytterligare behovet av att mäta laddningstider på e-handelssidor.

Med denna problemformulering är syftet med den här studien att göra en grundlig och aktuell jämförelse av ramverken React, Angular och Inferno.js. Genom att skapa tre e-handelssidor med tre olika ramverk, med samma struktur innehållande text och

bilder på produkter, ska det gå att jämföra och analysera prestandan i form av laddningstider på e-handelssidor. Studien strävar också efter att ge utvecklare den insikt och information som behövs för att kunna fatta ett informerat beslut vid valet av ramverk vid framtida utvecklingsprocesser av e-handelssidor med fokus på laddningstider.

Fråga: Vilket ramverk, mellan React, Angular och Inferno, visar de kortaste laddningstiderna för e-handelssidor?

Hypotes: Inferno förväntas visa de lägsta laddningstiderna jämfört med React och Angular.

3 Metod

Studien fokuserar på att utvärdera och jämföra laddningstiderna hos tre olika ramverk: React, Angular och Inferno.js. Målet är att identifiera vilket ramverk som ger den lägsta laddningstiden inom en e-handelssida. För att ta reda på det kommer inspiration tas från två tidigare studier, en av Singh et al. (2023) och en av Gizas et al. (2012). Studierna har använt metoder som gjort det möjligt att analysera prestandan hos olika JavaScript-ramverk, vilket har påverkat valet av metoder i denna studie. Det finns två olika sorter av metoder, kvalitativa och kvantitativa metoder. I kvantitativa metoder används bland annat experiment och surveyundersökning och i kvalitativa metoder finns exempelvis fallstudier och litteraturstudier (Creswell, 2009). För att uppnå målet med studien har olika metoder övervägts.

En metod som kan användas till denna studie är ett kontrollerat experiment. Ett kontrollerat experiment genomförs i en miljö där det finns hög kontroll över variablerna. En enda faktor manipuleras medan andra faktorer hålls konstanta för att isolera dess yttre påverkan på resultatet (Nayak & Singh, 2015). Till exempel, när laddningstiderna jämförs för olika ramverk, skulle endast ramverket som används ändras medan andra faktorer som internetanslutningens hastighet eller enhetens hårdvara förblir oförändrade. Kontrollerat experiment gör det möjligt att isolera effekten av den manipulerade variabeln (ramverket i detta fall) för att sedan bedöma ramverkets inverkan på laddningstiderna utan att påverkas av andra variabler.

En annan metod som kan användas är en litteraturstudie. Enligt Irene & Clark (2006) beskrivs det som att en litteraturstudie granskar vetenskapliga artiklar, böcker och andra källor, exempelvis avhandlingar och konferenshandlingar som är relevanta för ett ämne i en uppsats eller avhandling. Syftet med en litteraturstudie är att visa att författaren har insiktsfullt och kritiskt granskat relevant litteratur om sitt ämne för att övertyga en avsedd publik om att detta ämne är värt att ta upp.

En ytterligare metod som skulle kunna användas är fallstudier. Enligt (Runeson et al. 2012) skulle en fallstudie kunna förklaras som en empirisk undersökning som utforskar ett samtida fenomen inom dess verkliga livsmiljö, särskilt när gränserna mellan fenomen och sammanhang inte är tydligt uppenbara. Vilket innebär att i en fallstudie så testas inte bara webbsidorna i en kontrollerad miljö utan faktiskt används av riktiga användare på olika enheter, webbläsare och nätverkshastigheter. Vilket i sin tur skulle kunna ge en mer realistisk bild av resultaten och hur olika ramverk påverkar laddningstiderna. Nackdelen är att eftersom fallstudier utförs inom verkliga kontexter med många variabler som inte kan kontrolleras på samma sätt som i till exempel ett kontrollerat experiment, kan det vara svårt att exakt reproducera resultaten i framtida studier.

Denna studie syftar till att utvärdera laddningstiderna hos olika ramverk på e-handelssidor, med en metod som delvis liknar den i studien av Gizas et al. (2012) som användes för att bedöma kvalitet och prestanda hos ramverk. Liksom i Gizas et al. (2012) studie kommer även denna studie att inkludera en variant av kontrollerat experiment för att kunna isolera effekten av varje ramverk på laddningstiderna. En annan likhet mellan denna studies metod och Gizas et al. (2012) är användningen av specifika verktyg för att genomföra prestanda mätningarna. Precis som i studien av Gizas et al. (2012) där JSMeter användes för sina prestanda tester, så kommer denna studie använda ett annat verktyg som används för prestandatester, i form av

Tampermonkey för att kunna mäta laddningstiderna för olika ramverk på e-handelssidor. En skillnad mellan studien av Gizas et al. (2012) och denna studie är fokusområdet, där den föregående studien granskade kvalitet och prestanda över en rad olika faktorer, medan denna studie specifikt fokuserar på laddningstider inom e-handelssidor. Inom denna studie kommer även en litteraturstudie vara av särskild betydelse, inspirerad av tidigare studier såsom Singh et al. (2023), där litteraturstudien användes för att undersöka tidigare forskning och studier inom området för ramverk och deras prestanda. Litteraturstudien kommer att användas för att ta fram artefakterna genom att granska relevant litteratur, vilket kommer att möjliggöra en djupare förståelse för ämnet. Med hjälp av metoderna, kontrollerat experiment och litteraturstudie ska det i denna studie gå att utvärdera prestandan i form av laddningstider av de valda ramverken inom e-handelssidor.

3.1 Etiska aspekter

För att göra denna studie reproducerbar så kommer den att likt studien av (Marx-Raacz von Hidvég, 2022) göra flera körningar av testet på varje applikation för att verifiera att resultatet kan reproduceras. För att säkerställa noggrannheten i testet och minska störningar av bakgrunds nätverkstrafiken, kommer Chrome DevTools Throttling att användas för att kunna simulera specifika nätverksförhållanden. För att göra studien mer reproducerbar kommer github att användas för att dokumentera och dela koden som skrivits. Fördelarna med github är att koden kan spåras i vilka ändringar som gjorts, vem som bidragit till koden och slutligen att det enkelt går att dokumentera koden och dela den med allmänheten, vilket bekräftas i studien av (Karthik Ram 2013) där hon ger en översikt över Git och hur det verktyget kan utnyttjas för att göra vetenskap mer reproducerbar och transparent. Vilket gör det möjligt för andra att ta del av koden och kunna reproducera studien.

En annan viktig etisk aspekt som måste beaktas är när människor inkluderas och hur information om människor används och lagras. Till detta så finns det en europeisk förordning vid namn dataskyddsförordningen (GDPR). GDPR är en lagstiftning som syftar till att skydda individens integritet och personuppgifter. Den förordningen, som trädde i kraft den 25 maj 2018, påverkar alla organisationer som hanterar personuppgifter från EU-medborgare, oavsett var i världen de är baserade (IT Governance Privacy Team, 2016). Till denna studie kommer därför ingen information om verkliga människor att inkluderas.

Hållbarhet beskrivs som “Utveckling som möter nuvarande och framtida generationers krav samtidigt som den möter behoven hos framtida generationer”. I samband med mjukvaruutveckling innebär hållbarhet att mjukvaran kommer att vara kompatibel med framtida krav och behov och kan fungera på olika plattformar. Några av faktorerna som tagits i beaktning för att kunna utveckla mjukvara hållbart är faktorer som nämnts i studien av (Haider, Ilyas, Khalid, Ali, 2023). Faktorerna som tagits i beaktning är brist på kodningsstandard och dokumentation, energieffektiv kodning, bristande medvetenhet om hållbar mjukvaruteknik, brist på riktlinjer för utbildning och utveckling och den sista faktorn arbetsincitament. Framförallt är kodningsstandard och dokumentation faktorer som kommer tas i beaktning för att få en mer hållbar utveckling i denna studie.

Andra faktorer att tänka på inom hållbar utveckling är prestanda och utvecklingstid. Där prestanda och effektiviteten av exempelvis e-handelsapplikationer kan bidra till

att minska den totala energiförbrukningen av applikationen. Utvecklingstid omfattar inte bara tid som krävs för planering, design och implementation. Utan även andra faktorer som att utvecklare använder bilen för att pendla till arbetet, vilket också kan ha en negativ inverkan på en hållbar utveckling.

En ytterligare aspekt inom hållbar utveckling är inlåsningseffekter, vilket innebär att utvecklare blir beroende av olika teknologier eller ramverk som kan förändras på ett sätt som inte är bakåtkompatibelt. Det innebär att ramverk som React, Angular och Inferno får nya uppdateringar eller versioner, vilket kan introducera förändringar som kräver att koden skrivs om för att fungera korrekt. Inlåsningseffekter kan alltså leda till mer arbete, vilket medför högre kostnader och en längre utvecklingstid. Inlåsningseffekter är alltså en viktig aspekt för utvecklare att tänka på för att skapa mer hållbara applikationer.

4. Litteraturstudie

En litteraturstudie har genomförts för att samla in information och få en djupare förståelse för de tre valda ramverken React, Angular och Inferno. Under utvecklingen av artefakterna har bland annat sidor som stack overflow, ramverkens egna dokumentation samt böcker om ramverken varit till hjälp för att lösa olika problem, eller för att hämta kunskap om hur delar av ramverken fungerar och implementeras.

React

För React har flera resurser varit av nytta för att förstå dess koncept och implementeringdetaljer. Bland annat har boken *React 18 Design Patterns and Best Practices - Fourth Edition* (Roldán, 2023) varit till stor hjälp. Dessutom har sidan Stackoverflow utforskats, där gemenskapen delar insikter och lösningar på olika problem som har uppstått. Ramverkets officiella dokumentation[1] har även varit en viktig resurs till denna studie för att kunna få en djupare förståelse för ramverkets olika funktioner.

Angular

Samma gäller Angular där bland annat boken *Angular for Enterprise Applications - Third Edition* av (Uluca, 2024) har bidragit till värdefulla insikter i hur det går att bygga med Angular. Liksom med React har Stackoverflow använts för att hitta värdefulla svar och diskussioner om olika problem som går att stöta på vid utveckling med Angular. Sedan har även Angulars officiella dokumentation[2] varit en viktig resurs för att förstå dess struktur och användning.

Inferno

För ramverket Inferno upptäcktes det att den tillgängliga litteraturen är väldigt begränsad, troligtvis på grund av ramverkets relativt unga ålder jämfört med React och Angular. Även om det inte hittades någon litteratur i form av böcker med mera, har det ändå gått att dra nytta av diverse diskussioner på Stackoverflow. Dessutom har Infernos officiella dokumentation[3] utforskats för att förstå dess grundläggande koncept.

---

[1] https://legacy.reactjs.org/docs/getting-started.html
[2] https://angular.io/start
[3] https://www.infernojs.org/docs/guides/installation

5. Progression

Det började med att ett Github repository skapades, där alla implementationer och ändringar i koden skulle kunna sparas. Det valdes ett dataset som skulle passa in till de tänkta e-handelssidorna innehållandes data om olika produkter i form av kläder med tillhörande information och bilder. Med Github repository och dataset på plats skapades ett projekt för varje ramverk för att kunna skapa de olika applikationerna.

5.1 Utvecklingsmiljö

Det utvecklingsverktyg som har använts i denna studie är Visual Studio Code[4] som använts för att utveckla React, Angular och Inferno applikationerna.

5.2 Dataset till Databas

Det dataset som använts till alla E-handelssidor är Mango Products, vilket finns tillgängligt på kaggle.com. Datasetet innehåller data om olika produkter, innehållande bilder, namn, priser, beskrivningar och är tillgängligt under MIT-licensen[5]. När rätt dataset hittades, laddades det ner och importerades till databasen. Datasetet laddades in i databasen genom databasens webbadministration, PHPMyAdmin. Genom detta finns nu datasetet i databasen som ska användas till studien.

5.3 API

För att få tillgång till data från databasen som innehåller information om de olika produkterna, utvecklades ett API med namnet server.js[6]. Genom att implementera API:et kan applikationer hämta produktinformation och bilder från databasen för att presentera för användare. Utvecklandet av API:et gjordes genom att först skapa en anslutning till databasen genom att skriva in användarnamn, lösenord med mera till databasen. Sedan skapades olika endpoints för att kunna hantera förfrågningar.

```
app.get('/api/products', async (req, res) => {
  try {
    // Hämta produkter från databasen
    const limit = req.query.limit || 1500;
    const [rows, fields] = await pool.query(`SELECT * FROM store_mango
LIMIT ${limit}`);
    res.json(rows);
  } catch (error) {
    console.error('Error fetching products:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Figur 4. Endpoint för att hämta information om alla produkter.

Två olika endpoints skapades på server.js. En endpoint GET-route /api/products definieras för att hantera förfrågningar för att hämta alla produkter från databasen vilket kan ses i Figur 4. När en förfrågan görs till denna endpoint, utförs en SQL-fråga för att välja alla rader från tabellen store_mango i databasen. Resultatet av frågan blir att alla produkter skickas tillbaka som JSON till klienten som gjorde förfrågan. I

[4] https://code.visualstudio.com/
[5] https://www.mit.edu/~amini/LICENSE.md
[6] https://github.com/b21willu/Examensarbete/commit/93976e1b5a342f779f1ed3f8db24294d9bb61741

GET-routen för att hämta produkterna från databasen, har en parameter limit lagts till. Vilket gör det möjligt för klienten att specificera ett värde för antalet produkter som ska hämtas från databasen, då datasetet består av ca 3 000 produkter. Om ingen limit parameter specificeras i förfrågan, används ett standardvärde på 1500 produkter. Vid andra frågor som är mer specifika och vill ha data om en specifik produkt används en annan endpoint api/products/:sku[7].

```
app.get('/api/products/:sku', async (req, res) => {
  try {
    const sku = req.params.sku;

    // Hämta produkten från databasen baserat på SKU
    const [productRows, productFields] = await pool.query(`SELECT * FROM
store_mango WHERE sku = ?`, [sku]);
    const product = productRows[0];

    if (!product) {
      return res.status(404).json({ error: 'Produkt hittades inte' });
    }

    // Hämta bilderna för den specifika produkten från databasen
    const [imageRows, imageFields] = await pool.query(`SELECT
image_downloads FROM store_mango WHERE sku = ?`, [sku]);
    const images = imageRows[0].image_downloads.split(',').map(filename =>
filename.trim());

    product.image_downloads = images;

    res.json(product);
  } catch (error) {
    console.error('Error fetching product details:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Figur 5. Endpoint för att hämta information om produkter baserat på SKU värde.

Den andra endpointen används för att hämta information om produkter baserat på deras SKU värde se Figur 5. Enpointen api/products/:sku används vid produktlistan och till kundvagnen när användare klickar på en specifik produkt och bara information om just den produkten ska visas. Genom att hämta produkter via deras SKU (primär nyckel) värde hämtas bara en specifik produkt.

5.4 React

När utvecklingsmiljön var klar påbörjades utvecklingen av den första artefakten av en E-handelssida, som utvecklades med ramverket React. Först skapades ett nytt projekt i mappen Examensarbete som är länkat till en Github repository[8]. Sedan skapades ett React projekt genom att lokalisera till Github mappen innehållande examensarbete mappen via kommandotolken, och i den mappen skapades sedan projektet genom att skriva in ett kommando för att skapa ett nytt React projekt. När projektet var skapat och kopplat till github påbörjades utvecklingen av applikationen i Visual Studio Code. Först skapades komponenterna för grundstrukturen av sidan innehållande Header,

---

[7] https://github.com/b21willu/Examensarbete/commit/0820278453b330e5b82feee1e9d563ec6ec3c4fa

[8] https://github.com/b21willu/Examensarbete

Footer, Navigation, och MainContent. Vilka är baskomponenterna för alla sidor i applikationen. När komponenterna var skapade skulle logiken med navigationen att fixas med hjälp av react-router.

5.4.1 Routing

Routing i React används för att navigera mellan olika delar av webbapplikationen. Routing implementerades med hjälp av react-router-dom, som tillhandahåller komponenter för att hantera navigering och URL-matchning i React. I huvudkomponenten App.js[9] har en Router-komponent från react-router-dom använts för att omsluta hela applikationen. Inom Router-komponenten definieras olika Routes för att matcha olika URL-sökvägar och rendera motsvarande komponenter.

```
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
…

function App() {
  return (
    <CartProvider>
      <Router>
        <div>
          <Navigation />
          <Header />
          <Routes>
            <Route path="/" element={<MainContent />} />
            <Route path="/produkter" element={<ProductList />} />
            <Route path="/om-oss" element={<About />} />
            <Route path="/kontakt" element={<Contact />} />
            <Route path="/kundvagn" element={<ShoppingCart />} />
            <Route path="/product/:sku" element={<ProductDetail />} />
            <Route path="/kundvagn/:sku" element={<ShoppingCart />} />
          </Routes>
          <Footer />
        </div>
      </Router>
    </CartProvider>
  );
}

export default App;
```

Figur 6. Routing i React-komponenten App.js.

Till exempel för att definiera en Route för startsidan, används Route-komponenten med attributet path="/", vilket innebär att denna Route kommer att matcha när URL:en är “/”. Vilket resulterar i att <MainContent />-komponenten renderas när användaren besöker startsidan. Liknande Routes definieras för andra sidor i applikationen, såsom ProductList, About, Contact, ShoppingCart och ProductDetail-sidan se Figur 6. Varje Route specificerar en path som motsvarar den URL-sökväg som ska matchas. Och element-attributet anger vilken komponent som ska renderas när sökvägen matchas.

---

9 https://github.com/b21willu/Examensarbete/commit/343cae7625d36ee8d615de7acc3349aef34fa6df

```
import { Link } from 'react-router-dom';

function Navigation() {
    return (
        <nav>
            <ul>
                <li><Link to="/">Hem</Link></li>
                <li><Link to="/produkter">Produkter</Link></li>
                <li><Link to="/om-oss">Om oss</Link></li>
                <li><Link to="/kontakt">Kontakt</Link></li>
                <li><Link to="/kundvagn"><FaShoppingCart /></Link></li>
            </ul>
        </nav>
    );
}

export default Navigation;
```

Figur 7. Routing i React-komponenten Navigation.js.

I Navigation-komponenten Navigation.js[10] används Link-komponenten från react-router-dom för att skapa länkar till olika sidor i applikationen se Figur 7. När användaren klickar på en länk, sker navigeringen till sidan. Sammanfattningsvis gör användningen av react-router-dom det möjligt att hantera navigationen i React-applikationen och rendera olika komponenter beroende på den aktuella URL-sökvägen.

5.4.2 Hämta produkter

När komponenterna var skapade och routingen var fixad, skapades komponenten ProductList[11] för att kunna visa produkterna på sidan. ProductList hämtar data från server.js genom att använda fetch-metoden för att skicka en HTTP förfrågan till http://localhost:3001/api/products, där server.js hanterar denna förfrågan och returnerar data om produkterna. Som sedan används för att kunna presentera datan visuellt på sidan i form av bilder och text.

```
const ProductList = () => {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    fetch('http://localhost:3001/api/products')
      .then(response => response.json())
      .then(data => setProducts(data))
      .catch(error => console.error('Error fetching products:', error));
  }, []);

 return (
    <div>
      <h1>Produkter</h1>
      <div className="product-list">
        {products.map(product => (
          <div key={product.sku} className="product">
            <h2>{product.name}</h2>
```

[10] https://github.com/b21willu/Examensarbete/commit/343cae7625d36ee8d615de7acc3349aef34fa6df

[11] https://github.com/b21willu/Examensarbete/commit/37a5f7b489b43572e1499e1d9f2432d3eb6907e3

```
            <p>{product.description}</p>
            <p>Pris: {product.price} {product.currency}</p>
            <img src={product.images} alt={product.name} />
          </div>
        ))}
      </div>
    </div>
  );
};

export default ProductList;
```

Figur 8. Kod från komponenten ProductList för att fetcha och visa upp data från produkterna.

ProductList-komponenten utgör produktsidan och visar alla produkter från datasetet. I Figur 8 går det att se hur koden används för att hämta data från endpoint api/products, för att kunna hämta och visa upp data om alla produkter. Produktdatan skrivs ut genom att iterera genom produkterna med products.map(product =>) för att få ut alla produkter. Sedan bestäms det vilken data som ska hämtas genom att skriva i html-element, till exempel {product.name} för att skriva ut produktens namn. Samma logik implementerades för ProductDetails sida med skillnaden att den hämtar data ifrån endpoint api/products/${sku} för att få data om en specifik produkt.

5.4.3 ShoppingCart - CartContext

För att hantera kundvagnen implementerades logik för att kunna addera samt ta bort produkter i kundvagnen. För att göra detta har CartContext[12] skapats med hjälp av Reacts Context API. Context API:et möjliggör att data kan delas och nås av flera komponenter i React-applikationen.

```
import React, { createContext, useContext, useState } from 'react';

const CartContext = createContext();

export const useCart = () => useContext(CartContext);

export const CartProvider = ({ children }) => {
  const [cart, setCart] = useState([]);

  const addToCart = (product, sku) => {
    setCart([...cart, { ...product, sku: sku }]);
  };

  const removeFromCart = (skuToRemove) => {
    setCart(cart.filter(product => product.sku !== skuToRemove));
  };

  return (
    <CartContext.Provider value={{ cart, addToCart, removeFromCart }}>
      {children}
    </CartContext.Provider>
  );
};
```

[12] https://github.com/b21willu/Examensarbete/commit/cdae6c383a18b23a4f33b813658d030b0cc63ca5

```
export default CartContext;
```

Figur 10. Kod från CartContext som hanterar kundvagnens funktioner att ta bort/lägga till produkter .

I koden skapas först en CartContext med createContext()-funktionen från React, som ger tillgång till en Provider och en Consumer. CartProvider-komponenten hanterar kundvagnens data och funktionalitet genom att använda useState()-hooket för att spara produkterna i kundvagnen. Funktionen addToCart() lägger till produkter i kundvagnen genom att uppdatera cart-staten med den nya tillagda produkten. Funktionen removeFromCart() används för att ta bort produkter från kundvagnen genom att filtrera bort produkten med det angivna sku:värdet, vilket är den primära nyckeln för produkterna. CartProvider renderar CartContext.Provider och ger sina underliggande komponenter åtkomst till kundvagnens data och funktioner genom value-attributet, se Figur 10. Komponenterna kan nu använda kundvagnen genom useCart()-hooken. I komponenterna kan addToCart funktionen användas genom att importera useCart från CartContext och sedan använda den funktionen när exempelvis knappen "Lägg till produkt” klickas på.

```
import { useCart } from '../CartContext';

const ProductDetail = () => {
  const { sku } = useParams();
  const [product, setProduct] = useState(null);
  const { addToCart } = useCart();

  useEffect(() => {
    fetch(`http://localhost:3001/api/products/${sku}`)
      .then(response => {
        if (!response.ok) {
          throw new Error('Failed to fetch product');
        }
        return response.json();
      })
      .then(data => {
        setProduct(data);
      })
      .catch(error => console.error('Error fetching product details:',
error));
  }, [sku]);

  const handleAddToCart = (sku) => {
    addToCart(product, sku);
    alert('Produkten har lagts till i kundvagnen.');
  };
…
<button onClick={handleAddToCart} className="button">Lägg till i
kundvagnen</button>
```

Figur 11. Kod från ProductDetail-komponenten för att lägga till produkter i kundvagnen .

I Figur 11 går det att se kod som används för att lägga till produkter i kundvagnen. Liknande sker i Shoppingcart-komponenten genom att importera useCart från CartContext men istället använda funktionen removeFromCart när knappen “ta bort” klickas på.

5.4.4 Filtrering av produkter.

För att sidan ska likna en e-handelssida mer har en funktion adderats för att kunna filtrera produkterna baserat på vilken kategori de tillhör. Från databasen finns en kolumn som heter "terms" som innehåller vilken kategori av kläder produkterna tillhör, till exempel bags, shoes eller jackets. För att få till denna filtrering av kläder implementerades kod i ProductList[13].

```
const ProductList = () => {
  const [products, setProducts] = useState([]);
  const [filterTerm, setFilterTerm] = useState(null);

  useEffect(() => {
   if (filterTerm) {
      apiUrl += `?term=${filterTerm}`;
    }

  fetch(apiUrl)
   …
  }

 const handleFilterChange = (event) => {
    setFilterTerm(event.target.value);
  };

  const filterProducts = (product) => {
    if (!filterTerm) return true; // Om ingen term är vald, returnera true för
att visa alla produkter
    return product.terms.includes(filterTerm);
  };

<div className='filter-container'>
  <label htmlFor="filter" className="filter-label">Filter:</label>
  <select id="filter" onChange={handleFilterChange}  className="filter-select">
     <option value="">All</option>
     <option value="backpack">Backpack</option>
      <option value="bags">Bags</option>
      …
  </select>
</div>
```

Figur 12. Kod från ProductList-komponenten för att kunna filtrera produkter baserat på deras "term".

Funktionen handleFilterChange kallas på när användaren ändrar valet i filter-select-elementet. Den uppdaterar filterTerm med värdet från det valda alternativet i filter-select-element. Funktionen filterProducts används för att filtrera produkterna baserat på det aktuella värdet i filterterm. Om ingen term är vald returneras true, vilket innebär att alla produkter på sidan kommer att visas. Annars returneras true endast om produkten har den valda termen i dess terms-attribut (Se Figur 12).

[13] https://github.com/b21willu/Examensarbete/commit/cdae6c383a18b23a4f33b813658d030b0cc63ca5

5.5 Angular

Likt React projektet skapades ett nytt projekt i mappen “examensarbete” som är länkat till en Github repository. Projektet skapades genom att lokalisera till Github mappen innehållande examensarbete mappen med hjälp av kommandotolken. I den mappen skapades ett nytt Angular projekt genom att skriva in kommandot ng new angular-app. Sedan öppnades denna mapp i Visual Studio Code och utvecklingen av artefakten började genom att skapa komponenter. Komponenterna Header, Footer, Navigation samt MainContent skapades då de är grundstrukturen för alla sidor i applikationen.

5.5.1 Routing

Routing i Angular används för att navigera mellan de olika delarna av webbapplikationen. Routing implementerades med hjälp av Angulars routermodul, som är ett verktyg som tillhandahåller funktioner för att hantera navigering och URL-matchning i Angular-applikationer. I Angular-applikationen användes komponenten app-routing.module för att definiera och hantera routing. För att få till routing i applikationen implementerades kod i komponenten app-routing.module [14].

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { MainContentComponent } from
'./main-content/main-content.component';
import { ContactComponent } from './contact/contact.component';
import { AboutComponent } from './about/about.component';
import { ProductListComponent } from
'./product-list/product-list.component';
import { ProductDetailComponent } from
'./product-detail/product-detail.component';
import { ShoppingCartComponent } from
'./shopping-cart/shopping-cart.component';

const routes: Routes = [
  { path: 'home', component: MainContentComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'products', component: ProductListComponent },
  { path: 'product/:sku', component: ProductDetailComponent },
  { path: 'shopping-cart', component: ShoppingCartComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Figur 13. Routing i Angular-komponenten app-routing.module.

I app-routing.module importeras RouterModule och Routes från @angular/router. Därefter definieras en array med namnet routes som innehåller olika routes för applikationen. Varje route i arrayen har två värden, path och component. Path specificerar den URL-sökväg som ska matchas och component anger vilken komponent som ska renderas när URL-sökvägen matchas. Till exempel, för sökvägen

---

[14] https://github.com/b21willu/Examensarbete/commit/edb0ff10a8972565ba57c9b7a7239b5f2f58e95b

“products”, definieras ProductListComponent som komponenten som ska renderas. Liknande routes definieras för “about”, “contact”, “home” osv, var och en med motsvarande komponenter som ska renderas när URL sökvägen matchas (Se Figur 13).

5.5.2 Hämta produkter

För att hämta och visa upp produkterna skapades en ny komponent product-list för att visa upp datan om produkterna i form av bilder och text. För att hantera hämtningen av produkterna i Angular-applikationen har en angular service med namnet product.service[15] skapats. Denna service fungerar som ett gränssnitt för att interagera med API:et server.js. Liknande funktionalitet användes för komponenten product-detail men som hämtar och visar data om enskilda produkter baserat på vad användaren har valt för att sedan visa mer information om produkten.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private apiUrl = 'http://localhost:3001/api/products';

  constructor(private http: HttpClient) { }

  getProducts(): Observable<any[]> {
    return this.http.get<any[]>(this.apiUrl);
  }

  …

  getProduct(sku: string): Observable<any> {
    const url = `${this.apiUrl}/${sku}`;
    return this.http.get<any>(url);
  }
}
```

Figur 14. Kod för product.service.

I Figur 14 går det att se kod för product.service. Först finns tre imports, där Injectable används för att göra klassen injicerbar, vilket innebär att den är tillgänglig i hela applikationen. HttpClient är ett verktyg som används för att göra HTTP-förfrågningar till externa API:er. Observable används för att hantera asynkrona dataflöden, särskilt när det gäller hantering av HTTP-förfrågningar. Den första raden i tjänsten private apiUrl definierar den platsen där Angular-applikationen ska göra HTTP förfrågningar till, för att kunna hämta produktdatan från API:et. Funktionen getProducts() används för att hämta alla produkter från det definierade API:et. Den använder “HttpClient” för att göra en GET-förfrågan till den definierade “apiUrl” och returnerar en Observable för att få tillbaka produktdata. Och den sista funktionen getProduct(), är en metod för att hämta en specifik produkt baserat på dess SKU-värde. Den konstruerar den exakta

[15] https://github.com/b21willu/Examensarbete/commit/819e0b006dcd5c6db969f30905acd94f814ba9ff

URL:en för den specifika produkten och gör sedan en GET-förfrågan till den URL:en med “HttpClient” och returnerar likt getProducts() en Observable för att få tillbaka produktdata.

5.5.3 ShoppingCart

För att kunna hantera kundvagnen implementerades logik för att göra det möjligt att lägga till samt ta bort produkter från kundvagnen. För att fixa det har en Angular service skapats med namnet cart.service[16].

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class CartService {
  private cartKey = 'cart';
  private cart: any[] = this.getCartFromLocalStorage() || [];

  constructor() { }

  getCart(): any[] {
    return this.cart;
  }

  addToCart(product: any, sku: string): void {
    const productWithSku = { ...product, sku: sku };
    this.cart.push(productWithSku);
    this.saveCartToLocalStorage();
  }

  removeFromCart(skuToRemove: string): void {
    const indexToRemove = this.cart.findIndex(product => product.sku ===
skuToRemove);
    if (indexToRemove !== -1) {
      this.cart.splice(indexToRemove, 1);
      this.saveCartToLocalStorage();
    }
  }

  private saveCartToLocalStorage(): void {
    localStorage.setItem(this.cartKey, JSON.stringify(this.cart));
  }

  private getCartFromLocalStorage(): any[] {
    const cartData = localStorage.getItem(this.cartKey);
    return cartData ? JSON.parse(cartData) : [];
  }
}
```

Figur 15. Kod för cart.service.

Den första funktionen getCart() används för att hämta den aktuella kundvagnen. Med funktionen addToCart() kan produkter läggas till i varukorgen. Den tar emot produktens information och dess SKU-värde, lägger till SKU-värdet till produkten och lägger sedan till produkten i varukorgen. Därefter sparas varukorgen till local storage. För att ta bort produkter används funktionen removeFromCart(), där

16 https://github.com/b21willu/Examensarbete/commit/799f4f30cb90aba347bbb74374d68c197e91c69a

produkter tas bort baserat på deras SKU-värde. Funktionen söker igenom kundvagnen efter produkten med matchande SKU-värde och tar sedan bort den från kundvagnen. För att spara produkterna i kundvagnen, används LocalStorage. Med saveCartToLocalStorage() sparas den aktuella kundvagnen till local storage. Funktionen getCartFromLocalStorage() används för att kunna hämta produkterna i kundvagnen från local storage, för att sedan kunna visa upp produkterna på kundvagns sidan i applikationen. Local storage sparar datan som strängar. Genom att först konvertera kundvagnen till JSON-format så kan datan sparas som en sträng i local storage. När data sedan hämtas från local storage, konverteras den tillbaka till dess ursprungliga format för att kunna användas i applikationen (Se Figur 15).

5.5.4 Filtrering av produkter.

Till produktlistan används en funktion för att kunna filtrera produkterna baserat på vilken "terms" de har, "terms" värdet för produkterna innehåller vilken kategori av kläder de tillhör till exempel "T-shirt". Med filter-funktionen så ska det gå att filtrera produkterna efter kategorier som "shoes", "bags" eller "jackets". Funktionen getFilteredProducts() finns i tjänsten product.service [17].

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private apiUrl = 'http://localhost:3001/api/products';

  constructor(private http: HttpClient) { }

  …

  getFilteredProducts(filterTerm?: string): Observable<any[]> {
    let apiUrlWithFilter = this.apiUrl;
    if (filterTerm) {
      apiUrlWithFilter += `?term=${filterTerm}`;
    }
    return this.http.get<any[]>(apiUrlWithFilter);
  }

  …
}
```

Figur 16. Kod i product.service.

I Figur 16 visas hur Funktionen getFilteredProducts från ProductService används för att hämta produkter från API:et med en valfri filterterm. Metoden tar emot en parameter "filterTerm" , vilket är en sträng som används för att filtrera produkterna baserat på en viss term. Först skapas en variabel apiUrlWithFilter som innehåller den ursprungliga API-url:en. Om filterTerm-parametern är definierad läggs den till som en query parameter med namnet term till slutet av URL:en. Detta gör det möjligt att skicka med en filterterm till API:et för att filtrera produkterna baserat på termen.

[17] https://github.com/b21willu/Examensarbete/commit/9a6002edd8af54c4aaa5f04e163fdc2710815236

Sedan gör metoden en GET-förfrågan till API:et för att hämta de produkter som matchar med rätt "term".

5.6 Inferno

Likt de två tidigare projekten, skapades ett nytt Inferno projekt i mappen Examensarbete som är länkat till en Github-repository. Projektet skapades genom att navigera till Github-mappen innehållande examensarbete-mappen med hjälp av kommandotolken. I denna mapp skapades sedan ett nytt Inferno-projekt genom att använda kommandot create-inferno-app inferno-app. Sedan skapades komponenterna Header, Footer, Navigation och MainContent, som utgör grundstrukturen för alla sidor i applikationen.

5.6.1 Routing

Routing i Inferno används för att navigera mellan de olika delarna av webbapplikationen. För att implementera routing i Inferno användes inferno-router, som tillhandahåller komponenter för att hantera navigering och URL-matchning. Routing i Inferno implementeras genom att definiera olika routes som matchar olika URL-sökvägar och renderar motsvarande komponenter. I komponenten App.js[18], användes en Router-komponent för att omsluta hela applikationen. Inom Router-komponenten definieras sedan olika routes för att matcha olika URL-sökvägar och rendera de motsvarande komponenterna.

```
import { BrowserRouter, Route, Switch } from 'inferno-router';
…
export default class App extends Component {
  render() {
    return (
      <div>
        <BrowserRouter>
          <Navigation />
          <Header />
          <Switch>
            <Route exact path="/" component={MainContent} />
            <Route path="/products" component={ProductList} />
            <Route path="/about" component={About} />
            …
          </Switch>
        </BrowserRouter>
        <Footer />
      </div>
    );
  }
}
```

Figur 17. Kod för routing i App.js.

I koden från Figur 17 importeras BrowserRouter, Route och Switch från inferno-router. BrowserRouter används för att definiera routing-logiken för applikationen. Olika routes matchar olika URL-sökvägar och renderar motsvarande komponenter. Switch används för att endast rendera den första matchande routen

18

https://github.com/b21willu/Examensarbete/commit/c0faa6c613c997ab3077c7dc0d78f7993a9faa2f

inom switchen vilket innebär att komponenterna Navigation, Header och Footer kommer att visas på alla sidor. I routen så läggs det till vilken komponent som ska renderas när routens URL-sökväg matchas. Till exempel i Figur 17 så renderas komponenten ProductList när URL-sökvägen är /products. Vilket gör att när en användare navigerar till olika URL-sökvägar, matchas de med rätt route och renderar den.

```
import { Component } from 'inferno';
import { Link } from 'inferno-router';

class Navigation extends Component {
  render() {
    return (
      <nav>
        <ul>
          <li><Link to="/">Hem</Link></li>
          <li><Link to="/products">Produkter</Link></li>
          <li><Link to="/about">Om oss</Link></li>
          <li><Link to="/contact">Kontakt</Link></li>
          <li><Link to="/kundvagn">Kundvagn</Link></li>
          <li></li>
        </ul>
    </nav>
    );
  }
}

export default Navigation;
```

Figur 18. Kod för Navigation.js.

Genom att Navigation-komponenten innehåller länkar till olika URL-sökvägar, möjliggör den för användare att navigera mellan olika sidor i applikationen. När en användare klickar på en länk, matchas den URL-sökvägen med en route som definierats i App.js (Se Figur 18). Genom denna kod i Navigation.js[19] tillsammans med App.js möjliggörs navigeringen mellan de olika sidorna med inferno-router.

5.6.2 Hämta produkter

För att hämta och visa produkter i Inferno-applikationen skapades först en ny komponent med namnet ProductList[20]. Denna komponent hämtar och visar data om produkterna i form av bilder och text.

```
import { Component } from 'inferno';

class ProductList extends Component {
  constructor(props) {
    super(props);
    this.state = {
      products: []
    };
  }
```

[19] https://github.com/b21willu/Examensarbete/commit/c0faa6c613c997ab3077c7dc0d78f7993a9faa2f

[20] https://github.com/b21willu/Examensarbete/commit/77f42fe745ed276a6130db93c16baf4c828e01ac

```
  componentDidMount() {
    this.fetchProducts();
  }

  fetchProducts = async () => {
    try {
      const response = await fetch(`http://localhost:3001/api/products`);
      const data = await response.json();
      this.setState({ products: data });
    } catch (error) {
      console.error('Error fetching products:', error);
    }
  };

  render() {
    return (
      <div>
        <h1>Produkter</h1>
        <div className="product-list">
          {this.state.products.map(product => (
            <div key={product.sku} className="product">
              <h3>{product.name}</h3>
              <p className="product-content">{product.name}</p>
              <p className="product-content">Pris: {product.price}
{product.currency}</p>
            </div>
…
```

Figur 19. Kod för ProductList.js för att hämta och visa upp data om produkter.

I figur 19 hämtas och visas produkterna på produktsidan. För att hantera hämtningen av produkterna från API:et används en fetch-metod. Denna metod använder fetch-API för att göra en HTTP-förfrågan till API:et server.js och hämtar sedan produkterna från endpointen api/products. I render-metoden visas produkterna på sidan genom att först loopa igenom produkterna och sedan rendera dem med tillhörande information. Varje produkt renderas som en separat med produktens namn, pris, bilder med mera.

5.6.3 ShoppingCart

För att hantera kundvagnen i inferno applikationen implementeras kod i komponenten ShoppingCart[21]. Vilket gör det möjligt att ta bort samt lägga till produkter i kundvagnen. En uppdatering på metoden removeFromCart()[22] gjordes för att lösa ett problem som uppstod när två likadana produkter var i kundvagnen. Problemet var att när en av de två identiska produkter skulle tas bort, togs båda bort då de delar samma sku-värde .

```
import { Component } from 'inferno';
  …
```

[21] https://github.com/b21willu/Examensarbete/commit/f0095090d82c98b4ec69d7b56d0587866ee9f947

[22] https://github.com/b21willu/Examensarbete/commit/b8934f9a27053e479f9055d7ef18c75404adadae

```
  removeFromCart = (skuToRemove) => {
   const indexToRemove = window.cart.findIndex(product => product.sku ===
skuToRemove);
    if (indexToRemove !== -1) {
        window.cart.splice(indexToRemove, 1);
        this.forceUpdate();
    }
};
  …

  render() {
    const { images } = this.state;
    return (
      <div>
        {window.cart.map(product => (
          <div key={product.sku} className="shopping-cart-container">
            <div className="shopping-product-info">
              <p className="shopping-detail-content">{product.name}</p>
              <p className="shopping-detail-content">Pris: {product.price}
{product.currency}</p>
              <button onClick={() => this.removeFromCart(product.sku)}>Ta
bort</button>
            </div>
          </div>
        ))}
   …
```

Figur 20. Kod för ShoppingCart.js för att hantera kundvagnen.

Funktionen removeFromCart() används för att ta bort en produkt från kundvagnen, den tar emot SKU-värdet genom att använda findIndex-funktionen på window.cart-arrayen. Ifall rätt produkt hittas, tas produkten bort från kundvagnen med splice-funktionen som tar bort ett element från arrayen på den angivna positionen. Efter borttagningen av produkten uppdateras komponenten med this.forceUpdate() för att visa upp den uppdaterade kundvagnen. I Render-metoden, renderas produkterna i kundvagnen. För varje produkt i kundvagnen skapas en med en unik nyckel baserad på produkten SKU-värde. Produktnamn, pris och en "Ta bort" knapp renderas för varje produkt. När "Ta bort"-knappen klickas på anropas funktionen removeFromCart med produktens SKU-värde som parameter, för att ta bort den produkt med matchande SKU från kundvagnen (Se Figur 20). För att lägga till produkterna används funktionen addToCart() från komponenten ProductDetail[23].

```
…
addToCart = (product, sku) => {
    // Lägg till produkt i kundvagnen genom att uppdatera den globala
variabeln
    window.cart = [...(window.cart || []), { ...product, sku }];
    alert('Produkten har lagts till i kundvagnen.');
  };

…
 <button className="button" onClick={() => this.addToCart(product, sku)}>
    Lägg till i kundvagnen
```

[23] https://github.com/b21willu/Examensarbete/commit/f0095090d82c98b4ec69d7b56d0587866ee9f947

```
 </button>
…
```

Figur 21. Kod i ProductDetail.js för att lägga till produkter i kundvagnen.

Funktionen addToCart() används för att lägga till en produkt i kundvagnen. Den tar emot två parametrar: product och sku. Funktionen uppdaterar den globala variabeln window.cart genom att lägga till den nya produkten till den befintliga kundvagnen. Produkten läggs till som ett nytt objekt i formen { product, sku }, vilket inkluderar all produktinformation samt produktens sku-värde. Slutligen visas en alert för att visa användaren att produkten har lagts till i kundvagnen. När användaren klickar på knappen anropas funktionen addToCart() med den aktuella produkten och dess SKU:värde som parametrar (Se Figur 21).

5.6.4 Filtrering av produkter.

Likt de tidigare applikationerna har en funktion implementerats för att kunna filtrera produkterna baserat på vilken kategori de tillhör. Från databasen finns en kolumn men namnet “terms” som innehåller vilken kategori av kläder produkterna tillhör, till exempel bags, shoes eller jackets. För att få till denna filtrering av kläder implementerades kod i ProductList[24].

```
import { Component } from 'inferno';
…

class ProductList extends Component {
  constructor(props) {
    super(props);
    this.state = {
      products: [],
      filterTerm: '',
    };
  }
  …
  handleFilterChange = (event) => {
    const term = event.target.value;
    this.setState({ filterTerm: term });
  };

  filterProducts = (product) => {
    const { filterTerm } = this.state;
    if (!filterTerm) return true; // Om ingen kategori är vald returnera
true, för att visa alla produkter
    return product.terms.includes(filterTerm);
  };

  render() {
    const { products, images } = this.state;

    return (
    <div>
        <div className='filter-container'>
          <label htmlFor="filter" className="filter-label">Filter:</label>
          <select id="filter" onChange={this.handleFilterChange}
className="filter-select">
```

[24] https://github.com/b21willu/Examensarbete/commit/9cb74e644ccfc3a459c91fbb242575f5a8eb87b7

```
            <option value="">All</option>
            <option value="backpack">Backpack</option>
            <option value="bags">Bags</option>
            …
          </select>
      </div>
    );
  }
}

export default ProductList;
```

Figur 22. Kod i ProductDetail.js för att lägga till produkter i kundvagnen.

Funktionen handleFilterChange() används för att uppdatera filterTerm när användaren ändrar valet i filter-menyn. Funktionen filterProducts() används för att filtrera produkterna baserat på den aktuella filtertermen. Om ingen kategori är vald returneras true för att visa alla produkter. Annars kontrollerar funktionen om produkterna innehåller den valda kategorin, och visar de produkter som har matchande filterTerm. I render() visas filter-menyn och produkterna (Se Figur 22). När användaren ändrar valet i filter-menyn, uppdateras filterTerm och produkterna renderas om baserat på inputen från filtret. Koden i figur 22 möjliggör för användare att filtrera produkter efter en vald kategori för att endast se de produkter som matchar den valda kategorin.

5.7 Förberedelse inför pilotstudie

När React, Angular samt Inferno applikationerna var färdiga påbörjades förberedelserna inför pilotstudien. Till pilotstudien användes verktyget Tampermonkey för att kunna genomföra ett automatiserat script som laddar om sidan “products”. Scriptet mäter när sidan har laddat klart och lagrar sedan laddningstiden och fortsätter så 200 gånger för varje mätning. I varje iteration när scriptet laddar om sidan så sparas laddningstiden och lagras (200 iterationer), när scriptet kört klart konverteras laddningstiderna till en CSV fil och skapar en nedladdningsbar länk. Där CSV filen med mätningarna tillsammans med kod i utvecklingsverktyget PyCharm används för att kunna skapa grafer som används för att analysera mätningarna av React, Angular och Inferno.

6. Pilotstudie

För att undersöka ifall det går att jämföra prestandan i form av laddningstider mellan applikationerna har en pilotstudie genomförts. Pilotstudien har utförts genom att skapa ett automatiserat script för att ladda om produktsidan 200 gånger och mäta tiden det tog för produktsidan att ladda. En lokal servermiljö användes för att utföra mätningarna.

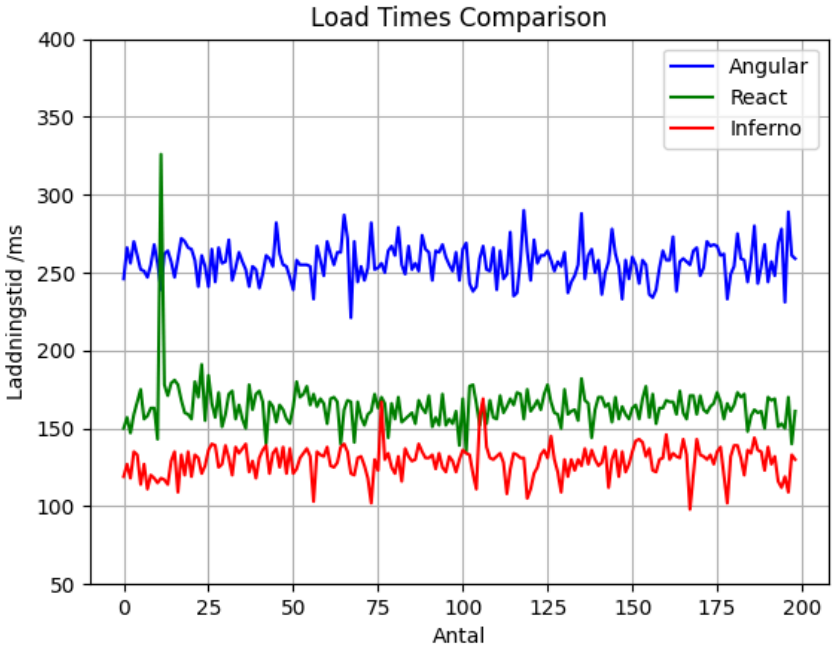
Operativ system	Windows 11 Home 64-bit version 22631.3447
GPU (Grafikkort)	AMD Radeon RX 6700 XT
CPU (Processor)	AMD Ryzen 7 5700x 8-Core Processor
Webbläsare	Google Chrome Version 124.0.6367.61 (Officiell version) (64 bitar)
React version	version 18.2.0
Angular version	version 17.3.2.
Inferno version	version 8.2.3

Tabell 1. Specifikationer för hårdvara och mjukvara från pilotstudie.

Specifikationer för hårdvaran och mjukvaran som användes till testerna finns i tabellen ovan (***Tabell 1***).

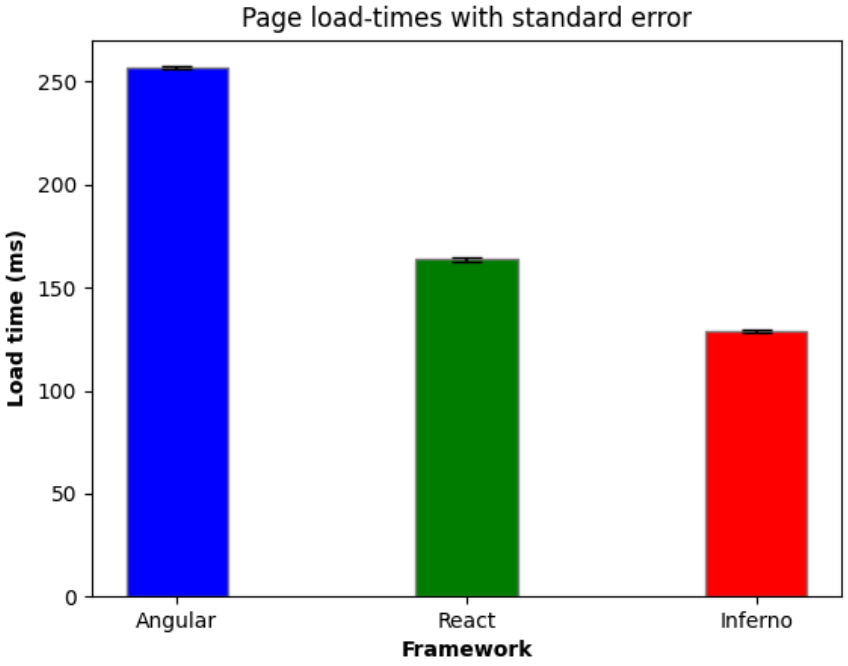
6.1 Grafer

Till pilotstudien har olika grafer skapats för att visualisera resultaten. Dels har linjediagram tagits fram för att visa värdena och jämföra de olika ramverkens resultat av laddningstider. Förutom linjediagram har även stapeldiagram med Standardfel, standardavvikelse samt konfidensintervaller tagits fram för att visa skillnaden mellan resultaten i pilotstudien. Sedan har även ett ANOVA-test gjorts för att bekräfta att det finns skillnader i resultaten av pilotstudien.



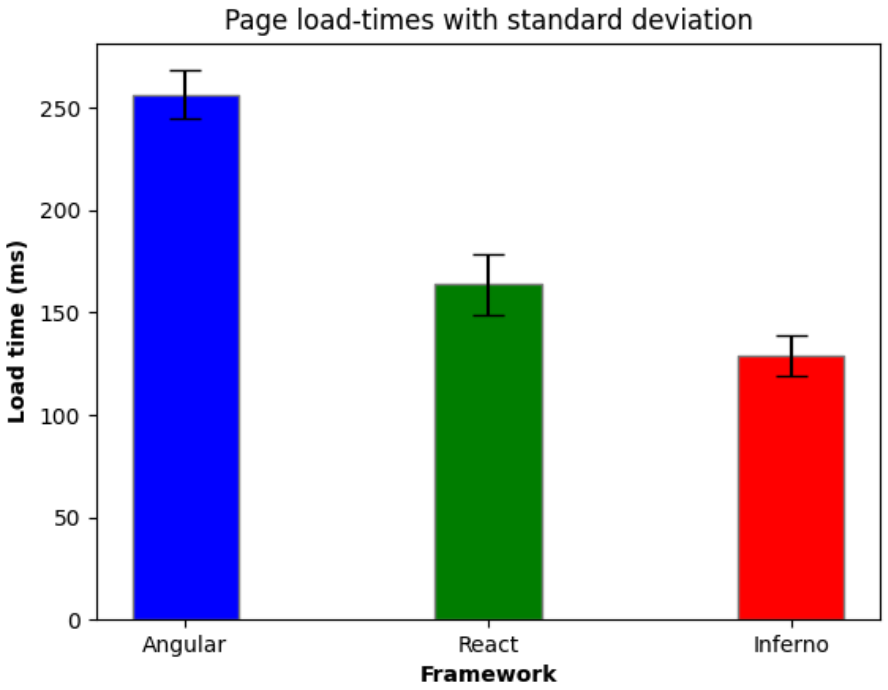
Figur 23. Linjediagram med laddningstider mellan de tre olika applikationerna.

Som det går att se i linjediagrammet (Se Figur 24) är skillnaderna mellan de tre ramverken ganska tydliga. Angular (Blå linje) visar sig ha de långsammaste laddningstiderna. Medan React (Grön linje) och Inferno (Röd linje) ligger lite närmare varandra. Trots det så är Inferno ganska tydligt ändå snabbast när det kommer till laddningstider baserat på linjediagrammet.



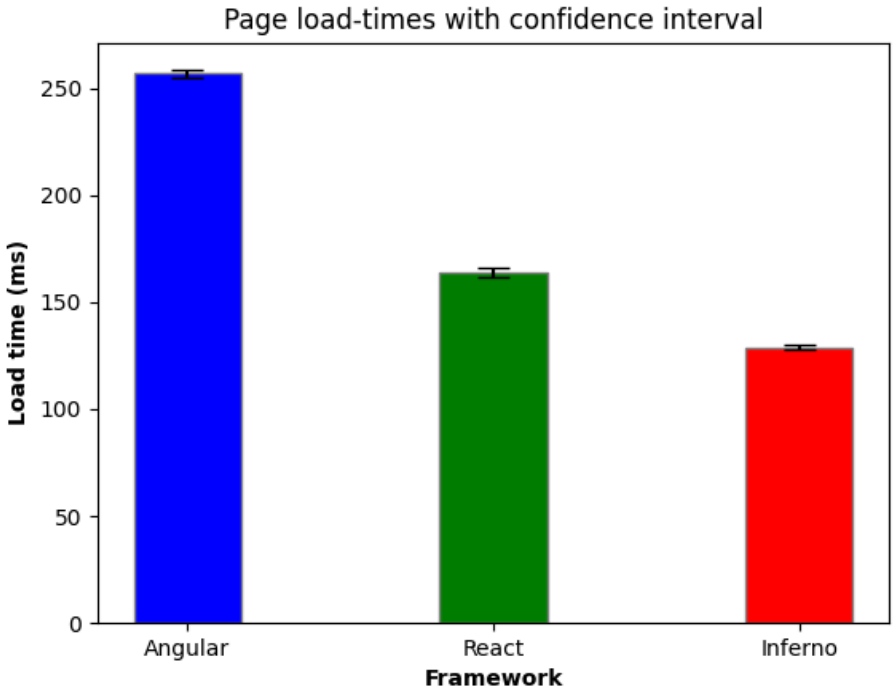
Figur 24. Stapeldiagram med standardfel från mätningar.

Stapeldiagrammet i Figur 25 illustrerar laddningstider för de tre ramverken Angular, React och Inferno. Varje stapel representerar den genomsnittliga laddningstiden för respektive ramverk, med standardfel inkluderat för att visa variationen i mätningarna.



Figur 25. Stapeldiagram med standardavvikelse från mätningar .

Stapeldiagrammet i Figur 26 illustrerar också laddningstider för de tre ramverken Angular, React och Inferno. Där varje stapel representerar den genomsnittliga laddningstiden för respektive ramverk, med deras standardavvikelse som är ett mått på spridningen av värdena från dess medelvärde.



Figur 26. Stapeldiagram med konfidensintervall från mätningar.

I Figur 27 visas ett stapeldiagram likt de två tidigare med staplar för respektive ramverk Angular, React och Inferno men i stapeldiagrammet är konfidensintervall inkluderat. Konfidensintervall

```
ANOVA Statistic: 5709.490235068091 and p-value: 0.0

The means are different.
```

Figur 27. ANOVA-test resultat.

6.3 Diskussion

I linjediagrammet (Se Figur 23) går det att tydligt se skillnader mellan laddningstiderna för ramverken React, Angular och Inferno. Denna observation stärks ytterligare ifall stapeldiagrammen inkluderas (se figur 24, 25 och 26), där det går att se att de olika ramverken uppvisar tydliga variationer i laddningstider. I stapeldiagrammen framgår också standardfel, standardavvikelse och konfidensintervall.

Sammanfattningsvis i denna genomförda pilotstudie visar de framtagna graferna tillsammans med ANOVA-testet på betydande skillnader i laddningstider mellan de olika ramverken React, Angular och Inferno. För att kunna fördjupa förståelsen och bekräfta resultaten från mätningarna, bör antalet mätpunkter i kommande tester ökas betydligt jämfört med pilotstudien. En ökad mängd mätpunkter bör ge en bättre möjlighet att kunna bedöma om skillnaderna i laddningstider mellan ramverken fortfarande är lika tydliga, eller om de eventuellt tenderar att jämna ut sig med en ökad datainsamling. Antalet produkter på sidan kommer också att varieras för att se hur ramverken hanterar laddningstiderna vid olika mängder data.

7. Utvärdering

I detta kapitel utvärderas experimentet som utförts och de resultat det innehöll. Experimentet gick ut på att jämföra laddningstider mellan tre applikationer skapade med ramverken React, Angular och Inferno. Applikationerna efterliknar e-handelssidor innehållandes produkter i form av bilder på kläder med tillhörande information, hämtade från ett dataset kallat Mango Products[25]. Pilotstudien visade att det var möjligt att göra mätningar på laddningstiden mellan de olika ramverken React, Angular och Inferno. Då pilotstudien genomfördes i en mindre skala med 200 mätpunkter så har en ny mycket större mätning gjorts för att kunna analysera och dra tydliga slutsatser om skillnaderna i laddningstid mellan ramverken. De nya mätningarna gjordes genom att öka antalet mätningar från 200 i pilotstudien till 1000 i den nya mätningen på laddningstider. Sedan har även nya mätningar inkluderats som mäter laddningstiden för applikationerna där mängden produkter har varierats genom att göra tre mätningar per applikation med 200, 1000 och 2000 produkter på sidan. För att kunna se hur laddningstiderna varieras beroende på datamängden, en annan skillnad från pilotstudien är att cache har inaktiverats för att få mer exakta mätningar på laddningstiderna. Cache inaktiverades eftersom cachelagrade resurser kan bidra till snabbare laddningstider vid upprepade sidladdningar, vilket kan ge en missvisande bild av den verkliga laddningstiden vid sidans första besök.

7.1 Experiment

I denna del kommer grafer i form av linjediagram och stapeldiagram innehållandes standardfel, standardavvikelse och konfidensintervaller att presenteras för att visa laddningstiderna för de olika applikationerna som utvecklats i React, Angular och Inferno. Först kommer det att presenteras grafer för ett ramverk i taget med varierande storlek på datasetet. Sedan kommer en jämförelse att göras mellan ramverken React, Angular och Inferno över olika storlekar på datasetet. Genom att analysera graferna kommer det att ge insikt i prestanda i form av laddningstid för varje ramverk med varierande datamängd. Det kommer att identifieras eventuella skillnader eller mönster mellan ramverken. De olika mätningarna som har gjorts i experimentet går att se i tabellen nedan (Se Tabell 2).

React (200 produkter)	1000 mätpunkter
React (1000 produkter)	1000 mätpunkter
React (2000 produkter)	1000 mätpunkter
Angular (200 produkter)	1000 mätpunkter
Angular (1000 produkter)	1000 mätpunkter
Angular (2000 produkter)	1000 mätpunkter
Inferno (200 produkter)	1000 mätpunkter
Inferno (1000 produkter)	1000 mätpunkter

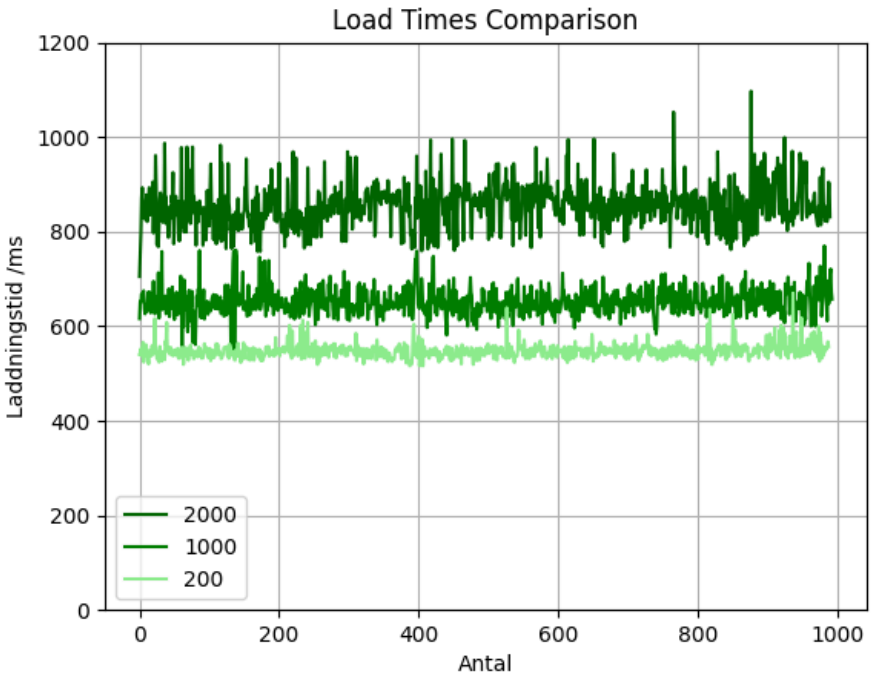
[25] https://www.kaggle.com/datasets/maparla/mango-products?select=store_mango.csv

Inferno (2000 produkter)	1000 mätpunkter

Tabell 2. Tabellen innehåller olika mätningar samt antal mätpunkter för varje mätning.

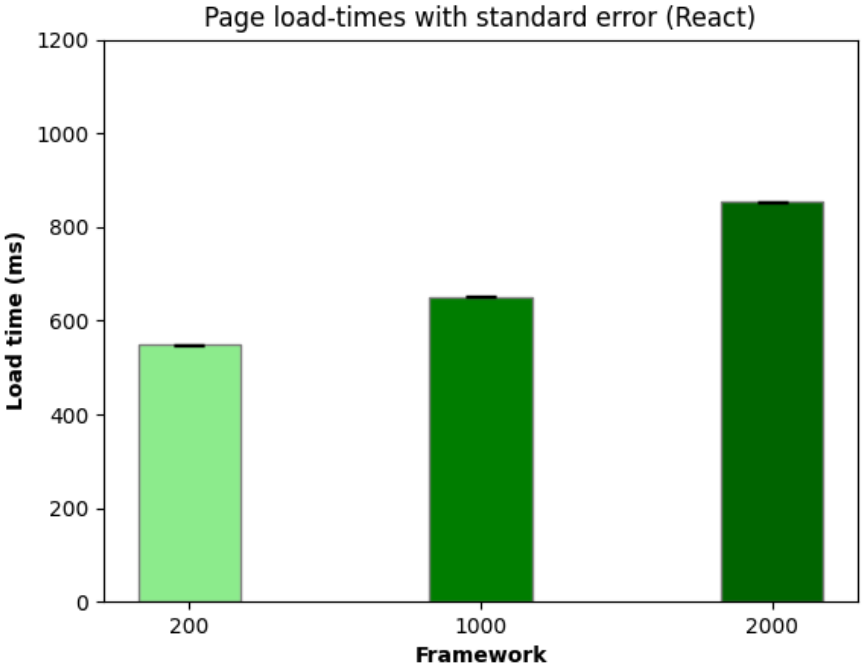
I tabell 2 går det att se de olika mätningar som har gjorts. Där till exempel React (200 produkter) innebär att e-handelssidan är utvecklad med ramverket React och innehåller 200 produkter från datasetet. Sedan innebär den högra kolumnen att det har gjorts 1000 mätningar på laddningstiden, det vill säga att produktsidan har laddats om och mätt laddningstiden 1000 gånger.

7.1.1 React



Figur 28. Linjediagram React 200, 1000 och 2000 produkter.

Som det går att se i linje diagrammet (Se Figur 29) är skillnaden mellan laddningstiderna på olika antal produkter relativt tydliga. Linjerna visar mätningar gjorda med olika antal produkter där 200 produkter (ljusgrön linje), 1000 produkter (grön linje) och 2000 produkter (mörkgrön linje). Ganska väntat så presterar applikationen de snabbaste tiderna på 200 produkter och långsammast med 2000 produkter. Linjen för 200 produkter (Linjen längst ner) är ganska rak utan mycket brus medan ju mer data som inkluderas ju mer brus verkar det bidra till på mätningarna, ifall man kollar på till exempel den översta linjen (2000 produkter) så har den mycket mer brus i sina mätningar än de andra två.



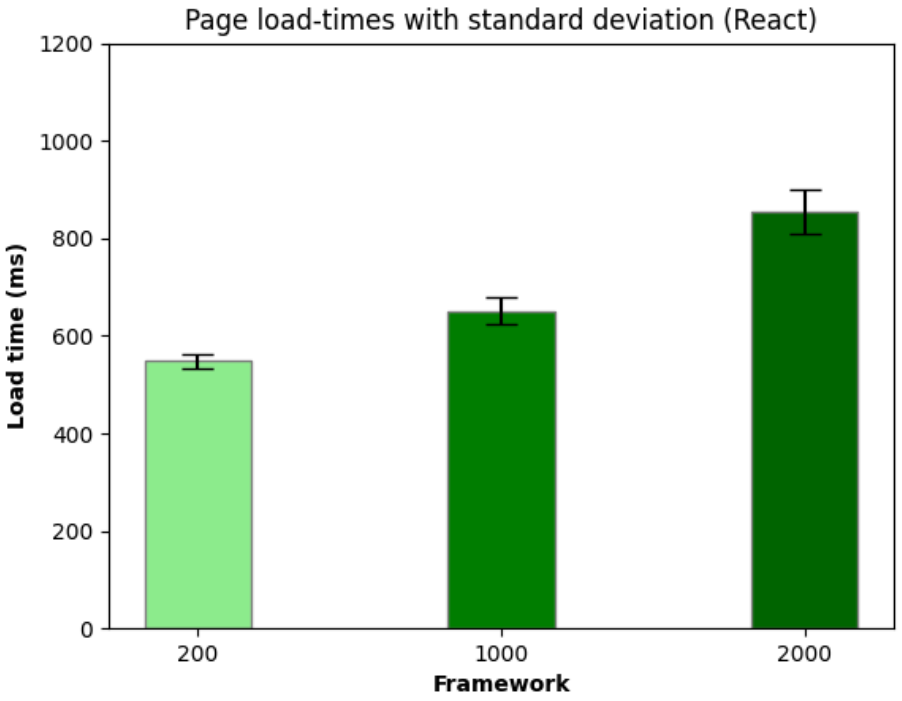
Figur 29. Stapeldiagram React 200, 1000 och 2000 produkter med standard fel.

I stapeldiagrammet ovan (se figur 30) visas en jämförelse mellan olika laddningstider gjorda på en React-applikation med olika datamängd, 200 produkter (ljusgrön), 1000 produkter (grön) och 2000 produkter (mörkgrön). Skillnaden i laddningstider beroende på antal produkter är ganska tydlig där ju mer produkter som inkluderas desto högre laddningstider. I detta diagram inkluderas även standardfel, se den svarta linjen på toppen av stapeldiagrammen. Standardfelet är visuellt ganska otydligt, men nedan kan du se siffrorna för standardfelet avrundat till tre decimaler (Se Tabell 3).

200 produkter	0.479
1000 produkter	0.878
2000 produkter	1.457

Tabell 3. Standardfel för 200, 1000 och 2000 produkter i React (avrundat till tre decimaler)

Som det går att se i Tabell 3 så ökar standardfelet ganska tydligt när datasets storleken ökar, vilket tyder på att medelvärdet för laddningstiden kan bli mindre tillförlitlig vid större dataset.



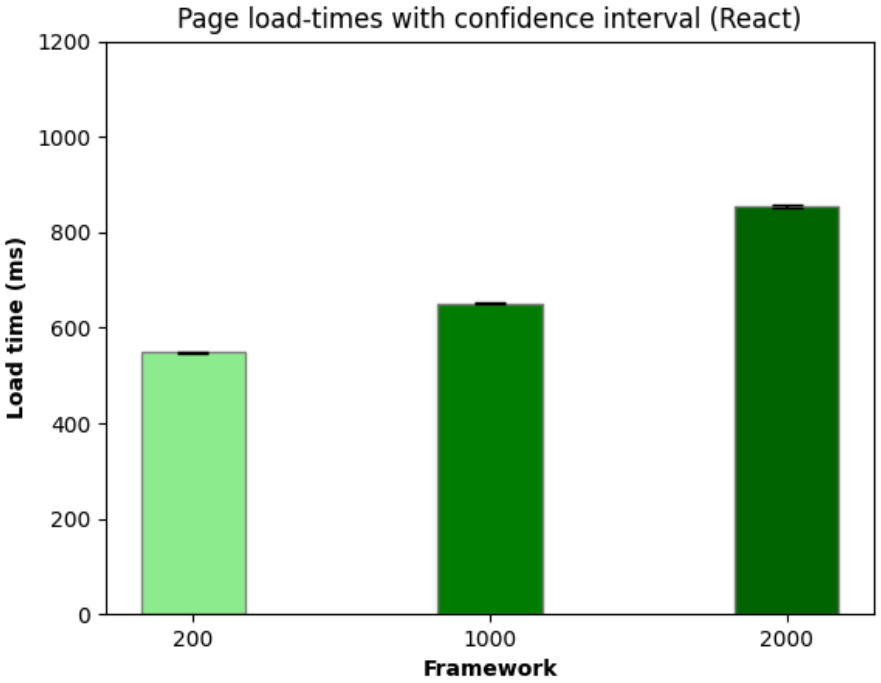
Figur 30. Stapeldiagram React 200, 1000 och 2000 produkter med standardavvikelse.

I stapeldiagrammet ovan (se figur 30) visas en jämförelse mellan olika laddningstider gjorda på en React-applikation med olika datamängd 200 produkter (ljusgrön), 1000 produkter (grön) och 2000 produkter (mörkgrön). I stapeldiagrammet inkluderas även standardavvikelsen, vilket går att se i figuren där felstaplarna i toppen av de färglagda staplarna representerar standardavvikelsen. En tabell har skapats för att tydligare visa standardavvikelsen (Se Tabell 4).

200 produkter	15.067
1000 produkter	27.698
2000 produkter	45.863

Tabell 4. Standardavvikelse för 200, 1000 och 2000 produkter i React (avrundat till tre decimaler).

Som det går att se i Tabell 4 så ökar standardavvikelsen tydligt när datasetets storlek ökar. Detta indikerar en ökad spridning eller variation i laddningstiderna när datasetets storlek ökar.



Figur 31. Stapeldiagram React 200, 1000 och 2000 produkter med konfidensintervall.

I stapeldiagrammet ovan (se figur 31) visas en jämförelse mellan olika laddningstider gjorda på en React-applikation med olika datamängd, 200 produkter (ljusgrön), 1000 produkter (grön) och 2000 produkter (mörkgrön). Konfidensintervallet för varje dataset visas genom felstaplarna på toppen av de färgade staplarna. För en tydligare presentation av konfidensintervallet har en tabell skapats (se Tabell 5).

Antal produkter	Lägre gräns	Övre gräns
200	546.996	548.877
1000	649.596	653.046
2000	851.488	857.208

Tabell 5. Konfidensintervall för 200, 1000 och 2000 produkter i React (avrundat till tre decimaler).

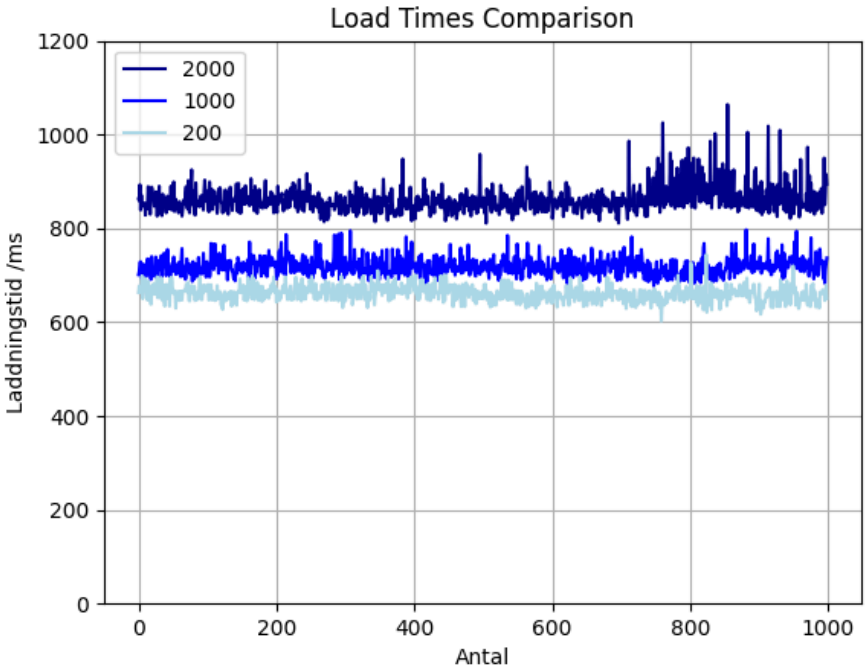
För konfidensintervallet har en konfidensnivå på 95% använts. Vilket innebär att det är 95% säkerhet att det sanna medelvärdet för populationen ligger inom det beräknade intervallet. Intervallet är mellan den lägre gränsen och den övre gränsen. För varje ökning av antalet produkter verkar både den lägre och övre gränsen för konfidensintervallet öka. Vilket kan vara en indikation på att osäkerheten i vårt uppskattade medelvärde ökar när antalet produkter ökar.

ANOVA Statistic: 22692.49810896744 and p-value: 0.0
The means are different.

Figur 32. Anova test för React mätningarna med 200, 1000 och 2000 produkter.

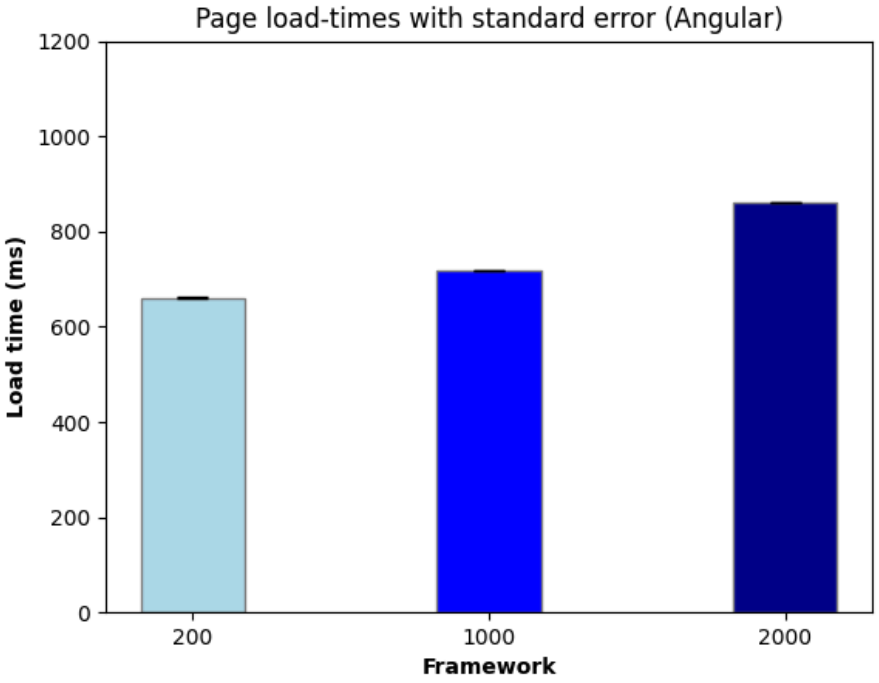
I ANOVA testet som gjordes för mätningarna av react applikationen så visas resultatet “The means are different” , vilket indikerar på att det finns statistiskt signifikanta skillnader mellan grupperna. Detta innebär att laddningstiderna för React-applikationen skiljer sig åt beroende på datamängden.

7.1.2 Angular



Figur 33. Linjediagram Angular 200, 1000 och 2000 produkter.

Som det går att se i linje diagrammet (Se Figur 33) är skillnaden mellan laddningstiderna på olika antal produkter relativt tydliga. Linjerna visar mätningar gjorda med olika antal produkter där 200 produkter (ljusblå linje), 1000 produkter (blå linje) och 2000 produkter (mörkblå linje). Ganska väntat så presterar applikationen de snabbaste tiderna på 200 produkter och långsammast med 2000 produkter .



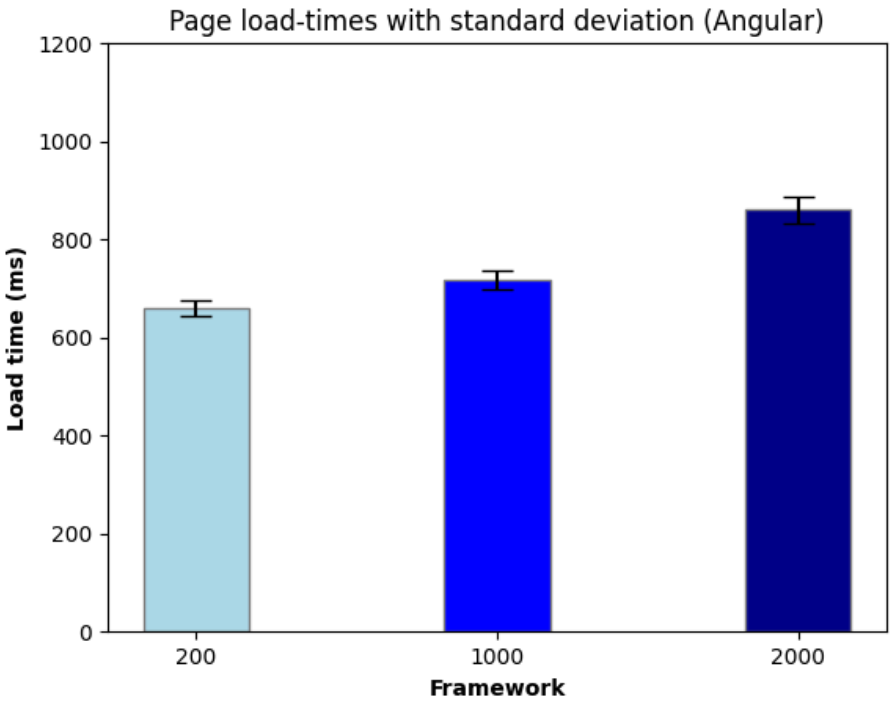
Figur 34. Stapeldiagram Angular 200, 1000 och 2000 produkter med standard fel.

I stapeldiagrammet ovan (se figur 34) visas en jämförelse mellan olika laddningstider gjorda på en Angular-applikation med olika datamängd, 200 produkter (ljusblå), 1000 produkter (blå) och 2000 produkter (mörkblå). Resultaten är tydliga, ju mer produkter desto längre blir laddningstiderna. I stapeldiagrammet inkluderas även standardfel, se den svarta linjen på toppen av stapeldiagrammen. Standardfelet är visuellt ganska otydligt, men nedan kan du se siffrorna för standardfelet avrundat till tre decimaler (Se Tabell 6).

200 produkter	0.499
1000 produkter	0.594
2000 produkter	0.865

Tabell 6. Stapeldiagram Angular 200, 1000 och 2000 produkter med standardfel (avrundat till tre decimaler).

Som det går att se i Tabell 6 så ökar standardfelet när datasets storleken ökar, vilket tyder på att medelvärdet för laddningstiden kan bli mindre tillförlitlig vid större dataset.



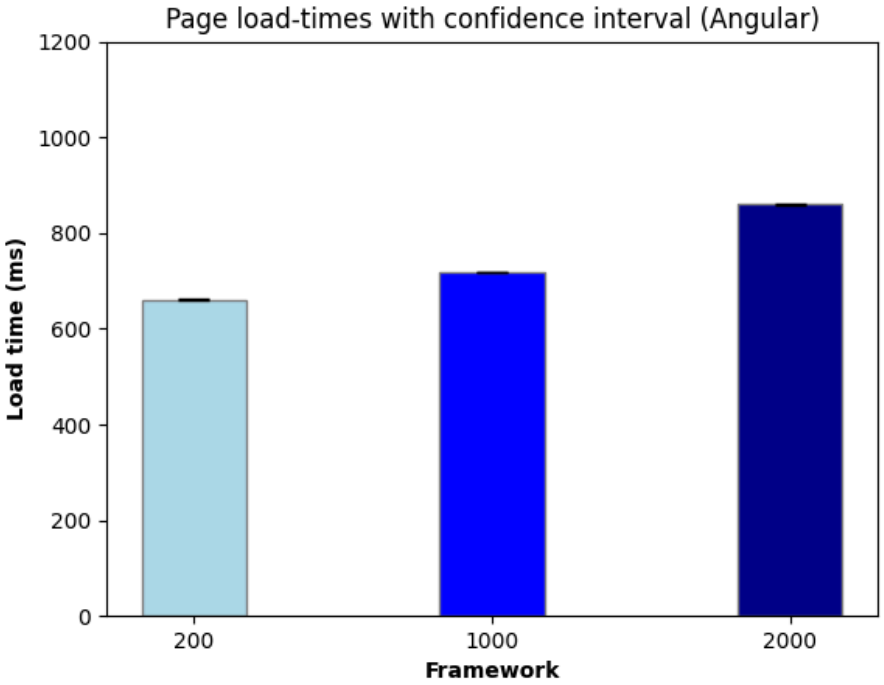
Figur 35. Stapeldiagram Angular 200, 1000 och 2000 produkter med standardavvikelse.

I stapeldiagrammet ovan (se figur 35) går det att se en jämförelse mellan olika laddningstider gjorda på en Angular-applikation med olika datamängd, 200 produkter (ljusblå), 1000 produkter (blå) och 2000 produkter (mörkblå). I stapeldiagrammet inkluderas även standardavvikelsen. En tabell har skapats för att tydligare visa standardavvikelsen (Se Tabell 7).

200 produkter	15.778
1000 produkter	18.793
2000 produkter	27.371

Tabell 7. Standardavvikelse för 200, 1000 och 2000 produkter i Angular (avrundat till tre decimaler).

I Tabell 7 visas standardavvikelsen för de mätningar som gjorts på angular-applikationen. I tabellen går det att se att standardavvikelsen ökar när datasetets storlek ökar. Vilket indikerar på en ökad spridning eller variation i laddningstiderna när datasetets storlek ökar.



Figur 36. Stapeldiagram Angular 200, 1000 och 2000 produkter med konfidensintervall.

I stapeldiagrammet i figur 36 visas en jämförelse mellan olika laddningstider gjorda på en Angular-applikation med olika datamängd, 200 produkter (ljusblå), 1000 produkter (blå) och 2000 produkter (mörkblå). Stapeldiagrammet innehåller även konfidensintervallet för varje gjord mätning och visas genom felstaplarna på toppen av de färgade staplarna. För en tydligare presentation av konfidensintervallet har en tabell skapats (se Tabell 8).

Antal produkter	Lägre gräns	Övre gräns
200	660.246	662.205
1000	717.295	719.629
2000	859.020	862.419

Tabell 8. Konfidensintervall för 200, 1000 och 2000 produkter i Angular (avrundat till tre decimaler).

För konfidensintervallet har en konfidensnivå på 95% använts. Detta innebär att det är 95% säkerhet att det sanna medelvärdet för populationen ligger inom det beräknade intervallet. Intervallet är mellan den lägre gränsen och den övre gränsen. Efter att ha jämfört konfidensintervallen för Angular märks det att intervallet är relativt likartat mellan de olika mätningarna. Vilket indikerar att osäkerheten i uppskattningen av medelvärdet för laddningstiden inte ökar mycket när antalet produkter ökar.

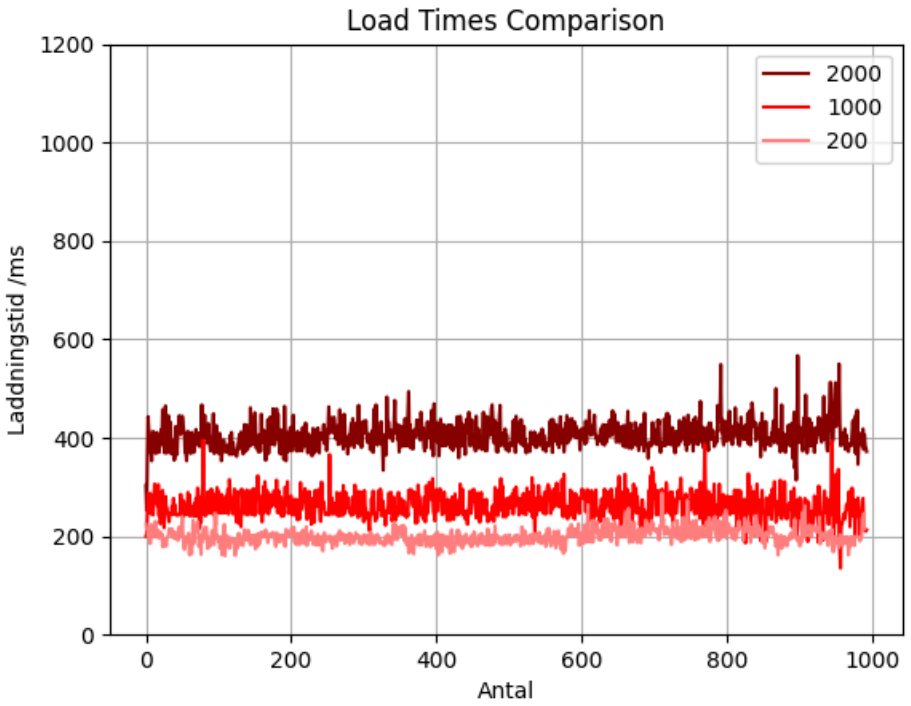
ANOVA Statistic: 23402.621476071556 and p-value: 0.0

The means are different.

Figur 37. Anova test för Angular mätningarna med 200, 1000 och 2000 produkter.

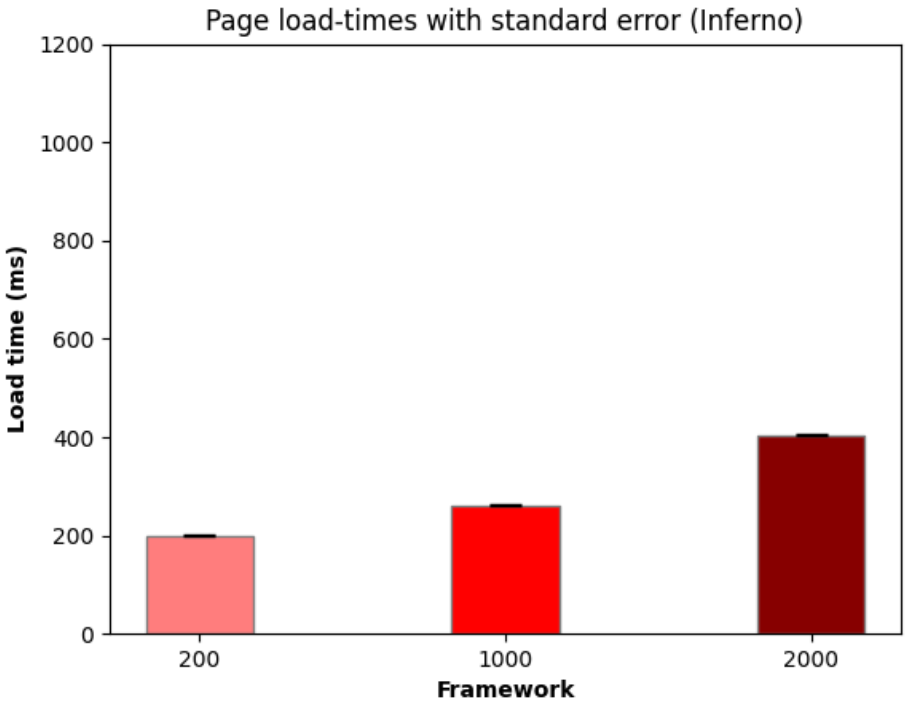
Enligt Anova testet i figur 37 så visar det sig att “The means are different” vilket tyder på att det finns statistiskt signifikanta skillnader mellan mätningarna. Vilket innebär att laddningstiderna för Angular-applikationen skiljer sig åt beroende på datamängden.

7.1.3 Inferno



Figur 38. Linjediagram Inferno 200, 1000 och 2000 produkter.

Som det går att se i linje diagrammet (Se Figur 38) finns det skillnader mellan laddningstiderna på olika antal produkter. Linjerna visar mätningar gjorda med olika antal produkter där 200 produkter (ljusröd linje), 1000 produkter (röd linje) och 2000 produkter (mörkröd linje). I tidigare linjediagram har skillnaderna varit större men i denna ändå visuellt möjligt att se skillnader. Linjen för 200 produkter ligger likt väntat med lägst laddningstider (dock inte tydligt) då linjen med 1000 produkter ligger väldigt nära. Och sedan ligger linjen med 2000 produkter ganska tydligt över med de längsta laddningstiderna.



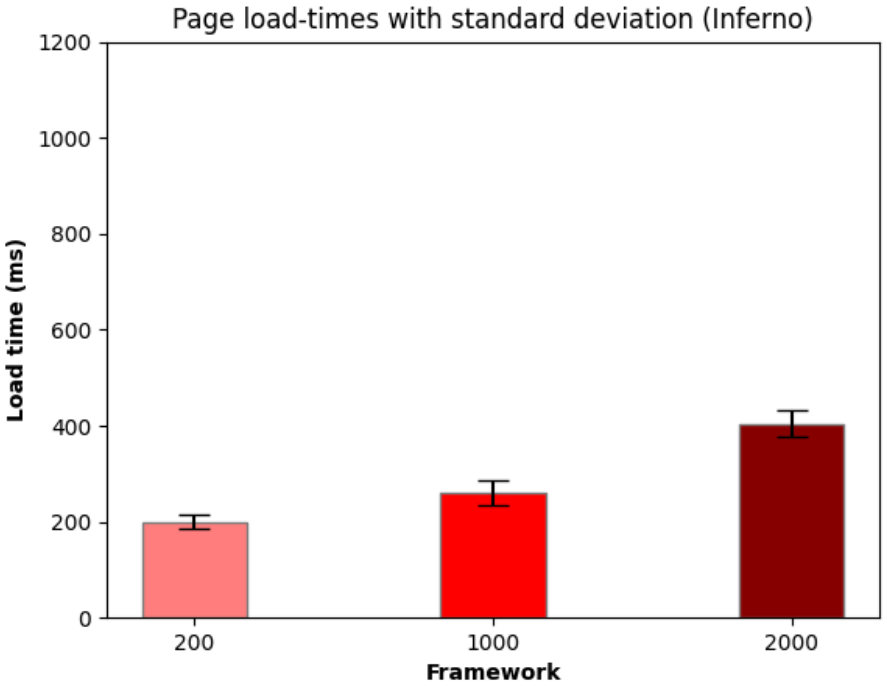
Figur 39. Stapeldiagram Inferno 200, 1000 och 2000 produkter med standardfel.

I stapeldiagrammet (Se figur 39) visas en jämförelse mellan olika laddningstider gjorda på en Inferno-applikation med olika datamängd, 200 (ljusröd), 1000 (röd) och 2000 produkter (mörkröd). Resultaten indikerar att ju större datamängd desto längre blir laddningstiden. I diagrammet visas även standardfel (den svarta linjen i toppen av staplarna). Standardfelet är svårt att tyda i stapeldiagrammet, därför har en tabell skapats för att visa siffrorna för standardfelet avrundat till tre decimaler (Se Tabell 9).

200 produkter	0.494
1000 produkter	0.841
2000 produkter	0.877

Tabell 9. Standardfel för 200, 1000 och 2000 produkter i Inferno (avrundat till tre decimaler).

I Tabell 9 går det att se att skillnaden mellan standardfelet är stort mellan 200 och 1000 produkter, medan skillnaden är ganska liten mellan 1000 och 2000 produkter. Inferno-applikationen reagerar olika på olika datamängder. Skillnaden i standardfel mellan 200 och 1000 produkter är tydligt, vilket antyder att prestandan och stabiliteten hos Inferno kan påverkas av storleken på datasetet. Det är möjligt att Inferno har svårigheter att hantera medelstora dataset jämfört med mindre eller större dataset. Dock är skillnaden i standardfel relativt liten mellan 1000 och 2000 produkter.



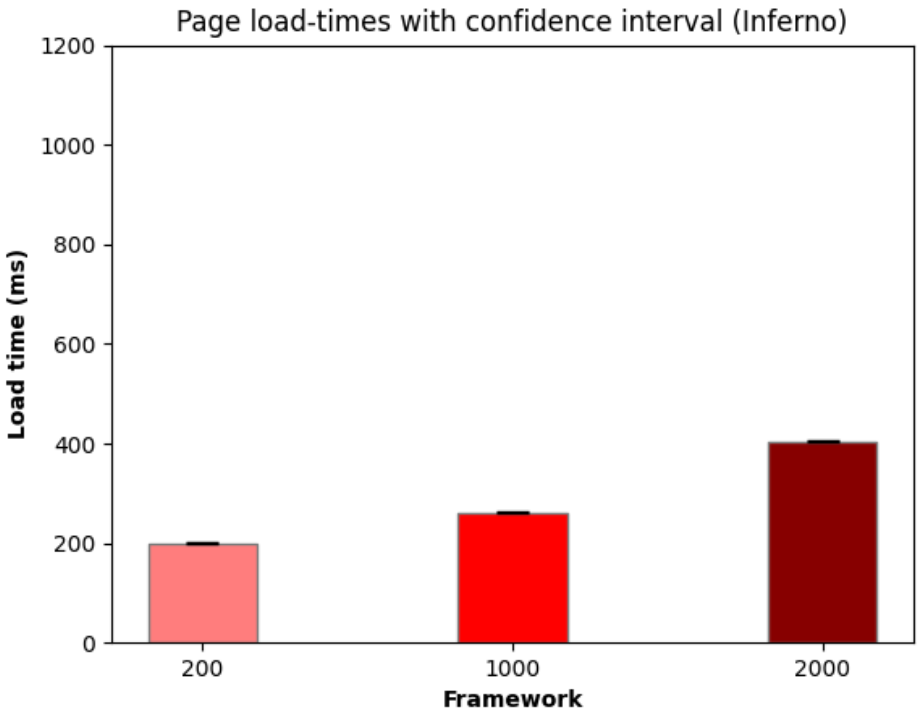
Figur 40. Stapeldiagram Inferno 200, 1000 och 2000 produkter med standardavvikelse.

I stapeldiagrammet ovan (se figur 40) går det att se en jämförelse mellan olika laddningstider gjorda på en Inferno-applikation med olika datamängd, 200 (ljusröd), 1000 (röd) och 2000 produkter (mörkröd). I detta diagram inkluderas även standardavvikelsen. En tabell har skapats för att tydligare visa standardavvikelsen (Se Tabell 10).

200 produkter	15.580
1000 produkter	26.521
2000 produkter	27.664

Tabell 10. Standardavvikelse för 200, 1000 och 2000 produkter i Inferno (avrundat till tre decimaler).

I Tabell 10 går det att se skillnader mellan standardavvikelsen för de mätningar som gjorts på Inferno-applikationen. I tabellen går det att se att skillnaden är markant mellan 200 och 1000 produkter vilket antyder att det finns en betydande ökning av spridningen i laddningstiderna när datamängden ökar från mindre datamängder till mer medelstora. Däremot är skillnaden i standardavvikelse mellan 1000 och 2000 produkter mindre, vilket tyder på att den ökade spridningen eller variationen i laddningstiderna inte ökar avsevärt ytterligare när datamängden ökar mer.



Figur 41. Stapeldiagram Inferno 200, 1000 och 2000 produkter med konfidensintervall.

I stapeldiagrammet i figur 41 visas en jämförelse mellan olika laddningstider gjorda på en Inferno-applikation med olika datamängd, 200 (ljusröd), 1000 (röd) och 2000 produkter (mörkröd). Stapeldiagrammet innehåller även konfidensintervallet för varje gjord mätning och visas genom felstaplarna på toppen av de färgade staplarna. För en tydligare presentation av konfidensintervallet har en tabell skapats (se Tabell 11).

Antal produkter	Lägre gräns	Övre gräns
200	199.340	201.283
1000	259.623	262.925
2000	403.076	406.520

Tabell 11. Konfidensintervall för 200, 1000 och 2000 produkter i Inferno (avrundat till tre decimaler).

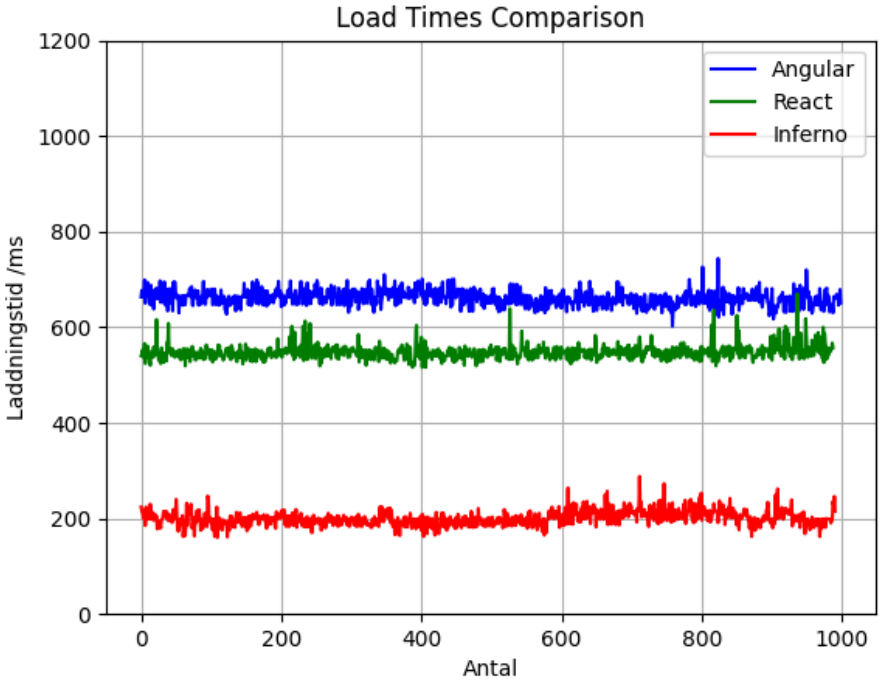
För konfidensintervallet har en konfidensnivå på 95% använts. Detta innebär att det är 95% säkerhet att det sanna medelvärdet för populationen ligger inom det beräknade intervallet. Intervallet är mellan den lägre gränsen och den övre gränsen. Efter att ha jämfört konfidensintervallen för Inferno märks det att intervallet är relativt lika mellan de olika datamängderna. Vilket kan indikera på att osäkerheten i uppskattningen av medelvärdet för laddningstiden inte ökar mycket när antalet produkter ökar.

ANOVA Statistic: 19122.96913901954 and p-value: 0.0 The means are different.

Figur 42. Anova test för Inferno mätningarna med 200, 1000 och 2000 produkter.

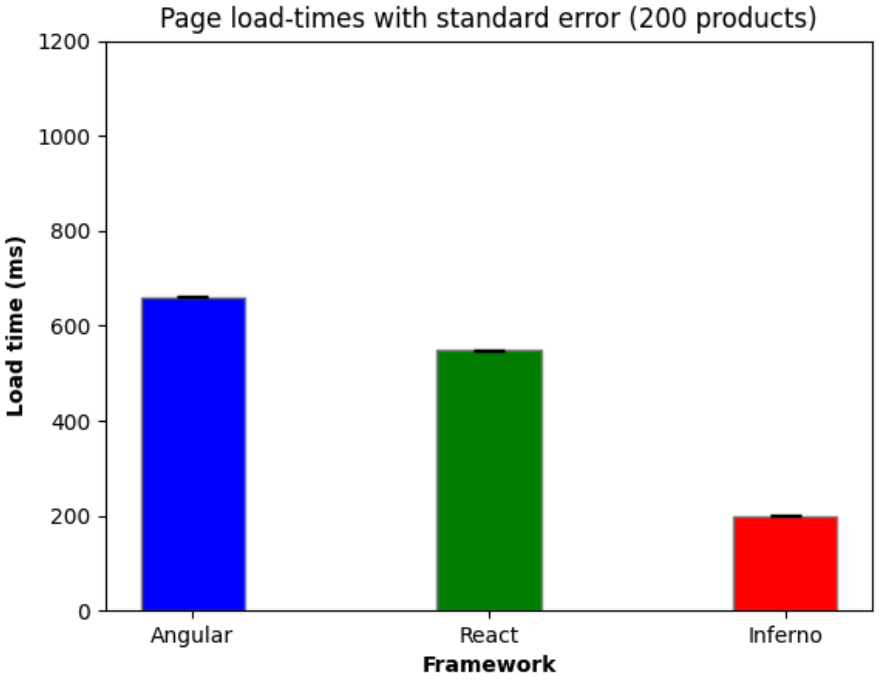
I ANOVA-testet visas resultatet "The means are different" vilket tyder på att det finns en tydlig skillnad på mätningarna gjorda på olika datamängder i Inferno-applikationen.

7.1.4 Jämförelse av ramverken med 200 produkter



Figur 43. Linjediagram React, Angular och Inferno med 200 produkter.

I linjediagrammet (se figur 43) går det att se skillnaderna på laddningstider mellan de tre olika ramverken React, Angular och Inferno med en datamängd av 200 produkter. I linjediagrammet visar sig Inferno ha de absolut snabbaste laddningstiderna till följd av React och sedan sist Angular med de längsta laddningstiderna på en datamängd av 200 produkter.



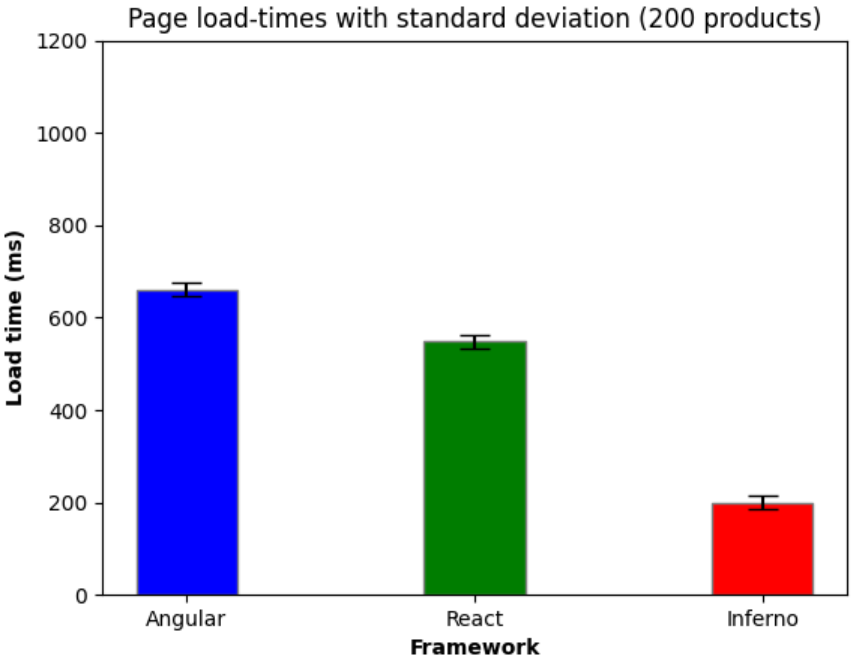
Figur 44. Stapeldiagram React, Angular och Inferno (200 produkter) med standardfel .

I stapeldiagrammet i figur 44 går det att se skillnader i laddningstid mellan React, Angular och Inferno med en datamängd på 200 produkter. Inferno visar de snabbaste laddningstiderna med React efter och sedan Angular med de längsta laddningstiderna med denna datamängd. I stapeldiagrammet har även standardfel inkluderats, vilket i stapeldiagrammet är ganska otydligt så därför har en tabell skapats för att visa siffrorna på standardfelet mellan ramverken (Se tabell 12).

Angular	0.499
React	0.479
Inferno	0.494

Tabell 12. Standardfel för Angular, React och Inferno med en datamängd på 200 produkter (avrundat till tre decimaler).

I Tabell 12 går det att se skillnader i standardfel mellan ramverken Angular, React och Inferno med en datamängd på 200 produkter. Trots att det finns variationer i standardfelet mellan ramverken är skillnaderna relativt små. Detta gör det utmanande att dra klara slutsatser enbart baserat på standardfelet. Det verkar finnas en liknande grad av osäkerhet i uppskattningen av medelvärdet för laddningstiden för de olika ramverken med denna datamängd.



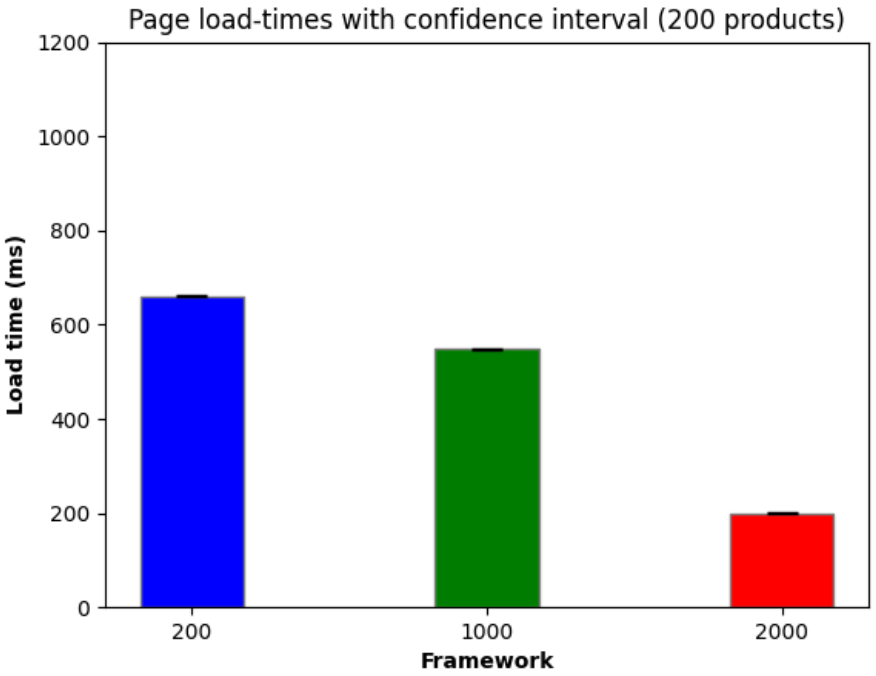
Figur 45. Stapeldiagram React, Angular och Inferno (200 produkter) med standardavvikelse.

I stapeldiagrammet i figur 45 går det att se skillnaderna på laddningstider mellan de tre olika ramverken React, Angular och Inferno med en datamängd av 200 produkter. I stapeldiagrammet är även standardavvikelse inkluderat, de svarta fel staplarna i toppen av de färglagda staplarna. Det är svårt att se några större skillnader i standardavvikelsen i stapeldiagrammet så därför har en tabell skapats för att visa skillnaderna i siffror (Se Tabell 13).

Angular	15.778
React	15.067
Inferno	15.580

Tabell 13. Standardavvikelse för Angular, React och Inferno med en datamängd på 200 produkter (avrundat till tre decimaler).

I tabell 13 går det att se standardavvikelsen mellan de olika ramverken React, Angular och Inferno med en datamängd på 200 produkter. Trots att det finns variationer i standardavvikelsen mellan ramverken är skillnaderna relativt små. React uppvisar de lägsta värdena av standardavvikelse, vilket indikerar en mindre spridning eller variation i laddningstiderna jämfört med Inferno och Angular. Inferno har något högre standardavvikelse än React, medan Angular har den högsta standardavvikelsen bland de tre ramverken.



Figur 46. Stapeldiagram React, Angular och Inferno (200 produkter) med konfidensintervall.

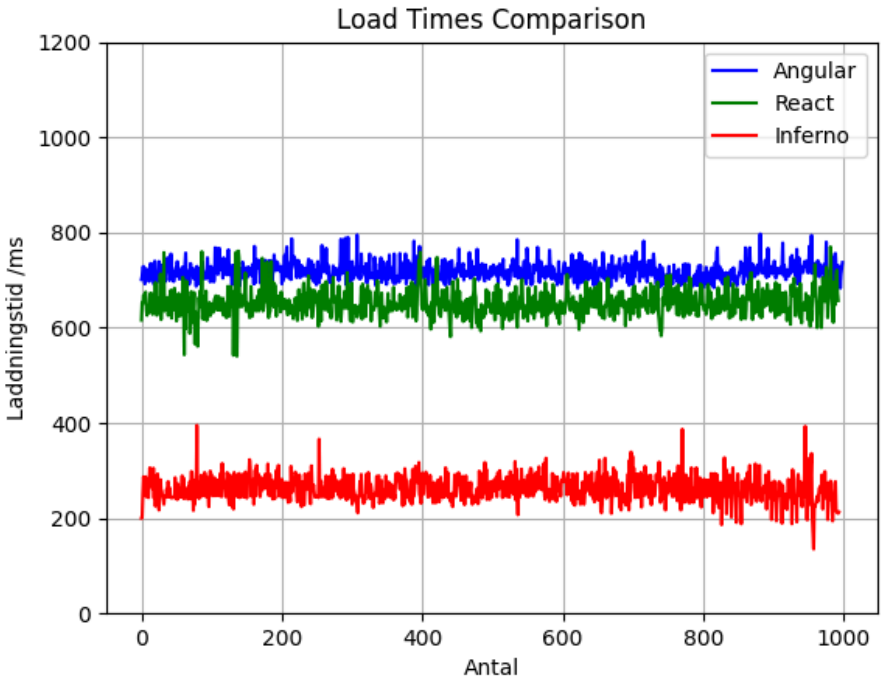
I stapeldiagrammet visas medelvärdet i laddningstid och konfidensintervall för de olika ramverken React, Angular och Inferno med en datamängd på 200 produkter. Då konfidensintervallet inte är tydligt nog i stapeldiagrammet har en tabell skapats för att skriva ut siffrorna av konfidensintervallet (Se Tabell 14).

Antal produkter	Lägre gräns	Övre gräns
Angular	660.246	662.205
React	546.996	548.877
Inferno	199.340	201.283

Tabell 14. Konfidensintervall React, Angular och Inferno med en datamängd på 200 produkter (avrundat till tre decimaler).

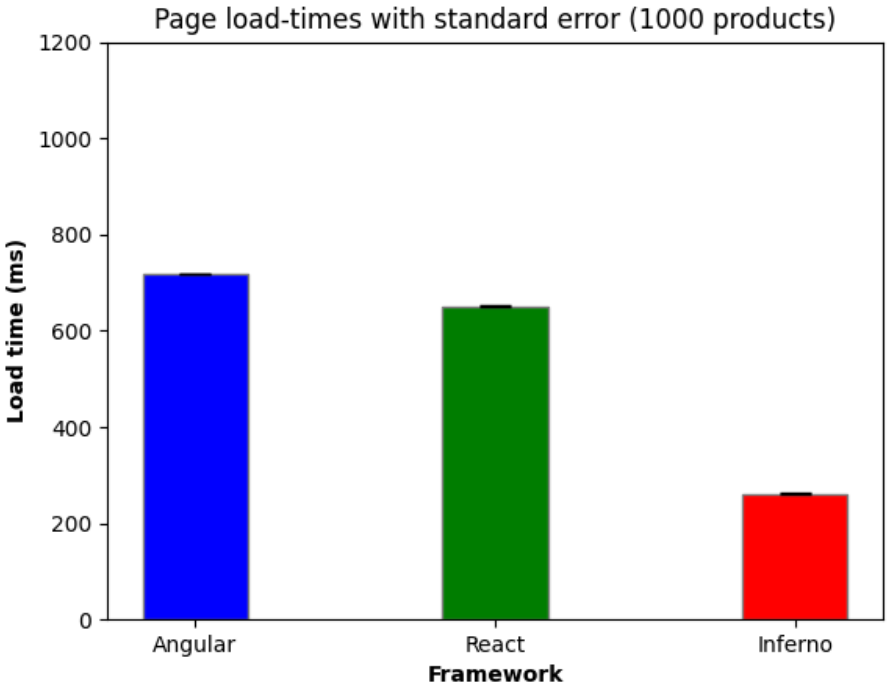
I Tabell 14 går det att konfidensintervallen mellan de tre olika ramverken Angular, React och Inferno med en datamängd på 200 produkter. Konfidensintervallet har en konfidensnivå på 95%, vilket innebär att det är 95% säkerhet att medelvärdet kommer att ligga innanför konfidensintervallet. I tabellen är intervallerna, alltså den lägre gränsen till den övre gränsen likartade mellan de olika ramverken.

7.1.5 Jämförelse av ramverken 1000 produkter



Figur 47. Linjediagram React, Angular och Inferno med 1000 produkter.

I linjediagrammet (se figur 47) går det att se skillnaderna på laddningstider mellan de tre olika ramverken React, Angular och Inferno med en datamängd på 1000 produkter. Inferno visar återigen de lägsta laddningstiderna med en liten ökning från mätningen med 200 produkter. Angular och React har också fått lite längre laddningstider men ligger nu närmare varandra än de gjorde i mätningen på 200 produkter. Vilket innebär att de presterar mer lika med en datamängd på 1000 produkter än vad de gjorde på 200 produkter.



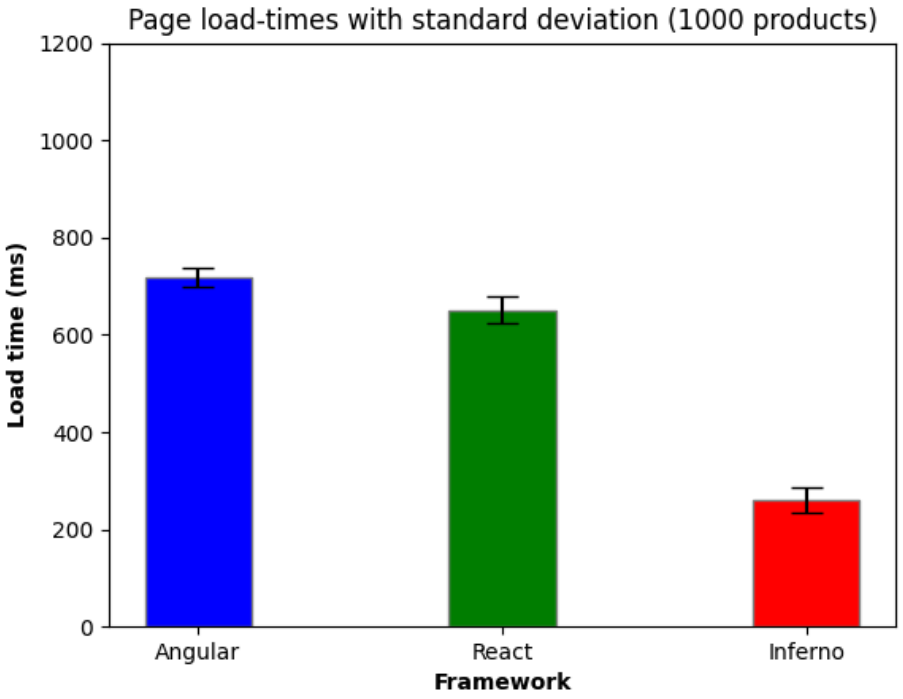
Figur 48. Stapeldiagram React, Angular och Inferno (1000 produkter) med standardfel.

I stapeldiagrammet (se figur 48) går det att se skillnader i laddningstid mellan React, Angular och Inferno med en datamängd på 1000 produkter. Inferno visar de snabbaste laddningstiderna med React efter och sedan Angular med de längsta laddningstiderna med denna datamängd. I stapeldiagrammet har även standardfel inkluderats, vilket i stapeldiagrammet kan vara ganska otydligt så därför har en tabell skapats för att visa siffrorna på standardfelet mellan ramverken (Se tabell 15).

Angular	0.594
React	0.878
Inferno	0.841

Tabell 15. Standardfel React, Angular och Inferno med en datamängd på 1000 produkter (avrundat till tre decimaler).

I tabellen ovanför (se tabell 15) går det att se skillnader i standardfel mellan ramverken Angular, React och Inferno med en datamängd på 1000 produkter. Både React och Inferno visar liknande standardfel medan Angular visar ett ganska tydligt lägre värde för standardfelet. Vid en datamängd på 200 produkter så visade alla ramverk liknande standardfel (se Tabell 12). Medan i denna datamängd (1000 produkter) så är Angulars standardfel lägre, vilket kan tyda på att Angular har mindre variation i mätningarna än React och Inferno och har en mer stabil prestanda vid större datamängd..



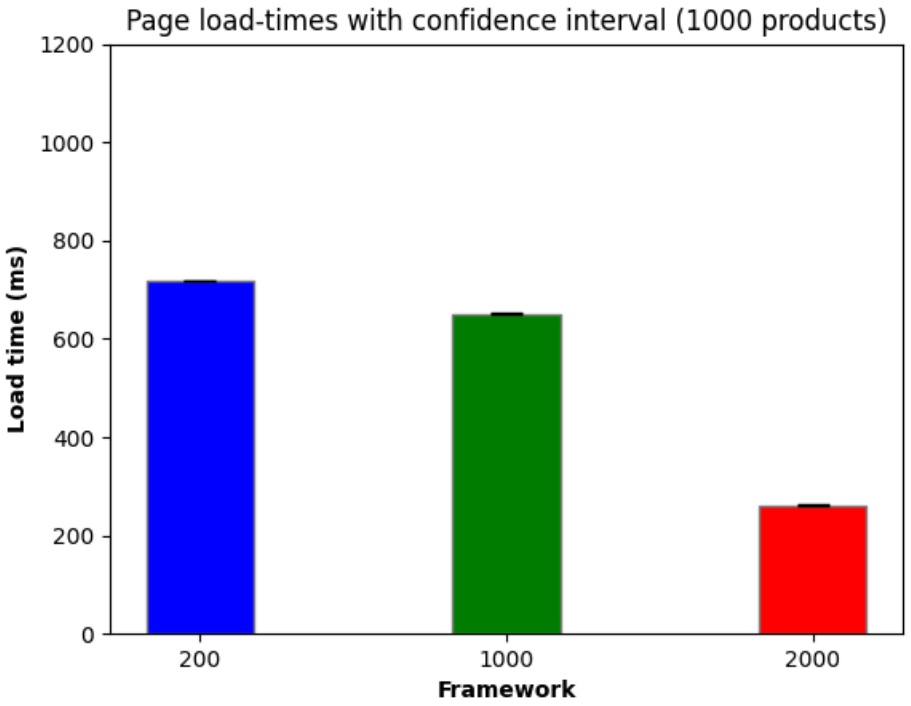
Figur 49. Stapeldiagram React, Angular och Inferno (1000 produkter) med standardavvikelse.

I stapeldiagrammet (Se figur 49) går det att se staplar som visar medelvärdet för laddningstiderna av de tre olika ramverken Angular, React och Inferno. I stapeldiagrammet går det även att se standardavvikelsen, då det är svårt att se stora skillnader i fel staplarna så har en tabell skapats för att visa tydligare skillnaden i standardavvikelse mellan mätningarna (Se Tabell 16).

Angular	18.793
React	27.698
Inferno	26.521

Tabell 16. Standardavvikelse React, Angular och Inferno med en datamängd på 1000 produkter (avrundat till tre decimaler).

I tabellen går det att se standardavvikelsen mellan de tre ramverken Angular, React och Inferno med en datamängd på 1000 produkter. Angular har lägst värde när det kommer till standardavvikelse vilket antyder en mindre spridning eller variation i laddningstiderna jämfört med React och Inferno. React och Inferno har snarlika värden vilket betyder att deras spridning eller variation är relativt lika vid denna datamängd.



Figur 50. Stapeldiagram React, Angular och Inferno (1000 produkter) med konfidensintervall.

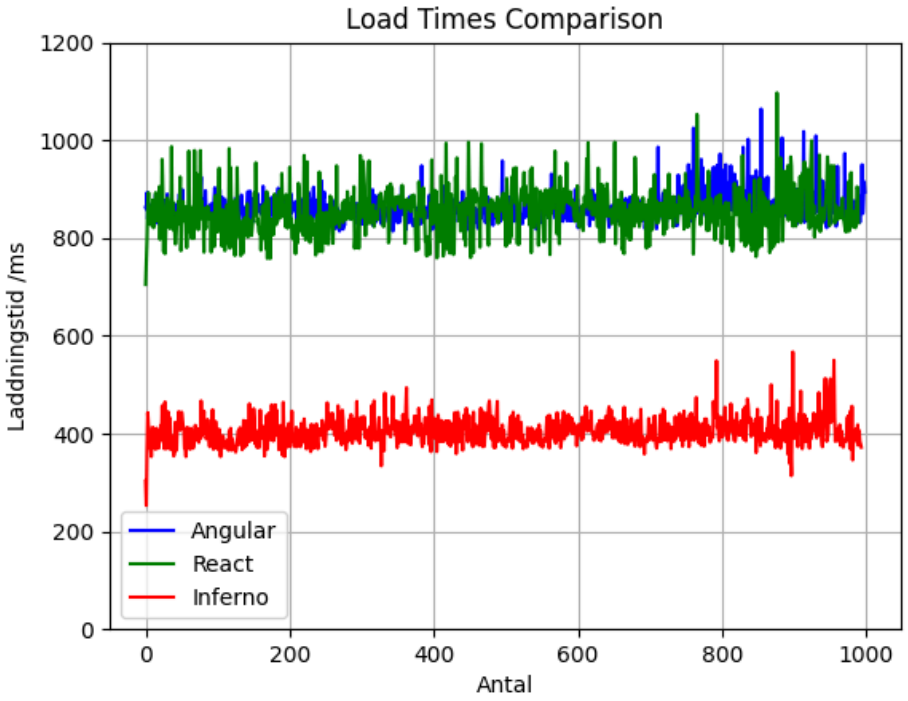
I stapeldiagrammet (Se figur 50) går det att se staplar som visar medelvärdet för laddningstiderna av de tre olika ramverken Angular, React och Inferno. Men i stapeldiagrammet går det även att se konfidensintervall. Då det är svårt att se skillnader i fel staplarna så har en tabell skapats för att tydligare visa skillnaden i konfidensintervallen mellan mätningarna (Se Tabell 17).

Ramverk	Lägre gräns	Övre gräns
Angular	717.295	719.629
React	649.596	653.046
Inferno	259.623	262.925

Tabell 17. Konfidensintervall React, Angular och Inferno med en datamängd på 1000 produkter (avrundat till tre decimaler).

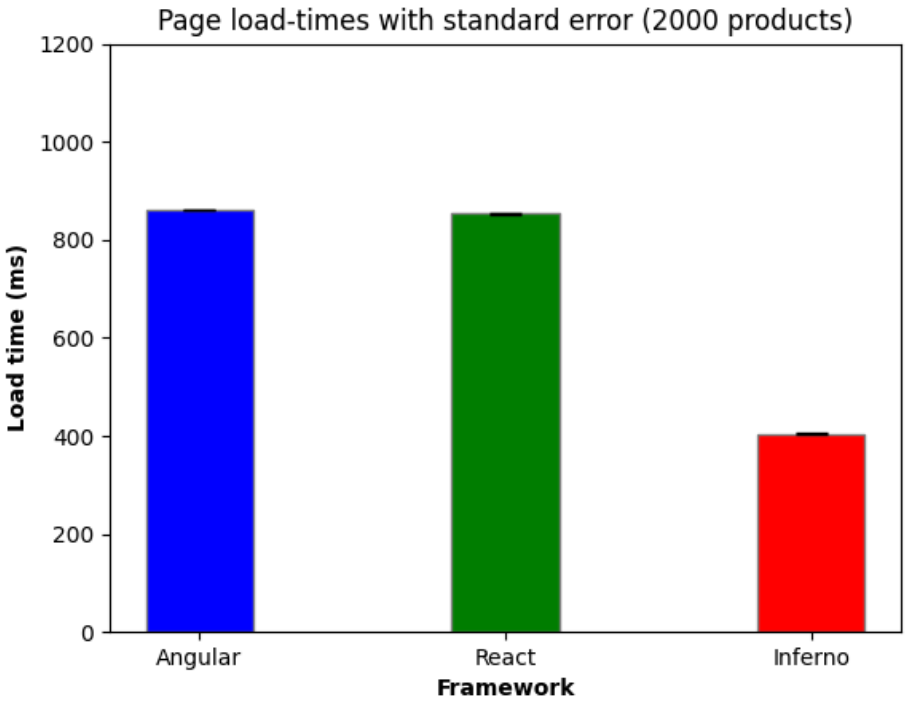
I tabell 17 går det att se konfidensintervallen mellan de tre olika ramverken Angular, React och Inferno med en datamängd på 1000 produkter. Konfidensintervallet har en konfidensnivå på 95% vilket innebär att det är 95% säkerhet att medelvärdet kommer att ligga innanför intervallet. Intervallet är mellan den lägre gränsen och den övre gränsen. Skillnaderna mellan de tre olika mätningarna är väldigt små så konfidensintervallet ligger runt samma för de tre olika ramverken vid denna datamängd.

7.1.6 Jämförelse av ramverken 2000 produkter



Figur 51. Linjediagram React, Angular och Inferno med 2000 produkter.

I detta linjediagram (Se figur 51) så går det att se laddningstiderna för e-handelssidan skapad i Angular, React och Inferno med en datamängd på 2000 produkter. Inferno visar de lägsta laddningstiderna på ett medelvärde på cirka 400 ms. I denna mätning till skillnad från de tidigare mätningarna så visar React och Angular ungefär samma medelvärde.



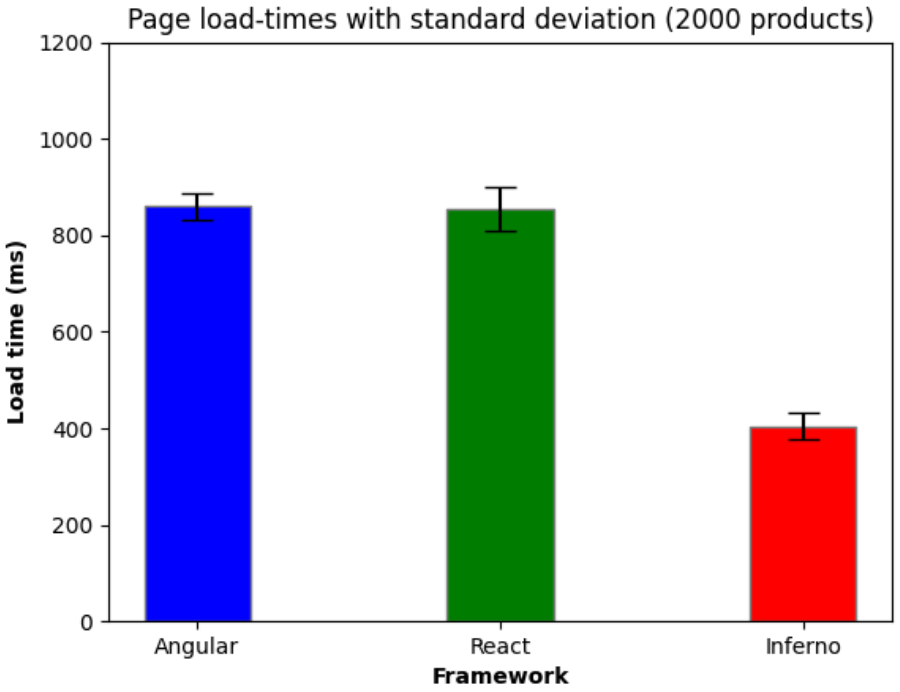
Figur 52. Stapeldiagram React, Angular och Inferno (2000 produkter) med standardfel. (Tabell 18)

I stapeldiagrammet i figur 52 går det att se ett stapeldiagram med medelvärdet av laddningstiden samt standardfel. Då fel staplarna inte visuellt går att tyda så har en tabell skapats för att presentera värdet för standardfelet (Se Tabell 18).

Angular	0.865
React	1.457
Inferno	0.877

Tabell 18. Standardfel React, Angular och Inferno med en datamängd på 2000 produkter (avrundat till tre decimaler).

I tabell 18 går det att se standardfelet för de olika applikationerna med en datamängd på 2000 produkter. Angular och Inferno visar liknande standardfel medan React sticker ut med en mycket högre siffra på ca 1.4. Vilekt kan tyda på att Angular och Inferno hanterar större datamängder med konsekvent än React, då värdet för React nästan är det dubbla gentemot Inferno och Angular.



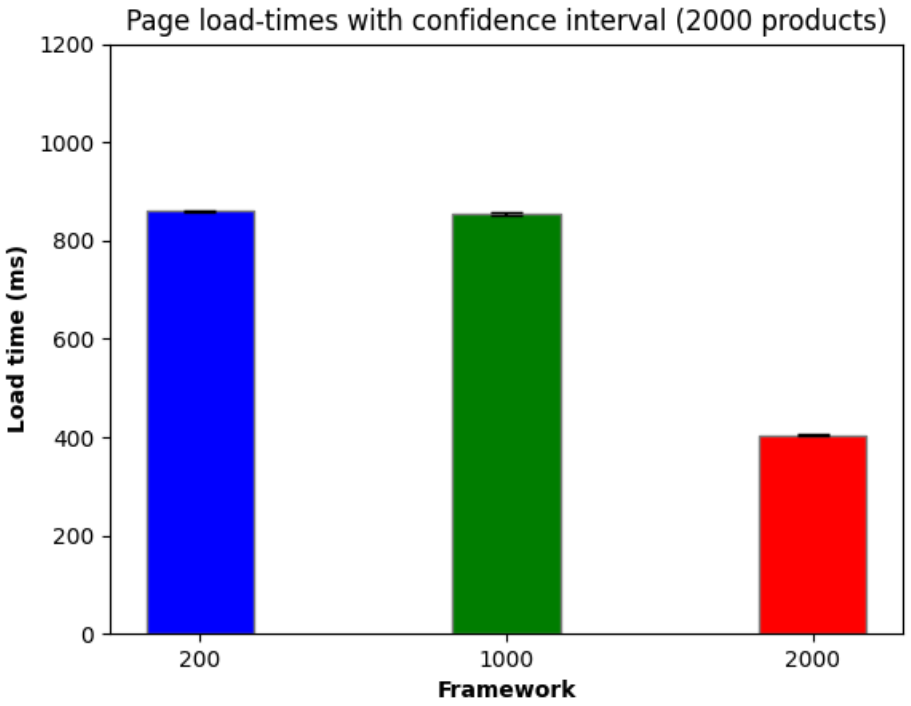
Figur 53. Stapeldiagram React, Angular och Inferno (2000 produkter) med standardavvikelse. (Tabell 19)

I stapeldiagrammet i figur 53 går det att se ett stapeldiagram med medelvärdet av laddningstiden men även standardavvikelse. Då det inte går att se större skillnader i fel staplarna så har en tabell skapats för att presentera värden (Se Tabell 19).

Angular	27.371
React	45.863
Inferno	27.664

Tabell 19. Standardavvikelse React, Angular och Inferno med en datamängd på 2000 produkter (avrundat till tre decimaler).

I denna tabell med standardavvikelsen för de olika applikationerna så går det att se skillnader och likheter. Angular och Inferno visar ungefär samma standardavvikelse omkring 27 medan React ännu en gång visar ett högre värde på ca 46. Vilket kan tyda på att React inte kan hantera större datamängder lika konsekvent som Inferno och Angular.



Figur 54. Stapeldiagram React, Angular och Inferno (2000 produkter) med konfidensintervall. (Tabell 20)

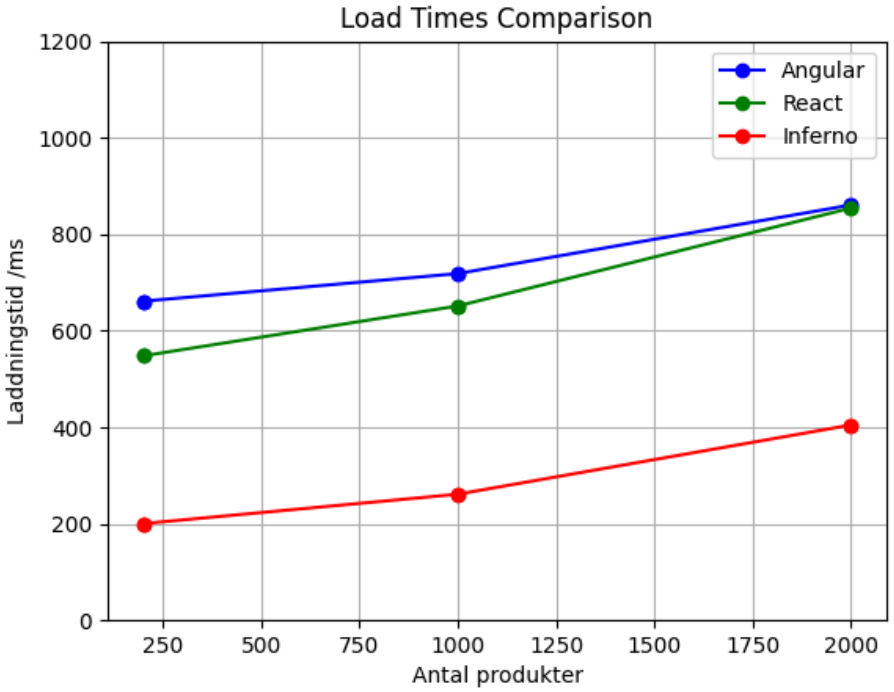
I stapeldiagrammet (se figur 54) går det att se staplar som visar medelvärdet för laddningstiderna av de tre olika ramverken Angular, React och Inferno med en datamängd på 2000 produkter. Men i stapeldiagrammet så går det även att se konfidensintervall. Då det är svårt att se skillnader i fel staplarna så har en tabell skapats för att tydligare visa skillnaden i konfidensintervallen mellan mätningarna (Se Tabell 20).

Ramverk	Lägre gräns	Övre gräns
Angular	859.020	862.419
React	851.488	857.208
Inferno	403.076	406.520

Tabell 20. Konfidensintervall React, Angular och Inferno med en datamängd på 2000 produkter (avrundat till tre decimaler).

I tabell 20 går det att se konfidensintervallen mellan de tre olika ramverken Angular, React och Inferno med en datamängd på 2000 produkter. Konfidensintervallet har en konfidensnivå på 95% vilket innebär att det är 95% säkerhet att medelvärdet kommer att ligga innanför intervallet. Intervallet är mellan den lägre gränsen och den övre gränsen. Angular och Inferno har ett liknande intervall medan React har ett intervall på nästan det dubbla jämfört med Angular och Inferno.

7.1.7 Samkörning Diagram React, Angular & Inferno



Figur 54. Linjediagram Angular (Blå linje), React (Grön linje) och Inferno (Röd linje) med punkter för de olika mätningarna.

I figur 54 går det att se ett linjediagram med alla gjorda mätningar. Linjediagrammet visar ökningen av laddningstiderna när datamängden ökar. På x-axeln visas ett antal produkter, där punkterna ligger på 200, 1000 och 2000 produkter. På y-axeln syns laddningstiden i millisekunder. Punkterna visar alltså medelvärdet av laddningstiden för ramverken på en viss datamängd. I figur 54 går det att se att inferno ökar mest i laddningstid när datamängden ökar, cirka 200 ms. Vid 200 produkter presterade React lägre laddningstider än Angular men det tenderar att jämna ut sig när datamängden ökar, vilket går att se på punkterna för 2000 produkter.

7.2 Slutsats

Slutsatsen med denna undersökningen var att ta reda på ifall det går att se skillnader i laddningstider hos React, Angular och Inferno, för att ta reda på vilket av ramverken som prestandamässigt är mest lämpligt för en e-handelssida med krav på snabba laddningstider. Till experimentet sattes frågeställningen och hypotesen:

Fråga: Vilket ramverk, mellan React, Angular och Inferno, visar de kortaste laddningstiderna för e-handelssidor?

Hypotes: Inferno förväntas visa de lägsta laddningstiderna jämfört med React och Angular.

Efter att ha genomfört flera tester och analyserat resultaten av laddningstiderna, visas det tydligt att Inferno var snabbast, vilket framförallt går att se i linjediagrammen i figur 43, 47 och 51. Figurerna illustrerar att Inferno konsekvent presterade bättre laddningstider än både React och Angular i tester med olika antal produkter. För 200 produkter hade Inferno de kortaste laddningstiderna med ett medelvärde på 200.311 ms jämfört med React som hade ett medelvärde på 547.937 ms och Angular med ett medelvärde på 661.226 ms. För 1000 produkter visade Inferno återigen de snabbaste laddningstiderna med ett medelvärde på 261.275 ms medan React hade 651.321 ms och Angular 718.462 ms. Även för 2000 produkter visade Inferno de lägsta laddningstiderna med ett medelvärde på 404.799 ms jämfört med Reacts 854.348 ms och Angulars 860.720 ms. Med det experiment som genomförts kan vi bekräfta att hypotesen visade sig stämma. Inferno hade de lägsta laddningstiderna i samtliga tester och är därför det ramverk som prestandamässigt är mest lämpligt för en e-handelssida med krav på snabba laddningstider.

Frågeställningen, "Vilket ramverk, mellan React, Angular och Inferno, visar de kortaste laddningstiderna för e-handelssidor?", har således fått ett tydligt svar genom experimentet. Inferno visade tydligt de kortaste laddningstiderna och kan därmed ses som det mest lämpliga valet för en e-handelssida med höga krav på snabba laddningstider.

8. Avslutande diskussion

8.1 Sammanfattning

Syftet med studien var att ta reda på prestanda skillnader i form av laddningstider mellan tre valda ramverk, React, Angular och Inferno. Artefakterna skapades för att efterlikna en e-handelssida med ett dataset som består av produkter i form av bilder och text om olika kläder. Datan som användes i artefakten hämtades från ett dataset kallat Mango Products[26] som används under MIT-licensen. Zhou, Giyane & Nyasha (2013) nämner att de flesta internetanvändarna helt enkelt kommer att välja att stänga ner webbplatsen om de inte lyckas ladda inom genomsnittet av 8 sekunder. Kravet på laddningstider kan vara avgörande särskilt för e-handelssidor där långa laddningstider kan bidra till förlorade kunder. Detta ligger till grund för experiment som jämför olika ramverks laddningstider på en e-handelssida. För att ta reda på vilket av de valda ramverken React, Angular och Inferno som presterar de kortaste laddningstiderna på en e-handelssida med varierande mängd produkter.

8.2 Diskussion

Arbetet har visat att det är möjligt att mäta laddningstider för en e-handelssida med tre olika ramverk: React, Angular och Inferno, samt identifiera skillnader i prestanda mellan dem. Resultaten från experimentet visar att Inferno konsekvent presterade de kortaste laddningstiderna, vilket stämmer överens med den ursprungliga hypotesen.

Inferno visade betydande fördelar när det gäller laddningstider för alla tester innehållandes olika mängder produkter 200, 1000 och 2000 (se figur 43, 47 och 51). Det är värt att notera att under mätningarna observerades en tydlig ökning av laddningstiderna för Inferno, vilket indikerar en exponentiell försämring av laddningstider med en ökande datamängd. Denna observation antyder att Infernos prestanda i form av laddningstider kan påverkas mer av stora datamängder jämfört med vad React och Angular gör. Trots skillnaderna, bekräftar resultaten att Inferno ändå var det snabbaste alternativet baserat på laddningstider för att ladda en e-handelssida.

I vissa mätningar för React med 2000 produkter (se figur 52 & 53) blev standardfelet och standardavvikelsen relativt stora jämfört med de andra ramverken samt de tidigare testerna. Denna observation väcker frågor om tillförlitligheten hos mätningarna för React. En av de möjliga förklaringarna till denna variation kan vara att det uppstod störningar i mätningarna, vilket kan ha påverkat resultaten. Även om sådana variationer förekommer, bekräftar de flesta andra mätningarna att våran huvudsakliga slutsats om prestandan i form av laddningstider stämmer för React, Angular och Inferno.

Sammanfattningsvis visar studien att Inferno är det ramverk som visade snabbast laddningstider, för att ladda en e-handelssida med varierande produktmängd. React visade de näst snabbaste laddningstiderna och Angular de långsammaste laddningstiderna. Ökningen, procentuellt mellan ramverken och datamängderna visar att Inferno hade störst procentuell ökning, React näst störst och Angular med den minsta ökningen procentuellt över de olika testerna.

[26] https://www.kaggle.com/datasets/maparla/mango-products?select=store_mango.csv

8.2.1 Etiska aspekter

All kod som har använts för att skapa artefakterna har sparats och finns på GitHub (GitHub Examensarbete 2024). Det bidrar till att det blir möjligt att återskapa artefakterna och göra en upprepning av experimentet. User Scriptet som använts i tampermonkey för att genomföra mätningarna av artefakterna, finns dokumenterade i denna rapport under Appendix A.

I och med att experimentet har utförts på en lokal dator finns det vissa aspekter att beakta när det gäller reproducerbarheten av resultaten. Den externa validiteten vilket gäller generaliserbarheten av resultaten kan påverkas. Skillnader i hårdvara och mjukvara mellan den lokala datorn och andra datorer kan potentiellt påverka framtida resultat. Detta innebär att experimentets resultat kanske inte är lätta att applicera i ett bredare sammanhang. För att underlätta för framtida replikeringar eller validering av experimentet har specifikationerna för den hårdvaran och mjukvaran som användes dokumenterats och kan ses i figur 23.

För att minimera variationer som kan uppstå till följd av nätverksanslutningar har Chrome DevTools Throttling använts. Genom att använda verktyget har samma nätverkshastighet konsekvent använts i alla mätningar, oavsett tidpunkt eller dag då mätningarna utfördes. Throttling bidrar till att minimera variationer och för att säkerställa jämförbara resultat över olika tidsperioder. Dessutom möjliggör verktyget att samma nätverksanslutning kan simuleras vid framtida mätningar. När man använder Chrome DevTools Throttling för att bevara en konsekvent nätverkshastighet kan det ha praktiska fördelar, men det är viktigt att överväga de potentiella validitetshoten detta kan leda till. Då detta kan ha påverkan på generaliserbarheten och den externa validiteten av studiens resultat, då resultaten kanske inte reflekterar hur användare kommer att uppleva applikationerna i verkliga scenarion.

Experimentet har utförts med ambitionen av att vara så objektiv som möjligt och resultaten har lämnats från modifiering i den mån det har gått, endast väldigt tydliga och icke-sammanhängande "spikar" har tagits bort från mätningarna för att bevara tillförlitligheten hos den insamlade datan. Att ta bort “spikar” från mätningarna kan ses som ett potentiellt validitetshot, då det kan leda till urvalsbias. Trots att objektiviteten har varit viktig i detta experiment så är det viktigt att vara medveten om risken för urvalsbias, och att överväga noggrant om att ta bort vissa “spikar”.

Inga människor har inkluderats i denna studie så därför har ingen person kommit till skada av arbetet.

8.2.2 Samhällsnytta

Genomförandet av experimentet syftar till att undersöka vilket av ramverken React, Angular och Inferno som presterade de kortaste laddningstiderna på en e-handelssida. Denna fråga är av stor vikt för företag online, då studier som Zhou, Giyane & Nyasha (2013) påpekar att majoriteten av internetanvändarna tenderar att överge en webbplats om den inte laddar inom genomsnittet av 8 sekunder.

Genom att identifiera vilket ramverk som presterar bäst i termer av laddningstider kan utvecklare och företag fatta mer välgrundade beslut vid utvecklingen av e-handelsplattformar. Vilket enligt Vihervaara & Alapahuluoma (2018) så har flera studier betonat vikten av att organisationer verifierar laddningstiderna för sina

webbsidor. Vilket är särskilt viktigt för e-handelssektorn eftersom snabba webbplatser uppnår högre lönsamhet. Laddningstiderna för e-handelssidor har en mätbar effekt på olika affärs metriker, som exempelvis varumärkesuppfattning, konvertering, intäkter, övergivna varukorgar och sidvisningar. Att vara medveten om skillnaderna i ramverkens prestanda är viktigt för både företag och användare.

Slutligen kan resultatet av experimentet också bidra till den bredare samhällsnyttan genom att främja utvecklingen av effektivare och mer användarvänliga webbteknologier. Genom att öka förståelsen för hur olika ramverk påverkar e-handel sidornas prestanda i form av laddningstider kan forskare och utvecklare fortsätta att förbättra och optimera teknikerna för att bli ännu snabbare.

8.3 Framtida arbete

I framtida arbeten hade det varit intressant att inkludera en ännu större datamängd eller ett annat dataset för att se hur skillnaderna mellan ramverkens prestanda i termer av laddningstider hade förändrats. Framförallt eftersom det i det här experimentet visade sig att Inferno hade de lägsta laddningstiderna men den största procentuella ökningen mellan testerna när datamängden ökade. Så genom att öka mängden produkter i framtida mätningar, så kanske resultaten visar att Inferno eventuellt är det långsammaste alternativet vid större datamängder.

En annan intressant aspekt i framtida arbeten hade varit att göra jämförelser på olika dataset, till exempel ett dataset som innehåller andra typer av produkter. Vidare skulle det vara relevant och intressant att undersöka andra aspekter av prestanda, såsom minnesanvändning och CPU-belastning, för att få en mer heltäckande bild av ramverkens prestanda.

Eftersom Chrome Dev Tools Throttling användes i experimentet för att ha samma nätverksanslutning, hade det i framtida arbeten varit intressant att utföra tester under olika nätverksförhållanden. För att ge insikter om hur React, Angular och Inferno presterar i varierande miljöer, vilket kan vara relevant för att förstå deras användbarhet i mer verkliga scenarier.

Då experimentet skedde på webbläsaren Google Chrome, skulle det vara intressant att inkludera tester över olika webbläsare för att se ifall det finns skillnader i hur ramverken presterar över olika webbläsare. Vilket kan ge en mer omfattande bild av ramverkens prestanda och deras användbarhet i olika miljöer och användarscenarier.

Ifall det fanns mer tid i framtida projekt hade det varit intressant att utforska möjligheten att skapa VR-baserade e-handelsplattformar där användare kan utforska och interagera med produkter i en realistisk 3D-miljö. Vilket skulle kräva mer tid och resurser än till detta experiment. Men det hade varit intressant att undersöka prestandan för olika ramverk på VR-baserade e-handelsplattformar.

Slutligen skulle det vara intressant att utforska andra ramverk som inte har ingått i just denna studie. Genom att inkludera fler eller andra ramverk kan vi få en bredare förståelse för de olika alternativ som finns tillgängliga för utvecklare, och hur ramverken jämför sig med varandra i termer av prestanda.

Referenser/References

Amjad, M., Hossain, M. T., Hassan, R., & Rahman, M. A. (2021). Web application performance analysis of E-commerce sites in Bangladesh: an empirical study. *International Journal of Information Engineering and Electronic Business*, 13(2), 47-54. DOI: 10.5815/ijieeb.2021.02.04 [2024-06-07]

Clark, I. L. (2006). Writing the Successful Thesis and Dissertation: Entering the Conversation. [2024-2-20]

Creswell, J. W. (2009). *Research Design – Qualitative, Quantitative and Mixed methods Approaches*. New Delhi: Sage Publication. [2024-2-14]

GitHub Examensarbete. https://github.com/b21willu/Examensarbete. [2024-5-15]

Gizas, A. B., Christodoulou, S. P., & Papatheodorou, T. S. (2012). *Comparative evaluation of JavaScript frameworks*. DOI: 10.1145/2187980.2188103. [2024-03-11]

Haider, W., Ilyas, M., Khalid, S., & Ali, S. (2023). *Factors influencing sustainability aspects in crowdsourced software development: A systematic literature review. Journal of Software: Evolution and Process.* https://doi.org/10.1002/smr.2630. [2023-12-14]

IT Governance Privacy Team. (2016) .EU General Data Protection Regulation (GDPR): An Implementation and Compliance Guide. [2024-2-20]

Mariano, C. L. (2017) *Benchmarking JavaScript Frameworks*. Masters dissertation, 2017. doi:10.21427/D72890 [2024-03-12]

Marx-Raacz Von Hidvég, T. (2022). Are the frameworks good enough? : A study of performance implications of JavaScript framework choice through load- and stress-testing Angular, Vue, React and Svelte, Dissertation,. [2023-12-12]

Nakajima, N., Matsumoto, S., & Kusumoto, S. (2019) ."Jact: A Playground Tool for Comparison of JavaScript Frameworks," *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Putrajaya, Malaysia, 2019, pp. 474-481, doi: 10.1109/APSEC48747.2019.00070. [2023-12-11]

Nayak, J. K., & Singh, P. (2015). Fundamentals of Research Methodology : Problems and Prospects. First ed. SSDN Publishers & Distributors [2024-2-13]

Pano, A., Graziotin, D., & Abrahamsson, P. (2018). Factors and actors leading to the adoption of a JavaScript framework. Empirical Software Engineering. 23. doi:10.1007/s10664-018-9613-x. [2023-12-11]

Ram, K. (2013). Git can facilitate greater reproducibility and increased transparency in science. *Source Code Biol Med* 8, 7 (2013). doi:10.1186/1751-0473-8-7. [2023-12-14]

Runeson, P., Höst, M., Rainer, A., & Regnell, B. (2012). .Case Study Research in Software Engineering: Guidelines and Examples. [2024-2-20]

Santana Roldán, C. (2023). *React 18 Design Patterns and Best Practices - Fourth Edition*. [2024-2-6]

Shavin, M (2023). *Learning Vue*. [2024-2-6]

Singh, P., Srivastava, M., Kansal, M., Singh, A. P., Chauhan, A., & Gaur, A. (2023). "A Comparative Analysis of Modern Frontend Frameworks for Building Large-Scale Web Applications," *2023 International Conference on Disruptive Technologies (ICDT)*, Greater Noida, India, 2023, pp. 531-535, doi: 10.1109/ICDT57929.2023.10150911. [2023-12-14]

Uluca, D. (2024). *Angular for Enterprise Applications - Third Edition*. [2024-2-6]

Vihervaara, J., & Alapaholuoma, T. (2018). "The impact of HTTP/2 on the service efficiency of e-commerce websites," *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, 2018, pp. 1317-1321, doi: 10.23919/MIPRO.2018.8400238. [2023-03-12]

Zhou, M., Giyane, M., & Nyasha, M. (2013). Effects of Web Page Contents on Load Time over the Internet. International Journal of Science and Research (IJSR). 2319-7064. [2024-2-13]

Appendix A - Tampermonkey userscript

```
(function() {
    'use strict';

    var maxDataLength = 1000; // Maximalt antal laddningstider att spara
    var loadTimesStr = localStorage.getItem('loadTimes'); // Hämta laddningstider
från localStorage
    var loadTimes = loadTimesStr ? JSON.parse(loadTimesStr) : []; // Konvertera
till array om det finns data, annars använd en tom array

    // Funktion för att mäta laddningstiden för sidan
    function measurePageLoadTime() {
        var loadTime = window.performance.timing.loadEventEnd -
window.performance.timing.navigationStart;
        loadTimes.push(loadTime); // Pusha laddningstiden till arrayen
        console.log('Load time:', loadTime, 'milliseconds');

        if (loadTimes.length >= maxDataLength) {
            console.log('Max data length reached. Stopping measurements.');
            saveLoadTimesToCSV(); // Spara laddningstiderna till en CSV-fil när
maxlängden har nåtts
        } else {
            saveLoadTimesToLocalStorage(); // Spara laddningstiderna till
localStorage innan sidan laddas om
            setTimeout(reloadProductPage, 2000); // Ladda om produktsidan efter en
fördröjning på 2 sekunder för att göra en ny mätning
        }
    }

    // Funktion för att ladda om produktsidan
    function reloadProductPage() {
        location.reload(); // Ladda om produktsidan
    }

    // Funktion för att spara laddningstiderna till localStorage
    function saveLoadTimesToLocalStorage() {
        localStorage.setItem('loadTimes', JSON.stringify(loadTimes)); // Spara
laddningstiderna till localStorage
    }

    // Funktion för att spara laddningstiderna till en CSV-fil
    function saveLoadTimesToCSV() {
        var csvContent = "data:text/csv;charset=utf-8,";
        loadTimes.forEach(function(time, index) {
            csvContent += time + "\n";
        });
        var encodedUri = encodeURI(csvContent);

        var appName = "Inferno";
        var fileName = "load_times_" + appName + ".csv";

        var link = document.createElement("a");
        link.setAttribute("href", encodedUri);
        link.setAttribute("download", fileName);
        document.body.appendChild(link);
        link.click();

        // Rensa localStorage efter att datan har sparats
        localStorage.removeItem('loadTimes');
    }
```

```
    // Funktion för att kontrollera om sidan är helt laddad
    function isPageLoaded() {
        return document.readyState === 'complete';
    }

    function waitForPageLoad() {
        if (isPageLoaded()) {
            measurePageLoadTime();
        } else {
            setTimeout(waitForPageLoad, 1000); // Kontrollera igen efter 1 sekund
        }
    }

    waitForPageLoad();

})();
```