İSTEMCİ TARAFLI JAVASCRIPT ÇERÇEVELERİNİN WEB SAYFASI YÜKLEME PERFORMANSINA ETKİLERİNİN İNCELENMESİ

Bekir YILMAZ

YÜKSEK LİSANS TEZİ
BİLGİSAYAR BİLİMLERİ ANA BİLİM DALI

GAZİ ÜNİVERSİTESİ

BİLİŞİM ENSTİTÜSÜ

ARALIK 2019

Bekir Yılmaz tarafından hazırlanan “İstemci Taraflı Javascript Çerçevelerinin Web Sayfası Yükleme Performansına Etkilerinin İncelenmesi” adlı tez çalışması aşağıdaki jüri tarafından OY BİRLİĞİ ile Gazi Üniversitesi Bilgisayar Bilimleri Anabilim Dalında YÜKSEK LİSANS TEZİ olarak kabul edilmiştir.

Danışman: Doç.Dr. Aslıhan TÜFEKCİ

Bilgisayar Bilimleri Ana Bilim Dalı, Gazi Üniversitesi

Bu tezin, kapsam ve kalite olarak Yüksek Lisans Tezi olduğunu onaylıyorum ...……

Başkan: Prof. Dr. Muhammet Ali AKCAYOL

Bilgisayar Bilimleri Ana Bilim Dalı, Gazi Üniversitesi

Bu tezin, kapsam ve kalite olarak Yüksek Lisans Tezi olduğunu onaylıyorum ...……

Üye: Doç.Dr. Gökhan ŞENGÜL

Bilgisayar Mühendisliği Ana Bilim Dalı, Atılım Üniversitesi

Bu tezin, kapsam ve kalite olarak Yüksek Lisans Tezi olduğunu onaylıyorum ...……

Tez Savunma Tarihi: 19/12/2019

Jüri tarafından kabul edilen bu tezin Yüksek Lisans Tezi olması için gerekli şartları yerine getirdiğini onaylıyorum.

……..…….

Doç. Dr. Aslıhan TÜFEKCİ

Bilişim Enstitüsü Müdürü

ETİK BEYAN

Gazi Üniversitesi Bilişim Enstitüsü Tez Yazım Kurallarına uygun olarak hazırladığım bu tez çalışmasında;

- Tez içinde sunduğum verileri, bilgileri ve dokümanları akademik ve etik kurallar çerçevesinde elde ettiğimi,
- Tüm bilgi, belge, değerlendirme ve sonuçları bilimsel etik ve ahlak kurallarına uygun olarak sunduğumu,
- Tez çalışmasında yararlandığım eserlerin tümüne uygun atıfta bulunarak kaynak gösterdiğimi,
- Kullanılan verilerde herhangi bir değişiklik yapmadığımı,
- Bu tezde sunduğum çalışmanın özgün olduğunu,

bildirir, aksi bir durumda aleyhime doğabilecek tüm hak kayıplarını kabullendiğimi beyan ederim.

Bekir YILMAZ

19/12/2019

İSTEMCİ TARAFLI JAVASCRIPT ÇERÇEVELERİNİN WEB SAYFASI YÜKLEME PERFORMANSINA ETKİLERİNİN İNCELENMESİ

(Yüksek Lisans Tezi)

Bekir YILMAZ

GAZİ ÜNİVERSİTESİ
BİLİŞİM ENSTİTÜSÜ
Aralık 2019

ÖZET

Bu çalışma kapsamında, istemci taraflı web uygulaması geliştirme sürecinde kullanılan çerçeveler ve ortaya koydukları yaklaşımların, web sayfası yükleme performansına etkileri araştırılmıştır. Çalışmada şablonlama ve dom oluşturma yaklaşımları referans uygulama üzerinde, araştırma kapsamında geliştirilen yardımcı performans ölçüm kütüphanesi ve gezinme zamanlaması programlama ara yüzü aracılığıyla farklı tarayıcı platformları üzerinde ölçülmüştür. Çalışmada 100 nesnelik veri boyutlarında çerçeve kullanımının dom oluşturma süresini ve buna bağlı olarak sayfa yükleme zamanını, kullanıcı tarafından farkedilebilir sürelerde etkilemediği sonucuna varılmaktadır. Veri boyutu 1000 nesneye çıkarıldığında ise sanal dom oluşturma yaklaşımının yükleme zamanları bağlamında düşük performans gösterdiği sonucuna ulaşılmaktadır. Sonuç olarak, istemci taraflı web uygulamalarında küçük veri boyutlarında ortaya konulan mimari yaklaşımların günümüz donanım kapasitesinde kullanıcı deneyimi düşünüldüğünde ihmal edilebilir sürelerde fark oluşturduğu gözlemlenmektedir. Ancak veri boyutu büyüdükçe çerçevelerin sayfa yükleme süreleri arasında oluşan farkın arttığı gözlemlenmiştir.

Bilim Kodu : 92408
Anahtar Kelimeler : Bilgisayar yazılımı, web uygulamaları, javascript çerçeveleri
Sayfa Adedi : 75
Danışman : Doç. Dr. Aslıhan TÜFEKCİ

ANALYZING THE EFFECTS OF CLIENT SIDE JAVASCRIPT FRAMEWORKS ON WEB PAGE LOAD PERFORMANCE

(M. Sc. Thesis)

Bekir YILMAZ

GAZİ UNIVERSITY

INFORMATICS INSTITUE

December 2019

ABSTRACT

In this study, the frameworks used in client-side web application development process and their effects on web page loading performance were investigated. Templating and dom construction approaches were measured on the reference application through different browser platforms with NavigationTimingAPI and with a performance measurement library developed within the scope of the research. We found that the usage of frameworks with data size up to 100 objects does not affect the page load time with a noticeable time period by the user. When the data size is increased to 1000 objects, it is shown that the virtual dom creation approach performs poorly in context of loading times. As a result, it is observed that the architectural approaches revealed in the small data sizes in client-side web applications make a difference in negligible periods in today's hardware capacity, considering the user experience. However, as the data size increased, the gap between the page load times of frameworks also increased.

Science Code : 92408
Keywords : Computer software, web applications, javascript frameworks
Page Count : 75
Supervisor : Assoc. Prof. Aslıhan TÜFEKCİ

TEŞEKKÜR

Bu çalışmanın gerçekleştirilmesinde Danışman Hocam Doç. Dr. Aslıhan TÜFEKCİ'ye, bana verdikleri destek için eşim Fulya ve Aileme sonsuz teşekkürlerimi sunarım.

İÇİNDEKİLER

Sayfa

Sayfa

ÇİZELGELERİN LİSTESİ

Çizelge **Sayfa**

ŞEKİLLERİN LİSTESİ

Şekil **Sayfa**

Şekil **Sayfa**

SİMGELER VE KISALTMALAR

Bu çalışmada kullanılmış bazı kısaltmalar, açıklamaları ile aşağıda sunulmuştur.

Kısaltmalar	Açıklama
API	Uygulama programlama arayüzü (Application programming ınterface)
AST	Soyut sözdizimi ağacı (Abstract syntax tree)
CDN	İçerik dağıtım ağı (Content delivery network)
CSS	Basamaklı stil sayfaları (Cascading style sheets)
CSSOM	Basamaklı stil sayfaları nesne modeli (Cascading style sheet object model)
DNS	Alan adı sunucusu (Domain name server)
DOM	Doküman nesne modeli (Document object model)
ECMA	Avrupa Bilgisayar Üreticileri Birliği
HTML	Hipermetin işaretleme dili (Hypertext markup language)
HTTP	Hipermetin transfer protokolü (Hypertext transfer protocol)
HTTPS	Güvenli hipermetin transfer protokolü
IP	Internet protokolü (Internet protocol)
JIT	Tam zamanında derleme (Just in time compilation)
JS	Javascript
JSON	Javascript nesne gösterimi (Javascript object notation)
LOC	Kod satırı sayısı (Lines of code)
MVC	Model görünüm denetleyici (Model view controller)
SLOC	Kaynak kod satır sayısı (Source lines of code)
TCP	İletim kontrol protokolü (Transmission control protocol)
TTI	Time to ınteractive
URL	Tektip kaynak konumlandırıcı (Uniform resource locator)
W3C	Dünya Çapında Ağ Konsorsiyumu (World wide web consortium)
WWW	Dünya çapında ağ
XML	Genişletilebilir işaretleme dili (Extensible markup language)

1. GİRİŞ

Internetin yaygınlaşması ile web uygulamaları, farklı alanlarda daha fazla sayıda kullanıcıya hizmet vermeye başlamıştır. Bu duruma bağlı olarak web uygulamalarının kullanılabilirlik, ölçeklenebilirlik ve performansa dayalı gereksinimleri artmış ve çeşitlilik göstermiştir. Artan kullanıcı ihtiyaçlarına yanıt verebilmek için web uygulamalarının işlevleri ve kullanılabirlik ihtiyaçları da artmış, bunun sonucu olarak uygulama işlevleri ve kod boyutları da büyüme göstermiştir. Büyüyen web uygulamalarının geliştirilmesi için daha geniş ekipler ile farklı yazılım mimarileri ve yazılım geliştirmeyi kolaylaştırıcı yaklaşımlar kullanılmaya başlanmıştır. Bu ihtiyaçlara yanıt verebilmek için web uygulama geliştirme sürecinde kullanılan kaynaklar, yaklaşımlar ve teknolojiler zamanla çeşitlilik göstermiştir (Mehdi, 2007).

Yazılım kodlarının geliştirici ekipleri tarafından sürdürülebilirliğinin sağlanması ve doğru şekilde yönetilebilmesi, web uygulamalarının artan ve değişen ihtiyaçlara hızlı şekilde cevap verebilmesi ve uygulama sürümlerinin hızlı ve doğru şekilde geliştirilerek devreye alınabilmesi gerekliliğini ortaya çıkarmıştır. Javascript çerçeveleri de istemci taraflı web uygulaması geliştirme süreçlerinde uygulama geliştirme sürecini hızlandırmak ve kolaylaştırmak, bakımı ve geliştirmesi daha kolay yapılabilir ve sürdürülebilir istemci taraflı web uygulamaları oluşturmak için ortaya çıkmıştır. Böylelikle geliştirme sürecinde seçilen teknoloji ve yaklaşımlar, web uygulaması mimarisinin doğru kurgulanabilmesini, yazılım geliştirme sürecini, yazılım performansını ve sonuç olarak yazılım geliştirme maliyetlerini etkileyen ana faktörler haline gelmiştir.

Google tarafından 2017 yılında yayınlanan araştırma raporu (Google, 2017), mobil platformlarda kullanıcıların %53'ünün yükleme süresi ortalama 3 saniyeyi geçen web sitelerini doğrudan terk ettiklerini ortaya koymaktadır. Bu doğrultuda sayfa yükleme zamanı, bir web sitesinin kullanıcı odaklı olarak performansını belirleyen en önemli ölçütlerden birisi olarak öne çıkmaktadır. Web uygulama performansı ile ilgili daha önce gerçekleştirilen çalışmalarda ise, uygulama performansınının ağırlıklı olarak Javascript performansına bağlı olduğu ortaya konulmuştur. Wang vd. diğerleri tarafından gerçekleştirilen çalışmada (Wang, Balasubramanian, Krishnamurthy and Wetherall, 2013) sayfa yükleme performansını analiz

edebilmek için WProf isimli bir araç önerilmiştir. Bu araç, tarayıcı içerisinde çalışarak, sayfa yüklemesini etkileyen aktivitelerin detaylı bağımlılık graflarını çizebilmektedir. Çalışmada sayfa yüklemesini etkileyen bağımlılıklar 4 ayrı kategori altında değerlendirilmiş ve sayfa yüklemesini en çok etkileyen etkenin %35 oranında Javascript kodunun çalıştırılmasına bağımlı olduğu belirlenmiştir. Bu doğrultuda, web uygulama geliştirme sürecinde, istemci tarafında tercih edilecek Javascript çerçevesi ve buna bağlı mimari yaklaşımların uygulama performansına göstereceği etkilerin öngörülebilmesi gerekmektedir.

Kullanılan çerçevelerin performansa etkisini öngörebilmek ve nedenlerini ortaya koyabilmek için literatürde daha önce çeşitli araştırmalar gerçekleştirilmiştir. Selakovic ve Pradel tarafından gerçekleştirilen araştırmada (Selakovic and Pradel, 2016), istemci ve sunucu taraflı 16 ayrı Javascript kütüphanesi için performansı etkileyen 8 ayrı temel neden ortaya koymuşlar, bu nedenlerin ortak özelliklerinin, uygulama programlama arayüzlerinin verimsiz olarak kullanımı olduğu sonucuna varmışlardır. Bunun yanı sıra, performans iyileştirmelerinin sadece birkaç satır kodu değiştirerek gerçekleştirilebileceği fikrini ortaya koymuşlardır.

Ramos vd. (Ramos, Valente and Terra, 2018), AngularJS çerçevesinin performansı konusunda 95 ayrı geliştiriciyle bir anket çalışması gerçekleştirmişlerdir. Çalışmalarında AngularJS çerçevesinde performans kayıplarına yol açan temel nedenlere, geliştiricilerin anket kapsamındaki sorulara verdikleri cevaplarla ulaşmaya çalışmışlardır. Çalışmaları sonucunda performansı olumsuz etkileyen nedenlerin başında AngularJS çerçevesini kullanmaktaki bilgi eksiklikleri ve bunun sonucunda ortaya çıkan yazılım mimarisindeki yanlış geliştirme yaklaşımlarının etkili olduğu sonucuna varmışlardır. Bunun yanı sıra, araştırmaları sonucunda, geliştirme sürecinde çerçeveye özel ng-repeat direktifinin hatalı kullanımı ve iki yönlü veri bağlamanın gereksiz kullanımlarının performans kayıplarına neden olduğunu ortaya koymuşlardır.

Izquierdo vd. çalışmalarında (Garcia-Izquierdo and Izquerdo, 2012) istemci taraflı şablonlama tekniklerinin genel amaçlı şablon sistemlerine uygulanabilirliğini araştırmışlardır. Çalışmada, tasarım ve programlama süreçlerinin ayrılmasıyla, uygulama geliştirme sürecini hızlandırıcı, çift model süreci gibi etkiler ortaya konmuş, performans iyileştirmeleri ve önbellekleme teknolojileri gibi istemci taraflı şablonlamanın kullanımını teşvik edici bulgulara yer verilmiştir. Ancak bununla birlikte, kullanılan tekniklerin web

görünürlüğünü etkileyen ve arama motorları üzerinden erişilebilirliği kısıtlayan, anlamsal kayıplara neden olduğu da tespit edilmiştir.

Gizas vd. tarafından yapılan çalışmada (Gizas, Christodoulou and Papathedorou, 2012) ise Dojo, ExtJS, JQuery gibi 7 farklı popüler Javascript çerçevesi karşılaştırılmaktadır. Gerçekleştirilen değerlendirmelerde büyüklük metrikleri (kod satır sayısı, yorum satırlarının kod satırlarına oranı), karmaşıklık metrikleri (McCabe's döngüsel karmaşıklığı, kod derinliği), sürdürülebirlik metrikleri (Halstead metrikleri) açısından sonuçlar ortaya koyulmaktadır. Çalışma Javascript çerçevelerinin karşılaştırılmasıyla ilgili olarak Geçerlilik (Validation), Kalite (Quality) ve Performans bölümlerinden oluşan bir yöntem ortaya koymaktadır. Ortaya konulan yöntem daha sonra Graziotin ve Abrahamsson tarafından yapılan çalışmada (Graziotin and Abrahamsson, 2013) genişletilerek yeni bir yöntem önerilmiştir. Önerilen yöntemde örnek bir uygulama kullanılması tavsiye edilmiş ve örnek uygulama üzerinde Dokümantasyon, Topluluk (Community) ve Fayda (Pragmatics) parametrelerinin de modele eklenmesi gerektiği savunulmuştur.

Ocariza vd çalışmalarında (Ocariza, Frolin, Pattabiraman and Mesbah, 2015) Javascript çerçevelerini güvenilirlik ve sürdürülebilirlik açısından değerlendirmişlerdir. Çalışmalarında çerçevelerde belirteçler (identifier) ve türler (type) açısından tutarsızlıklar bulunduğu ve bu tutarsızlıkları ortaya koyabilmek için ortaya çıkan bir yöntem ihtiyacından bahsetmişlerdir. Bununla ilgili olarak, AngularJS çerçevesi üzerinde, tutarsızlıkları tespit eden bir araç önermişlerdir.

Web uygulamalarında istemci taraflı Javascript çerçeveleri tarafından ortaya konulan yaklaşımların sayfa yükleme zamanına etkisinin araştırılması, bu tez çalışmasının amacını oluşturmaktadır. Bu çalışmada, önceki çalışmalar doğrultusunda incelenmiş olan farklı yazılım mimarisi yaklaşımları sağlayan ve en çok kullanılan üç ayrı Javascript çerçevesi (Angularjs, Reactjs ve Vuejs) karşılaştırma için seçilmiştir. Araştırma kapsamında, üç ayrı Javascript çerçevesi tarafından istemci taraflı web uygulaması geliştirme süreçlerinde ortaya konulan şablonlama ve doküman nesne modeli (DOM) oluşturma yaklaşımlarının, sayfa yükleme zamanına etkisi değerlendirilmiştir. Bu bağlamda günümüzde yaygın olarak kullanılan üç ayrı tarayıcı üzerinde (Chrome, Firefox ve Internet Explorer) ve masaüstü ortamında testler gerçekleştirilmiştir.

Bu çalışmaya benzer olarak, daha önce Davila tarafından gerçekleştirilen çalışmada (Davila, 2015), Angularjs, Backbonejs, Emberjs, Marionettejs and Reactjs çerçeveleri, Chrome ve Firefox tarayıcıları üzerinde, 1000 satırlık veri boyutu ile 5'er kez TodoMVC referans uygulaması (Osmani, 2015) üzerinde karşılaştırılmaktadır.

Koetsier tarafından gerçekleştirilen çalışmada ise (Koetsier, 2016), Angularjs, Reactjs, EmberJS ve Vuejs çerçeveleri sadece performans açısından değil, Olgunluk (Maturity), Kullanım kolaylığı (ease of use), tarayıcı desteği ve uyumluluk (browser support), tekrar kullanılabilirlik (reusability), test edilebilirlik (testability), yönlendirme (routing) ve şablonlama (templating) bakımından karşılaştırılmaktadır. Çalışmada performans testleri kısmı çalışmanın küçük bir bölümünü oluşturmakta, gerçekleştirilen karşılaştırmada, sadece Chrome tarayıcısı üzerinde 10 ve 1000 satırlık veri boyutları üzerinde ilgili çerçevelerle 5'er kez testler gerçekleştirilmektedir.

Molin tarafından gerçekleştirilen çalışmada (Molin, 2016), tek sayfa Javascript uygulamaları için Angularjs, Angular2, React çerçeveleri kullanılarak 100 satırlık ve 1000 satırlık veri boyutları ile 1'er kez TodoMVC uygulaması üzerinde gerçekleştirilmektedir.

Bu çalışmada gerçekleştirilen testlerde daha önceki çalışmalar doğrultusunda 100 ve 1000 nesnelik veri boyutları kullanılmıştır. Veri boyutunun seçimi konusunda daha önceki çalışmalar incelendiğinde testlerin 5'er kez gerçekleştirildiği görülmektedir. Bu doğrultuda bu çalışma kapsamında da testler 5'er kez gerçekleştirilmiştir.

Gerçekleştirilen testlerde, sayfa yükleme zamanını ölçebilmek için, tarayıcı çalışma zamanı üzerinde Gezinti Zamanlaması Programlama Arayüzü (NavigationTimingAPI) ve araştırma kapsamında geliştirilen yardımcı performans ölçüm kütüphanesi kullanılmaktadır. Elde edilen bulgular doğrultusunda, çerçevelere ait farklı yaklaşımların uygulama yükleme performansına etkisi, avantajlı ve dezavantajlı yönleri ile değerlendirilmekte ve öneriler ortaya konulmaktadır. Bu çalışma bilimsel olarak yazılım performans mühendisliğinin web uygulamalarına uygulanması kapsamında katkı sunmakta, değişken veri boyutu ve yazılım mimarisine göre web sayfası yükleme performansının değerlendirilmesi konusunda bir yöntem ortaya koymaktadır.

Bu tez çalışması beş bölümden oluşmaktadır. İkinci bölümde web uygulamaları ve Javascript hakkında temel bilgilere yer verilmiştir. Üçüncü bölümde yöntem bölümü bulunmaktadır. Dördüncü bölümde bulgulara yer verilmekte, beşinci bölümde ise gerçekleştirilen çalışmaya dair sonuçlar yer almaktadır.

2. WEB UYGULAMALARI VE JAVASCRIPT

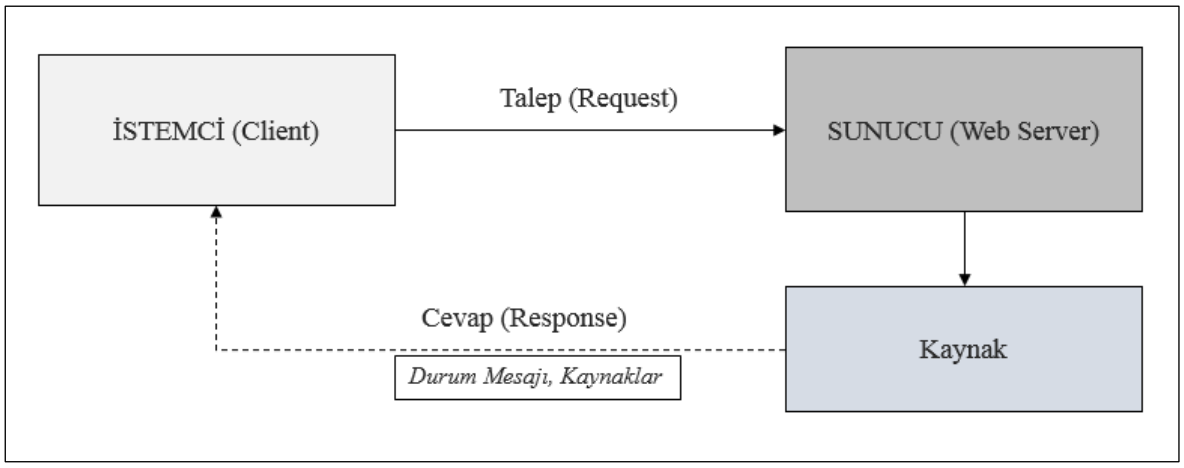
Bu bölümde web uygulama mimarisinin temelleri, sunucu ve istemci taraflı programlama ve web uygulamalarında istemci taraflı programlamada kullanılan javascript programlama dili ve javascript çerçeveleri hakkında temel bilgilere yer verilmektedir.

2.1. Web Uygulamaları

Internetin ilk zamanlarında, web uygulamaları daha çok, kullanıcılara sabit içerik sunan yapılarda geliştirilmişlerdir. Bu dönemde kullanıcıların, uygulamadan talep ettikleri içerikler çok fazla değişkenlik göstermemekle birlikte, web sitelerine erişen kullanıcılar, temelde tüm kullanıcılara gösterilen ortak sabit (statik) içeriklere ulaşmaktaydılar (Aghaei, Nematbakhsh, and Farsani, 2012). Zamanla, internetin yaygınlaşmasıyla birlikte, web uygulamaları daha çeşitli amaçlarla ve farklı alanlarda kullanıcılara özel içerik sunabilecek şekilde kullanılmaya başlanmıştır. Ortaya çıkan yeni ihtiyaçlar sonucunda, statik web sayfalarının yerine, daha yüksek kalitede, karmaşık ve dinamik web sayfaları üretme gerekliliği ortaya çıkmıştır (Offutt, 2002). Böylelikle daha önceleri kullanıcılara genel ve içeriği değişmeyen statik içerikler sunulurken, bu amaçla isteklerin akıllı bir şekilde işlenebilmesi için sunucu tarafında çalışan, veri ve iş mantığıyla bağlantı kuran yazılımlar (dinamik web uygulamaları) geliştirmeye başlanmıştır. Kullanıcılara ihtiyaçları doğrultusunda dinamik içerik sağlayabilmek için web uygulamaları daha akıllı hale dönüşmüşlerdir (O'Reilly, 2007). Günümüzde web uygulamaları sadece sabit veriler sunmaktan ziyade, birçok farklı alanda kullanılmakta, kişisel ve kurumsal ihtiyaçlara yanıt vermekte, e-ticaretten sosyal ağlara hayatın her alanında faaliyet göstermektedir.

2.2. İstemci Sunucu Mimarisi

Web uygulamaları temelde istemci sunucu mimarisi üzerinde çalışmaktadır (Oluwatosin, 2014). Şekil 2.1'de istemcinin sunucudan talep ettiği istek ve sunucudan istemciye verilen yanıta dair kavramsal çalışma şeması verilmektedir. İstemci sunucu mimarisinde, istemci tarafından gerçekleştirilen bir istek, sunucuya iletildikten sonra, sunucu istemciden gelen talebe göre, çeşitli işlemler gerçekleştirmekte ve yanıtını istemciye bildirmektedir. Web sunucusu üzerinde çalışan bir web uygulaması, temelde kullanıcıdan alınan talepleri işleyerek, karşılığında istemci tarafına bir yanıt döndürmektedir.



Şekil 2.1. İstemci sunucu mimarisi temel çalışma mantığı

İstemci ve sunucu aynı platform veya aynı fiziksel ortam üzerinde birlikte çalışabileceği gibi, farklı fiziksel ortam ve platformlarda da olabilmektedir. Sunucu, kullanıcıdan (istemciden) gelen talebi değerlendirerek, ilgili kaynaklara erişmekte ve yanıt döndürmektedir. Daha sonra istemci, sunucudan kendisine gelen yanıtı işleyerek gerekli eylemleri (sonucun kullanıcıya gösterilmesi, uygulamanın davranışının değiştirilmesi vb.) gerçekleştirmektedir.

Web uygulama sunucuları temelde HTTP protokolü üzerinde çalışmaktadır. Bu protokolde istemci-sunucu mimarisi arasındaki iletişime ilişkin olarak, bir istek-yanıt standardı tanımlanmıştır. RFC 2616 standardı içerisinde, HTTP/1.1 protokolü için istemci ve sunucu arasındaki iletişimin kurgulanması ve bu iletişimin dayandırıldığı kurallar yer almaktadır (Internet Engineering Task Force, 1999).

İstemci, HTTP protokol kuralları çerçevesinde, sunucuya isteğine dair bir HTTP komutu göndererek (GET, POST vb.), bu komutu karşılığında sunucudan yanıt almaktadır. Sunucu kendisine iletilen isteğe yanıt olarak, bir durum mesajı üretmektedir. Sunucu, istemciden gelen isteği alıp işlediğinde bu isteğe yanıt olarak başlık ve gövde verisinin yanı sıra bir HTTP durum mesajı döndürmektedir. İstemci tarafından iletilen bir isteğe karşılık, eğer kaynak sunucuda bulunuyorsa, sunucu, 200 (Tamam) yanıt koduyla ve kaynağın kendisiyle dönüş yapmaktadır. Bununla birlikte, kaynak bulunamadıysa, istemciye 404 (bulunamadı) yanıt kodunu döndürerek, kaynağın sunucuda bulunamadığı mesajını iletmektedir. İstemci de bu durum mesajını ve ek bilgileri alarak yanıtı işlemektedir.

HTTP protokolü üzerinde istemci sunucu mimarisi temel alındığında, sunucu yanıtını istemciye gönderdikten sonra, sürecin geri kalanıyla ilgilenmemektedir. İstemci tarafından gerçekleştirilen istek sonucunda, sunucu tarafından bağlantı kapatılmaktadır. İstemci ve sunucu arasında bu bağlamda durumsuz (stateless) bir iletişim gerçekleştirilmektedir. İsteğe dair yanıt istemciye gönderildikten sonra, sunucu sürecin geri kalanıyla ilgilenmemektedir. Daha sonraki süreç, istemcinin kararına bırakılmaktadır. İstemci de bu yanıtı işleyerek, sunucunun cevabını kullanıcıya gösterebilmekte, hata durumunda tekrar istek başlatabilmekte veya başka bir eylemde bulunabilmektedir (Totty, Gourley, Sayer, Aggarwal and Reddy, 2009).

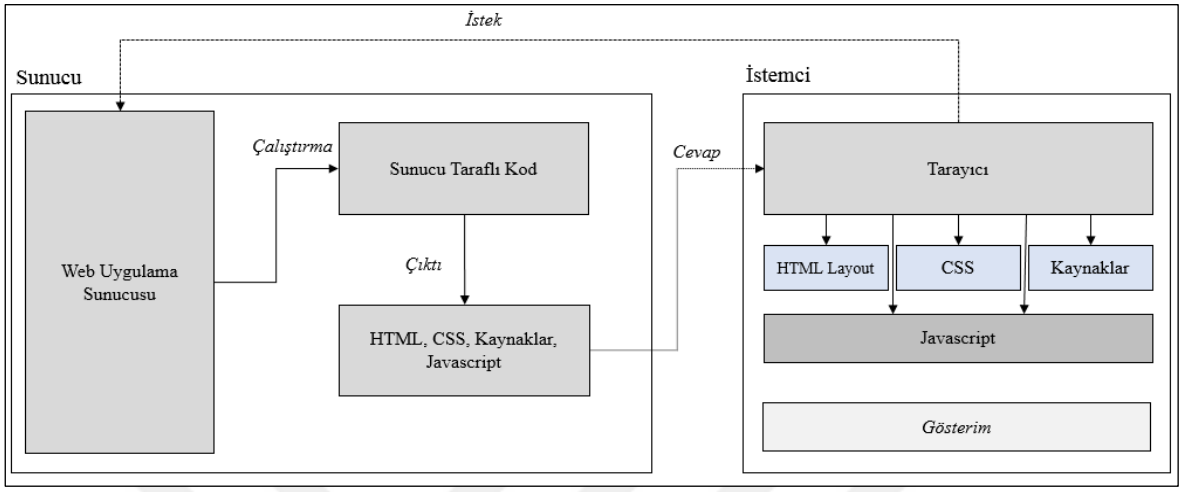
Web uygulamaları istemci sunucu mimarisi temel alındığında, iki parçadan oluşmaktadır. Sunucu tarafında, uygulamanın istemci tarafı dışında kalan uygulama mantığı ile ilgili kodlar bulunmaktadır. Sunucu bu kodlar aracılığıyla kaynaklara ulaşarak, kullanıcı talebine göre bu kaynaklar üzerinde düzenleme yapmakta ve/veya talep edilen kaynakların tümünü veya bir kısmını istemciye göndermektedir. Bu kaynaklar, değişmeyen (statik) kaynaklar olabileceği gibi, veritabanı veya üçüncü sistemlerle bağlantılar ve entegrasyonlar olabilmektedir.

2.3. Sunucu Taraflı Programlama

Sunucu üzerinde çalışan uygulama kodları, sunucu taraflı programlamanın kapsamını oluşturmaktadır. Sunucu taraflı web uygulama kodları, kullanılan platforma bağlı olarak farklı dillerde ve platformlarda geliştirilebilmektedir. Bu kapsamda, bir web uygulamasının hizmet verebilmesi için web sunucusu kullanılmaktadır (Connolly and Hoar, 2015).

Web sunucusu, herhangi bir istemciden bir istek aldığında, bu isteği işleyebilmek için, gelen isteğe göre uygun olan sunucu taraflı uygulama kodlarını çalıştırmaktadır. Örneğin; istemci tarafında bulunan kullanıcı adı ve şifre alanlarını içeren, kullanıcının sisteme giriş yapması ve yetkilendirilmesi için tasarlanmış bir giriş formu verisi, istemci tarafından sunucuya gönderilmektedir. Daha sonra, sunucu istemciden gelen kullanıcı adını ve şifreyi alarak, veritabanına bağlanmakta, böyle bir kullanıcı olup olmadığını kontrol etmektedir. Ardından şifrenin doğru ya da yanlış girilmesi durumuna göre istemciyi yönlendirmekte veya doğrudan cevap döndürmektedir. Ya da bunun yerine, sabit (statik) bir kaynak (dosya, resim vb.) talep edilmesi durumunda sunucu, içeriği önceden belirlenen belirli şartlar doğrultusunda istemciye gönderebilmektedir. Uygulama ve sunucu davranışı, büyük ölçüde

kurgulanan mimariye, yaklaşıma, teknolojiye ve geliştirilen çözüme bağlı olarak değişkenlik göstermektedir.



Şekil 2.2. Web uygulama sunucusu çalışma şeması

Şekil 2.2'de web uygulamasında sunucu taraflı ve istemci taraflı kodun birlikte kavramsal çalışma diagramı verilmektedir. Web sunucusu üzerinde çalışan sunucu taraflı web uygulaması kodu, istemcinin gerçekleştirdiği eyleme veya isteğe bağlı olarak, HTML, CSS, Javascript ve Kaynaklar'dan (resim, video, ses vb.) oluşan bir çıktı üreterek, istemciye göndermektedir (Maras, Carlson and Crnkovic, 2012).

Sunucu taraflı programlama, sunucu kaynaklarına ihtiyaç duyulması durumunda, iş mantığını oluşturan kodların veri kaynaklarına erişiminde ve kaynaklara erişimde güvenlik gereksinimi bulunduran, geliştirme süreçlerinde kullanılmaktadır. İstemciden alınan isteklerin, sunucu tarafından doğrulanması ve istekler karşılığında yanıtlar oluşturulması sunucu taraflı programlamayla çözümlenmektedir.

2.4. İstemci Taraflı Programlama

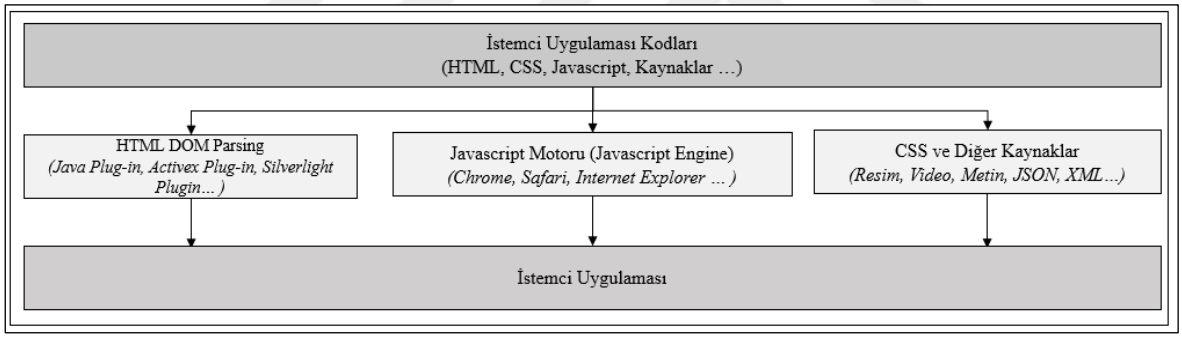
Web uygulamasının istemci tarafında, son kullanıcı bilgisayarında çalışan, kullanıcıya arayüz sağlamanın yanı sıra, sunuculara ve diğer kaynaklara çeşitli istekler gönderebilen ve karşılığında karşı taraftan gelen yanıtları işleyerek son kullanıcıya gösteren uygulama bulunmaktadır.

Sunucu veya sunuculara yapılan istekler doğrultusunda yanıt olarak alınan HTML, CSS, Javascript ve kaynaklar, istemci bilgisayarında tarayıcı tarafından önceden belirlenen kurallar doğrultusunda oluşturularak kullanıcıya web sayfası gösterilmektedir.

2.5. Web Tarayıcıları

Web tarayıcısı istemci bilgisayarında çalışmakta, kullanıcının talebine karşılık olarak sunucudan dönen yanıtları işlemektedir. İstemci taraflı web uygulaması da tarayıcı üzerinde uygulama arayüzünü oluşturmakta, uygulama kapsamındaki veriyi ve uygulama içerisinde işlemlerin gerçekleştirilebilmesi için çeşitli fonksiyonları kullanıcılara bir web tarayıcısı üzerinde sunmaktadır.

Şekil 2.3'de görüldüğü üzere, tarayıcı talep edilen istek sonucunda dönen, HTML, CSS, Kaynaklar ve Javascript kodlarını işleyerek istemci bilgisayar üzerinde arayüzü oluşturmaktadır.



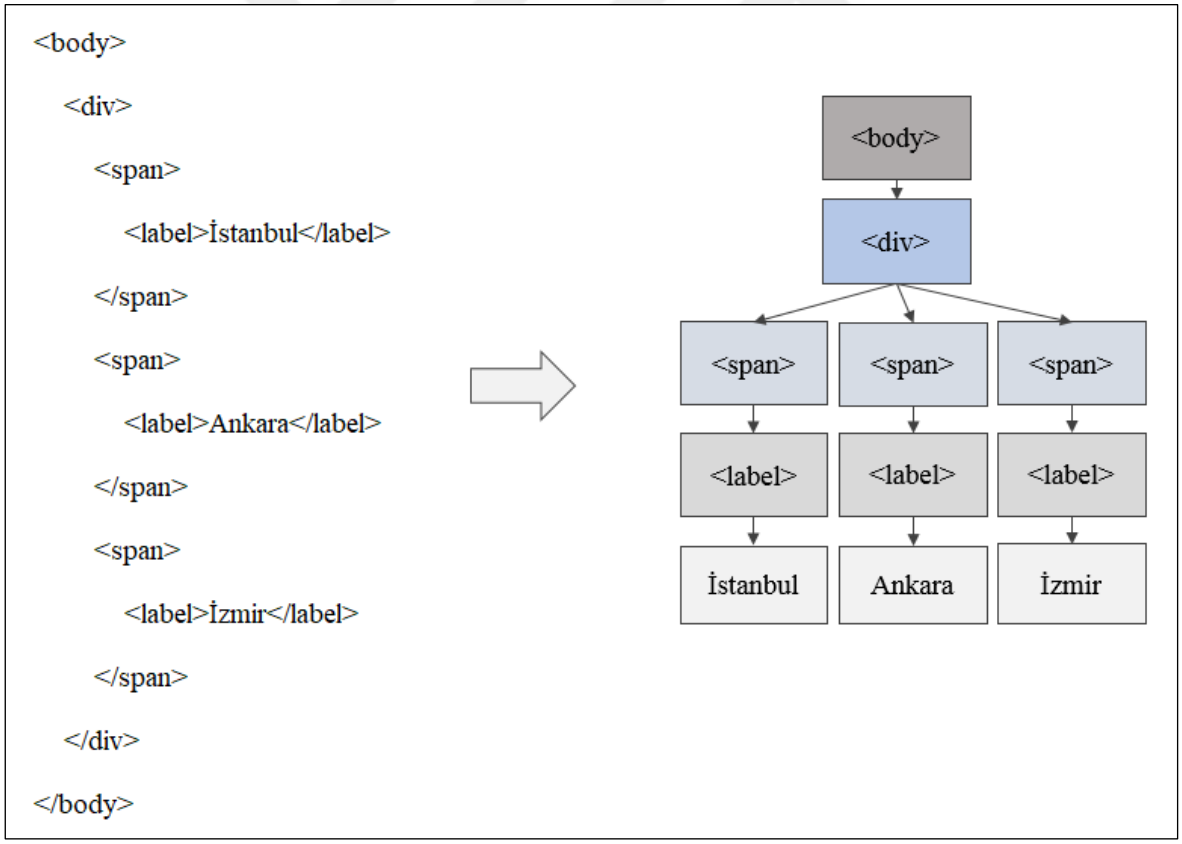
Şekil 2.3. Tarayıcı çalışma şeması

Tarayıcı sunucudan aldığı HTML sayfasına ait kodları işleyerek DOM (Document Object Model) oluşturmakta, yine CSS kodlarını işleyerek CSSOM'a aktararak, DOM içerisindeki nesnelere atanmış stil bilgilerini uygulamakta, diğer yandan sayfa içerisinde yer alan Javascript kodlarını çalıştırmaktadır.

2.5.1. Doküman nesne modeli

Bir HTML dokümanı tarayıcıya yüklendiğinde, tarayıcı dokümanı işleyerek, HTML elemanlarını, yapısal bir modele aktarmaktadır. Bu yapısal modele Doküman Nesne Modeli (DOM- Document Object Model) adı verilmektedir. DOM ağaç yapısına sahip olmakla birlikte, aynı zamanda HTML belgesini işlemek için bir uygulama programlama arayüzü (API) sunmaktadır. Bu uygulama programlama arayüzü aracılığıyla, HTML dokümanının içeriği ve yapısı üzerinde işlemler gerçekleştirilebilmektedir (Flanagan, 2011).

HTML içerisindeki elemanlar ayrıştırılırken, DOM üzerinde düğümlere aktarılmaktadır. Şekil 2.4'de görüldüğü üzere bu düğümler arasında DOM içerisinde bir hiyerarşi kurgulanmaktadır. Bu hiyerarşi doğrultusunda, bu yapı bir ağaç oluşturduğu için DOM Ağacı (DOM Tree) olarak da isimlendirilmektedir.



Şekil 2.4. HTML içeriğinin DOM'a aktarılması

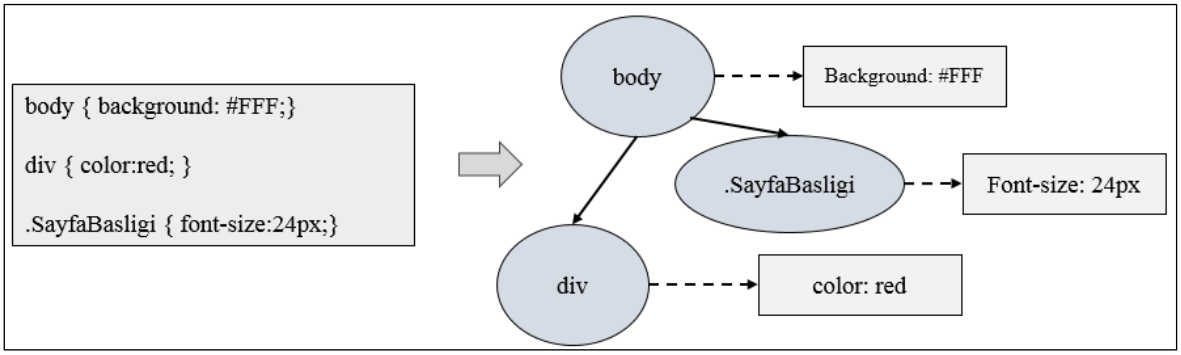
DOM ağacı boyutunun büyümesi, web uygulama performansını birçok açıdan etkilemektedir. Büyük bir HTML dosyası, küçük bir HTML dosyasına göre, sunucudan indirilmesi için daha fazla zaman gerektirmekte, bununla birlikte indirilen HTML belgesinin

DOM'a ayrıştırılması (parsing) ve nihai sayfa oluşturma (render) işlemini de uzatmaktadır. Bununla birlikte büyük DOM ağacı üzerinde Javascript ile gerçekleştirilecek işlemler, daha küçük boyutlu bir DOM ağacına göre çok daha maliyetli olabilmektedir (Smith, 2013).

Günümüzde DOM yapısına dair teknik özellikler (spesifikasyonlar), World Wide Web Consortium (W3C) tarafından belirlenerek, tavsiye niteliğinde geliştirilmektedir. DOM için en son 2004 yılında W3C tarafından üçüncü seviye çekirdek teknik özellikler yayımlanmıştır (Le Hors vd., 2004). Söz konusu teknik özelliklerin dördüncü seviye sürümü W3C tarafından halen geliştirilmeye devam etmektedir (W3C, DOM4 Working Draft, 2015).

2.5.2. Basamaklı stil sayfası nesne modeli

CSS (Cascading Style Sheets- Basamaklı Stil Sayfası) ile web sayfalarının arayüzlerinin biçimlendirilmesi ve belirli kullanıcı hareketlerine göre sayfa üzerindeki DOM nesnelerinin görsel stillerinin belirlenmesi sağlanmaktadır. HTML kodlarının DOM'a aktarılmasına benzer bir şekilde, CSS kodları da tarayıcı tarafından Basamaklı Stil Sayfası Nesne Modeli'ne (CSSOM- CSS Object Model) aktarılmaktadır (W3C, 2019). Şekil 2.5'de CSS'in CSSOM yapısına aktarım süreci gösterilmektedir.

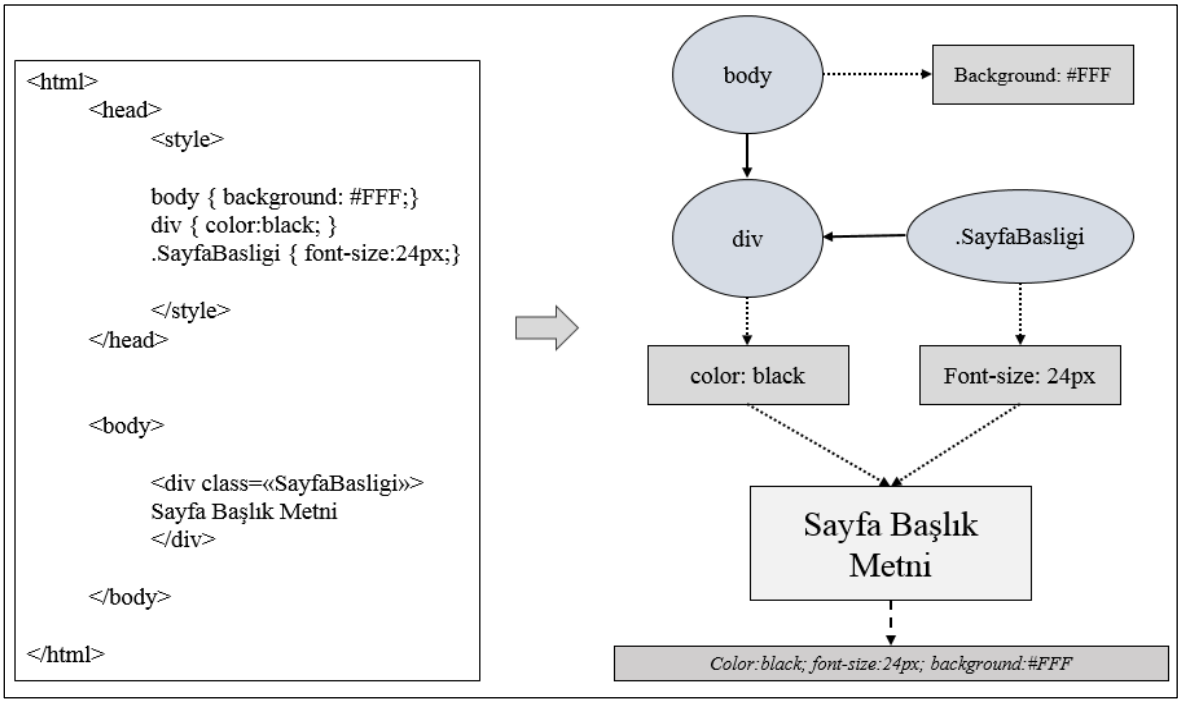


Şekil 2.5. CSS içeriğinin CSSOM'a aktarılması

CSSOM, DOM'a benzer bir şekilde, ağaç yapısında saklanmaktadır. Böylece bu ağaç yapısı üzerinde tarayıcı ilgili düğümlerin belirttiği stil tanımlarını DOM elemanlarına uygulamaktadır.

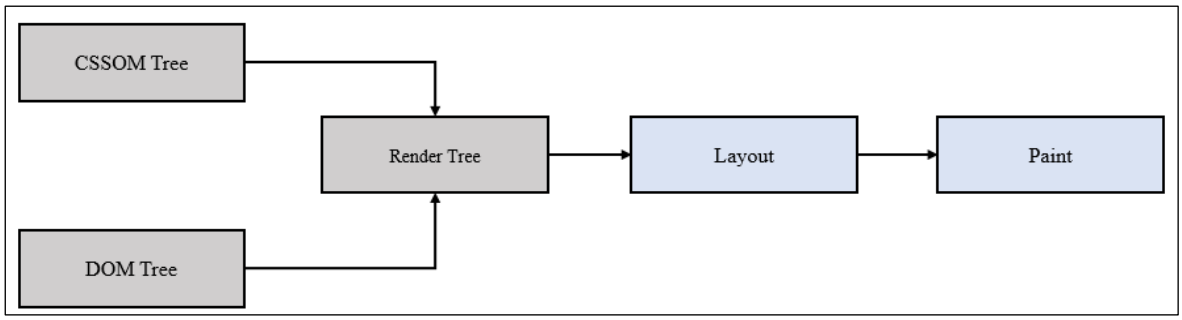
2.5.3. Oluşturma ağacı

Tarayıcı üzerinde, web sayfası arayüzünün oluşturulması sürecinde Gösterim Motoru (Rendering Engine) kullanılmaktadır. HTML ve CSS kodları, tarayıcı tarafından DOM ve CSSOM'a ayrıştırıldıktan sonra, bu iki ayrı ağaç birleştirilerek, Oluşturma Ağacı (Render Tree) oluşturulmaktadır. Oluşturma ağacı, gösterim motoru tarafından işlenerek arayüz oluşturulmaktadır. Şekil 2.6'da oluşturma ağacında gösterim motoru, her bir düğümün kök düğümden başlayarak ve tek tek düğümlerin üstünden geçerek, kullanıcıya gösterilecek nihai arayüzü oluşturmaktadır.



Şekil 2.6. Oluşturma ağacının tarayıcı tarafından oluşturulması

Oluşturulan nihai arayüz, oluşturma ağacının ve bu ağacın bağımlı olduğu, CSSOM ve DOM'un oluşturma süresine bağlı olarak kullanıcıya gösterilebilmektedir. DOM ve CSSOM'un elde edilmesiyle birlikte, oluşturma ağacının oluşturulması, ardından bu ağacın düğümlerinin hesaplanarak ekranda piksellere dönüştürme ve nihai görüntünün oluşturulması süreci, Kritik Oluşturma Yolu (Critical Rendering Path) olarak adlandırılmaktadır. Kritik oluşturma yoluna ait kavramsal diagram Şekil 2.7'de görülmektedir.

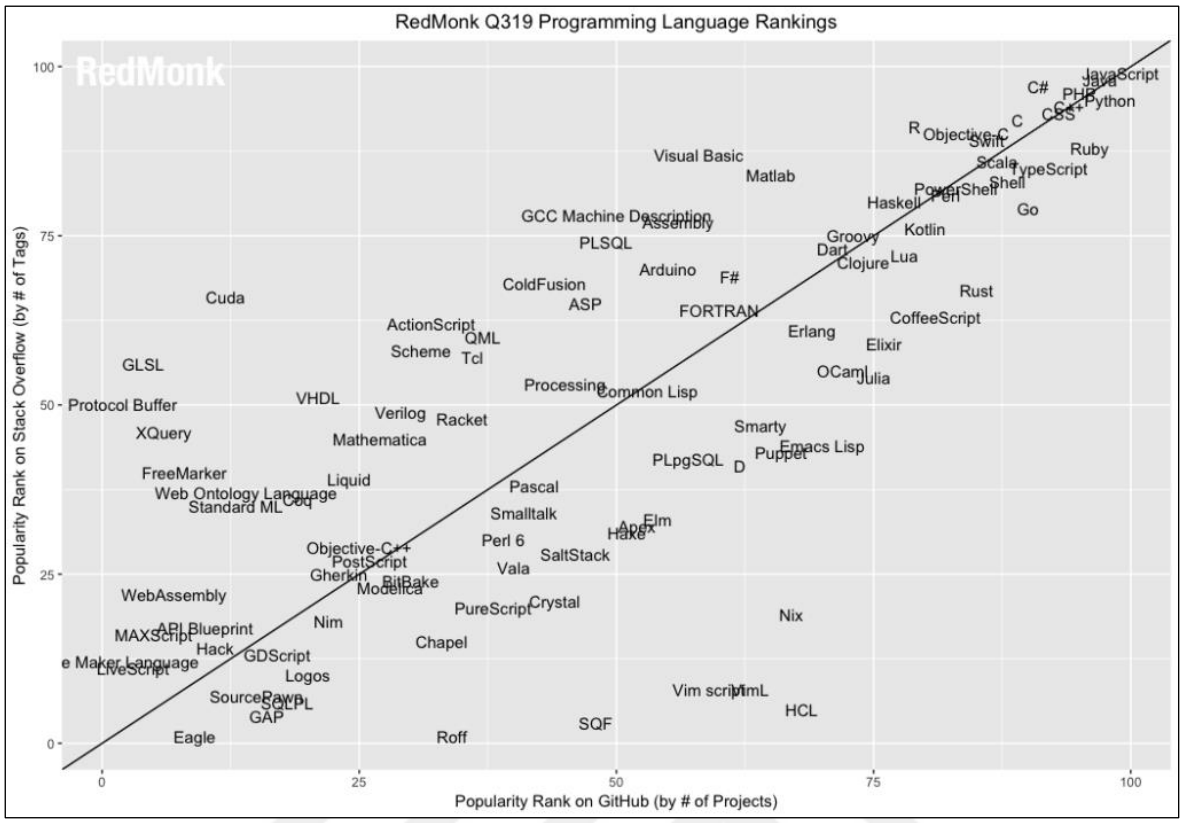


Şekil 2.7. Kritik oluşturma yolu

Kritik oluşturma yolunda, işlemler sıralı ve kademeli bir şekilde gerçekleştirilmektedir. Gösterim motoru, kullanıcıya daha iyi bir deneyim sunabilmek için olabilecek en hızlı şekilde içeriği kullanıcıya göstermeye çalışmaktadır. Bu kapsamda, oluşturma ağacının oluşturulması sırasında, tüm HTML dokümanının ayrıştırılarak DOM ağacına aktarılmasının bitişi beklenmemektedir. DOM ve CSSOM ağaçlarının ayrıştırılan bölümleri, oluşturma ağacında hızlıca birleştirilerek doğrudan ekranda gösterilmeye çalışılmaktadır. Bu kapsamda, HTML ve CSS tanımlarının doğru kurgulanmaması durumunda, kritik oluşturma yolunda, oluşturma engelleme (render blocking) durumu ortaya çıkmaktadır. Oluşturma engellenmesi durumu, kullanıcıların web sayfasını daha geç görmelerine sebep olmaktadır (Grigorik, 2018).

2.6. Javascript

Javascript, ilk olarak 1995 yılında Brendan Eich tarafından LiveScript adında Netscape Communicator tarayıcısı üzerinde çalışmak üzere geliştirilmiştir (Peyrott, 2017). Javascript günümüzde hemen hemen tüm web uygulamalarında kullanılmaktadır. Şekil 2.8'de tüm programlama dilleri arasında, O'Grady tarafından 2019 Haziran ayında gerçekleştirilen araştırmaya göre (O'Grady, 2019), programcıların sıklıkla kullanmakta oldukları Github (Github, 2019) ve Stackoverflow (Stackoverflow, 2019) üzerinde Javascript en popüler programlama dili olarak gösterilmektedir.



Şekil 2.8. Programlama dilleri kullanım sıklıkları (O'Grady, 2019)

Javascript'in ilk ortaya çıktığı zamanlarda henüz bir standart bulunmamakta, her bir tarayıcı Javascript kodlarını farklı şekilde yorumlamaktaydı. Javascript'in standardizasyonunun sağlanması 1996 yılında ECMA (European Computer Manufacturers Association- Avrupa Bilgisayar Üreticiliği Birliği)'ne verilmiştir. ECMA Javascript adını resmi olarak ECMAScript olarak belirlemiştir (Chapman, 2017).

Javascript'in ECMA'ya devredilmesinden sonra, bu farklılıkları ortadan kaldırmak, Javascript söz dizimini ve çalışma ortamını standartlaştırmak amacıyla, ECMA-262 altında bir teknik özellikler (spesifikasyon) yayınlanmıştır. Günümüzde ECMA-262 standardının dokuzuncu sürümü, ECMAScript 2018 standardı kullanılmaktadır (ECMA, 2018). Bu standardizasyon yaklaşımının yanı sıra, web uygulamaları geliştikçe, endüstriye yönelik olarak geliştirilen araçlar, kütüphaneler ve çerçeveler aracılığıyla Javascript, belirli standartlar altında toplanmaya başlanmıştır.

2.6.1. Javascript motoru

Tarayıcı üzerinde bir web sayfasının çalıştırılabilmesi için iki ayrı bileşen bulunmaktadır. Sayfanın tarayıcı penceresine yerleşimi ve gösterimi için, Gösterim motoru (Rendering Engine), Javascript kodunun çalıştırılması için Javascript motoru (Javascript Engine) kullanılmaktadır.

Javascript motoru (Javascript engine) tarayıcı üzerinde Javascript kodunu çalıştırmaktadır. DOM'a ayrıştırılan HTML elemanları üzerinde, olay yakalayıcılarının (event handler) bu elemanlara bağlanması (event binding) Javascript motoru tarafından yönetilmektedir. Böylece olay yakalayıcıları aracılığıyla, DOM Javascript kodu tarafından erişilebilmekte ve üzerinde çeşitli işlemler gerçekleştirilebilmektedir.

Çizelge 2.1'de günümüzde kullanıcılar tarafından sık kullanılan belirli tarayıcıların (Browser Market Share, 2019) kullandıkları Gösterim ve Javascript Motorları görülmektedir. Web tarayıcıların yeni sürümleri ortaya çıktıkça, gösterim ve Javascript motorlarında da gelişmeler ortaya çıkmaktadır.

Çizelge 2.1. Sık kullanılan tarayıcılara ait gösterim ve javascript motorları

Tarayıcı	**Versiyon**	**Gösterim Motoru**	**Javascript Motoru**
Internet Explorer / Edge (Microsoft)	9+	Trident	Chakra
Chrome (Google)	40+	Blink	V8
Firefox (Mozilla)	35+	Gecko	SpiderMonkey
Opera	27+	Blink	V8
Safari (Apple)	8+	WebKit	Nitro

Önceki zamanlarda geliştirilen tarayıcılarda, Javascript motoru, tarayıcının içerisinde bütünleşik olarak bulunurken, günümüzde tarayıcıdan bağımsız ayrı katmanlar ve uygulamalar olarak geliştirilebilmektedir. Böylece aynı zamanda, Javascript kodu, tarayıcı dışında da doğrudan Javascript motoru üzerinde de çalıştırılabilmektedir. Bunun sonucu olarak günümüzde Javascript kodu tarayıcı dışında farklı amaçlarla Node.js gibi farklı

platformlar üzerinde de çalışabilmektedir. Bu yönüyle Javascript motoru kavramını tarayıcıdan bağımsız olarak ele almak gerekmektedir.

Javascript motoru temelde, bir sanal makine (virtual machine) ortamı sağlamaktadır. Sanal makine kavramı, bir bilgisayar sisteminin, dış dünyadan izole edilmiş bulunan, yazılım tabanlı bir öykünme (emulation) ortamıdır. Sanal makine içerisinde çalışan kodlar dış dünyadan izole olarak çalışmakta, diğer uygulamaların çalıştıkları kaynaklara ve alanlara erişememektedir. Böylelikle geliştirilen Javascript kodunun, güvenli olarak dış dünyada yalıtılmış bir ortamda çalışması sağlanmaktadır.

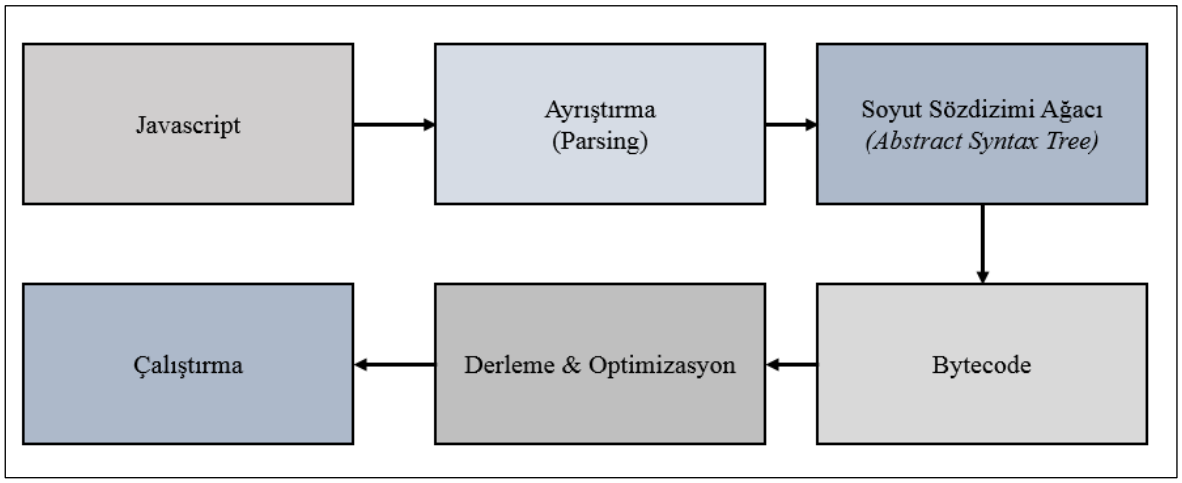
Sanal makine türleri sınıflandırılırken, fiziksel makinelerin sunduğu çalışma ortamındaki fonksiyonlara öykünme dereceleri referans alınarak değerlendirme yapılmaktadır. Bir sistem sanal makinesi, bir bilgisayar sistemini baştan sona taklit ederek öykünme gerçekleştirebilmektedir. Böylece sanal makine üzerinde, tamamen ayrı bir işletim sistemi ya da benzer büyüklükte platformlar çalıştırılabilmektedir. Sanallaştırma teknolojisi ile tek bir fiziksel bilgisayar üzerinde, farklı işletim sistemlerine sahip birbirlerinden soyutlanmış olarak çalışan sanal makineler oluşturulabilmektedir.

Bununla birlikte, süreç tabanlı sanal makine ise sistem sanal makinesine göre daha kısıtlı fonksiyon ve özellikler sunmaktadır. Bu sanal makine türü, bir programı ya da program parçacığını (process) daha basit şartlarda öykünerek çalıştırmaktadır. Bu duruma Wine Platformu (Wine HQ, 2019) örnek olarak gösterilebilir. Wine platformu, Linux üzerinde Windows uygulamalarının çalıştırılmasını sağlayan, bir süreç (process) sanal makinesi olarak görev yapmaktadır. Bu platform ile Windows üzerinde çalışan çeşitli programlar, Linux üzerinde de bu sanal makine platformunu kullanarak çalıştırılabilmektedir. Ancak bu platform, Windows işletim sisteminin baştan sona tüm fonksiyonlarını Linux ortamı üzerinde sağlamamaktadır.

Javascript motoru bu kapsamda değerlendirildiğinde, Javascript kodunu yorumlayıp çalıştırması yönüyle bir süreç sanal makinesidir. Süreç sanal makinesi içerisinde uygulama kodu derlenerek veya yorumlanarak, dış çevreden soyutlanmış/izole edilmiş bir ortamda, diğer yazılımların kullandıkları bellek alanlarına müdahale etmeden çalıştırılmaktadır.

2.6.2. Javascript'in çalıştırılması

Geliştirilen herhangi bir uygulamadaki gerçek uygulama kodu, concrete syntax (somut sözdizimi) olarak tanımlanmaktadır. Javascript motoru, kodu çalıştırırken öncelikle ilgili kodu ayrıştırarak (parse) bir soyut sözdizimi ağacı (AST- Abstract Syntax Tree) oluşturmaktadır. Daha sonra bu ağaç üzerinde ilerleyerek, kodu bytecode'a dönüştürmektedir. Javascript çalıştırma sürecine ait kavramsal şema, Şekil 2.9'da görülmektedir.



Şekil 2.9. Javascript çalıştırma süreci

Bytecode, Javascript kodunun dönüştürüldüğü bir ara formdur. Bytecode üzerinde derleme, yorumlama ve iyileştirme süreçleri işletilmektedir. Bu süreci takiben, Javascript motoru içerisindeki belirli kısımlar, optimize edilerek nihai makine kodu, ilgili bilgisayardaki işlemci mimarisine göre derlenerek Javascript motoru içerisindeki sanal makine üzerinde çalıştırılmaktadır.

Yazılım kodu çalıştırılırken yorumlama ya da derleme yaklaşımları bulunmaktadır. Böylece bu yaklaşımları gerçekleyebilmek için yorumlayıcı (interpreter) ya da derleyici (compiler) kullanılmaktadır. Bir yorumlayıcı, kodu satır satır yorumlayarak, doğrudan makine koduna ya da ara koda çevirdikten sonra çalıştırmaktadır. Derleyici (compiler) ise kendisine verilen kodun tamamını makine koduna çevirdikten sonra çalıştırmaktadır.

Yorumlama yaklaşımının avantajı, kodu çalıştırmadan önce ek bir zamana ihtiyaç duymamasıdır. Böylece uygulama kodu, kodun tamamını makine koduna çevirmeye gerek

kalmadan satır satır yorumlanarak, vakit kaybetmeden çalıştırılmaya başlanır. Ancak döngü içeren kodlarda (örneğin bir dizi üzerinde kodun iterasyon yapması vb.) yorumlayıcı aynı kodu tekrar yorumlamak zorunda kalmaktadır. Bu durumda da aynı işlem tekrarlı olarak yorumlanarak ek zaman maliyeti ortaya çıkarmaktadır.

Derleyicilerde ise ilk önce kodun tamamını derlemek için derleme süresi gerekmektedir. Derleme sürecinde, kod çalıştırılmaya başlamadan önce derleme süresi kadar beklemek gerekmektedir. Ancak kod çalıştırılmaya başladıktan sonra, önceden derleme zamanında optimize edildiği için yorumlanan koda göre daha hızlı çalışmaktadır. Kod derlenirken, döngüler ve iterasyonlar gibi maliyetli kod bloklarını derleyici, derleme zamanında optimize etmektedir. Yorumlayıcı ise, derleyiciden farklı olarak, bu tür kod bloklarıyla, kodun çalışma sürecinde ilk kez karşılaştığı için bu iyileştirmeleri gerçekleştirememektedir.

Derleyici ve yorumlayıcı yaklaşımlarının kodun çalıştırılmasından önce ve çalıştırma sırasında birbirlerine göre avantajları ve dezavantajları bulunmakla birlikte, karşılaşılan her durum için uygun olan tek bir yaklaşımı doğru olarak belirlemek mümkün değildir. Özellikle web uygulamaları düşünüldüğünde, her bir ayrı web uygulaması için Javascript kodunun daha önceden temin edilerek derlenmesi ve istemci bilgisayarında saklanması mümkün değildir. Böylece bu uygulama kodlarının yorumlayıcı üzerinde çalıştırılması, web uygulaması içerisinde çalışan Javascript'in temin edilmesi, derlenmesi zamanları düşünüldüğünde, zaman açısından daha avantajlı olmaktadır.

Bununla birlikte, yorumlayıcı ve derleyicilerin ortaya koyduğu, iki ayrı yaklaşımın da avantajlı yönleri birleştirilerek Tam Zamanında Derleme (JIT- Just In Time Compilation) yaklaşımı ve bu yaklaşımı uygulayabilmek için özel derleyiciler geliştirilmiştir. Tam zamanında derleme yaklaşımında, öncelikle tüm kod bloğu yorumlayıcıya gönderilmektedir. Daha sonra kodun çalışması, yorumlanma sürecinde izlenmeye başlanmaktadır. Kod içerisindeki sıcak noktalar (döngüler, tekrarlayan kod blokları), tespit edildikten sonra, kodun tekrar çalışması durumunda daha hızlı davranabilmesi için maliyetli kod blokları, derlenerek çalıştırılmaktadır. Böylece ilk başta kodun çalıştırılması için derleme zamanı beklenmemekte, sıcak kod bölümleri izlenerek, ilgili noktalar o anda derlenerek çalıştırılmakta ve böylece iki ayrı yaklaşımın avantajlı yönleri birleştirilerek performans iyileştirmesi ve kodun daha hızlı çalıştırılması hedeflenmektedir.

2.6.3. Javascript çalışma zamanı

Javascript Çalışma Zamanı (Javascript Runtime), Javascript motoru tarafından, platforma özel olarak geliştirilen bir Javascript uygulamasına söz konusu platformla ilgili olarak nesne, fonksiyon ve özellikler sağlayan bir API (uygulama programlama arayüzü) görevi görmektedir.

Bir javascript uygulaması geliştirilirken, üzerinde geliştirildiği platformun amacına göre Javascript Çalışma zamanı tarafından sağlanan nesneler ve fonksiyonlar kullanılmaktadır. Tarayıcıya özel olan nesne ve fonksiyonlar (window, document vb.), Javascript motorunun içerisinde hazır olarak bulunmamaktadır. Bunun için Javascript motoru üzerinde, çalışılan platforma yönelik olarak bir uygulama programlama arayüzü sağlanmaktadır.

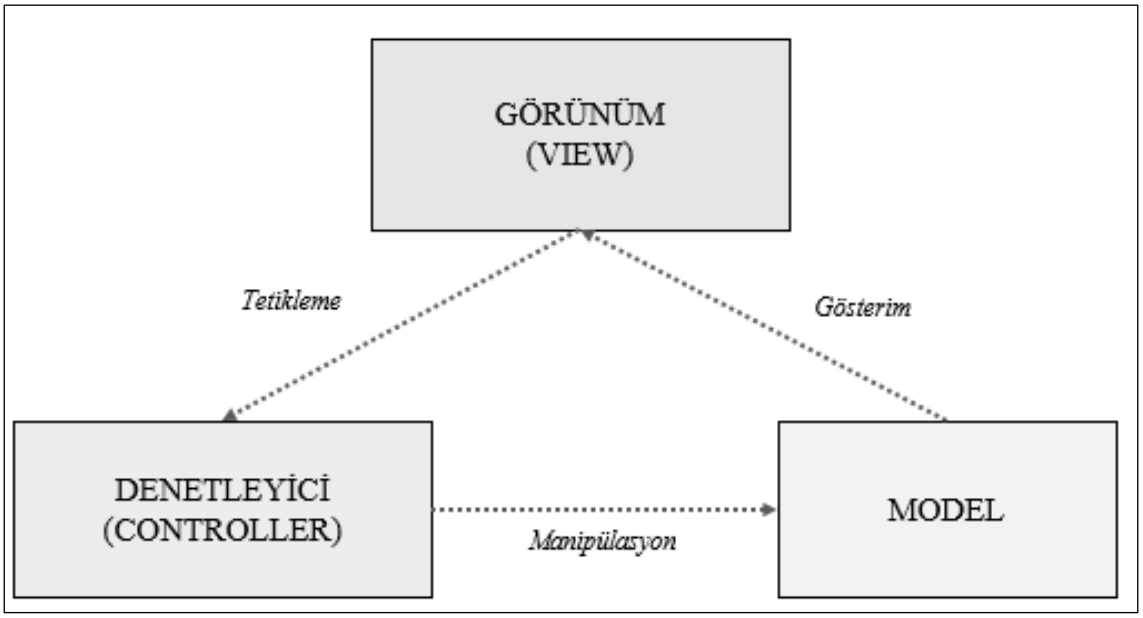
Tarayıcı üzerinde, Javascript Çalışma Zamanı (Javascript Runtime) tarafından "Window" ve "Document" özellikleri başta olmak üzere çeşitli nesne ve fonksiyonlar sağlanarak, Javascript uygulamasının tarayıcı üzerindeki DOM'a erişebilme ve işleyebilme olanağı sunulmaktadır. Böylece istemci taraflı olarak, Javascript kodu tarayıcı tarafından sağlanan API aracılığıyla, web sayfası üzerinde iş ve işlemleri gerçekleştirebilmektedir.

Bununla birlikte, tarayıcı dışında Node.js platformunda (Tilkov and Vinoski, 2010), DOM yerine sunucu tarafında bulunan nesneler (http, server vb.) ile ilişkili özellikler ve araçlar sağlanmaktadır. Böylece bu platform üzerinden yerel sistem kaynaklarına erişilerek, platform üzerinde sunucu taraflı programlama gerçekleştirilebilmektedir.

2.7. Javascript Çerçeveleri

Model-Görünüm-Denetleyici (MVC) mimarisi uygulamanın çeşitli katmanlara bölünerek geliştirilmesini ve sürdürülebilirliğini kolaylaştırmak üzere kullanılan bir yaklaşımdır. MVC yaklaşımında, veri, iş mantığı ve görünümü oluşturan bileşenler, birbirlerinden ayrılarak geliştirilmektedir (Leff and Rayfield, 2001). Böylece yazılım herhangi bir yerinde gerçekleşebilecek değişikliklerin ve eklemelerin en az maliyetle ve en kısa sürede gerçekleştirilmesi hedeflenmektedir. Bunun yanı sıra uygulama kodları, geliştirici ekipleri tarafından daha okunabilir bir hale gelmekle birlikte, yazılımın sürdürülebilirliğini ve genişletebilirliği de artmaktadır.

MVC modeli ilk kez Trygve Reenskaug tarafından Smalltalk-76 adlı programlama üzerinde tanıtılmıştır (Reenskaug, 1979). MVC yaklaşımı aslında bir tasarım deseni (design pattern) olarak da değerlendirilmektedir. MVC mimarisinde, kullanıcı görünüm (view) ile etkileşime girmektedir. Görünüm katmanında, verilerin kullanıcıya gösterimi ve sunum şekli ele alınmaktadır. Görünüm, arayüzü oluşturması anlamında, kullanıcıyla etkileşimin başlangıç noktasıdır. Uygulama kullanıcısı, görünüm aracılığıyla, eylemler gerçekleştirerek, denetleyicide karşılık gelen eylemleri tetiklemektedir. Bu etkileşim aracılığıyla, kullanıcı talebi doğrultusunda denetleyici aracılığıyla model üzerinde çeşitli işlemler gerçekleştirilmektedir. MVC mimarisinin kavramsal modeli Şekil 2.10'da verilmektedir.

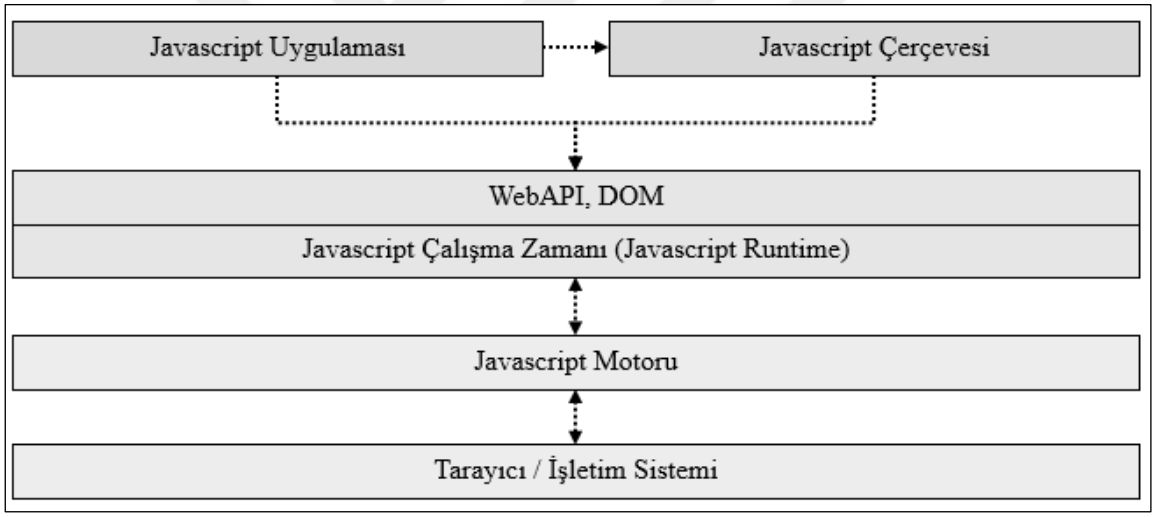


Şekil 2.10. Model-görünüm-denetleyici yapısı

Görünüm kullanıcı isteklerini controller (denetleyici) aktardıktan sonra, denetleyici (Controller), kullanıcı tarafından gelen istekleri işleyen ve model üzerinde bu taleplere göre aksiyonları (işlemleri) gerçekleştiren, MVC tasarım modelinin merkezi katman olarak görev yapmaktadır. Denetleyici (Controller) tarafında geliştirilen kod, kullanıcının View (Görünüm) ile etkileşime geçmeye başladığında çağırılan fonksiyonları barındırmaktadır. Denetleyici (Controller), görünümden aldığı taleplere karşılık gelen fonksiyonları çalıştırmakta, bu talepler doğrultusunda model üzerindeki veri üzerinde çeşitli değişiklikler gerçekleştirmekte veya kullanıcıya gösterimi için görünümün, modelden alınan veriye göre tetiklenmesini sağlamaktadır.

Modelde, üzerinde çalışılan ve kullanıcıya gösterimi sağlanan veriye dair işlemler gerçekleştirilmektedir. Ancak uygulamaya dair iş mantığı kodu (business logic), model içerisinde tutulmamaktadır. Modelin tek amacı verileri ilgili kayıt işlemek, göndermek veya diğer bileşenler tarafından kullanılmak üzere görünüm katmanı için hazırlamaktır. Modelde oluşan değişiklikleri gerçekleştirildikten ya da model katmanından ilgili isteğe karşılık gelen veri alındıktan sonra, görünüm güncellenerek kullanıcıya geri bildirim sunulmaktadır.

Javascript çerçeveleri kullanılarak, Model-Görünüm-Denetleyici yaklaşımı, web uygulamalarında istemci tarafına uygulanmaktadır. Böylece, istemci web uygulamasının, daha yapısal, genişleyebilir ve yönetilebilir bir geliştirme sürecine evrilmesi hedeflenmektedir. Javascript Çerçevesinin, istemci bilgisayarındaki tarayıcı üzerinde kavramsal çalışma şeması Şekil 2.11'de görülmektedir.



Şekil 2.11. Javascript çerçevesi çalışma ortamı

Web uygulamasında bulunan Javascript kodları, doğrudan DOM ve Javascript çalışma zamanına erişebilmelerinin yanı sıra, Javascript çerçevesinin sağladığı özellik ve fonksiyonları kullanarak çalışmaktadırlar. Javascript Çerçeveleri, Javascript ile geliştirilmiş, uygulama geliştirme kütüphanesi veya birden fazla kütüphaneden oluşmaktadır. Çerçeve, kodun yazımı hakkında geliştiriciye altyapı, araçlar ve yazılım geliştirme yaklaşımı sunmaktadır. Çeşitli işlemleri gerçekleştirmek için fonksiyonlar, özellikler ve araçlar barındırmakta, bunun yanı sıra farklı tarayıcılar ve farklı Javascript motorları üzerinde, çapraz tarayıcı uyumluluğunun sağlanabilmesi (cross browser compatibility) için standart sağlamaktadır.

Javascript çerçeveleri, görünüm katmanında, modelden gelen verinin arayüze bağlanabilmesi için, şablon yaklaşımı, model (veri) ve view (görünüm) katmanlarını net bir şekilde ayırabilmesiyle, tasarımcılar ve programcıların birlikte ve bağımsız çalışmalarını sağlamaktadır (Garcia-Izquierdo and Izquerdo, 2012). Görünüm şablonu (view template), verinin görünüm katmanında gösterimi için arayüz kodunu ve ilgili verinin gösterimi için yer tutucuları (placeholder) bünyesinde barındırmaktadır. Görünüm şablonları, geliştirme sürecini hızlandırmakta ve kod yönetilebilirliğini kolaylaştırmaktadır. Böylelikle arayüz tasarımcıları ve uygulama geliştiricileri hızlı şekilde web uygulaması geliştirebilmektedirler (Garcia-Izquierdo and Izquerdo, 2012).

2.7.1. Şablonlama

Javascript kodu içerisinde veri doğrudan kod içerisinde sayfaya yazdırılabilmekte ya da bir HTML şablonu kullanılarak ilgili DOM elemanlarına bağlanabilmektedir. Şekil 2.12'de verilen kod bloğunda, bir menü ve menüyü oluşturan menü elemanları, HTML şablon kodunun döngüsü içerisinde bir liste modeli kullanılarak oluşturulmaktadır. Şablon içerisinde tanımlanan döngüye ait kısımlar {{#each}} – {{/each}} blokları arasında gerçeklenmektedir. Şablon kodu bu kod bloğu arasında kalan kısımlar için, kendisine verilen liste üzerinde liste boyutu kadar döngüye girerek çıktı üretmektedir.

```
<script type="text/html">

        <ul>
            {{#each sehirler}}

            <li>{{SehirAdi}}</li>

            {{/each}}
        </ul>

    </script>
```

Şekil 2.12. Liste şablon kodu

Örnekte arayüze dair HTML kodlarının yanı sıra, veri modeline ait parantezler "{ }" ile işaretlenmiş yer tutucular (placeholder) bulunmaktadır. Şablon oluşturulduktan sonra, veriyi bir şablona bağlamak gerekmektedir. Veri modeli örneği Şekil 2.13'de görülmektedir.

```
var Country = {

    Name: "Türkiye",

    Code: "90",

    Population: "80M",

    Cities: {

        City:{ Id:6, Name: "Ankara" },
        City:{ Id:34, Name: "İstanbul" },
        City:{ Id:35, Name: "İzmir" },

    }
}
```

Şekil 2.13. Model nesnesi örneği

Şablon modeli veri modeline bağlandıktan sonra, ilgili şablonlama çerçevesi, şablonu inceleyerek, ilgili veri modelinin karşılık gelen alanlarını, veriye ait değişken değerleriyle değiştirmektedir. Böylece Şekil 2.14'de verilen çıktı üretilmektedir.

```
<ul>

    <li>İstanbul</li>

    <li>Ankara</li>

    <li>İzmir</li>

    <li>Bursa</li>

    <li>Adana</li>

    <li>Antalya</li>

</ul>
```

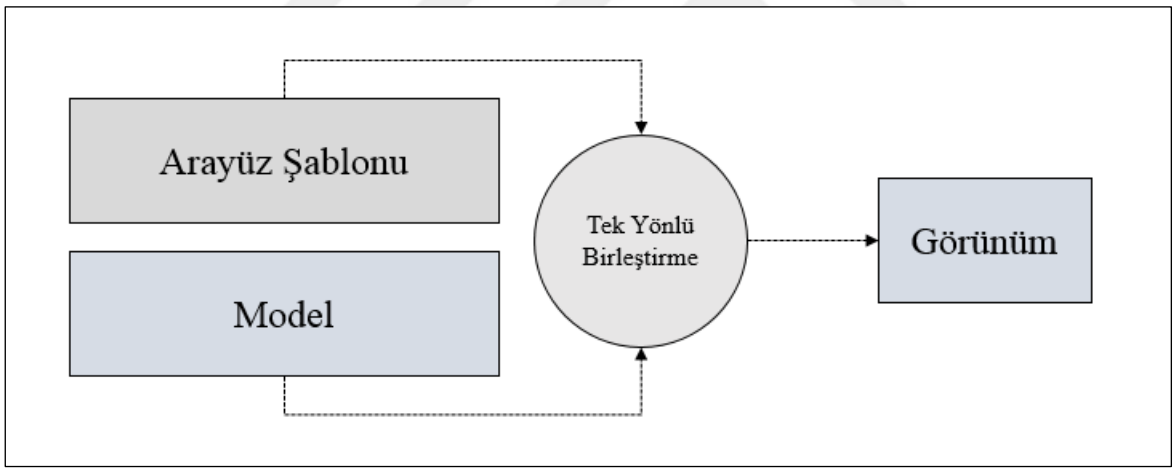
Şekil 2.14. Liste Şablon HTML çıktısı örneği

HTML şablonlama yaklaşımı, değişik şekillerde de ifade edilebilir ve geliştirilebilir. İş mantığı içeren kodların bir kısmı, şablon koduna aktarılabilir. Gösterimi yapılacak verinin tekil ya da çoklu olma durumuna göre ya da şarta ve durumlara (if-else vb) bağlı olarak şablonlar çeşitli kodlar içerebilmektedirler.

2.7.2. Tek ve çift yönlü veri bağlama

Javascript çerçevelerinde şablon yaklaşımı temelinde, gösterimi gerçekleştirilmesi istenilen verinin ilgili DOM elemanlarına bağlanması bulunmaktadır. Modelde saklanan veri, ilgili DOM elemanına bağlanarak, veride oluşan herhangi bir güncellemenin doğrudan DOM'a yani arayüze yansıtılması ya da DOM üzerinde gerçekleştirilen verideki değişimin doğrudan modele yansıtılması amaçlanmaktadır. Bu yönteme veri bağlama (data binding) adı verilmektedir.

Veri bağlama, tek yönlü veya çift yönlü olarak iki farklı yaklaşımla gerçekleştirilebilmektedir. Tek Yönlü veri bağlama kavramı, modelde bulunan verinin doğrudan Javascript kodu içerisinde veya HTML şablonu kullanılarak arayüze aktarılmasıyla uygulanmaktadır. Şekil 2.15'de görüldüğü üzere, bu yöntemde tek yönlü olarak veri, görünüme aktarılmaktadır.



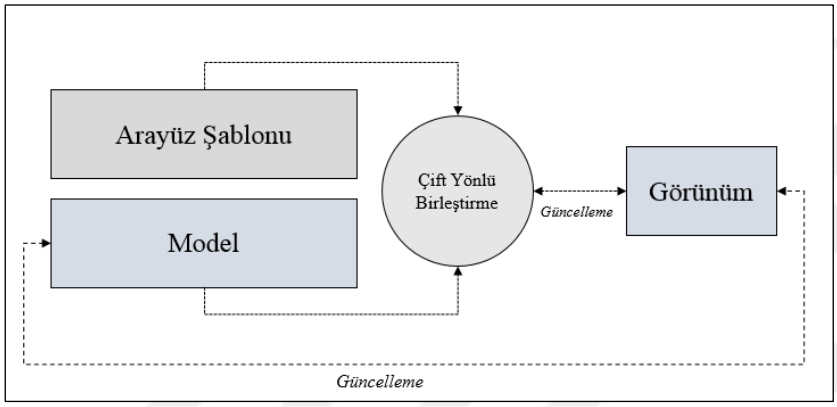
Şekil 2.15. Tek yönlü veri bağlama

Bu yöntemde, model verisi görünüme aktarıldıktan sonra, modelde oluşacak herhangi bir değişiklik arayüzü otomatik güncellememektedir. Şablon ve model verisi yalnızca bir seferliğine birleştirilerek görünüm katmanına yansıtılmaktadır. Bu yaklaşımda görünümün güncellenmesi geliştiriciye bırakılmıştır.

Tek yönlü veri bağlama yönteminde, verinin güncellenmesi durumunda, sadece model güncellendiği için, değişiklikleri görünüm katmanına yansıtabilmek için ek geliştirmeler yapmak gerekmektedir. Görünüm katmanında ilgili veri için gösterim yapılan ilgili DOM ağacı düğümü bulunduktan sonra, değişikliğin arayüze yansıtılması gerekmektedir.

Sık güncellenmeyen ve/veya kritik olmayan uygulamalarda verinin gösterimi için tek yönlü veri bağlama iyi bir seçenek olarak değerlendirilebilir. Verinin güncellenmesi durumunda görünümü güncellemek için ek uygulama kodu çalışmadığı için, bu tür uygulamalarda performans olarak da kazanç sağlayacağı düşünülmektedir.

Çift yönlü veri bağlama yaklaşımında ise, modelde yer alan veri güncellendiğinde, görünüm de güncellenmektedir. Tam tersi durumda veri, görünüm katmanında güncellenmesi durumunda, model katmanında da güncelleme sağlanmaktadır. Çift yönlü veri bağlama yöntemine ait kavramsal şema Şekil 2.16'da görülmektedir.



Şekil 2.16. Çift yönlü veri bağlama

Bu yöntem, geliştirilen uygulamanın özelliklerine göre, sürekli güncellenen ve/veya kritik olan uygulamalarda, verinin arayüzde gösterimi ve tutarlılığının sağlanabilmesi için tercih edilebilmektedir. Veri güncellemesinin sık olarak gerçekleştiği uygulamalarda, iş mantığının model ve arayüzde eşzamanlamasını (senkronizasyon) sağlamak için bu yaklaşım daha uygun olarak düşünülebilmektedir.

2.7.3. Sanal ve gerçek dom oluşturma

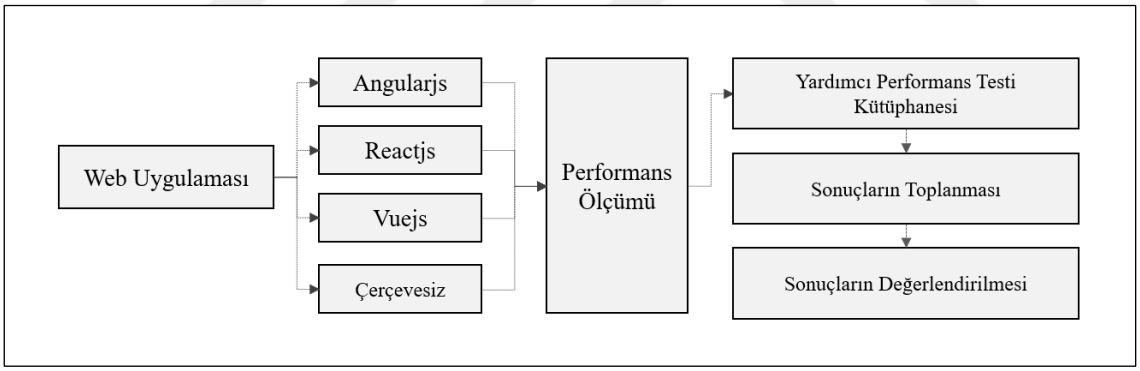
Dom ağacı üzerindeki düğümlerden herhangi birisinde değişiklik oluşması durumunda, tüm dom ağacı ekranda yeniden oluşturulmaktadır. Sanal doküman nesne modeli (Virtual dom), gerçek doküman nesne modelinin hafızada tutulan bir kopyasıdır. Dom üzerindeki bir durum değişikliğinde, doğrudan gerçek dom güncellenmek yerine, sanal dom ağacı güncellenmektedir. Ekrana yansıtılacak dom ağacındaki değişiklikler sanal dom'a aktarıldıktan sonra, sanal ve gerçek dom ağacı arasında oluşan farklılıklar, fark (diff) algoritmasıyla hesaplanarak, farklı kısımlar gerçek DOM'a aktarılır. Böylece gerçek ve

sanal dom ağaçları eşitlenmiş olur. Sanal DOM güncellemeleri biriktirerek gerçek DOM'un tek seferde ve kısmi olarak güncellenmesini sağlar (Zou, Chen, Zheng, Zhang and Gao, 2014). Böylece her durum değişikliğinde gerçek dom ağacının tüm düğümleriyle birlikte tekrar tekrar yeniden oluşturulması yerine kısmi güncelleme ile performans artışı sağlanmaktadır.

3. YÖNTEM

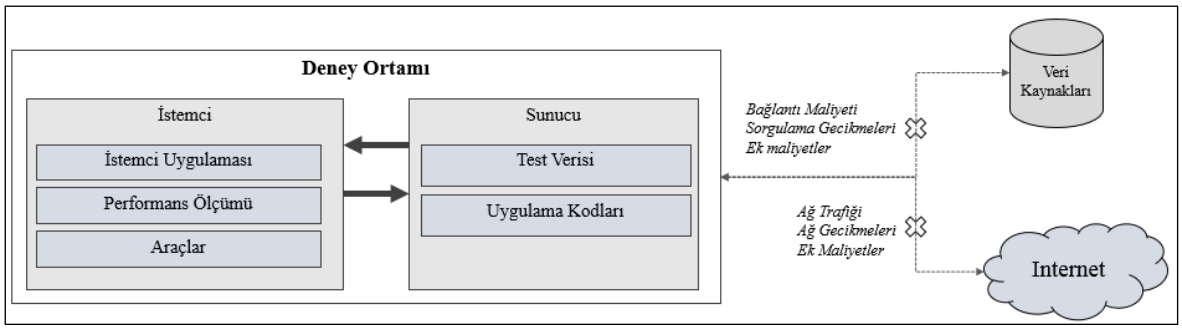
Bu bölümde Javascript çerçevesi kullanımının web uygulama yükleme performansına etkisinin araştırılması kapsamında araştırma soruları, araştırma yöntemi ve araştırma sürecinde oluşturulan deney ortamı ve kullanılan araçlara yer verilmektedir.

Bu çalışma kapsamında, araştırma sorularına yanıt bulabilmek için, kontrollü deney ortamında gerçekleştirilen testler sonucunda ortaya çıkan nicel veriler doğrultusunda bulgular incelenmekte ve değerlendirilmektedir. Şekil 3.1'de araştırma yöntemi modeli görülmektedir. Oluşturulan kontrollü deney ortamında, istemci taraflı Javascript çerçevelerinin web uygulaması yükleme performansına etkisini incelemek, ölçümleyebilmek ve gözlem verileri elde edebilmek için aynı uygulamanın farklı çerçevelerle geliştirilen sürümleriyle, farklı tarayıcılar üzerinde performans testleri gerçekleştirilmektedir. Yardımcı performans testi kütüphanesi sonucunda toplanan sonuçlar, karşılaştırma ve ayrıntılı şekilde incelenerek değerlendirilmiştir.



Şekil 3.1. Araştırma yöntemi akış diyagramı

Deneyleri gerçekleştirebilmek için, deney ortamının diğer dış faktörlerden etkilenmemesi ve iyi tanımlanmış belirli niteliklere sahip olması gerekmektedir. Böylece gerçekleştirilen çalışmanın benzer olarak hazırlanan bir ortamda tekrarlandığında, yakın ve benzer sonuçlar vermesi hedeflenmektedir.



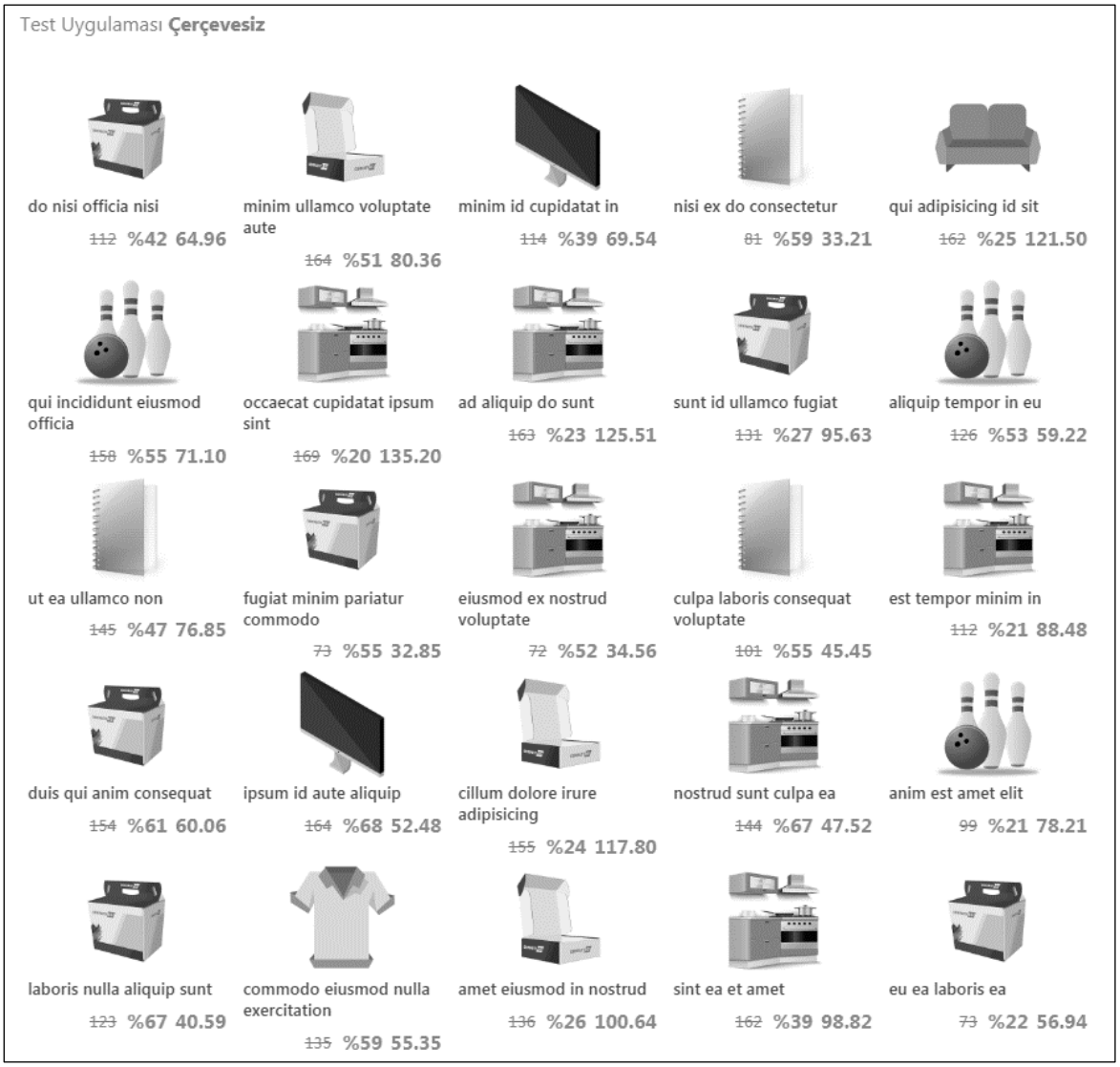
Şekil 3.2. Deney ortamı

Tez çalışması kapsamında oluşturulan deney ortamına ait kavramsal şema Şekil 3.2'de verilmektedir. Deney ortamı için aşağıdaki özelliklerde bir platform kullanılmıştır**.**

- **İşletim Sistemi:** Windows 10 x86 64 bit İşletim Sistemi (Sürüm 10.0.17763.805)
- **Birincil Hafıza:** 16 GB DDR4 2033 Mhz RAM
- **İşlemci:** Intel Core i7-6700HQ 2.6 GHZ Quad Core
- **Sunucu:** Node.js LiveServer v1.2.1

Tez çalışması kapsamında, performansa etkinin belirlenebilmesi için araştırma kapsamında bir web uygulaması oluşturulmuştur. Geliştirme ortamında test edilen her bir çerçeve için ayrı ayrı uygulama sürümleri geliştirilmiştir. Geliştirilen örnek web uygulamasına ait ekran görüntüsü Şekil 3.3'de verilmektedir. Oluşturulan uygulamanın, çalışma kapsamında seçilen her bir çerçeve kullanarak ayrı sürümleri oluşturulmuştur. Aynı uygulama farklı çerçeveler kullanılarak ve herhangi bir çerçeve kullanılmadan tekrar geliştirilmiştir. Böylece her bir Javascript Çerçevesinin ve mimari yaklaşımlarının web uygulama yükleme süresine etkileri, çerçevesiz kullanım da dâhil olmak üzere aynı uygulamanın 4 farklı sürümünde karşılaştırmalı olarak test edilebilmektedir.

Wang vd. (Wang, Krishnamurthy and Wetherall, 2016) tarafından yapılan çalışmada, sayfa yüklemesi sırasında CSS'nin dörtte üçünün kullanılmadığı, CSS ve JavaScript'i ayrıştırmayı bloke etmenin, sayfa yüklemelerini % 20 yavaşlattığını tespit etmişlerdir. Bu nedenle deney ortamında, oluşturulan tüm uygulama sürümlerinde performansı eşit şartlarda ölçebilmek için html düzeni ve css yapısı, kaynaklar, kullanılan veri seti ve javascript yükleme sırası, birebir aynı olacak şekilde oluşturulmuştur.



Şekil 3.3. Test uygulaması

Kontrollü deney ortamında gerçekleştirilen test sayısını ve veri büyüklüğünü tespit edebilmek için, daha önce gerçekleştirilen benzer çalışmalar incelenmiştir. Söz konusu çalışmalar göz önüne alındığında, daha önce Davila (Davila, 2015) tarafından 1000 satırlık veri boyutu ile 5'er kez ölçüm gerçekleştirildiği gözlemlenmektedir. Bununla birlikte Koetsier (Koetsier, 2016) tarafından gerçekleştirilen çalışmada, 1000 satırlık veri boyutu üzerinde 1'er kez ölçüm gerçekleştirildiği, Molin (Molin, 2016) tarafından gerçekleştirilen çalışmada, 100 ve 1000 satırlık veri boyutları üzerinde 1'er kez ölçümler gerçekleştirildiği gözlemlenmiştir. Bu doğrultuda, önceki çalışmalara paralel olarak nicel veriler üzerinden, örneklem kümesinin temsiliyetini arttırabilmek, veri boyutundaki değişimin de çerçeve bazında uygulama yükleme performansına etkisi değelendirebilmek için uygulama sürümlerinde sırasıyla 10 ve 1000 adet nesne içeren veri setleri kullanılmış ve her veri setiyle

5’er kez testler gerçekleştirilmiştir. Veri setleri JsonGenerator uygulaması (Json Generator, 2019) kullanılarak JSON formatında üretilmiştir.

3.1. Kullanılan Araçlar

Bu bölümde tez kapsamında gerçekleştirilen çalışmada kurgulanan deney ortamında, performans deneylerinin gerçekleştirilebilmesi için kullanılan araçlara yer verilmektedir.

3.1.1. Tarayıcılar

Web sayfası yükleme performansına etkinin araştırılabilmesi için, bu tez çalışması kapsamında Çizelge 3.1’de verilen tarayıcılar karşılık gelen versiyonlarıyla birlikte kullanılmaktadır.

Çizelge 3.1. Çalışmada kullanılan tarayıcılar ve sürümleri

Tarayıcı Adı	**Sürüm**	**Platform**
Chrome	78.0.3904.70	Windows 10 x64 Bit
Firefox	69.0	Windows 10 x64 Bit
Internet Explorer	11.0.155	Windows 10 x64 Bit

Çalışmada her bir tarayıcı üzerinde, uygulama yükleme performansına etkide bulunmaması için herhangi bir eklenti kurmadan tarayıcıların sadece yalın sürümleri üzerinde deneyler gerçekleştirilmiştir.

3.1.2. Gezinme zamanlaması uygulama programlama arayüzü

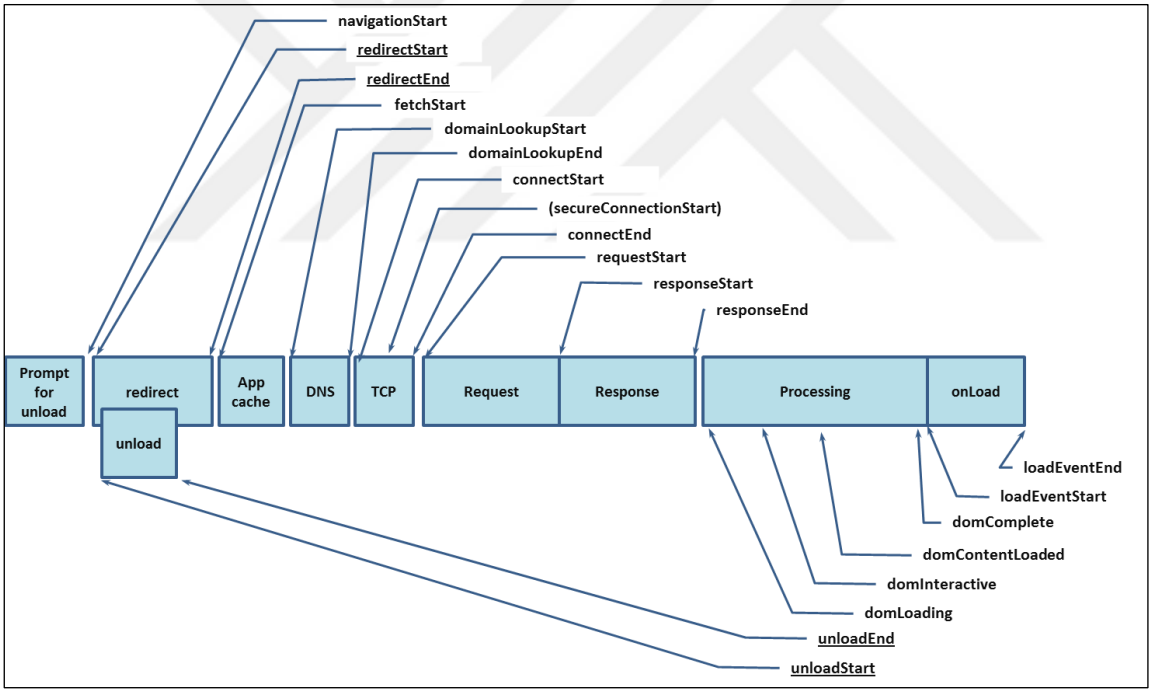
Gezinme Zamanlaması Uygulama Programlama Arayüzü (NavigationTimingAPI), bir web sayfasının yükleme evrelerine dair zaman damgalarını (timestamp) oluşturmakta, sayfa performansıyla ilgili belirli kayıtlar tutmaktadır.

NavigationTimingAPI, Tarayıcı Tabanlı Javascript Çalışma Zamanı üzerinde sunulmaktadır. Bu kapsamda, bir web sayfasının ve bu web sayfasına bağlı ilgili kaynakların sunucudan alınarak, ekranda gösterimine kadar geçen süreçte gerçekleşen tüm olaylar için zaman damgası bilgisi yüksek çözünürlüklü olarak oluşturulmaktadır. Oluşturulan bu

yüksek çözünürlüklü zaman damgasına, DOMHighResTimestamp adı verilmektedir. İşaretsiz büyük sayı (unsigned long) türünde veri döndürmekte, gezinme başlangıcı (navigationStart) olayından itibaren geçen zaman değerini, milisaniye (ms) veya mikrosaniye (µs) cinsinden vermektedir.

Bu çalışmada NavigationTimingAPI tarafından sağlanan yüksek çözünürlüklü zaman damgaları kullanılarak, sayfa yüklemesi sırasında, belirli olaylar arasında geçen zamanlar hesaplanmaktadır. Böylece istemci saat ve tarih bilgisine bağlı kalmadan, doğrudan zaman damgaları üzerinden hesaplama yapma imkânı bulunmaktadır.

NavigationTimingAPI üzerinde tutulan yüksek çözünürlüklü zaman damgaları ve bu zaman damgaları içeren bölümlere ait olay tetikleyicileri Şekil 3.4'de görülmektedir.



Şekil 3.4. Sayfa gezinmesi süreçleri ve olay tetikleyicileri (W3C, NavigationTimingAPI, 2012)

Gezinme Başlangıcı (NavigationStart)

Bu bölüm kullanıcıdan yeni bir sayfa isteği gelmeden hemen öncesine (Prompt for unload) ait bir zamanı bildirmektedir. Yeni bir gezinme durumu, kullanıcı sayfayı yenilediğinde, yeni bir bağlantıyı adres çubuğuna girip enter tuşuna bastığında veya bir bağlantıya tıkladığında, kısaca kullanıcı yeni bir sayfayı talep etmeden hemen önce ortaya çıkmaktadır. Kullanıcı yeni bir bağlantı (url) talep ettikten sonra, mevcut sayfa kaldırılarak yeni bağlantıya

ulaşılabilmesi için kaldırma olayı (unload event) fırlatılarak, navigationStart zaman damgası oluşturulmaktadır.

Redirect (Yönlendirme)

Bu bölüm, yönlendirme başlangıcı ve bitişi zaman damgalarından oluşmaktadır. İstemci yeni bir bağlantıyı ya da kaynağı talep ettikten sonra, Gezinme başlangıcından (NavigationStart) sonra bu bölüm çalışmaktadır. Eğer sunucu tarafından bir yönlendirme varsa, yönlendirme başlangıcı (redirectStart) ve yönlendirme sonu (redirectEnd) zaman damgaları oluşturulmaktadır. Bu iki zaman damgası arasında geçen süre yönlendirme süresidir. Herhangi bir yönlendirme olmamaması durumunda, "redirectStart" ve "redirectEnd" zaman damgaları, Gezinme başlangıcıyla (navigationStart) aynı değeri almaktadır.

Uygulama Ön Belleği (Appcache)

Bu bölümde istemci önbelleğiyle ilgili zaman damgaları saklanmaktadır. İstemci sunucudan bir kaynak talep ettiğinde, talep edilen kaynak ilk olarak tarayıcı tarafından tarayıcı önbelleğinde aranmaktadır. Bu kısmın başlangıcında bulunan "fetchStart" zaman damgası uygulama ön belleğini kontrol etmeye başlanılan zamanı bildirmektedir. Eğer kaynak önbellekte bulunuyorsa, doğrudan önbellekten alınarak kullanıcıya göstermektedir. Önbellekte ilgili kaynağın bir kopyası bulunamaması durumunda, bu değer yönlendirme bitişi (redirectEnd) zaman damgasıyla aynı değere sahip olmaktadır.

DNS (Domain Name Service) Lookup

İstemci, ilgili sunucunun IP sini çözümleyebilmek için, DNS sunucusuna bir istek göndermekte ve yanıt olarak kaynağın bulunduğu IP adresini elde etmektedir. Bu talebin iletildiği zaman damgası "domainLookupStart "olarak tanımlanmakta, talebe cevap verilen zaman damgası da "domainLookupEnd" olarak oluşturulmaktadır. Eğer talep edilen kaynak uygulama önbelleğinde mevcutsa bu iki zaman damgasının değerleri, "Appcache" bölümünde bulunan "fetchstart" zaman damgası ile aynı değeri almaktadır.

TCP (Transmission Control Protocol)

Bu kısımda "connectStart", "secureConnectionStart" ve "connectEnd" isminde üç ayrı zaman damgası bulunmaktadır. Başlangıçta "connectStart" zamanında, istemciden sunucuya kaynak hakkında giden talep (request) için yapılan bağlantı zamanı saklanmaktadır. SecureConnectionStart zamanı, eğer oluşturulan kaynak talebi güvenli bir bağlantı üzerinden

sağlanabiliyorsa oluşturulmaktadır. Bağlantı talebinin gönderilmesinden sonra, HTTPS protokolü kullanılması durumunda güvenli bağlantı için istemci ve sunucu arasındaki el sıkışma (handshaking) zamanını saklamaktadır. Eğer ortada bir güvenli bağlantı kurulmamışsa bu zaman damgası 0 (sıfır) ya da "undefined" değerini almaktadır. Daha sonra istemci ve sunucu arasındaki bağlantı kurulunca connectEnd zaman damgası değeri oluşturulmaktadır.

Request (İstek)

İstemci ve sunucu arasında TCP protokolü üzerinden bağlantı kurulduktan sonra, istemci sunucudan ilgili kaynağı talep etmektedir. Bu zaman damgası Request aşamasında, "requestStart" olarak kaydedilmektedir.

Response (Cevap)

İstemci tarafından talep edilen kaynağa verilen cevap Response aşamasında saklanmaktadır. Response aşamasının büyüklüğü, daha önceki bölümlerde verilen ağ taraflı ve sunucu taraflı performans faktörlerine, istemci sunucu arasındaki kaynak transferinin boyutlarına ve aradaki bağlantı hızına bağlıdır. Sunucunun cevabı vermeye başlama süresi "responseStart" ve verilen cevabın bitiş zamanı da "responseEnd" olarak oluşturulmaktadır.

Processing (İşleme)

Bu kısım tamamen istemci bölümünde gerçekleşmektedir. İstemci tarafından talep edilen HTML dokümanı işlenmekte, işleme sürelerine dair zaman damgaları bu bölüm içerisindeki zaman damgalarında tutulmaktadır.

DomLoading Zaman Damgası

DOM İşleme (processing) sürecinin başlangıcı 'domLoading' zaman damgasıyla başlamaktadır. Bu zaman damgası HTML'in ayrıştırılmaya (parsing) başlandığını bildirmektedir. Bunun için başlangıç zamanı "domLoading" olarak belirlenmektedir. Bu kısımda Javascript çalışma zamanı tarafından sağlanan, document nesnesinin readyState özelliği "loading" (yükleniyor) durumunda bulunmaktadır. Bu süre içerisinde istemci HTML belgesini çözümleyerek DOM ağacını oluşturmaktadır.

DomLoading zaman damgasının önemi, HTML dokümanın işlenmeye başladığını ancak diğer kaynakları işlemeye başlamadığını bildirmektedir. Sayfa gösterim zamanını

hesaplayabilmek için HTML'in ilk ayrıştrılmaya başlandığı süreyi referans almak gerekmektedir.

DomInteractive Zaman Damgası

DomInteractive zaman damgası, tarayıcının HTML ayrıştırma işlemini tamamladığını ve DOM ağacını oluşturduğunu belirtmektedir. Bundan sonraki süreç CSS'in ayrıştırılarak CSSOM oluşturulması ve kaynakların (resim vb.), alınarak yerleştirilmesidir. İstemci DOM ağacını oluşturduktan sonra sayfayı görüntülemeye başlamaktadır. Bu kısımda, kullanıcı web sayfasıyla etkileşimde bulunabilmektedir. Bu zamanın başlangıcı "domInteractive" zaman damgasında bulunmaktadır. Bu aşamada document nesnesinin readyState özelliği, "interactive" durumuna geçmektedir. DomInteractive zaman damgasının önemi, DOM ayrıştırma süresinin tamamlandığı bildirmesidir.

DomContentLoaded Zaman Damgaları

CSSOM oluşturulduktan sonra, tarayıcı Oluşturma Ağacını (Render Tree) oluşturmaya başlamaktadır. Bu sürecin başlangıcı "domContentLoadedEventStart" ve bitişi "domContentLoadedEventEnd" zaman damgalarında saklanmaktadır. Böylece Oluşturma Ağacının ne kadar sürede oluşturulduğu hesaplanabilmektedir.

İşleme aşamasının başlangıcını oluşturan "domLoading" olayının bitişi "domContentLoadedEnd" zaman damgasında saklanmaktadır. Böylece HTML içeriğinin tamamen ayrıştırılarak DOM'a yüklendiği bildirilmektedir. DomContentLoadedEnd zaman damgası, DOM'un hazır olduğu ve JavaScript yürütmesini engelleyen herhangi bir stil sayfasinin CSS bulunmadigi noktayı işaretlemektedir. DomContentLoaded zaman damgalarıyla oluşturma engelleyici Javascript olup olmadığı ve bu Javascript'in kritik oluşturma yolunu engelleyip engellemediği belirlenebilmektedir.

DomComplete Zaman Damgası

Tüm kaynakların indirilmesi ve ardından DOM'un yüklenmesinin ve işlenmesinin tamamlanmasıyla birlikte "domComplete" değişkenine zaman damgası, bitiş zamanı olarak atanmaktadır. Bu aşamada tüm sayfanın ve alt kaynaklarının hazır olduğu bildirilmekte, tarayıcının yükleniyor simgesi kaybolmaktadır. Bu aşamada sayfadaki Javascript kodları çalışmaya başlamaktadır.

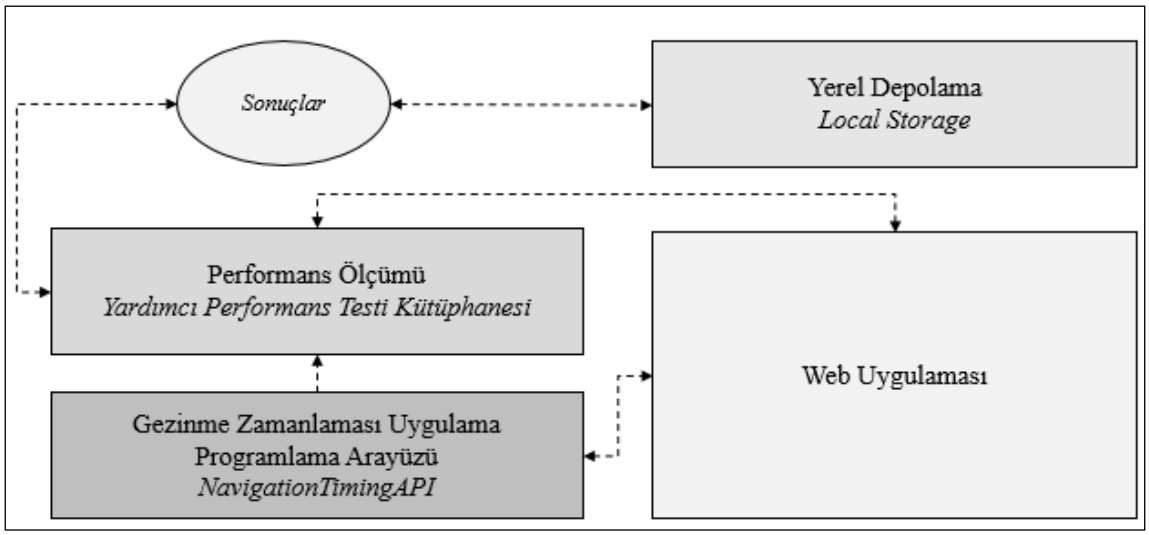
OnLoad (Yükleme)

Bu bölümde tüm işlemler tamamlandıktan sonra, istemci tüm DOM için yükleme olayı başlangıcını (LoadEventStart) saklamaktadır. Böylece geliştirici tarafından belgenin yüklenmesinin tamamlanıp tamamlanmadığı anlaşılabilmektedir. "LoadEventEnd" zaman damgası ile sayfanın tamamen yüklendiği bilgisine ulaşılmaktadır.

Bu çalışma kapsamında sayfa yükleme zamanı, Sampson (Sampson, Cascaval, Ceze, Montesinos and Suárez, 2012) ve Butkiewicz (Butkiewicz, Madhyastha and Sekar, 2011) tarafından gerçekleştirilen daha önceki çalışmalardan hareketle gezinme başlangıcından (NavigationStart) yükleme bölümünün sonuna kadar (OnLoadedEventEnd) geçen tüm süre dikkate alınarak hesaplanmıştır. Gezinme Zamanlaması Uygulama Programlama Arayüzü ile Sayfa Yükleme Zamanını hesaplayabilmek için, "onLoad" bölümünün bitişinde bulunan "yükleme zamanı bitişi (LoadEventEnd)" ile "Processing (İşleme)" bölümünün başlangıcında bulunan DOM Yükleme (domLoading) zaman damgaları arasındaki yüksek çözünürlüklü zaman farkı milisaniye olarak hesaplanmaktadır.

3.1.3. Yardımcı performans testi kütüphanesi

Araştırma kapsamında performans testlerinin tekrarlı ve kolay şekilde gerçekleştirilebilmesi ve sonuçların kaydedilebilmesi için yardımcı performans testi kütüphanesi geliştirilmiştir. Geliştirilen bu yardımcı araç ile tarayıcı üzerindeki Javascript çalışma zamanında (Javascript Runtime) bulunan Gezinme Zamanlanması Uygulama Programlama Arayüzü (NavigationTimingAPI) ve Yerel Depolama (Local Storage) nesnelerini ve bu nesnelere dair özellik ve fonksiyonları kullanılmaktadır.

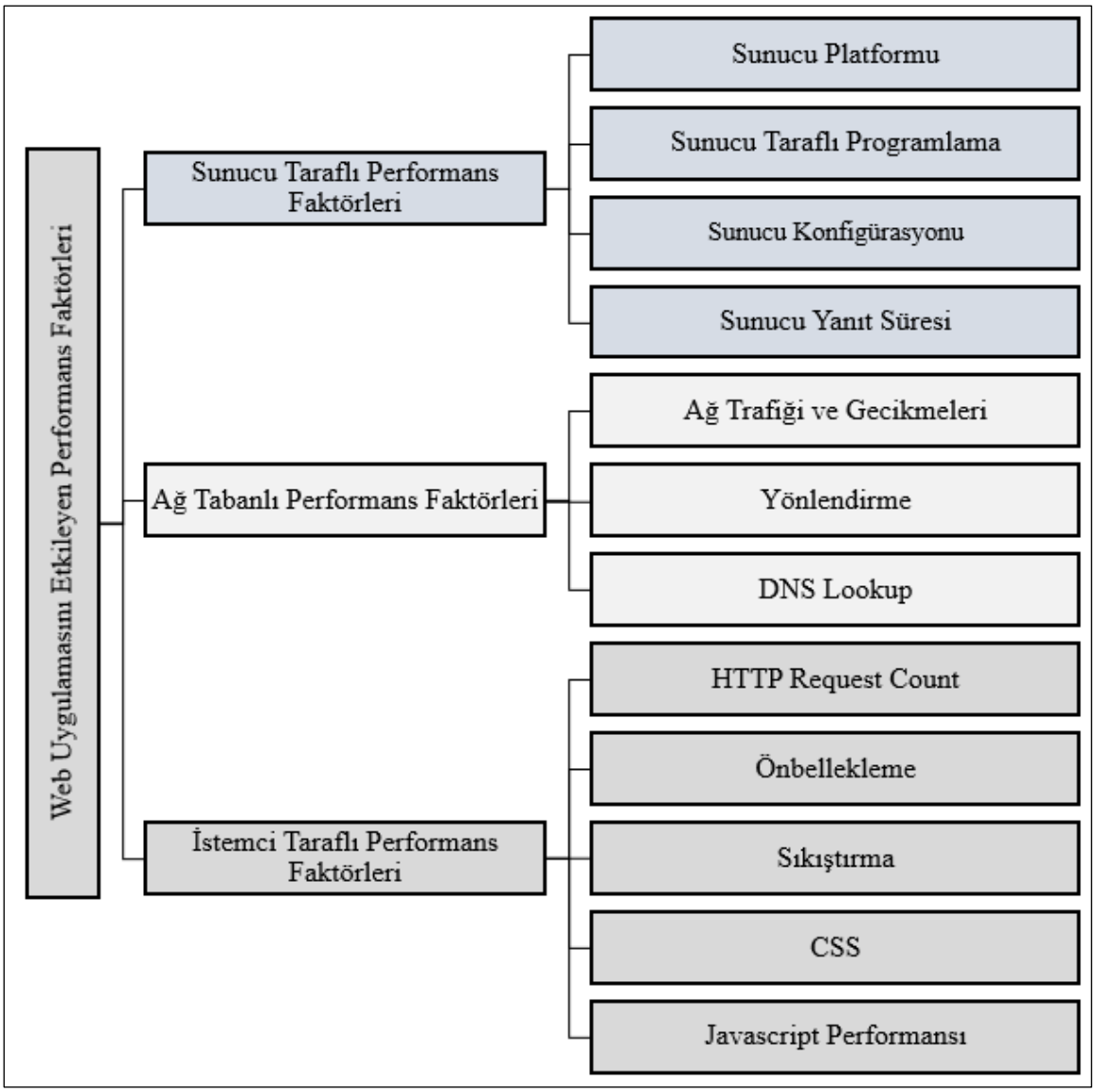


Şekil 3.5. Yardımcı performans testi kütüphanesi

Yardımcı performans testi kütüphanesi çalışma mantığına ait kavramsal şema Şekil 3.5'de görülmektedir. Yardımcı kütüphane, gezinme zamanlaması uygulama programlama arayüzü kullanılarak ölçülen web uygulaması yükleme zaman damgalarını otomatik olarak yerel depolamaya kaydetmektedir.

3.2. Web Sayfası Yükleme Performansını Etkileyen Faktörler

Web sayfası yükleme performansını etkileyen faktörler, Şekil 3.6'de görüldüğü üzere, 3 ayrı temel kavrama; istemci, sunucu ve bu ikisi arasındaki iletişimi sağlayan ağ performansına bağlıdır. Sunucu taraflı programlamanın yanı sıra, ağ üzerinde gerçekleştirilen iletişim ve istemci tarafında çalıştırılan kod parçacıklarının tümü web sayfası yükleme performansına etki etmektedir.



Şekil 3.6. Web sayfası yükleme performansını etkileyen faktörler

Sunucu taraflı performans faktörlerinde, kullanılan platformun sağladığı özellikler, performansı yakından etkilemektedir. Kodun derlenerek ya da derlenmiş hazır bir kopyasının çalıştırılması ya da hafıza kullanımı gibi etkenler seçilen platformun daha hızlı veya daha yavaş çalışmasına yol açmaktadır. Sunucu taraflı programlamada kullanılan algoritma, veri büyüklüğü ve kullanılan yaklaşımlar sunucu kısmında yanıt verme süresini etkilemektedir. Özellikle veritabanı ve diğer veri kaynaklarına erişim süreleri sunucu taraflı uygulama performansına ciddi etkide bulunmaktadır.

Bunun yanı sıra, seçilen platform ve uygulamaya dair düzenlemeler (sunucunun istekleri karşılama biçimi, sunucu yapılandırması vb.) çok sayıda kullanıcıya sahip uygulamalarda performansı etkileyen etkenler arasındadır. Sunucu yanıt süresi ise tüm bunların yanı sıra,

sunucunun donanım kapasitesine, sunucunun kullandığı diğer kaynakların hızına ve işlenen veri boyutu gibi diğer etkenlere bağlıdır.

Ağ taraflı performans faktörleri ise büyük ölçüde istemci ve sunucu arasındaki iletişimde yaşanan hıza, bant genişliğine, iletişim sırasında yaşanabilecek muhtemel gecikmelere ve hata oranına bağlıdır. İstemcinin mevcutta sahip olduğu bant genişliğinin yanı sıra, sunucu tarafında kurgulanan bant genişliği, DNS adresinin çözümlenmesi, web sayfasının yönlendirmeleri de yanıt verme süresine etki etmektedir.

DNS Arama (DNS Lookup), uygulama performansını etkileyen ağ taraflı performans faktörlerinden birisidir. Kullanıcılar erişmek istedikleri sunuculara ait IP adresleri yerine, kendileri için daha kolay hatırlanabilir anlamlı kelimelerden oluşan domain adları kullanmaktadır. İstemci bilgisayar bir sunucuya erişmek istediğinde, istemci bilgisayarı kendisinde tanımlı DNS sunucusuna veya internet servis sağlayıcısı üzerinde kayıtlı DNS sunucusuna, bu adresin hangi IP ye ait olduğu bilgisini talep etmektedir. Daha sonra DNS sunucusu istemci bilgisayara, bu domain adresine karşılık gelen IP numarasını döndürmektedir. Böylece istemci bilgisayar sunucuya ulaşabilmektedir. Alan adı sistemi (DNS-Domain Name System) çalışma mantığı bu yaklaşıma dayanmaktadır.

Sunucu tarafından istemciye döndürülen cevapta, birden fazla sunucu üzerinde bulunan kaynaklar referans olarak döndürülebilmektedir. Bu referans kaynaklar başka sunucularda, örneğin bir İçerik Dağıtım Ağı (CDN-Content Delivery Network) üzerinde bulunan kaynak olabilir. Bu durumda istemci referans verilen kaynakları, belirtilen Tekil Kaynak Tanımlayıcı bilgisiyle (URL-Uniform Resource Locator) ilgili sunucudan temin etmesi gerekecektir. Bu durumda tekrar DNS sunucusuna talepte bulunarak, söz konusu domainlere ait IP adreslerini de talep etmektedir.

Bu yönüyle DNS Arama işlemi, sunucu tarafından istemciye gönderilen kaynaklarda, çok sayıda alan adı bulunması durumunda bu alan adlarına dair DNS çözümleme istekleri ve bu isteklere dair yanıt süreleri, sayfa yüklenmesini geciktirmektedir.

Bunun yanı sıra, ağ tabanlı performans faktörlerinden, HTTP yönlendirmesi, istemcinin talebini başka bir adrese yönlendirme cevabı verilmesi durumunu tanımlamaktadır. İstemci sunucundan bir kaynağın adresini talep ettiğinde, sunucu, kaynağı bulması durumunda

istemciye “200” (ok) yanıtıyla birlikte ardından kaynağı gönderebilmektedir. Ancak bunun yanı sıra diğer durumlarda, HTTP yanıt mesajları bölümünde bahsedildiği üzere, istemciye 300 (çoklu seçenek – multiple choices), 301 (kalıcı taşınma – moved permanently), 302 (bulundu- found) veya 101 (protokol değişikliği – switching protocols) kodlarıyla kaynağın kendisi yerine diğer türlerde yanıtlar verebilmektedir. İstemci uygulama HTTP protokolü seviyesinde kendisine verilen bu yanıtı işleyerek, yanıtın türüne göre gerekli eylemleri gerçekleştirmekte, örneğin yönlendirme yapılan adresten kaynağı talep etmektedir. Bu süreçte, istemci birden fazla kez ilgili kaynağa ulaşabilmek için yönlendirilen bağlantıya talepte bulunmaktadır. Bu durumda ilgili kaynağın istemci tarafından elde edilmesi için gereken zaman uzamaktadır.

İstemci taraflı performans faktörleri, yazılımın istemci tarafında kurgulanan uygulama yapısı ve kullanılan yaklaşımlara bağlıdır. Bu yaklaşımlar doğrultusunda, istemcinin sunucuya kaç kere ve hangi sıklıkta istekte bulunduğu, kaynakların önbelleklenmesi, talep edilen kaynakların sıkıştırılıp sıkıştırılmadığı, CSS dosyasında stillerin tanımlanma şekli gibi etmenler, istemci tarafında performansı etkileyen faktörler arasında sayılabilmektedir.

İstemci tarafında, HTTP İstek Sayısının uygulama performansını etkileme durumu, istemci ve sunucular arasındaki iletişimin ve kaynak taleplerinin doğru şekilde kurgulanmasına bağlıdır. Bir web uygulamasının sunucudan gerçekleştirdiği, HTTP İstek Sayısı (HTTP Request Count), istemciden diğer uzaktaki kaynaklara (sunuculara vb.) ne kadar sayıda talep gönderildiğiyle yakından ilgilidir. Örneğin, sunucuda 4 (dört) ayrı stil dosyası (CSS) bulunması durumunda, bu dosyaları istemci HTML sayfasında tek tek referans verildiğinde, pratik olarak istemci önce HTML belgesini sunucudan almakta, daha sonra referans verilen bu dört ayrı stil dosyası için, dört ayrı istek sunucuya göndermektedir. Böylece HTML belgesi ve CSS dosyaları toplam 5 (beş) istek süreci sonucunda, sunucudan istemciye aktarılmaktadır. Ancak bu durumda, istemci sunucuya beş kez istek göndermekte, istek beş kez ağ üzerinden geçmekte, sunucu da beş kez talep işlemek durumunda kalmaktadır. Bu durumda her istek gönderildiğinde, ağ üzerinde paketler transfer edilirken, uygulama üzerinde ağ yükü (network overhead) oluşmaktadır. Bu durum da teknik olarak uygulamada herhangi bir sayfa yüklenirken ağ taraflı performans kayıplarına yol açmaktadır.

Günümüzde stil dosyalarının boyut olarak görece küçük olmasının yanı sıra, internet hızının yüksek olması da bu problemi çok farkedilir kılmamaktadır, ancak daha yüksek boyutlu

kaynaklarda (resim, ses gibi diğer öğeler) ve/veya daha düşük hızlı bağlantılarda bu faktörden kaynaklanan gecikmeler oldukça farkedilir olmaktadır. Bu probleme yanıt olarak CSS ve Javascript vb. kaynaklar, bundling (paketleme/sıkıştırma) işleminden geçirilerek, boyutları küçültülüp tek bir dosyada (bundle) birleştirilmektedir. İstemci tarafına gönderilen HTML dosyasında da bu tek dosya referans verilerek, istemciyle sunucu arasındaki talep sayısının azalması ve ağ trafiği yükünün hafifletilmesi sağlanmaktadır. İstemci bu durumda sunucudan istekte bulunduğunda, sadece HTML belgesi ve bu tek stil dosyası için olmak üzere sunucudan yalnızca 2 ayrı talepte bulunmaktadır. Böylece ağ trafiğinden kaynaklanabilecek muhtemel gecikmelere dair sürelerin en aza indirilmesi hedeflenmektedir.

Bir diğer yöntem ise, sunucudan tekrar tekrar istenen kaynakların, istemci tarafında önbelleklenmesidir. Bu yöntem daha çok, sıklıkla değişim göstermeyen statik kaynaklar için kullanılmaktadır. Sunucudan talep edilen kaynaklar, istemci bilgisayarında geçici bellekte saklanarak, aynı kaynak için tekrar tekrar sunucudan istekte bulunmadan, istemci ve sunucu arasındaki ağ trafiğinin azaltılması ve sonuç olarak sayfa gösteriminin hızlandırılması amaçlanmaktadır.

Bu yöntemler uygulanırken, tek ve ideal bir çözüm bulunmamaktadır. Örneğin kullanıcıya ait hassas verilerin (kredi kartı bilgisi, kullanıcı adı/şifre vb.), güvenlik açığı oluşturacakları için önbellekte veya istemci bilgisayarında tutulmaması gerekmektedir. Bu durumda performans yerine güvenliği ön planda tutmak uygulama için daha önemlidir. Bununla birlikte sık değişen bilgilerin veya kaynakların önbellekte tutulması uygulamada kullanıcıyı yanıltan ve/veya geçersiz bilgilerin de kullanıcıya gösterilmesine yol açmaktadır. Gereksiz (redundant) kaynakların da önbelleklenmesi durumunda, fazla hafıza kullanımına yol açmasına sebep olmaktadır. Bu yüzden uygulamanın istemci tarafında kurgulanacak yazılım mimarisinin, çalışacağı istemci cihazların kapasiteleri ve durumlarına göre ölçeklendirilmeleri, planlanmaları, sunucu ve ağ tarafında kurgulanan yaklaşımlarla uyumlu olması gerekmektedir.

Bu çalışma kapsamında, deney ortamının belirlenmesi ve oluşturulmasında, sadece istemci taraflı sayfa yükleme performansını ölçümleyebilmek için, sunucu taraflı ve ağ taraflı performans faktörlerinin ve diğer dış faktörlerin deney ortamından çıkarılması ve etkilerinin minimuma indirgenmesi gerekmektedir. Aksi takdirde, sunucu ve ağ tabanlı faktörlerdeki

gecikmelere bağlı olarak istemci tarafında uygulama yükleme süreleri de etkilenecek ve test sonuçları değişiklik gösterecektir.

Bu durumu engellemek ve test ortamının dışa bağımlılığı tamamen ortadan kaldırabilmek için, istemci ve sunucu bilgisayar aynı bilgisayar olarak belirlenmiştir. Böylelikle, sunucu ve istemcinin aynı bilgisayar üzerinde bulunması, ağ tabanlı performans faktörlerinin çalışmaya etkisini ortadan kaldırmaktadır.

Uygulama performansının ölçümü sırasında, performansa etki eden bir diğer faktör ise tarayıcı önbelleğidir (browser cache). Önbellekleme faktörünün devreden çıkarılabilmesi için HTML dokümanı içerisinde Şekil 3.7'da görülen kodlara yer verilmektedir.

```
<meta charset="utf-8">
<meta http-equiv="Cache-Control" content="no-cache, no-store, must-revalidate" />
<meta http-equiv="Pragma" content="no-cache" />
<meta http-equiv="Expires" content="0" />
```

Şekil 3.7. Tarayıcı önbelleklemenin engellenmesi

Tarayıcı önbellekleme web uygulaması seviyesinde engellenirken, istemciye gönderilen son çıktıda, <head></head> etiketleri arasında html sayfasının metaverisinde "http-equiv" direktifi kullanılmaktadır. http-equiv direktifinin özelliklerinden birisi Cache-control özelliğidir. Cache-control sayfanın tarayıcı tarafından nasıl önbellekleneceğini tarayıcıya anlatmak için vardır. Cache-control'un içerik (content) özelliği 4 ayrı değer alabilmektedir.

- public: Birden fazla istemci için önbellleklemeyi etkinleştirmektedir. Genel (public) önbellekte tutulması talep edilen kaynaklar için kullanılmaktadır.
- private: Sadece istemciye ait önbellekte tutulması istenen kaynaklar için kullanılmaktadır.
- no-cache: Önbellekleme istenmediği durumda kullanılmaktadır. Ancak istemci bu durumda da kaynağı önbellekte tutabilmektedir. Sunucuda değişim yoksa ya da sunucuya ulaşılamadığı durumda istemci kaynağını önbellekten almaktadır.
- no-store: Sunucudaki kaynak durumunda değişiklik olsun ya da olmasın, istemci her durumda sunucuya giderek kaynağın orijinalini talep etmektedir.

“pragma” özelliği, HTTP/1.0 protokolü üzerinde önbellekleme kontrolü gerçekleştirmektedir. HTTP/1.1 protokolünün kullanılmadığı istemci sunucu haberleşmesinde önbellekleme istenmediği durumda “no-cache” özelliği ile birlikte kullanılmaktadır (Fielding and Nottingham, 2019).

Bununla birlikte expires özelliği önbellekte tutulması durumunda hangi tarihe kadar kaynağın önbellekte tutulması gerektiğini tarayıcıya bildirmek için kullanılmaktadır. Bu çalışmada önbellekleme faktörünü devreden çıkararak, uygulamanın her seferinde sıfırdan yüklenmesini sağlayabilmek için Cache-control=”…”, Pragma=”no-cache” ve Expires=”0” olarak ayarlanmıştır. Böylece herhangi bir web sayfası tekrarlı olarak çağırıldığında, tarayıcının ilgili sayfaya ait HTML, CSS ve Javascript kodlarını ve diğer bileşenleri önbellekten alması engellenerek, her bir test iterasyonunda, sayfayı en baştan yükleyerek çalıştırması sağlanmaktadır.

3.3. Test Edilen Çerçeveler ve Mimari Yaklaşımları

Literatürde, Javascript Çerçevelerinin kullanım oranlarına dair, resmi bir araştırma olmadığından dolayı, çerçeveler ortaya koydukları mimari yaklaşım farklılıklarına göre, geliştiriciler tarafından sıklıkla kullanılan Github ve Stackoverflow siteleri, veritabanları üzerinde araştırılarak belirlenmiştir. Gerçekleştirilen araştırmaya göre geliştiriciler arasında en çok kullanılan ve popüler istemci taraflı 3 çerçeve, Çizelge 3.2’de verilmektedir.

Çizelge 3.2. Çerçevelere ait istatistikler [30,31]

Çerçeve	Github			StackOverflow	
	Watch	**Star**	**Fork**	**Tag**	**Followers**
Angularjs	4362	58345	28904	251223	129300
Vuejs	4835	91760	13477	16392	12100
Reactjs	5776	94126	17742	82163	53100

İstatistikler incelenirken, her iki platformun da çalışma mantığının temelini oluşturan ölçütler seçilmiştir. Github için değerlendirme aşamasında ele alınan kriterler Watch (İzleme), Star (Beğeni), Fork (proje kopyası) olarak seçilmiştir. Burada “Watch” parametresi, kütüphanenin kaç kişi tarafından takip edildiğini, “Star” parametresi kaç kişi tarafından beğeni aldığını, “Fork” ise bu ilgili çerçevenin topluluk tarafından replike

edilerek, geliştirme safhalarına ne kadar yöneldiğini belirtmektedir. Bu kriterlere göre, bu çalışma kapsamında seçilen 3 ayrı çerçeve, diğer istemci taraflı Javascript Çerçeveleri arasında öne çıkmaktadır.

Çizelge 3.3. Test edilen Javascript çerçevelerine ait özellikler

Çerçeve	**Versiyon**	**Toplam Boyut (kb) (Sıkıştırılmamış)**	**Toplam Boyut (Kb) (Sıkıştırılmış)**
Angularjs	1.6.9	1260 Kb	166 Kb
Vuejs	2.5.16	283 Kb	85 Kb
Reactjs	0.14.3	629 Kb	134 Kb

Test edilen Javascript Çerçevelerine dair toplam dosya boyutları Çizelge 3.3'de gösterilmektedir. Javascript Çerçevelerine ait kütüphanelerin iki ayrı sürümü bulunmaktadır. Geliştiriciler için bu kütüphanelerin sıkıştırılmamış (Uncompressed) sürümü ya da boyuttan tasarruf edebilmek amacıyla sıkıştırılmış sürümü (compressed/minified) bulunmaktadır. Sıkıştırılan Javascript kodlarının boyutları ve sahip oldukları kod satır sayıları küçülmektedir. Böylece Javascript motoru tarafından daha hızlı şekilde ayrıştırılabilmeleri ve ağ üzerinde transfer boyutunun küçültülmesi hedeflenmektedir.

Çizelge 3.4. Çerçevelere ait kod satırı sayıları

Çerçeve	**SLOC**		**SLOC Physical**		**SLOC Logical**	
	U	**C**	**U**	**C**	**U**	**C**
Angularjs	34359	338	12838	271	9636	1801
Vuejs	10948	6	8516	1	5062	884
Reactjs	18837	28	10120	6	7200	1675

Test edilen çerçevelere ait kütüphanelerin kod satır sayıları Çizelge 3.4'de gösterilmektedir. Bu çizelgede, SLOC (Source Lines of Code) metriği ilgili çerçeve kütüphanelerinin toplam kod satır sayısını belirtmektedir. SLOC Physical metriği fiziksel olarak çalıştırılabilir kod satır sayısını, "SLOC Logical" metriği mantıksal olarak çalıştırılabilir kod satır sayısını belirtmektedir. Her bir metrik altında gözlemlenen "U" (Uncompressed) her bir çerçeveye ait kütüphanelerin sıkıştırılmamış sürümüne ait kod satır sayısını, "C" (Compressed) ise sıkıştırılmış sürümüne ait kod satır sayısını göstermektedir.

Gerçek hayatta, üretim (production) ortamında çalışan bir web uygulamasında genellikle kütüphanelerin sıkıştırılmış sürümleri tercih edilmektedir. Böylece uygulama boyutu azaltılmakta, ağ taraflı ve istemci taraflı maliyet faktörlerinin minimize edilmesi hedeflenmektedir. Araştırma kapsamında, sonuçların gerçeğe en yakın değerleri verebilmesini sağlayabilmek için, ağ taraflı performans faktörleri devreden çıkarılmasına rağmen, eşitliği sağlamak adına kullanılan tüm çerçevelerin ve çerçeve kullanmadan geliştirilen uygulamaya ait kodların sıkıştırılmış (compressed) sürümleri kullanılmaktadır.

3.3.1. Angularjs

Angularjs, Google tarafından 2010 yılında geliştirilmiş, istemci taraflı bir Javascript uygulama çerçevesidir (Ramos, Valente and Terra, 2018). Angularjs ile geliştirilen istemci taraflı bir web uygulamasında, HTML etiketlerine çeşitli eklenti özellikler ekleyerek çalışmaktadır (AngularJS, tarih yok).

Şekil 3.8'de örnek bir Angularjs uygulamasına dair ön yüz kodları sunulmaktadır. Angularjs uygulamasının çalışacağı HTML elementi üzerinde "ng-app" direktifi yer almaktadır. Bu eleman içerisindeki bir alt elemanda da (div), denetleyici direktifi "ng-controller" yer almaktadır. Bu HTML elemanı içerisinde de "{{message}}" adında bir şablon yer tutucusu tanımlanmaktadır.

```
<div class="Main" ng-app="WebApp" ng-controller="WebController">
    <table>
        <tr>
            <td>
                <div class="ProductListContainer">
                    <div class="ProductList">
                        <div class="Product" ng-repeat="product in products">
                            <div class="ProductImageContainer">
                                <img class="ProductImage" src="../../_data/img/{{product.picture}}" />
                            </div>
                            <div class="ProductTitle">{{product.title}}</div>
                            <div class="ProductPriceContainer">
                                <div class="ProductPriceDiscountedPrice">
                                    {{(product.price - (product.price * product.discount / 100)).toFixed(2)}}
                                </div>
                                <div class="ProductPriceDiscount">%{{product.discount}}</div>
                                <div class="ProductPrice">{{product.price}}</div>
                            </div>
                            <div class="ProductInlineToolbar"></div>
                        </div>
                    </div>
                </div>
            </td>
        </tr>
    </table>
</div>
```

Şekil 3.8. Angularjs uygulaması HTML kodu

Oluşturulan bu nesneye bağlı olarak, çeşitli denetleyiciler (controller) oluşturulmakta ve denetleyicilerin gerçekleştirdikleri işlemler bu modül nesnesi üzerinden ilerlemektedir. Şekil 3.9'de verilen denetleyici kodu içerisinde, bu senaryoda kullanılan örnek uygulamada, şablondaki yer tutucular değerleriyle değiştirilerek ekrana gösterilmektedir.

```
1   var app = angular.module("WebApp", []);
2
3   app.controller("WebController", function ($scope) {
4           $scope.products = products;
5   });
```

Şekil 3.9. Angularjs javascript kodu

"$scope" değişkeni içerisindeki ilişkili model verisi olan "message" değişkeni, her değiştiğinde, uygulamada görüntülenen değişkene ait veri de görünüm katmanında güncellenmektedir. Angularjs yaklaşımında veri HTML direktifi içerisinde ng-repeat direktifi ile model ve view katmanı arasında çift yönlü olarak bağlanmaktadır.

3.3.2. Reactjs

Reactjs (React, 2019), 2013 yılında yayınlanan, istemci taraflı ve mvc mimarisinde görünüm katmanını odaklayan Javascript uygulama geliştirme çerçevesidir (Fedosejev, 2015). Reactjs ile iki farklı türde uygulama geliştirilebilmektedir. React tarafından kullanılan JSX formatı ile oluşturulan kod TypeScript söz dizimi kullanılarak geliştirildikten sonra Javascript'e çevrilerek kullanılmakla birlikte (Vipul and Sonpatki, 2016), doğrudan sadece Javascript tabanlı olarak da geliştirilebilmektedir. Reactjs ile JSX formatı kullanılarak geliştirilen kodlar, önişleyici (preprocessor) yardımıyla Javascript koduna dönüştürülerek çalıştırılmaktadır.

Reactjs kullanılarak geliştirilen uygulamanın HTML bölümüne dair kodlar Şekil 3.10'da verilmektedir. Angularjs çerçevesindeki yaklaşımın tersine Reactjs ile geliştirilen uygulamada herhangi bir HTML eklentisi kullanılmamaktadır.

```
<body>
    <div id="reactApp">
        <div class="wrapper">
            <div class="TopHeader">
                <table class="WebsiteTitle">
                    <tr>
                        <td>
                            <div class="TopHeader">
                                <img class="Logo" src="../../_data/img/2.png" />Test Uygulaması
                                <b>React</b>
                            </div>
                        </td>
                    </tr>
                </table>
            </div>
            <div class="Main">
                <table>
                    <tr>
                        <td>
                            <div class="ProductListContainer">
                                <div class="ProductList" id="ProductListContainer">
                                </div>
                            </div>
                        </td>
                    </tr>
                </table>
            </div>
        </div>
    </div>
</body>
```

Şekil 3.10. Reactjs uygulaması HTML kodu

Tez çalışması kapsamında, çerçevelerin web uygulama performansına etkilerini eşit şekilde test edebilmek için, Reactjs ile uygulama geliştirirken, sadece istemci taraflı React kütüphaneleri kullanılmış, Reactjs ile gelen JSX teknolojisi bu nedenlerle kullanılmamıştır, JSX tarafından üretilen Javascript kodu doğrudan uygulamada kullanılmıştır.

```
const ProductListConstructor = function(ProductList) {
    return (
      React.createElement("div", { className: "ProductList"},
        ProductList.map(function (product) {
          return React.createElement("div",{ className: "Product"},
            React.createElement("div",{ className: "ProductImageContainer" },
            React.createElement("img", { className: "ProductImage", src: "../../_data/img/" + product.picture })),
            React.createElement("div",{ className: "ProductTitle" },product.title
            ),
            React.createElement("div",{className: "ProductPriceContainer" },
            React.createElement("div",{ className: "ProductPriceDiscountedPrice" },
            (product.price - (product.price * product.discount / 100)).toFixed(2)),
              React.createElement("div",{ className: "ProductPriceDiscount" },"%",product.discount),
              React.createElement("div",{ className: "ProductPrice" },product.price)
            ),
            React.createElement("div", {className: "ProductInlineToolbar" })
          );
        })
      )
  );
}
ReactDOM.render(ProductListConstructor(products), document.getElementById('ProductListContainer'))

```

Şekil 3.11. Reactjs javascript kodu

Reactjs çerçevesiyle geliştirilen uygulama kodları, Şekil 3.11'da verilmektedir. Reactjs ile diğer çerçevelerden farklı olarak, bir ana uygulama sınıfı oluşturulmamaktadır. Uygulama davranışı, doğrudan çerçeve üzerinde tanımlanan React nesnesi üzerinden oluşturulan sınıf, nesne ve fonksiyon üzerinden kontrol edilmektedir. Reactjs yaklaşımında veri Javascript içerisinde yazdırılarak, model ve view katmanı arasında tek yönlü olarak bağlanmaktadır.

3.3.3. Vuejs

Vuejs, 2014 yılında geliştirilmiş bir javascript çerçevesidir (You, 2018). Vuejs çerçevesi kullanılarak geliştirilen bir uygulamada, ilgili HTML elemanının "id" özelliğine ait değerin çerçevenin kurucu fonksiyonuna gönderilmesiyle uygulama oluşturulur. Vuejs uygulaması da Angularjs uygulaması gibi oluşturulan ana uygulama nesnesi özelliğindeki HTML elemanı içerisinde çalışmaktadır.

```
10  <body>
11      <div id="vueApp">
12          <div class="wrapper">
13              <div class="TopHeader">
14                  <table class="WebsiteTitle">
15                      <tr>
16                          <td>
17                              <div class="TopHeader">
18                                  Test Uygulaması<b>VueJS</b>
19                              </div>
20                          </td>
21                      </tr>
22                  </table>
23              </div>
24              <div class="Main">
25                  <table>
26                      <tr>
27                          <td>
28                              <div class="ProductList">
29                                  <div class="Product" v-for="product in products">
30                                      <div class="ProductImageContainer">
31                                          <img class="ProductImage" v-bind:src="'../../_data/img/' + product.picture" />
32                                      </div>
33                                      <div class="ProductTitle">{{product.title}}</div>
34                                      <div class="ProductPriceContainer">
35                                          <div class="ProductPriceDiscountedPrice">
36                                              {{(product.price - (product.price * product.discount / 100)).toFixed(2)}}
37                                          </div>
38                                          <div class="ProductPriceDiscount">%{{product.discount}}</div>
39                                          <div class="ProductPrice">{{product.price}}</div>
40                                      </div>
41                                      <div class="ProductInlineToolbar"></div>
42                                  </div>
43                              </div>
44                          </td>
45                      </tr>
46                  </table>
47              </div>
48          </div>
49  </body>
```

Şekil 3.12. Vuejs uygulaması HTML kodu

Vuejs ile geliştirilen uygulamaya dair HTML kodları Şekil 3.12'de verilmektedir. Vuejs kullanılarak bir uygulama oluşturulurken, çerçeve kütüphanesi içerisindeki "Vue"

fonksiyonu üzerinden, “id” özelliğinin “container” değerine sahip HTML elemanı üzerinde, bir uygulama nesnesi oluşturulmaktadır. Örnek kodda görülen Vuejs yaklaşımında veri HTML direktifi kullanılarak model ve view katmanı arasında tek yönlü olarak bağlanmaktadır.

```
1  var vueApp=  new Vue({
2
3    el: '#vueApp',
4
5    data: {
6        products: products
7    }
8  });
```

Şekil 3.13. Vuejs javascript kodu

Vuejs ile geliştirilen istemci uygulamaya dair Javascript kodları Şekil 3.13’de verilmektedir. Angularjs’deki yaklaşıma oldukça benzerlik göstermektedir. Javascript bölümü içerisinde, “Vue()” fonksiyonu dönüşünde oluşturulan nesne “vueApp” isimli bir değişkene atanmaktadır. Angularjs çerçevesindeki yaklaşıma benzer olarak burada da bir ana uygulama nesnesi içerisinde, products değişkeni bulunmaktadır. Bu değişkene bağlanan veri, uygulamanın çalışması sırasında bağlanmakta ve dinamik olarak ekranda gösterilmektedir. Vuejs çerçevesindeki yaklaşımın Angularjs’den temel farkı, Gerçek DOM yerine Sanal DOM üzerinde oluşturma yöntemini gerçekleştirmesidir.

3.3.4. Çerçevesiz uygulama

Tez çalışması kapsamında, çerçeve performanslarının yanı sıra, çerçeve kullanmadan geliştirilen uygulamanın yükleme süreleri de hesaplanmıştır. Çerçeve kullanmadan geliştirilen uygulamada, herhangi bir yardımcı Javascript Çerçevesi ve/veya kütüphane kullanılmamış, doğrudan tarayıcı javascript çalışma zamanı üzerinde bulunan fonksiyonlar kullanılarak uygulama tekrar geliştirilmiştir. Böylece çalışma kapsamında, çerçeve kullanılarak geliştirilen uygulamalar ile çerçevesiz kullanımın, web uygulama yükleme performansına etkisi karşılaştırılmaktadır.

Şekil 3.14’de çerçevesiz uygulamaya ait HTML kodları verilmektedir. Çerçevesiz uygulamanın HTML kodlarında herhangi bir direktif ya da eklenti bulunmamaktadır.

```
<body>
    <div class="wrapper">
        <div class="TopHeader">
            <table class="WebsiteTitle">
                <tr>
                    <td>
                        <div class="TopHeader">
                            <img class="Logo" src="../../_data/img/2.png" />Test Uygulaması
                            <b>Çerçevesiz</b>
                        </div>
                    </td>
                </tr>
            </table>
        </div>
        <div class="Main">
            <table>
                <tr>
                    <td>
                        <div class="ProductListContainer">
                            <div class="ProductList" id="ProductList">
                            </div>
                        </div>
                    </td>
                </tr>
            </table>
        </div>
    </div>
</body>

```

Şekil 3.14. Çerçevesiz uygulama HTML kodu

Çerçeve kullanmadan geliştirilen istemci uygulamaya ait Javascript kodları Şekil 3.15'de verilmektedir. Çerçevesiz kullanımda, HTML elemanları, Javascript kodu içerisinde oluşturularak, döngü içerisinde dokümana yerleştirilmektedir. Veri bağlama işlemi de bu döngü içerisinde sağlanmaktadır.

```
var html = "";
for (i = 0; i < products.length; i++) {
    var product = products[i];
    html += "<div class='Product' id='" + i + "'>" +
        "<div class='ProductImageContainer'>" +
        "<img class='ProductImage' src='../../_data/img/" + product.picture + "' />" +
        "</div>" +
        "<div class='ProductTitle'>" + product.title + "</div>" +
        "<div class='ProductPriceContainer'>" +
        "<div class='ProductPriceDiscountedPrice'>"
        + (product.price - (product.price * product.discount / 100)).toFixed(2) + "  </div>" +
        "<div class='ProductPriceDiscount'> %" + product.discount + "  </div>" +
        "<div class='ProductPrice'>" + product.price + " </div>" +
        "</div>" +
        "<div class='ProductInlineToolbar'>" + "</div>" +
        "</div>";
}
document.getElementById("ProductList").innerHTML = html;

```

Şekil 3.15. Çerçevesiz uygulama javascript kodu

Çerçevesiz kullanımdaki yaklaşımın Reactjs'den temel farkı, Sanal DOM ağacı yerine doğrudan Gerçek DOM ağacı üzerinde işlem gerçekleştirmesidir. Çerçevesiz kullanımda veri doğrudan DOM ağacına bağlanmaktadır.

3.4. Çerçevelere Ait Yaklaşımlar

Araştırma kapsamında test edilen çerçeveler, belirli yönleriyle ve yaklaşımlarıyla birbirlerinden farklılık göstermektedirler. Çizelge 3.5 üzerinde çerçevelerin DOM Oluşturma/Güncelleme, Veri Bağlama ve Şablonlama yaklaşımları gösterilmektedir.

Çizelge 3.5. Çerçevelere ait yaklaşımlar

Çerçeve	DOM Oluşturma	Şablonlama
Angularjs	Gerçek	HTML Şablonu
Reactjs	Sanal	Javascript
Vuejs	Sanal	HTML Şablonu
Çerçevesiz	Gerçek	Javascript

Angularjs çerçevesi veriyi bağlama ve veriyi güncelleme sırasında gerçek DOM üzerinde işlem gerçekleştirmektedir. Başka bir deyişle oluşturma ve bağlama zamanında veride olabilecek muhtemel değişiklikler tüm veri ağacının güncellenmesi anlamına gelmektedir. Reactjs çerçevesi ise Vuejs'ye benzer şekilde DOM oluşturma sırasında Sanal DOM Oluşturma yaklaşımını kullanmaktadır.

Yine çerçevelerin ve çerçevesiz kullanımın ortaya koyduğu farklılıklar veri bağlama yaklaşımında da kendisini göstermektedir (Xing, Huang and Lai, 2019). AngularJS çift yönlü veri bağlama yaklaşımı kullanmaktadır. Vuejs ile hem tek hem de çift yönlü veri bağlama gerçekleştirilebilmektedir. Bununla birlikte Reactjs çerçevesi ve tez kapsamında geliştirilen çerçevesiz uygulama kodları tek yönlü veri bağlama yaklaşımı kullanmaktadır.

Angularjs çerçevesinde HTML şablonu yöntemi kullanılmaktadır. HTML çeşitli direktifler ile genişletilerek uygulama yönetilmektedir. Oluşturma (Render) işlemi Angularjs Controller üzerinden sağlanmaktadır. Reactjs çerçevesinde ise doğrudan Javascript üzerinde HTML kodları oluşturularak sayfaya bu HTML kodları yerleştirilmekte ve DOM ağacına bağlanmaktadır. Vuejs çerçevesinde Angularjs'ye benzer bir yapı kullanılmaktadır. HTML eklentileri ile direktifler aracılığıyla uygulama bu direktifler üzerinden yönetilmektedir.

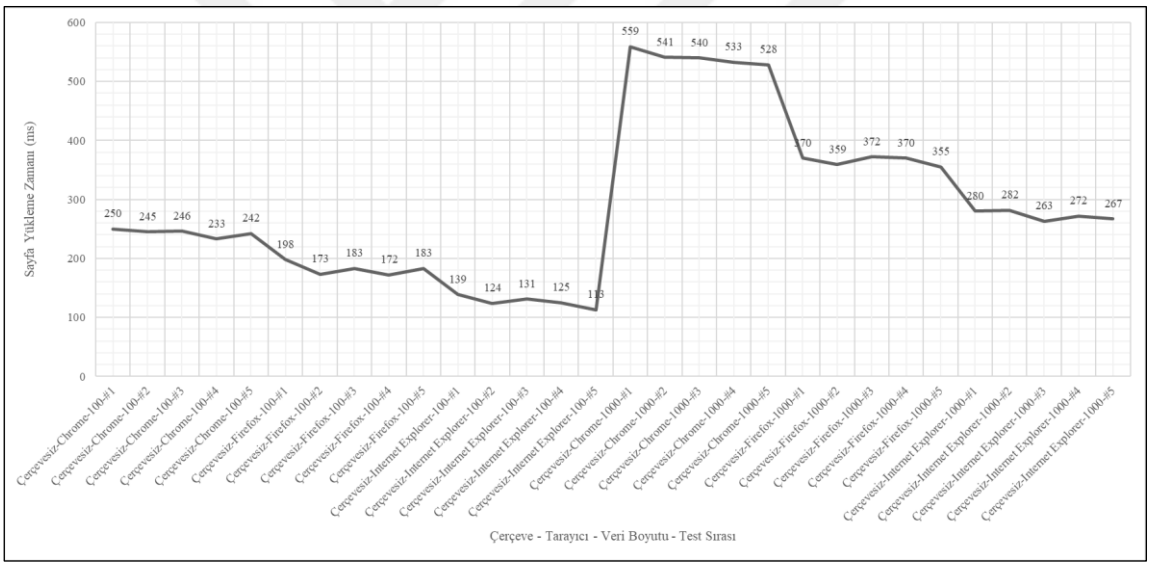
Çerçevesiz kullanımda ise HTML, Reactjs çerçevesine benzer şekilde doğrudan javascript kodu içerisinde oluşturularak, dokümana yerleştirilmektedir.

4. BULGULAR

Bu bölümde Javascript çerçeveleri tarafından ortaya konulan yaklaşımların, web sayfası yükleme süresine etkisinin araştırılması için gerçekleştirilen test ve deneyler sonucunda ortaya çıkan bulgular ortaya konulmakta ve değerlendirilmektedir.

4.1. Çerçeve Kullanımının Sayfa Yükleme Zamanına Etkisine İlişkin Bulgular

Çerçeve kullanımının sayfa yükleme performansını gözlemleyebilmek için ilk olarak herhangi bir çerçeve kullanmadan uygulama sürümü oluşturulmuş, değişken veri boyutları ve farklı tarayıcılar üzerinde test edilmiştir. Şekil 4.1 üzerinde çerçevesiz uygulamaya ait yükleme zamanları görülmektedir.



Şekil 4.1. Çerçeve kullanılmadan geliştirilen uygulamaya ait sayfa yükleme süreleri

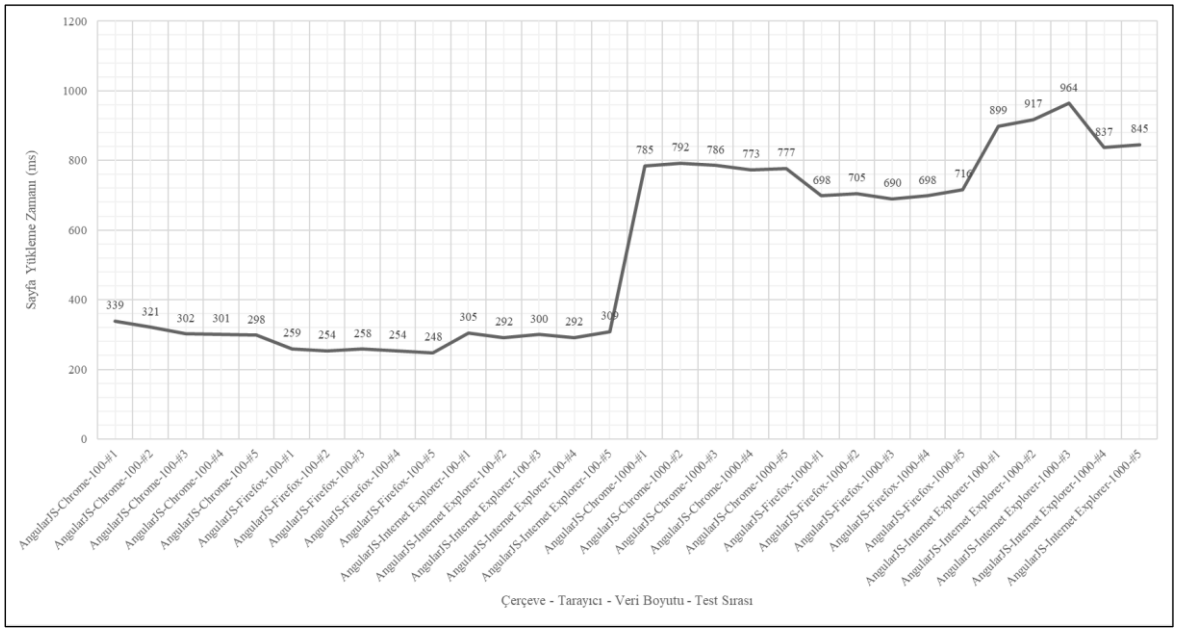
Çerçeve kullanılmadan geliştirilen uygulamaya ait sayfa yükleme zamanları incelendiğinde 100 nesnelik veri boyutu için tarayıcılarda sırasıyla Chrome: 233-250 ms, Firefox: 172-198 ms, Internet Explorer: 113-139 ms aralıklarında süreler gözlemlenmektedir. Çerçeve kullanılmadan geliştirilen uygulamadaki sayfa yükleme süreleri bu durumda tarayıcı bazında 17-27 ms arasında sapma göstermektedir. Bununla birlikte 1000 nesnelik veri boyutu için tarayıcı bazında yükleme süreleri sırasıyla Chrome: 528-559 ms, Firefox: 355-371 ms, Internet Explorer: 263-280 ms değerleri aralığında sayfa yükleme zamanları elde

edilmektedir. Tarayıcı bazında sayfa yükleme süreleri sapması 1000 nesnelik veri boyutunda, 16-31 ms arasında gözlemlenmektedir.

Çerçeve kullanmadan geliştirilen örnek web uygulaması üzerinde gerçekleştirilen testlerden aşağıdaki bulgular ortaya çıkmaktadır:

- Tarayıcı bazında veri boyutu 100'den 1000'e çıkarıldığında ortalama sayfa yükleme süresi sapması minimum 1 ms, maksimum 4 ms değişkenlik göstermektedir. (100 nesne: 17-27 ms, 1000 nesne: 16-31 ms).
- Çerçeve kullanılmadan geliştirilen web uygulaması genel olarak her veri boyutunda ve her tarayıcı üzerinde istikrarlı performans göstermektedir. Yükleme süreleri arasında ciddi farklar bulunmamaktadır.
- Veri boyutu 10 katına çıkarıldığında yükleme süresi tarayıcı bazında ortalama iki katına çıkmaktadır.

Angularjs çerçevesi kullanılarak geliştirilen uygulamaya ait sayfa yükleme zamanları Şekil 4.2 üzerinde görülmektedir.



Şekil 4.2. Angularjs ile geliştirilen uygulamaya ait sayfa yükleme süreleri

Uygulamaya ait sayfa yükleme süreleri 100 nesnelik veri boyutu için tarayıcılarda sırasıyla Chrome: 298-339 ms, Firefox: 248-259 ms, Internet Explorer: 292-309 ms aralıklarında değer almaktadır. Sayfa yükleme süreleri bu durumda tarayıcı bazında 11-41 ms arasında

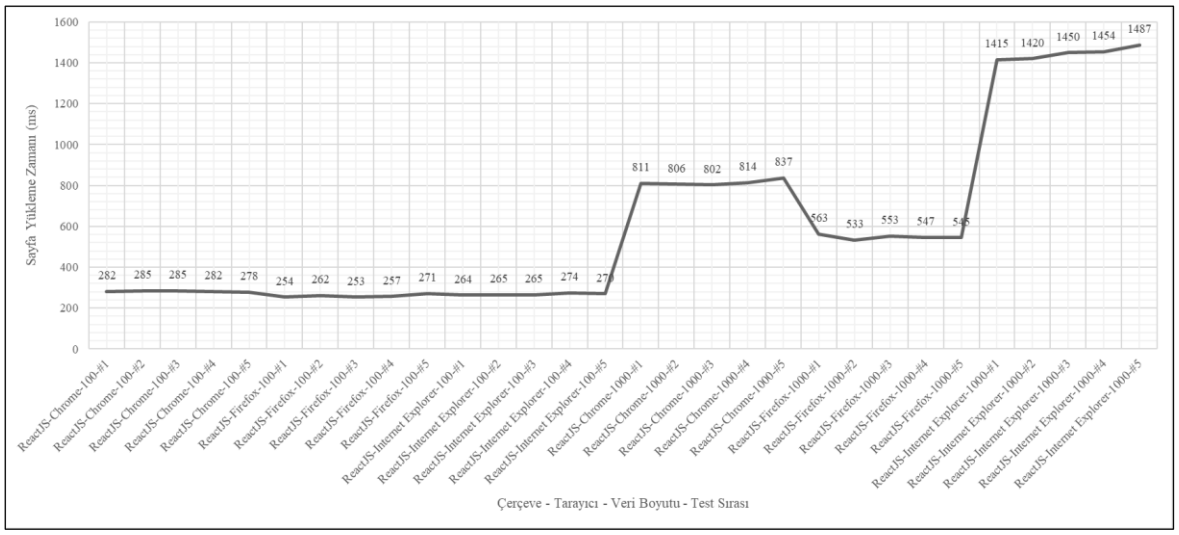
sapma göstermektedir. Bununla birlikte 1000 nesnelik veri boyutu için tarayıcı bazında yükleme süreleri sırasıyla Chrome: 773-792 ms, Firefox: 690-716 ms, Internet Explorer:837 ile 964 ms değerleri aralığında gözlemlenmektedir. Tarayıcı bazında sayfa yükleme süreleri 1000 nesnelik veri boyutunda, 19-127 ms arasında değişkenlik göstermektedir.

Angularjs çerçevesi ile geliştirilen web uygulaması ile gerçekleştirilen testlerde aşağıdaki bulgular ortaya çıkmaktadır:

- Tarayıcı bazında veri boyutu 100'den 1000'e çıkarıldığında ortalama sayfa yükleme süresi sapması minimum 8 ms, maksimum 86 ms değişkenlik göstermektedir. (100 nesne: 11-41 ms, 1000 nesne: 19-127 ms).
- Çerçevesiz uygulamaya benzer şekilde, veri boyutu 10 katına çıkmasına rağmen yükleme süresi tarayıcı bazında yaklaşık iki katına çıkmaktadır.
- Çerçeve kullanmadan geliştirilen uygulamaya göre kıyaslandığında, Angularjs kullanılarak geliştirilen uygulamanın sayfa yükleme sürelerinde artış gösterdiği gözlemlenmektedir.
- Çerçevesiz uygulamaya kıyasla Angularjs çerçevesinin kullanımı 100 nesnelik veri boyutunda ortalama 68,5 ms – 174 ms arasında yükleme süresi farkı oluşturmaktadır. 1000 nesnelik veri boyutunda ise 239 ms ile 628 ms fark oluşmaktadır.

Angular js ile geliştirilen uygulama genel olarak belirli yükleme zamanları aralığında, istikrarlı performans gösterdiği gözlemlenmektedir. Yükleme süreleri arasında kullanıcı tarafından fark edilebilecek ciddi zamanlar bulunmamaktadır.

Reactjs çerçevesi kullanılarak geliştirilen uygulamaya ait sayfa yükleme zamanları Şekil 4.3 üzerinde görülmektedir.



Şekil 4.3. Reactjs ile geliştirilen uygulamaya ait sayfa yükleme süreleri

Uygulamaya ait sayfa yükleme süreleri 100 nesnelik veri boyutu için tarayıcılarda sırasıyla Chrome: 278-285 ms, Firefox: 253-271 ms, Internet Explorer: 264-274 ms aralıklarında gözlemlenmektedir. Sayfa yükleme süreleri bu durumda tarayıcı bazında 7 ile 18 ms arasında sapma göstermektedir.

Bununla birlikte 1000 nesnelik veri boyutu için tarayıcı bazında yükleme süreleri sırasıyla Chrome: 802-837 ms, Firefox: 533-563 ms, Internet Explorer: 1415-1487 ms değerleri aralığında gözlemlenmektedir.

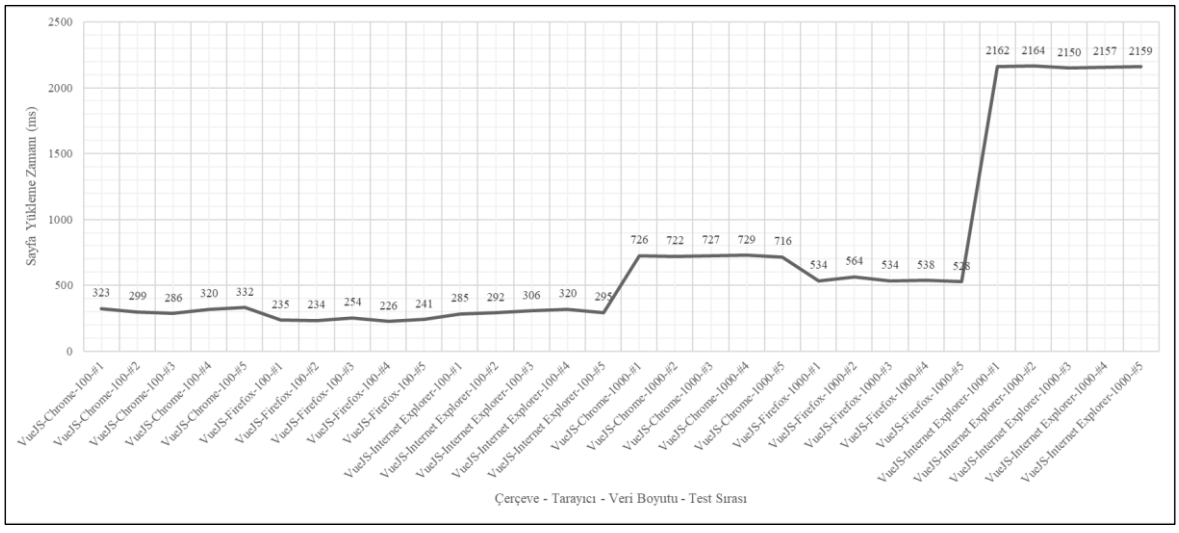
Sayfa yükleme süreleri 1000 nesnelik veri boyutunda, 30 ile 72 ms arasında değişkenlik göstermektedir. Reactjs çerçevesi ile geliştirilen web uygulaması ile gerçekleştirilen testlerde aşağıdaki bulgular ortaya çıkmaktadır:

- Tarayıcı bazında veri boyutu 100'den 1000'e çıkarıldığında ortalama sayfa yükleme süresi sapması minimum 12 ms, maksimum 51 ms değişkenlik göstermektedir. (100 nesne: 7-18 ms, 1000 nesne: 30-72 ms).
- Çerçevesiz uygulamaya benzer şekilde, veri boyutu 10 katına çıkmasına rağmen yükleme süresi tarayıcı bazında yaklaşık iki katına çıkmaktadır.
- Çerçeve kullanmadan geliştirilen uygulamaya göre kıyaslandığında, Reactjs kullanılarak geliştirilen uygulamanın sayfa yükleme sürelerinde artış gösterdiği gözlemlenmektedir.

- 1000 nesnelik veri boyutu için Chrome tarayıcısında sayfa yükleme zamanı yaklaşık 3 katına, Firefox üzerinde iki katına, Internet Explorer üzerinde ise 7 katına çıkmaktadır.
- Uygulama genel olarak 100 nesnelik veri boyutunda istikrarlı performans göstermektedir. Ancak 1000 nesnelik veri boyutuna çıkıldığında, tarayıcı kaynaklı istikrarsız yükleme süreleri ortaya çıkmaktadır. Buna rağmen uygulama yükleme süresi uzasa da yükleme sürelerindeki sapmalar göz önüne alındığında, çerçevenin öngörülebilir performans gösterdiği bulgusuna ulaşılmaktadır.
- Çerçevesiz uygulamaya kıyasla Angularjs çerçevesinin kullanımı 100 nesnelik veri boyutunda ortalama 68,5 ms – 174 ms arasında yükleme süresi farkı oluşturmaktadır. Bununla birlikte 1000 nesnelik veri boyutunda ise 239 ms ile 628 ms fark oluşmaktadır.

Reactjs ile geliştirilen uygulamanın genel olarak 100 nesnelik veri boyutunda belirli yükleme zamanları aralığında, çerçevesiz uygulamaya çok benzer şekilde tarayıcıdan bağımsız olarak öngörülebilir ve istikrarlı performans gösterdiği gözlemlenmektedir. Yükleme süreleri arasında kullanıcı tarafından fark edilebilecek ciddi süre farkları bulunmamakta ancak yüksek veri boyutlarında tarayıcı kaynaklı olarak yükleme süreleri farkları oluşmaktadır. Bununla birlikte yüksek veri boyutlarında yükleme süreleri her bir tarayıcı özelinde farklılık gösterdiği bulgusuna ulaşılmaktadır.

Vuejs çerçevesi kullanılarak geliştirilen uygulamaya ait sayfa yükleme zamanları Şekil 4.4 üzerinde görülmektedir.

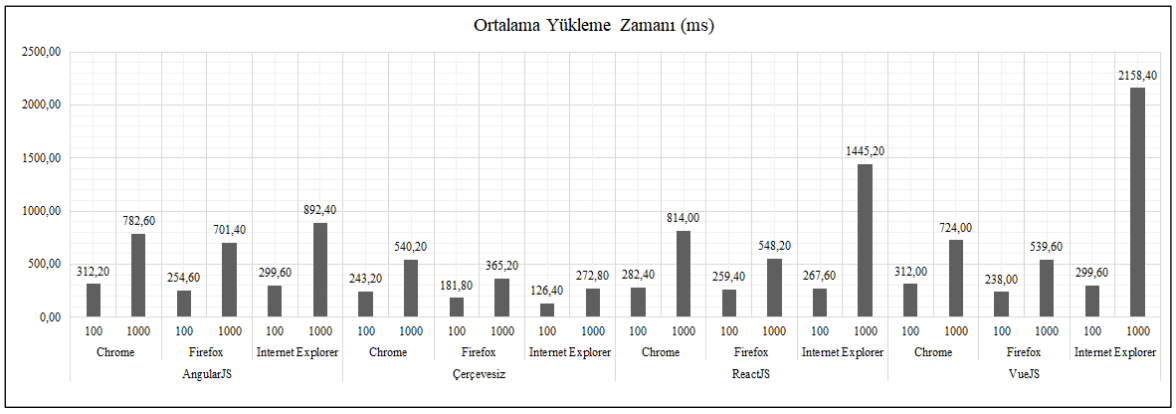


Şekil 4.4. Vuejs ile geliştirilen uygulamaya ait sayfa yükleme süreleri

Uygulamaya ait sayfa yükleme süreleri 100 nesnelik veri boyutu için tarayıcılarda sırasıyla Chrome: 286-332 ms, Firefox: 226-254 ms, Internet Explorer: 285-320 ms aralıklarında gözlemlenmektedir. Sayfa yükleme süreleri bu durumda tarayıcı bazında 28 ile 46 ms arasında sapma göstermektedir. 1000 nesnelik veri boyutu için tarayıcı bazında yükleme süreleri sırasıyla Chrome: 716-729 ms, Firefox: 528-564 ms, Internet Explorer: 2150-2164 ms değerleri aralığında gözlemlenmektedir. Sayfa yükleme süreleri 1000 nesnelik veri boyutunda, 13 ile 36 ms arasında değişkenlik göstermektedir. Reactjs çerçevesi ile geliştirilen web uygulaması ile gerçekleştirilen testlerde aşağıdaki bulgular ortaya çıkmaktadır:

- Sayfa yükleme süreleri 1000 nesnelik veri boyutu için Chrome ve Firefox üzerinde sayfa yükleme zamanı 100 nesnelik veri boyutuna göre yaklaşık 2 katına, Internet Explorer'da ise 9 katına çıkmaktadır.
- Uygulama genel olarak göre 100 nesnelik veri boyutunda istikrarlı performans göstermektedir. Ancak 1000 nesnelik veri boyutuna çıkıldığında, Chrome ve Freifox tarayıcıları üzerinde birbirine yakın performans değerleri elde edilse de Internet Explorer tarayıcısı üzerinde yükleme süreleri diğer tarayıcılara göre 3 katına çıkmaktadır.
- Çerçevesiz uygulamaya kıyasla Vuejs çerçevesinin kullanımı 100 nesnelik veri boyutunda ortalama 56,20 ms – 173,20 ms arasında yükleme süresi farkı oluşturmaktadır. Bununla birlikte 1000 nesnelik veri boyutunda ise 174,40 ms ile 1885,60 ms fark oluşmaktadır.
- Uygulama genel olarak 100 nesnelik veri boyutunda istikrarlı performans göstermektedir. Ancak 1000 nesnelik veir boyutuna çıkıldığında, IE üzerinde istikrarsız yükleme süreleri ortaya çıkmaktadır. Bu da söz konusu çerçevenin tarayıcı kaynaklı olarak performansının, yüksek veri boyutlarında öngörülemez olduğu bulgusunu elde etmemizi sağlamaktadır.

Tüm uygulama sürümlerine ait veri boyutu ve tarayıcı bazında ortalama sayfa yükleme süreleri Şekil 4.5 üzerinde gösterilmektedir.



Şekil 4.5. Çerçeve, veri boyutu ve tarayıcı bazında uygulama sürümlerine ait ortalama sayfa yükleme süreleri

Çerçeve bazında ortalama yükleme zamanları incelendiğinde, 100 nesnelik veri boyutu üzerinde gerçekleştirilen testlerde, çerçeve kullanımının sayfa yükleme performansına etkisinin tarayıcıdan bağımsız olarak daha az olduğu ve daha küçük aralıklarda değişkenlik gösterdiği gözlemlenmektedir. Bununla birlikte, veri boyutu değiştikçe çerçeve ve tarayıcı bazında yükleme süreleri arasında farklar tarayıcı ve kullanılan çerçeveye bağlı olarak ortaya çıkmaktadır. Çerçeve kullanmadan geliştirilen uygulamada veri boyutunun değişkenliği, sayfa yükleme süresini daha az etkilerken, çerçeve kullanılarak geliştirilen uygulamalarda, Chrome ve Firefox tarayıcısı üzerinde 1000 nesnelik veri boyutuna çıkıldığında, ortalama yükleme süreleri 100 nesnelik veri boyutuna göre 2 ile 3 kata kadar artmaktadır. Vuejs ve Reactjs çerçevelerinde 1000 nesnelik veri boyutunda yükleme sürelerinin 100 nesnelik veri boyutuna göre Internet Explorer üzerinde 7-8 katına kadar çıktığı ve ciddi yükleme süreleri farkı oluşturduğu gözlemlenmektedir.

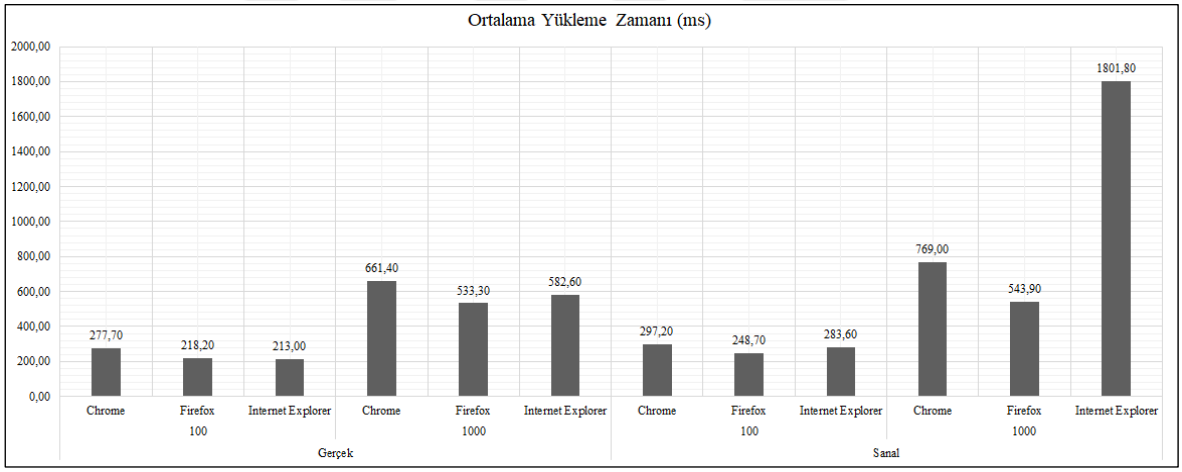
Çerçeve kullanımının sayfa yükleme performansına etkisi düşünüldüğünde, tarayıcı ve veri boyutunun değişimine göre farklılıklar oluştuğu gözlemlenmektedir. Çerçeve kullanılarak geliştirilen uygulamalar incelendiğinde, özellikle Reactjs ve Vuejs ile geliştirilen uygulamaların veri boyutu değiştiğinde Internet Explorer tarayıcısı üzerinde daha çok sapma gösterdiği ortaya çıkmaktadır.

Çerçeve kullanımının sayfa yükleme süresine etkisi incelendiğinde, çerçeve kullanımının her ne kadar yazılımın sürdürülebilirliği ve geliştirilebilirliği üzerine olumlu etkileri olsada, bu çalışma kapsamında gerçekleştirilen testler sonucunda, sayfa yükleme zamanları referans alındığında, performans açısından öngörülemezliğe yol açtığı bulgusuna ulaşılmaktadır.

4.2. DOM Oluşturma Yaklaşımının Sayfa Yükleme Zamanına Etkisine İlişkin Bulgular

Bu bölümde çerçeveden bağımsız olarak, gerçek ve sanal dom oluşturma yöntemlerinin sayfa yükleme zamanlarına etkisi farklı veri boyutları ve farklı tarayıcılar üzerinde değerlendirilmektedir.

Veri boyutu 100'den 1000 nesne boyutuna çıkarıldığında, gerçek dom oluşturma yönteminde yükleme süreleri arasındaki artışlar sırasıyla Chrome: 383 ms, Firefox: 315,10 ms, Internet Explorer: 369,6 ms olarak gözlemlenmektedir. Elde edilen test sonuçları doğrultusunda gerçek dom oluşturma yönteminin ortalama 355,85 ms yükleme süresine sahip olduğu gözlemlenmektedir. DOM oluşturma yöntemine göre (Sanal/Gerçek) ortalama sayfa yükleme zamanları Şekil 4.6 üzerinde verilmektedir.



Şekil 4.6. DOM oluşturma yaklaşımlarının, veri boyutu ve tarayıcı bazında ortalama sayfa yükleme süreleri

Sanal dom oluşturma yöntemi incelendiğinde, veri boyutu artırıldığında sayfa yükleme sürelerinde tarayıcı bazında sırasıyla Chrome: 471,80 ms, Firefox: 295,20, Internet Explorer: 1219,20 ms olarak ölçülmektedir. Gerçek dom oluşturma yöntemiyle kıyaslandığında firefox tarayıcısı üzerinde sanal dom oluşturma yöntemi gerçek dom oluşturma yöntemine göre sayfa yükleme zamanıda en az, Internet Explorer tarayıcısı üzerinde ise en çok sapmayı gösterdiği bulgusuna ulaşılmaktadır.

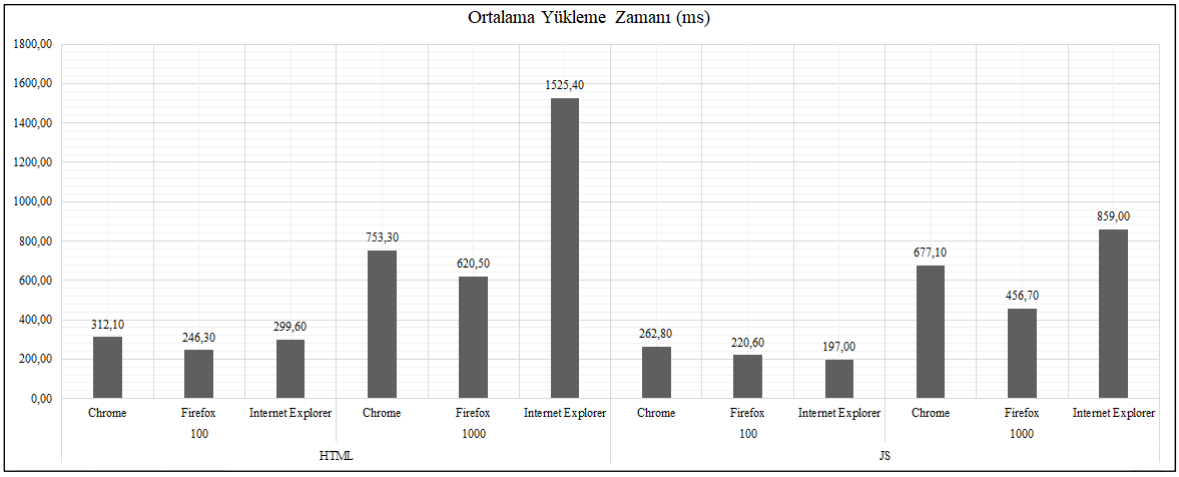
Gerçek ve Sanal DOM oluşturma yöntemleri arasında, genel ortalamalar referans alındığında sayfa yükleme süreleri arasında 100 nesnelik veri boyutu üzerinde çok fazla fark

bulunmamakla birlikte, veri boyutu arttıkça, yükleme sürelerindeki artış tarayıcılarda farklı şekilde gözlemlenmektedir. Bu yönüyle Gerçek DOM oluşturma yönteminin büyük veri boyutlarında, sayfa yükleme performansının öngörülebilir olması için sanal dom oluşturma yöntemi yerine tercih edilmesi gerektiği bulgusuna ulaşılmaktadır.

4.3. Şablonlama Yaklaşımının Sayfa Yükleme Zamanına Etkisine İlişkin Bulgular

Bu bölümde çerçeveden bağımsız olarak, şablonlama yöntemlerinin sayfa yükleme zamanlarına etkisi farklı veri boyutları ve farklı tarayıcılar üzerinde değerlendirilmektedir.

Çerçeveden bağımsız olarak şablonlama yöntemine göre (HTML/JS) ortalama sayfa yükleme zamanları Şekil 4.7 üzerinde verilmektedir.



Şekil 4.7. Şablonlama yaklaşımlarının, veri boyutu ve tarayıcı bazında ortalama sayfa yükleme süreleri

HTML şablon yöntemi incelendiğinde veri boyutunun 100'den 1000 nesne boyutuna çıkarıldığı durumda sayfa yükleme sürelerinde tarayıcı bazında sırasıyla Chrome: 441,20 ms, Firefox: 374,20, Internet Explorer: 1225,8 ms süre artışı gözlemlenmektedir. Bununla birlikte Javascript şablon yöntemi incelendiğinde sayfa yükleme sürelerinde tarayıcı bazında sırasıyla Chrome: 414,30 ms, Firefox: 236,10, Internet Explorer: 662 ms süre artışı gözlemlenmektedir.

HTML ve Javascript şablonlama yöntemleri arasında, genel ortalamalar referans alındığında sayfa yükleme süreleri arasındaki sayfa yükleme sürelerindeki sapmalar göz önüne

alındığında, Javascript içerisinde HTML metni tanımlanarak ve oluşturulan nihai HTML üzerinden DOM oluşturularak ekranda gösterilmesi tercih edilmesi gerektiği ortaya çıkmaktadır. HTML şablonlama yöntemi kod okunabilirliğini arttırsa da bu yöntemi uygulamak veri boyutu büyüdükçe tarayıcı kaynaklı olarak yükleme sürelerinde sapmaya neden olduğu bulgusuna ulaşılmaktadır.

5. SONUÇLAR

Bu çalışma kapsamında, istemci taraflı web uygulaması geliştirme sürecinde kullanılan mimari yaklaşımların, web sayfası yükleme performansına etkileri araştırılmıştır. Çalışmada farklı Javascript çerçeveleri tarafından ortaya konulan veri bağlama, şablonlama ve dom oluşturma yaklaşımları referans uygulama üzerinde, araştırma kapsamında geliştirilen yardımcı performans ölçüm kütüphanesi ve gezinme zamanlaması programlama arayüzü aracılığıyla farklı tarayıcı platformları üzerinde ölçülmüştür.

Bu çalışmada ortaya çıkan bulguların sonucu olarak, istemci taraflı web uygulamalarında küçük veri boyutlarında ortaya konulan mimari yaklaşımların günümüz donanım gücünde kullanıcı deneyimi düşünüldüğünde ihmal edilebilir sürelerde fark oluşturduğu gözlemlenmektedir. Ancak veri boyutu büyüdükçe ortaya konulan mimari yaklaşımlarda uygulama yükleme süreleri arasında oluşan farkların arttığı gözlemlenmiştir. Ağ hızı arttıkça istemci tarafına aynı anda gönderilen veri boyutunun da zamanla yüksek boyutlara ulaşacağı düşünüldüğünde, istemci tarafında doğru mimari seçim ve yaklaşım önem arz etmektedir.

Bu çalışmaya benzer olarak gerçekleştirilen daha önceki çalışmalarda ortaya çıkan sonuçların karşılaştırması Çizelge 4.1 üzerinde verilmektedir.

Daha önce Davila tarafından gerçekleştirilen çalışmada (Davila, 2015), Angularjs, Backbonejs, Emberjs, Marionettejs and Reactjs çerçeveleri, Chrome ve Firefox tarayıcıları üzerinde, 1000 satırlık veri boyutu ile 5'er kez TodoMVC referans uygulaması (Osmani, 2015) üzerinde karşılaştırılmaktadır. Çalışma sonucunda Reactjs çerçevesinin diğer çerçevelere göre genel olarak daha iyi performans gösterdiği sonucuna ulaşmışlardır. Bu çalışmada Chrome ve Firefox tarayıcıları üzerinde benzer veri boyutlarında paralel sonuçlar elde ederken, Davila tarafından gerçekleştirilen çalışmaya ek olarak kullandığımız Internet Explorer tarayıcısını da göz önüne aldığımızda genel performans ölçütleri düşünüldüğünde, bu çalışmada Reactjs çerçevesi yerine Angularjs çerçevesi ön plana çıkmaktadır.

Çizelge 4.1 Daha önce gerçekleştirilen çalışmalar ile karşılaştırma

	Davila, 2015	**Koetsier, 2016**	**Molin, 2016**	**Tez Çalışması**
Çerçeveler	•Angularjs •Backbonejs •Emberjs •Marionettejs •Reactjs	•Angularjs •Reactjs •EmberJS •Vuejs	•Angularjs •Angular2 •Reactjs	•Angularjs •Reactjs •Vuejs •Çerçevesiz
Tarayıcılar	•Chrome	•Chrome	•Chrome	•Chrome
Veri Boyutu	•1000	•10 •1000	•10, 100, 500, 1000, 2000, 3000, 4000, 5000	•100 •1000
Test Sayısı	5	5	1	5
Kullanılan Araç	Tarayıcı Eklentisi	JavaScript Framework Benchmark.	BenchmarkJS	NavigationTimingAPI
Sonuçlar	Reactjs çerçevesi Firefox ve Chrome üzerinde 1000 nesnelik veri boyutu üzerinde performans açısından öne çıkmaktadır.	Reactjs çerçevesi performans açısından öne çıkmaktadır. Sanal DOM yaklaşımı veri boyutu büyüdükçe ve daha fazla verinin güncellenmesi gerektiğinde kullanılmalıdır.	Reactjs çerçevesi 1000 nesnelik veri boyutu üzerinde Chrome tarayıcısında performans açısından öne çıkmaktadır.	Reactjs çerçevesi performans açısından öne çıkmaktadır. Ancak Internet Explorer tarayıcısı da göz önüne alındığında genel olarak Angularjs çerçevesi ön plana çıkmaktadır. Veri boyutları büyüdüğünde sanal dom yerine gerçek dom yaklaşımı tercih edilmelidir.

Koetsier tarafından gerçekleştirilen çalışmada ise (Koetsier, 2016), Angularjs, Reactjs, EmberJS ve Vuejs çerçeveleri sadece performans açısından değil, Olgunluk (Maturity), Kullanım kolaylığı (ease of use), tarayıcı desteği ve uyumluluk (browser support), tekrar kullanılabilirlik (reusability), test edilebilirlik (testability), routing ve şablonlama (Templating) bakımından karşılaştırılmaktadır. Çalışmada performans testleri kısmı çalışmanın küçük bir bölümünü oluşturmakta, gerçekleştirilen karşılaştırmada, sadece Chrome tarayıcısı üzerinde 10 ve 1000 satırlık veri boyutları üzerinde ilgili çerçevelerle 5'er kez testler gerçekleştirilmektedir. Çalışmada Reactjs çerçevesi performans olarak öne çıkmaktadır. Ancak bununla birlikte sadece tek bir tarayıcı üzerinde (Chrome) testler gerçekleştirilmektedir. Çalışmada, performansın yanı sıra pek çok açıdan karşılaştırmalar yapılması nedeniyle, performans açısından yapılan karşılaştırmalar sadece verilerin eklenmesi, güncellenmesi ve silinmesi konusunda yapılan testlerle sınırlı kalmaktadır. Söz konusu çalışmada Sanal DOM yaklaşımı veri boyutu büyüdükçe ve daha fazla verinin güncellenmesi gerektiğinde tercih edilebilirliği bildirilmektedir, ancak bu çalışmada veri boyutu büyüdükçe sanal dom yaklaşımının gerçek dom yaklaşımına göre daha uzun süre yükleme sürelerine yol açtığını gözlemlenmektedir. Aradaki fark çalışmada sanal dom yaklaşımının sadece reactjs üzerinden değerlendirilmesinden kaynaklanmaktadır, ancak bu çalışmada Vuejs yükleme süreleri de hesaba katılarak sanal dom yaklaşımı çerçeveden bağımsız olarak değerlendirilerek gerçekleştirilmektedir. Bu durum da sanal dom

yaklaşımının sayfa yükleme süresine ve genel uygulama performansına olan etkilerinin çerçeveden bağımsız olarak değerlendirilmesi gerektiğini ortaya çıkarmaktadır.

Molin tarafından gerçekleştirilen çalışmada (Molin, 2016), tek sayfa Javascript uygulamaları için Angularjs, Angular2, React çerçeveleri kullanılarak 100 satırlık ve 1000 satırlık veri boyutları ile 1'er kez TodoMVC uygulaması üzerinde gerçekleştirilmektedir. Çalışmada sadece yükleme zamanı ve veri boyutu arasındaki ilişki açısından karşılaştırmalar yapılmaktadır. Gerçekleştirilen testlerde, bu çalışmayla ortak olarak test edilen Javascript çerçevelerinden Reactjs çerçevesinin Angularjs çerçevesine göre daha iyi performans gösterdiği sonucuna ulaşmışlardır. Bu sonuçlar genel olarak bu çalışmayla Chrome tarayıcısı üzerinde gerçekleştirilen testlerde paralellik gösterse de söz konusu çalışmada Firefox ve Internet Explorer tarayıcılarında test yapılmadığı için karşılaştırma imkanı bulunmamaktadır.

Bu çalışmada 100 nesneye kadar veri boyutu ile geliştirilecek uygulamalarda, çerçeve kullanımının sayfa yükleme zamanını, kullanıcı tarafından farkedilebilir sürelerde etkilemediği sonucuna varılmaktadır. Veri boyutu 1000 nesneye çıkarıldığında ise Reactjs ve Vuejs gibi sanal dom oluşturma yaklaşımı kullanan çerçeveler için sayfa yükleme sürelerinin daha fazla uzadığı gözlemlenmektedir. Bu yönüyle sanal dom oluşturma yaklaşımı veri boyutlarının artması durumunda özellikle Internet Explorer tarayıcısında, yükleme zamanları bağlamında düşük performans gösterdiği sonucuna ulaşılmaktadır.

Bu çalışma, istemci taraflı web uygulama performansının değerlendirilmesinde, yazılım mühendisliği ve yazılım performans mühendisliği kapsamında katkı sağlaması yönünden önem arz etmektedir. Bu çalışma halihazırda endüstride kullanılan yazılım mimarilerinin uygulama performansına etkisinin değerlendirilmesinde, bugüne kadar yapılan yazılım performansı değerlendirme çalışmalarını genişletmekte, bu çalışmalara yöntem ve araç olarak katkı sunmaktadır.

KAYNAKLAR

Aghaei, S., Nematbakhsh, M. A., and Farsani, H. K. (2012, Ocak). Evolution of The World Wide Web: From Web 1.0 to Web 4.0. *International Journal of Web and Semantic Technology (IJWesT), 3*(1), 1-10.

Butkiewicz, M., Madhyastha, H. V., and Sekar, V. (2011). Understanding Website Complexity: Measurements, Metrics, and Implications. *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, 313-328. Berlin.

Connolly, R., and Hoar, R. (2015). Web Servers, *Fundamentals of Web Development, Global Edition*, 91-92. Pearson Education Limited.

Davila, H. (2015, Ocak). *Performance of Javascript Frameworks on Web Single Page Applications.* Pontificia Universidad Catolica de Chile.

Flanagan, D. (2011). Overview of the DOM. D. Flanagan içinde, *Javascript: The Definitive Guide (6th Edition)*, 361, Kanada: O'Reilly.

Freeman, A. (2014). *Pro ASP.NET MVC 5 Platform.* APress.

Garcia-Izquierdo, F. J., and Izquerdo, R. (2012, Mart). Is the Browser the Side for Templating? *IEEE Internet Computing*, 61-68.

Gizas, A., Christodoulou, S., and Papathedorou, T. (2012). Comparative evaluation of javascript frameworks. *WWW '12 Companion Proceedings of the 21st International Conference on World Wide Web*, 513-514, Lyon.

Graziotin, D., and Abrahamsson, P. (2013). Making Sense Out of a Jungle of JavaScript Frameworks: Towards a Practitioner-Friendly Comparative Analysis. J. Heidrich, o. M. Oiv, J. A., and M. Baldassarre (Dü) içinde, *Product-Focused Software Process Improvement. PROFES 2013*, 334-337, Paphos: Springer.

Guardia, C. d. (2016). *Python Web Frameworks.* O'Reilly.

Hartl, M. (2012). *Ruby on Rails Tutorial Second Edition.* Addison Wesley.

İnternet: AngularJS. URL: https://angularjs.org, Son Erişim Tarihi: 01.09.2019.

İnternet: Browser Market Share, Market Share Statistics for Internet Technologies. URL: https://netmarketshare.com, Son Erişim Tarihi: 10.06.2019.

İnternet: Chapman, S. *History of Javascript*. URL: https://www.thoughtco.com/a-brief-history-of-javascript-2037675, Son Erişim Tarihi: 14.10.2019.

İnternet: ECMA, *ECMAScript® 2018 Language Specification.* URL: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, Son Erişim Tarihi: 14.08.2019.

İnternet: Fedosejev, A. (2015, Nisan 25). *React.js Essentials.* Packt Publishing. ReactJS: URL: https://reactjs.org, Son Erişim Tarihi: 11.10.2019.

İnternet: Fielding, R., Nottingham, R., *Hypertext Transfer Protocol (HTTP/1.1): Caching.* Internet Engineering Task Force, URL:https://tools.ietf.org/pdf/rfc7234.pdf, Son Erişim Tarihi: 11.09.2019.

İnternet: Github. URL: http://www.github.com, Son Erişim Tarihi: 10.09.2019.

İnternet: Google, *Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed.* URL: https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf, Son Erişim Tarihi: 20.10.2019.

İnternet: Grigorik, I., *Render Blocking CSS*. Google Web Fundamentals: URL: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-blocking-css, Son Erişim Tarihi: 17.07.2019.

İnternet: IETF, *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616, URL:https://www.rfc-editor.org/rfc/pdfrfc/rfc2616.txt.pdf, Son Erişim Tarihi: 21.07.2019.

İnternet: Json Generator., URL: https://www.json-generator.com, Son Erişim Tarihi: 04.10.2019.

İnternet: Le Hors, A., Hégaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., and Byrne, S. (2004), Document Object Model (DOM) Level 3 Core Specification: URL: https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407, Son Erişim Tarihi: 14.07.2019.

İnternet: O'Grady, S. (2019, 6 1). *The RedMonk Programming Language Rankings: June 2019,* URL:http://www.redmonk.com/sogrady/2017/06/08/language-rankings-6-17, Son Erişim Tarihi: 12.07.2019.

İnternet: Osmani, A. (2015). *Helping you select an MV* framework*. URL: http://todomvc.com/, Son Erişim Tarihi: 08.08.2019.

İnternet: Peyrott, S. (2017, Ocak 16). *A Brief History of Javascript*, URL: https://auth0.com/blog/a-brief-history-of-javascript, Son Erişim Tarihi: 12.07.2019*İnternet:* Github React, URL: https://github.com/facebook/react, Son Erişim Tarihi: 04.05.2019.

İnternet: Reenskaug, T. (1979). *MVC - XEROX PARC 1978-79*, URL: http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html, Son Erişim Tarihi: 19.04.2019.

İnternet*: Stackoverflow*, URL:http://www.stackoverflow.com adresinden alındı, Son Erişim Tarihi: 19.07.2019.

İnternet: W3C, *NavigationTimingAPI* W3C Recommendation, URL: https://www.w3.org/TR/navigation-timing/, Son Erişim Tarihi: 07.06.2019.

İnternet: W3C (2015). *DOM4 Working Draft*, W3C Recommendation, URL: https://www.w3.org/TR/domcore/, Son Erişim Tarihi: 07.06.2019.

İnternet: W3C (2019),*CSS Object Model (CSSOM).* URL:https://drafts.csswg.org/cssom, Son Erişim Tarihi: 07.06.2019.

İnternet: W3C, *DOM Living Standard*, URL: https://dom.spec.whatwg.org/, Son Erişim Tarihi: 07.06.2019.

İnternet: Wine HQ, About Wine Project. URL: https://www.winehq.org, Son Erişim Tarihi: 18.04.2019.

İnternet: You, E. (2018), *Vue.js*, URL: https://vuejs.org/Q, Son Erişim Tarihi: 14.07.2019.

Koetsier, J. (2016, Haziran 8). *Evaluation of JavaScript frameworks for the development of a web-based user interface for Vampires.* Universiteit van Amsterdam.

Leff, A., and Rayfield, J. (2001). Web-Application Development Using the Model/View/Controller Design Pattern. *EDOC '01 Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, 118-127.

Lockhart, J. (2015). *Modern PHP*. O'Reilly.

Maras, J., Carlson, J., and Crnkovic, I. (2012). Extracting Client-side Web Application Code. *WWW '12 Proceedings of the 21st international conference on World Wide Web*, 819-828, Lyon: ACM Digital Library.

Mehdi, J. (2007). Some Trends in Web Application Development. *Future of Software Engineering*, 199-213, Washington: IEEE Computer Society.

Molin, E. (2016). *Comparison of Single-Page Application Frameworks.* KTH Royal Institue of Technology.

Nadhom, M., and Loskot, P. (2018). Survey of public data sources on the Internet Usage and Other Internet Statistics. *Data in Brief, 18*, 1914-1929.

Ocariza, J., Frolin, S., Pattabiraman, K., and Mesbah, A. (2015). Detecting inconsistencies in JavaScript MVC applications. *ICSE '15 Proceedings of the 37th International Conference on Software Engineering, 1*, 325-335.

Offutt, J. (2002). Quality attributes of Web software applications. *IEEE Software*, 25-32.

Oluwatosin, H. S. (2014). Client-Server Model. *IOSR Journal of Computer Engineering (IOSR-JCE), 16*(1), 67-71.

O'Reilly, T. (2007). What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Communications and Strategies*, 17.

Pradel, M., Schuh, P., Necula, G., and Sen, K. (2014). EventBreak: Analyzing the responsiveness of user interfaces through performance-guided test generation. *OOPSLA '14- Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 33-47.

Ramos, M., Valente, M. T., and Terra, R. (2018, Mart/Nisan). AngularJS Performance: A Survey Study. *IEEE Software, 35*(2), 72-79.

Ratanaworabhan, P., Livshits, B., Simmons, D., and Zorn, B. (2010). JSMeter: Measuring Javascript Behaviour In The Wild. *Proceedings of the USENIX Conference on Web Application Development*, 1-12.

Ruparelia, N. B. (2010, Mayıs). Software Development Lifecycle Models. *ACM SIGSOFT Software Engineering Notes, 35*(3), 8-13.

Sampson, A., Cascaval, C., Ceze, L., Montesinos, P., and Suárez Gracia, D. (2012). Automatic discovery of performance and energy pitfalls in HTML and CSS. *IEEE International Symposium on Workload Characterization (IISWC)*, 82-83.

Selakovic, M., and Pradel, M. (2016). Performance Issues and Optimizations in JavaScript: An Empirical Study. *2016 IEEE/ACM 38th IEEE International Conference on Software Engineering (ICSE).* Austin, ABD.

Smith, P. (2013). The Document Object Model. P. Smith içinde, *Professional Web Site Performance: Optimizing The Front End And The Back End* (s. 116). Indiana, ABD: Wiley.

Tilkov, S., and Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing, 14*(6), 80-83.

Totty, B., Gourley, D., Sayer, M., Aggarwal, A., and Reddy, S. (2009). O'Reilly Media.

Vipul, A., and Sonpatki, P. (2016). *ReactJS by Example - Building Modern Web Applications with React.* Packt Publishing.

Wang, X. S., Balasubramanian, A., Krishnamurthy, A., and Wetherall, D. (2013). Demystifying Page Load Performance with WProf. *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)* , 473-485.

Wang, X. S., Krishnamurthy, A., and Wetherall, D. (2016). Speeding Up Web Page Loads with Shandian. *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, 109-122, Santa Clara, CA: USENIX Association.

Woodside, M., Franks, G., and Petriu, D. C. (2007). The Future of Software Performance Engineering. *Future of Software Engineering, FOSE '07*, 171-187, IEEE Computer Society.

Xing, Y., Huang, J., and Lai, Y. (2019). Research and Analysis of the Front-end Frameworks and Libraries in E-Business Development. *ICCAE 2019 Proceedings of the 2019 11th International Conference on Computer and Automation Engineering*, 68-72, Perth.

Zou, Y., Chen, Z., Zheng, Y., Zhang, X., and Gao, Z. (2014). Virtual DOM coverage for effective testing of dynamic web applications. *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 – Proceedings.*

EKLER

(Ekler Tezin arka kapağında CD ortamında verilmiştir.)

ÖZGEÇMİŞ

Kişisel Bilgiler

Soyadı, adı	: YILMAZ, Bekir
Uyruğu	: T.C.
Doğum tarihi ve yeri	: 06.09.1984, Afyonkarahisar
Medeni hali	: Evli
Telefon	: 0 (312) 207 61 77
E-mail	: bekiryilmaz.ce@gmail.com

Eğitim

Derece	Eğitim Birimi	Mezuniyet Tarihi
Yüksek Lisans	Gazi Üniversitesi / Bilgisayar Bilimleri	Devam ediyor
Lisans	Eskişehir Osmangazi Üniversitesi / Bilgisayar Mühendisliği	2012
Lise	Afyon Milli Piyango Anadolu Lisesi	2002

İş Deneyimi

Yıl	Yer	Görev
2012-Halen	Su Yönetimi Genel Müdürlüğü	Uzman
2009-2012	Ptt Genel Müdürlüğü	Mühendis
2008-2009	Inveon Bilgi Teknolojileri	Yazılım Geliştirme Uzmanı

Yabancı Dil

İngilizce

GAZİLİ OLMAK AYRICALIKTIR...