Ontwerpdocumentatie Inside Airbnb

Niels Bosman | ITA-NOTS-A-f

Auteur: Niels Bosman
Studentnummer: 646983
Klas: ITA-NOTS-A-f
Docent: Marcel Verheij
Course: NOTS
Datum: 07-06-2022
Versie: 1.0

Inhoudsopgave

1 Inleiding

Dit document is het ontwerp document voor de webapplicatie “Inside Airbnb”. Met deze webapplicatie kunnen gebruikers de data van alle Airbnb listings in Amsterdam zien op een dynamische kaart met behulp van Mapbox[1]. Deze datapunten zijn ook filterbaar op prijs, buurt en aantal reviews. Naast de basis features van deze app is er ook de mogelijkheid om interessante statistieken uit de data in te zien indien de gebruiker een administrator rol toegekend gekregen heeft.

Het doel van dit document is om te documenteren welke ontwerpkeuzes er gemaakt zijn in het development proces van Inside Airbnb. Hierbij wordt het functioneel en technisch ontwerp toegelicht maar ook de keuzes die gemaakt zijn op het vlak van performance en security.

[1] https://www.mapbox.com/

2 Functioneel ontwerp

Voordat er aan het ontwikkelen van Inside Airbnb begonnen kan worden moet er eerst in kaart gebracht worden welke use cases er zijn. Dit is in dit geval erg makkelijk omdat de casus dit expliciet heeft gedocumenteerd.

Use case	Prioriteit
Registreren en inloggen	Must
Filter op prijs	Must
Filter op buurt	Must
Filter op review	Must
Kaart is clickable, details rechts op pagina, maakt gebruik van de mapbox API	Must
Details per item waarop gefilterd is: #overnachtingen, #opbrengst in de maand	Must
Er moeten rollen toegevoegd en toegekend worden aan de geregistreerde gebruikers	Must
Resultaten zoals trends, totalen, gemiddelden, etc. worden weergegeven in charts, alleen te bekijken voor admins. Denk daarbij aan bv. gemiddelde beschikbaarheid per maand, gemiddelde beschikbaarheid per buurt, overzicht van gemiddelde huurprijs per buurt. Andere managementoverzichten zijn ook mogelijk, ga daarvoor op zoek naar online voorbeelden.	Must
Locaties van zoekresultaat zichtbaar op kaart.	Could
Layout idem als insideairbnb.com	Could

Figuur 1: Use cases in tabelvorm

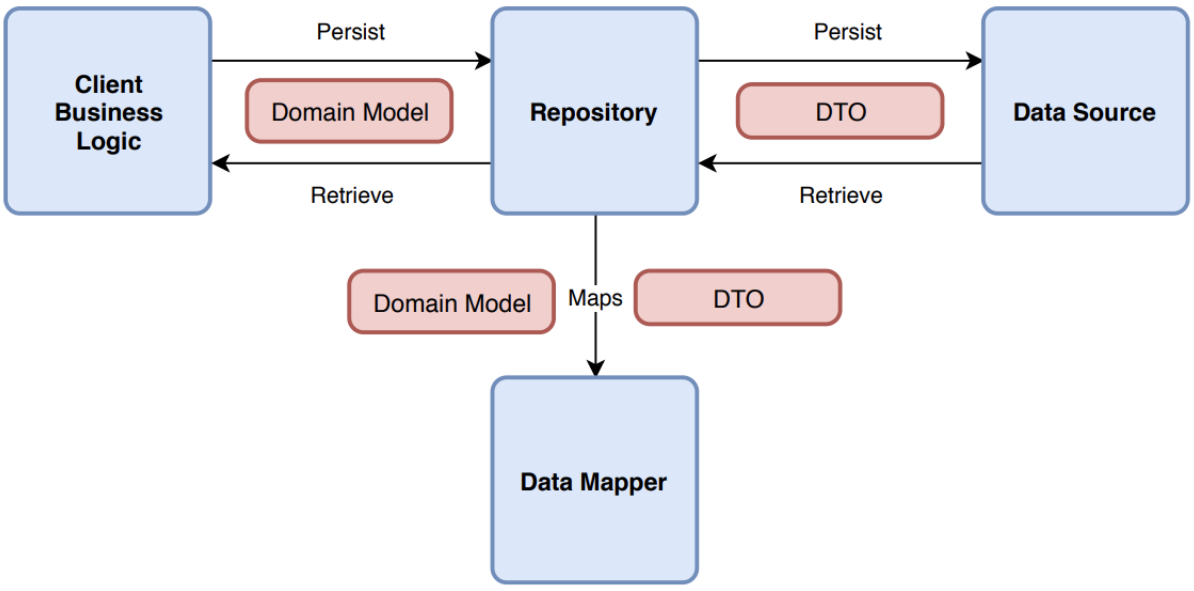
3 Technisch ontwerp

In dit hoofdstuk wordt er wat meer verteld over de technische ontwerpkeuzes voor Inside Airbnb.

De data voor de applicatie komt uit een MSSQL Azure database[2]. Deze data is geïmporteerd uit een .bacpac bestand van de provider Inside Airbnb. De data uit de database wordt uitgelezen door een ASP.NET Web Api[3]. Deze API draait op de nieuwste stabiele versie van .NET (6.0) om er voor te zorgen dat de laatste features gebruikt kunnen worden.

De hierboven aangesproken API is natuurlijk niet genoeg om een gebruiker van Inside Airbnb aan de eerder besproken use cases te laten voldoen. Om deze reden is er voor gekozen Next.js[4] te gebruiken voor de client side logica. Next.js maakt het makkelijk om door middel van javascript interactieve web applicaties te maken met behulp van het React[5] framework. Deze applicatie zal de API aanspreken en met behulp van die data de gebruiker de use cases laten uitvoeren.

Om in de API op een nette manier met de data uit te database te werken is er gebruik gemaakt van het repository pattern[6]. Door gebruik te maken van het repository pattern is het gemakkelijk om vanuit je applicatie data uit een database te verkrijgen, zonder direct met de database te praten in je “business logic”. Dit heeft als resultaat dat de business logic een stuk gemakkelijker te lezen is omdat hier geen logica in staat over hoe bepaalde data uit een database opgevraagd wordt. Hieronder is gevisualiseerd hoe dit werkt.



Figuur 2: Repository pattern gevisualiseerd

[2] https://azure.microsoft.com/en-us/products/azure-sql/database/#overview
[3] https://dotnet.microsoft.com/en-us/apps/aspnet/apis
[4] https://nextjs.org/
[5] https://reactjs.org/
[6] https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30

Om het authenticeren van gebruikers op een veilige manier te doen is er gebruik gemaakt van Auth0[7]. Deze tool zorgt ervoor dat je in de applicatie zelf geen authenticatie in hoeft te bouwen maar dat dit systeem door hun geregeld wordt. Het enige wat er gedaan is om dit te gebruiken is in de client side code de gebruiker redirecten naar de Auth0 inlog pagina. Wanneer de gebruiker hier inlogt wordt hij daarna weer terug naar de applicatie gestuurd waar het inloggen in de browser sessie onthouden wordt. Via de client side code wordt de access token van Auth0 opgevraagd waarmee de gebruiker de API kan laten weten welke rol deze heeft. In het onderstaande figuur staat het stuk code dat in de API staat om te controleren of de gebruiker wel de juiste rechten heeft om de specifieke API route te bereiken.

```
[HttpGet("listing-amount-per-neighbourhood")]
[Authorize(Policy = "ReadStatisticsAccess")]
public async Task<IActionResult> GetListingAmountPerNeighbourhood()
{
        // Controller code
}
```

Figuur 3: Auth0 verificatie in API controller

3.1 Packages

3.1.1 API

Bij het ontwikkelen van de .NET Web API zijn de volgende packages gebruikt.

Package		Doel
Microsoft.AspNetCore.Authentication.JwtBearer	6.0.5	Het afhandelen van authenticatie met Jwt tokens mogelijk maken.
Microsoft.EntityFrameworkCore.Design	6.0.5	Het entity framework
Microsoft.Extensions.Caching.StackExchangeRedis	7.0.0	Caching door middel van Redis
Swashbuckle.AspNetCore	6.2.3	Swagger API documentatie voor ASP.NET
Microsoft.EntityFrameworkCore.SqlServer	6.0.4	Verbinden met de MSSQL server
Microsoft.VisualStudio.Web.CodeGeneration.Design	6.0.3	Het mogelijk maken van het automatisch genereren van controllers.

Figuur 4: ASP.NET Web Api packages

[7] https://auth0.com/

3.1.2 Client

Bij het ontwikkelen van de client side Next.js code zijn de volgende packages gebruikt

Package		Doel
@auth0/auth0-react	1.10.1	Authenticatie met Auth0 gemakkelijker implementeren in React door middel van een officiële wrapper.
@heroicons/react	1.0.6	Icons van Heroicons gebruiken in React.
axios	0.27.2	Het fetchen van data simpeler maken en minder boilerplate
chart.js	3.8.0	Statistieken voor admins in grafieken laten zien.
mapbox-gl	2.8.2	Kaart tonen met listings.
rc-slider	10.0.0	Een slider met een minimale en maximale waarde gebruiken in React voor filters.
react-chartjs-2	4.1.0	Het gebruik van Chart.js gemakkelijker maken in React.
react-map-gl	7.0.12	Het gebruik van Mapbox gemakkelijker maken in React.
react-select	5.3.2	Een select veld gebruiken in React waar de selectie mee verwijderd kan worden.
typescript	4.6.4	Typescript support voor type safety in Javascript.

Figuur 5: Client Next.js applicatie packages

4 Performance

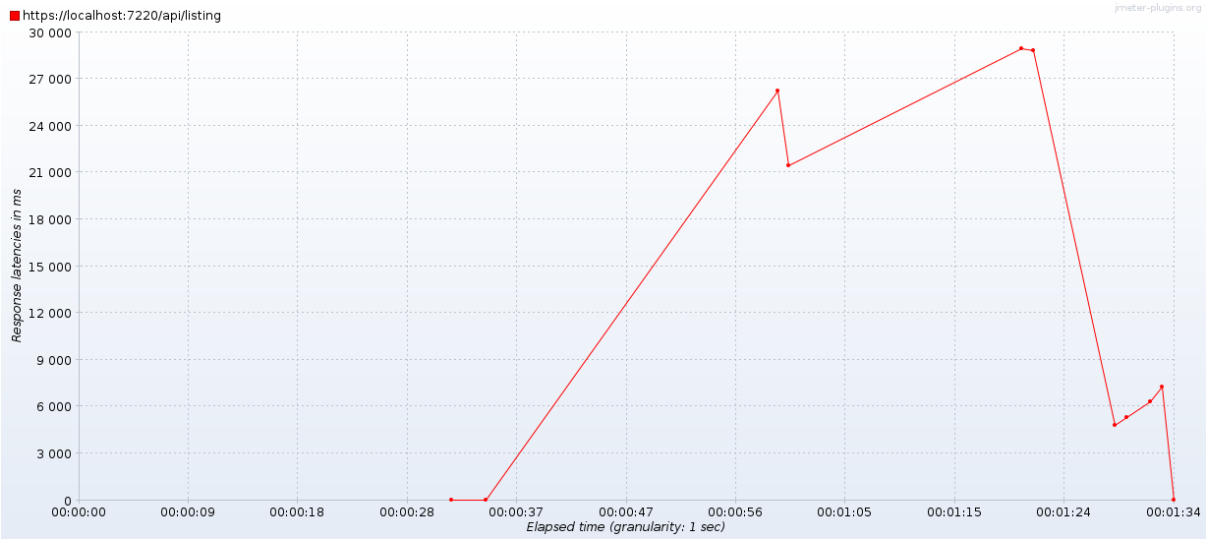
4.1 Baseline

Om goed in te kunnen schatten hoe de performance van de applicatie verbeterd op basis van aanpassingen wordt er een baseline test uitgevoerd. Deze test zal door middel van Apache JMeter[8] gedaan worden. Hierbij worden de volgende instellingen gebruikt.

Target rate	Ramp Up Time	Ramp Up Steps	Hold Target Rate Time
30 arrivals/s	40 seconden	5	20 seconden

Figuur 6: Instellingen JMeter voor performance tests

Met deze instellingen wordt de meest eisende API route getest. Dit is de API route die alle listings ophaal, dit zijn er meer dan 19.000 waardoor deze erg veeleisend is.



Figuur 7: Baseline test voor de de API route van alle listings.

Zoals hierboven te zien is zijn deze instellingen te zwaar voor de route. Dit omdat er na ongeveer 30 seconden pas de eerste response kwam in vorm van een error. Omdat dit geen goede baseline is om naar te kijken omdat de dataset zo groot is ga ik eerst een simpele optimalisatie toevoegen waardoor een groot probleem al opgelost wordt: Het aantal kolommen dat moet worden opgehaald.

[8] https://jmeter.apache.org/

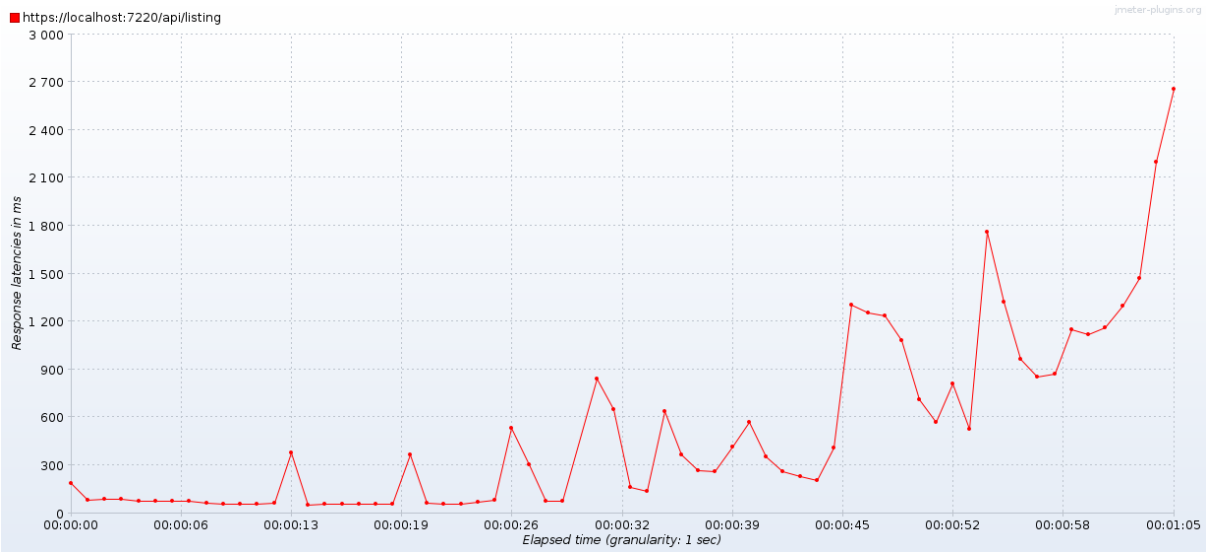
4.1.1 Relevante kolommen selecteren

Omdat deze dataset van 19.000+ records erg groot is is het belangrijk dat we van deze data alleen de kolommen ophalen waar we daadwerkelijk wat mee willen doen. In dit geval zijn dat: Id, Latitude, Longitude, Price, Neighbourhood, NumberOfReviews en ReviewScores rating. Deze kolommen selecteren we door een extensie te maken op de Listing DbSet met een LINQ select statement.

```
public static IQueryable<ListingDto> AsDto(this DbSet<Listing> listings)
    {
        return listings.Select(listing => new ListingDto {
            Id = listing.Id,
            Latitude = listing.Latitude,
            Longitude = listing.Longitude,
            Price = listing.Price,
            Neighbourhood = listing.NeighbourhoodCleansed,
            NumberOfReviews = listing.NumberOfReviews,
            ReviewScoresRating = listing.ReviewScoresRating,
        });
    }
```

Figuur 8: Relevante kolommen selecteren in C# dmv. LINQ

Door dit toe te voegen behaalt de performance test de volgende resultaten:



Figuur 9: Resultaten performance test na selecteren belangrijke kolommen.

Hier is te zien dat de resultaten een stuk beter zijn. De API regeert in het begin erg snel en loopt uiteindelijk op naar meerdere seconden, wat natuurlijk te lang is voor een API. Maar wat wel opvalt is dat de API alle requests aan kan, hij crasht niet meer.

De volgende statistieken kunnen uit de performance test gehaald worden. Door deze statistieken als baseline te gebruiken kan er worden gekeken of bepaalde optimalisaties de snelheid van de API verbeteren.

Minimale responstijd	Maximale responstijd	Gemiddelde responstijd
0.74 s	16.54 s	5.254 s

Figuur 10: Resultaten performance test baseline in tabelvorm

4.2 AsNoTracking

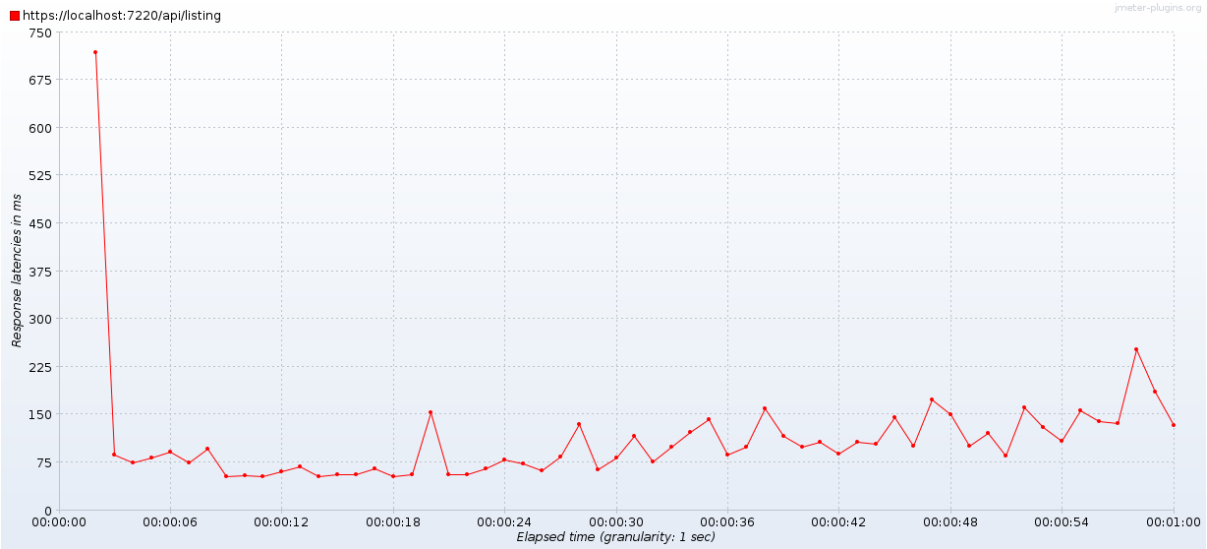
De volgende performance verbetering die toegevoegd zal worden is het gebruik van AsNoTracking()[9] in C#. AsNoTracking zorgt ervoor dat de context van EF Core niet hoeft te onthouden welk record er opgehaald is. Dit is in dit geval niet nodig omdat wij ook niet deze listing aan willen passen, in dat geval is het nodig dat de context onthoudt over welke listing het gaat.

Het toepassen van AsNoTracking is erg gemakkelijk. Dit kan op elke IQueryable lijst in C# toegevoegd worden op de volgende manier:

```
public async Task<IEnumerable<ListingDto>?> GetAll()
{
  return await _listings
    .AsDto()
    .AsNoTracking()
    .ToListAsync();
}
```

Figuur 11: Toepassen van AsNoTracking() in de ListingRepository

Het toevoegen van AsNoTracking had de volgende performance test resultaten.



Figuur 12: Performance test resultaten na toevoegen AsNoTracking()

Zoals hier te zien is verbeterd dit de resultaten erg. Met uitzondering van de eerste request blijft de lijn een stuk vlakker waar uit te zien is dat de API bijna de hele test de zelfde performance had.

De volgende statistieken zijn uit de test gekomen:

[9] https://stackoverflow.com/questions/12211680/what-difference-does-asnotracking-make

	Minimale responstijd	Maximale responstijd	Gemiddelde responstijd
Baseline	0.74 s	16.54 s	5.254 s
Met AsNoTracking	0.69 s	2.153 s	1.85 s
Verschil	- 0.05 s	- 14.387	- 3.404

Figuur 13: Resultaten performance test met AsNoTracking() in tabelvorm

Er kan dus worden geconcludeerd dat het toevoegen van AsNoTracking() een erg goede invloed heeft op de performance van de applicatie. En aangezien dit geen nadelen heeft in dit geval is het een erg goede keuze.

4.3 Redis cache

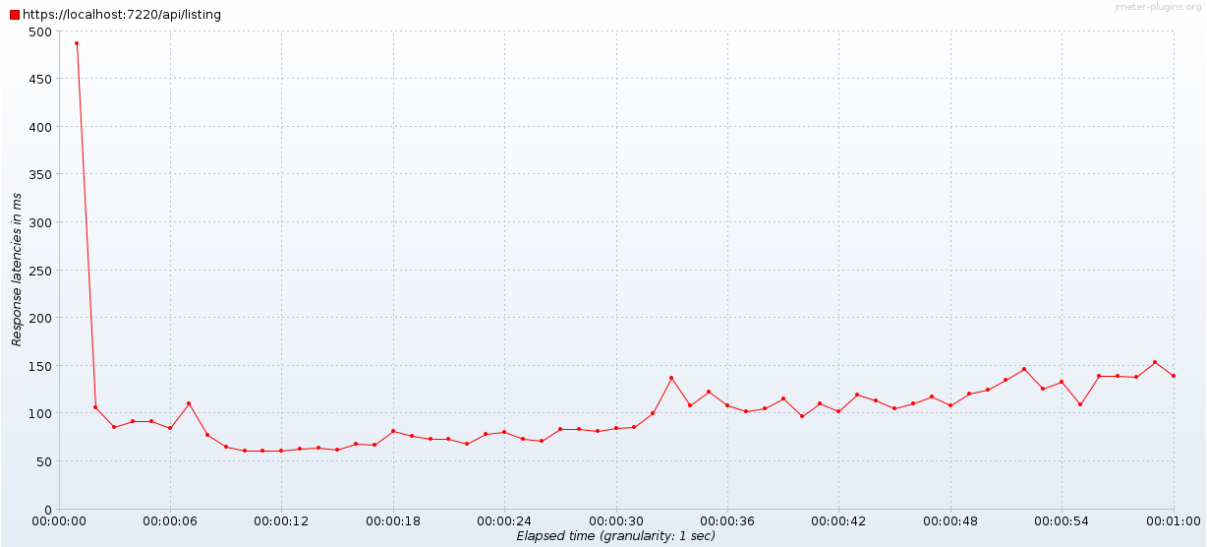
Om de load op de database te verlagen omdat er weinig veranderd in de database kan verstandig zijn cache te gebruiken in de vorm van Redis Cache[10]. Dit zorgt ervoor dat de data in een redis geheugen database opgeslagen wordt waardoor het een stuk sneller opgehaald kan worden. Het implementeren van Redis is erg gemakkelijk. Na een aantal regels toe te voegen aan de middleware in Program.cs kan er door middel van dependency injection de cache geïnjecteerd worden in de controller. Op de volgende manier kan daarna de cache gebruikt worden:

```
public async Task<ActionResult<IEnumerable<Listing>>> GetAll()
{
  const string cacheKey = "all-listings";
  var cachedListings = await _cache.GetStringAsync(cacheKey);

  if (cachedListings != null)
  {
    return Ok(JsonConvert.DeserializeObject<List<ListingDto>>(cachedListings));
  }

  var listings = await _repository.GetAll();

  await _cache.SetStringAsync(
    cacheKey,
    JsonConvert.SerializeObject(listings),
    new DistributedCacheEntryOptions().SetSlidingExpiration(_cacheTimeout)
  );

  return Ok(listings);
}
```

Figuur 14: Toepassing Redis in .NET

[10] https://redis.io/

Allereerst wordt er gekeken of er al een item in de cache staat voor de opgegeven key (in dit geval “all-listings”. Wanneer dit zo is pakt het systeem de cachede listings uit Redis en retourneerd dit. Wanneer dit niet zo is haalt het systeem de listings op via de normale manier en zet hij de waarde in de cache voor de volgende requests. Er is voor een cache timeout van 15 minuten gekozen. Het toevoegen van deze caching service had de volgende impact op de performance test:



Figuur 15: Performance test na toepassen redis cache.

In deze grafiek is te zien dat de performance erg vergelijkbaar is met de vorige, maar toch een stuk vlakker is. Dit betekend dus dat de performance een stuk voorspelbaarder is dan zonder redis cache, wat er waarschijnlijk voor zal zorgen dat bij nog meer requests deze variant dit nog beter aankan. Maar om echt goed te zien wat de resultaten zijn is het goed om de echte getallen te vergelijken met die van AsNoTracking().

	Minimale responstijd	Maximale responstijd	Gemiddelde responstijd
Met AsNoTracking	0.69 s	2.153 s	1.85 s
Met Redis	0.74 s	1.321 s	1.48 s
Verschil	+ 0.05 s	- 0.832 s	- 0.37 s

Figuur 16: Performance verschil redis in vergelijking met alleen AsNoTracking()

Uit deze tabel kan worden opgedaan dat de minimale responstijd niet echt wijzigt. Sterker nog, dit is zelfs een beetje slechter. De maximale responstijd is echter bijna de helft als dat het was vóór het implementeren van de redis cache, wat echt een groot verschil is. Ook misschien wel de belangrijkste statistiek, de gemiddelde responstijd is een stuk sneller.

Al met al kan worden geconcludeerd dat het toepassen van Redis caching een positief effect heeft op de performance van de API.

4.3 Next.js

Naast alle hierboven gedemonstreerde performance optimalisaties op de API biedt het client side framework Next.js ook nog mogelijkheid de client side code te optimaliseren door middel van static props.

Wat Next.js doet als je getStaticProps()[11] gebruikt op je pagina is wanneer de app gebuild wordt alle properties ophalen die in deze functie staan. Dit wordt dan lokaal in een json bestand opgeslagen waardoor dit super snel in de pagina laadt. Het nadeel van het gebruiken van getStaticProps() is dat het statisch is (en dus niet aangepast is als de database veranderd), maar hier is een oplossing voor. Wanneer je dit gebruikt kan je in de return statement van de functie een *revalidate* property meesturen. Dit geeft aan om de hoeveel seconden deze statische props opnieuw opgehaald moeten worden. Dit werkt dus ongeveer hetzelfde als de redis cache, maar dan op de front-end. Hieronder is het voorbeeld te zien hoe ik dit heb toegepast om de API aan te roepen en op te slaan.

```
export const getStaticProps: GetStaticProps = async () => {
  const { NEXT_PUBLIC_API_URL } = process.env
  const allListings: Listing[] = (await axios.get(`${NEXT_PUBLIC_API_URL}/listing`)).data

  return {
    props: { allListings },
    revalidate: 30,
  }
}
```

Figuur 17: getStaticProps in Next.js

In het figuur is te zien dat ik de revalidate prop een waarde van 30 heb gegeven. Dit om er voor te zorgen dat de cache op de front-end niet leidend zal zijn en er vaak genoeg een call gedaan wordt naar de API.

[11] https://nextjs.org/docs/basic-features/data-fetching/get-static-props

4.4 Conclusie

Door gebruik te maken van de drie optimalisaties: Relevante kolommen selecteren, AsNoTracking() en Redis cache is de performance van de API drastisch veranderd. Na het toepassen van het selecteren van de juiste kolommen crashte de applicatie niet meer en na het toepassen van AsNoTracking() en Redis cache is de gemiddelde responstijd met 71.83%[12] verlaagt.

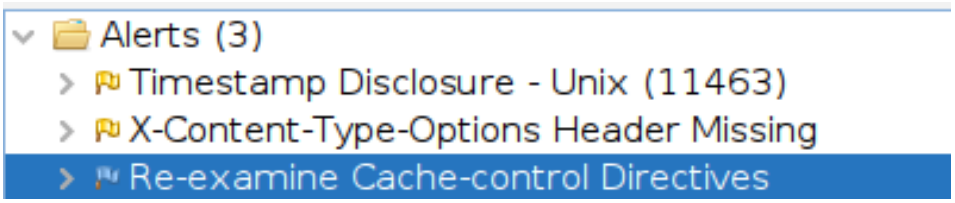
Naast deze optimalisaties biedt het client side framework Next.js nog meer optimalisaties wat ervoor zorgt dat de webapplicatie als nog sneller ervaren wordt. Dit door middel van het server side ophalen van de listings en deze “cachen” in een bestand.

[12] (1.48 s - 5.254 s) / 5.254 s * 100

5 Security

5.1 OWASP ZAP

Om goed te checken of de gemaakte applicatie veilig is voeren we een security check uit met OWASP ZAP[13]. Deze software crawlt de gegeven URL's aan geeft aan wat hier aan verbeterd kan worden.



Figuur 18: API test voor verbeteringen

Om goed te kunnen weten wat er verbeterd kan worden is er een scan uitgevoerd voordat er verbeteren aan de veiligheid aangebracht zijn. Uit deze scan is te zien dat er twee problemen zijn op het gebied van security. "X-Content-Type-Options Header Missing" en "Re-examine Cache-control Directives". Deze twee waarschuwingen zijn erg gemakkelijk op te lossen door een aantal regels code toe te voegen de aan Program.cs middleware in de API.

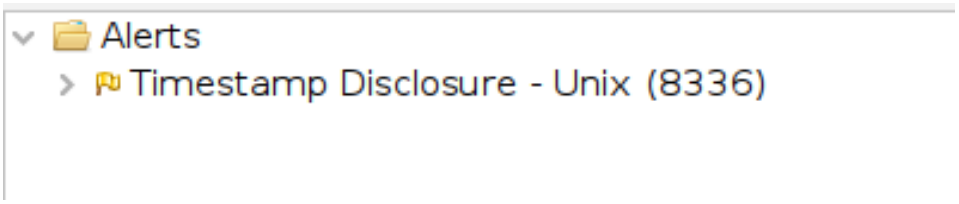
De eerste twee regels zorgen ervoor dat het duidelijk is dat de content type is en dat deze niet exploit kunnen worden. De laatste header die toegevoegd wordt zorgt ervoor dat de data uit de pagina zelf niet gecached wordt door andere browsers (dit betekent niet dat de redis niet gebruikt kan worden op de API zelf).

```
app.Use(async (context, next) =>
{
  context.Response.Headers.Add("Content-Type", "application/json");
  context.Response.Headers.Add("X-Content-Type-Options", "nosniff");
  context.Response.Headers.Add("Cache-Control", "nocache");
  await next();
});
```

Figuur 19: Aanpassingen security API

Na het toevoegen van deze code is te zien dat de alerts weg zijn (behalve Timestamp Disclosure, maar dit is geen echte alert).

[13] https://www.zaproxy.org/



Figuur 20: ZAP resultaten na verbeteringen