Foodiee vol.2

Food ordering web application

Final project course

Implementation

Zacharias-Christos Argyropoulos

SAE Athens 2022

Index

1. Introduction and deployment

The main focus of this final project will be to optimize and transform a previously developed React web application. Foodiee, as it is called, is a food ordering web application, in which the users can order food from multiple restaurants and get it delivered at their location at the same time. The users can additionally search restaurants by their location, filter the returned results, explore their menu, add items to their cart and finally checkout among other features. The main goal of this final project would be to further investigate and explore state of the art technologies and frameworks as well as refactoring existing code. This transformation should eventually lead to a better performance and SEO (search engine optimization) for most cases and as a result to a better user experience.

Both the service and the final user interface have being deployed on Vercel and the UI can be accessed publicly here; https://foodiee.vercel.app/

Additionally the source code of this project is publicly accessible here (branch master); https://github.com/z-argyropoulos/Foodiee-Ordering-Web-App .
Feedback and PR's are always welcomed.

2. Technologies and benefits

To accomplish the previously mentioned targets , three main technologies, tools and frameworks will be utilized. Next.js will be the main React framework used, TypeScript will be used over Javascript for the new components and logic as well as GraphQL for the data queries.

TypeScript, as stated in their official documentation, is a strongly typed programming language that builds on JavaScript, which gives you better tooling at any scale. Therefore, we can define types on our data and components, which catches many errors and mistakes in development and gives us better autocompletion in the editor.

Next.js is a framework that allows us to write React components as usual that run on the server which ultimately gives us more rendering capabilities than the typical client side rendering. Additionally, here we can specify which rendering/generation method of the following will be used on each page of our application;

- **Static site generation (SSG)**: The page gets generated at build time and then it gets cached and served from CDNs. Next will fetch at build time the data needed and generate the HTML along with some minimal javascript code to run on the client. This method is the fastest of all in terms of TTFB(time to first byte) but the data might be stalled as they are fetched only when the project builds.
- **Server side rendering (SSR)**: The page gets generated at runtime after every request to that page/resource. Here the TTFB is higher but data is guaranteed to be the latest available.
- **Incremental Static Regeneration (ISR)**: This method is a mix of SSG and SSR as it gives you the option to regenerate pages after the build stage. The exported page still can get cached but it has an expiration date. After the specified duration has passed, the page will get regenerated and so on.
- **Client side rendering (CSR)**: When the data on the page needs to get updated frequently then CSR is the way to go. While the data is the latest available, it has a negative impact on SEO. For this option Next also provides useful hooks for data fetching like SWR.

GraphQL will work as a middleware in the existing Node.js application and will reside besides the already defined REST APIs. With GraphQL the client can query exactly the data that it needs, which reduces over fetching data (when some properties in the response body won't be used at all) and under fetching data (having to make multiple separate API calls to get the final desired structured data).

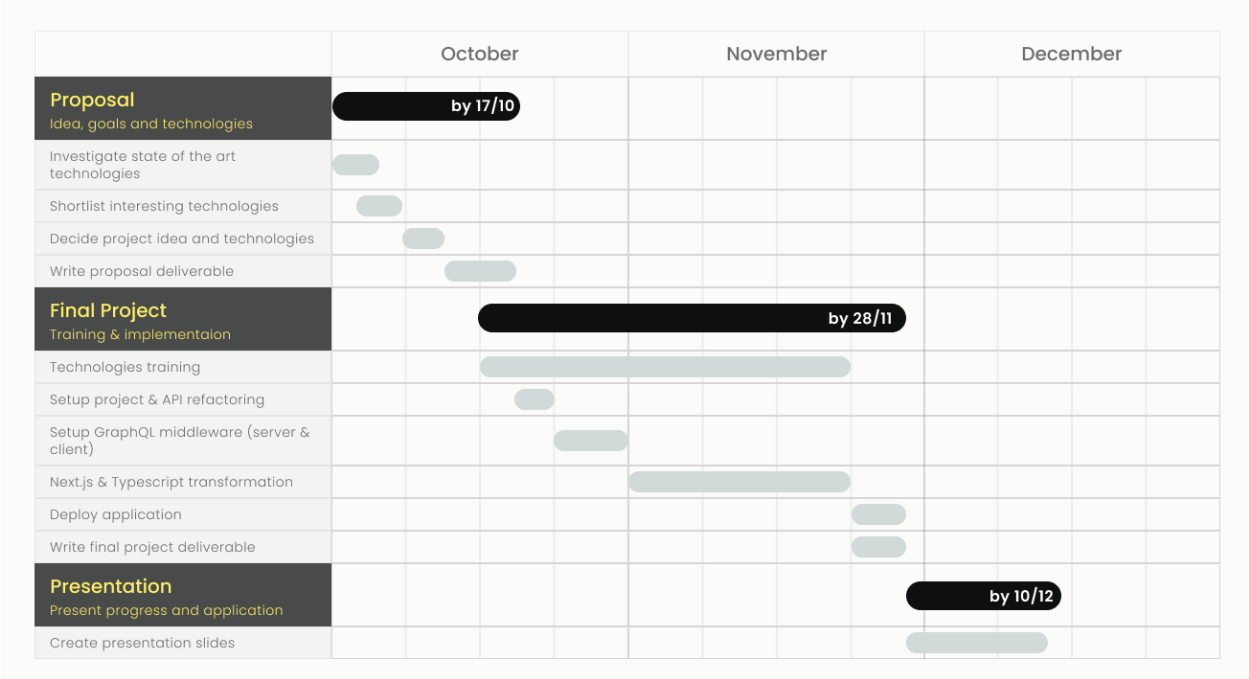
Technology stack overview

Next.js - React
TypeScript
MUI (Material UI)
Node.js
Express.js
MongoDB
GraphQL
Docker

3. Project management and timeline

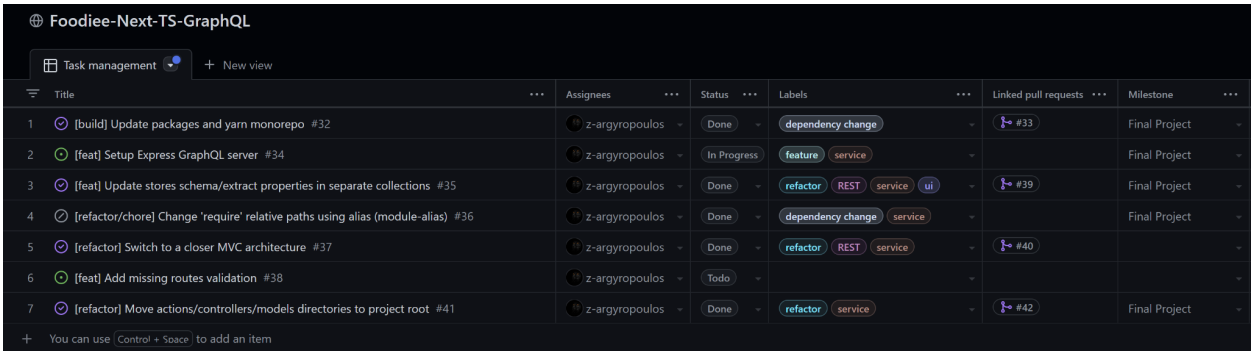
Project timeline

In order to keep track of the deadlines and tasks that need to be fulfilled for this project, an initial top level timeline has been created as show below;



Project management

Additionally for the tasks and the project management Github's Project feature will be used. All tasks (features, fixes, ci/cds, deployments etc) that need to be completed for this project will be created here while specifying more details for this task for example the status, milestones, topic, linked PRs etc. This will also automatically create an issue with all these details.



Conventions

Issues

For the naming of the issues the following format should be used;

[{type_of_issue}] {issue_title}

The type of issue can be one of 'build', 'chore', 'ci', 'docs', 'feat', 'fix', 'perf', 'refactor', 'revert', 'style', 'test'.

The issue title should begin with an uppercase letter.

Example; *[feat] Setup Express GraphQL server*

Branches

For the naming of the branch the following format should be used;

FD{issue_number}

After an issue has been created a unique number will be assigned to this issue, which will be used to checkout to this branch and initiate the implementation.

Example; *FD32*

Commit messages

For the messages of the individual commits in a branch, the following format should be used;

{branch_name} | {type_of_issue}({scope?}): {commit_description}

The type of issue can be one of 'build', 'chore', 'ci', 'docs', 'feat', 'fix', 'perf', 'refactor', 'revert', 'style', 'test'.

The scope is optional and can be left out of the commit message.

The commit description should be short, in present tense and begin with a lowercase letter.

For example; *FD32 | feat(profile-page): add avatar in user information*

Pull requests

For the naming of the PR the following syntax should be followed;

{branch_name} - {issue_title}

When the task is completed a pull request(PR) should be created linking to the issue and the relevant branch.

The pull request should also include a detailed description of what changes in the codebase.

For example; *FD32 - Update packages and yarn monorepo*

4. Service refactoring and GraphQL

Collection refactoring

The implementation started with making an overall refactoring on the db collections. It was decided to split the store collection into several ones for better scaling and for GraphQL (resolvers) to make more sense. For 1:1 relationships the properties remained embedded to the restaurant schema (e.g. the rating, the store address, the minOrder properties etc). For 1: N relationships the linking between different collections was achieved with a foreign key (e.g. the items included in the restaurant catalog). For example the restaurant catalog has many items, but a single item is unique for that store and therefore only refers to one restaurant. For this case a restaurantId was added to each item document referring to the _id of the restaurant. Below is presented a comparison of the collections/documents before and after the refactoring.

Before (1 collection)

Stores collection


```
_id: ObjectId('61f191f16cc66fffeaeb1ade')
name: "Sushi Land"
description: "Sushi is a traditional Japanese food consisting of a steam-cooked rice…"
address: Object
   street: "Sushi Street 27"
   region: "Pagkrati"
   postal_code: "12359"
   city: "Athens"
   _id: ObjectId('61f191f16cc66fffeaeb1adf')
categories: Array
   0: "Sushi"
   1: "Japanese"
   2: "Tuna"
rating: 9.5
deliveryTimeRange: Array
minOrder: "12€"
isOpen: true
catalog: Array
   0: Object
      category: "Bao Buns"
      products: Array
         0: Object
            title: "Chicken Bao Buns"
            description: "Description description description description."
            price: 6.54
            quantity: 30
            _id: ObjectId('61f191f16cc66fffeaeb1ae1')
         1: Object
      _id: ObjectId('61f191f16cc66fffeaeb1ae0')
   1: Object
__v: 0
```

After (3 collections)

Restaurants collection


```
_id: ObjectId('61f191f16cc66fffeaeb1ade')
name: "Sushi Land"
description: "Sushi is a traditional Japanese food consisting of a steam-cooked rice…"
address: Object
    street: "Sushi Street 27"
    region: "Pagkrati"
    postal_code: "12359"
    city: "Athens"
    _id: ObjectId('61f191f16cc66fffeaeb1adf')
categories: Array
    0: "Sushi"
    1: "Japanese"
    2: "Tuna"
rating: 9.5
deliveryTimeRange: Array
    0: 40
    1: 50
minOrder: "12€"
isOpen: true
__v: 0
```


Restaurant_categories collection


```
_id: ObjectId('634b48d705543076967b4cdf')
name: "Sushi"
description: "Category description"
__v: 0
```


Items collection


```
_id: ObjectId('61f191f16cc66fffeaeb1ae2')
title: "Tempura Bao Buns"
restaurantId: ObjectId('61f191f16cc66fffeaeb1ade')
description: "Description description description description."
price: 6.24
quantity: 10
category: "Bao Buns"
```

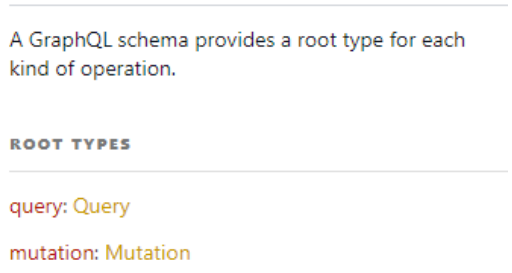

MVC architecture refactoring

To follow a more MVC-like project architecture (without the “view” part) and best practices an additional directory with controllers has been introduced in the project structure. These controllers contain the code that is responsible to call various db actions and return a response. This code was previously located in the routes directory inside each endpoint callback function. Now each route file is responsible on which controller needs to be called to return the appropriate response/do mutations in the database.

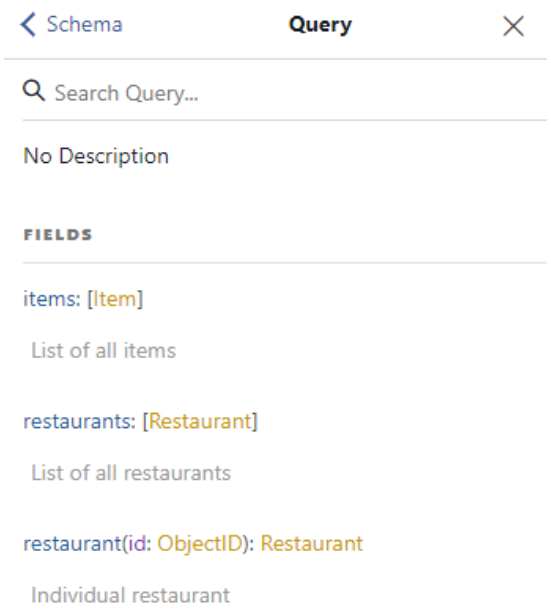
GraphQL middleware/endpoint

Aside from the REST APIs that were previously developed a new GraphQL endpoint was introduced that was later used in the client to consume data. In the “types” directory the types of all needed properties in the schemas are defined. The “schema” directory is divided into queries and mutations in order to get and mutate data respectively. Having strongly typed the properties and the connections between them, a documentation of the graph is being displayed on the /graphql endpoint. A quick overview of this can be found bellow;

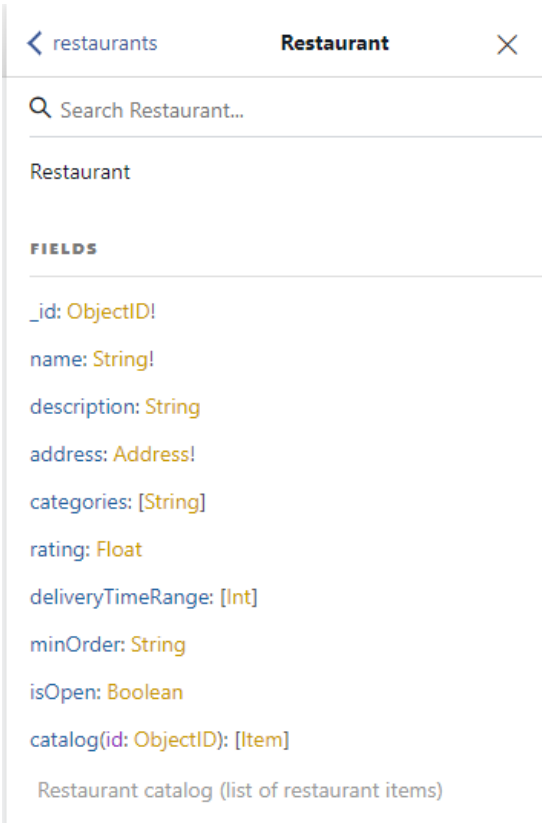
Root schema (queries/mutations)



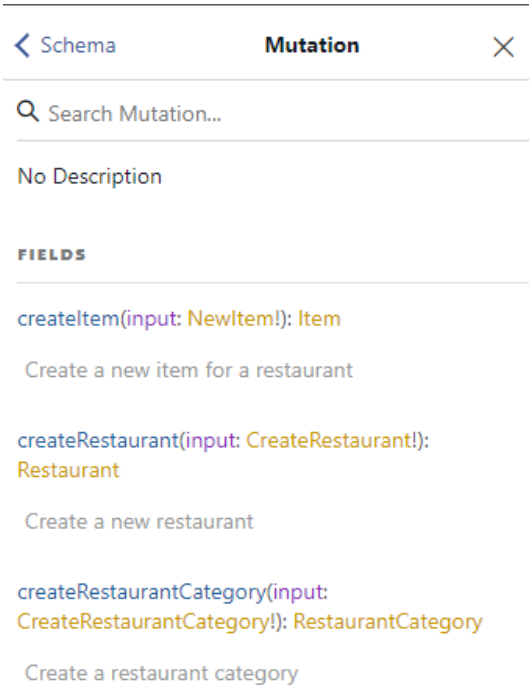
Query schema



Restaurant type



Mutation schema



5. Client Next.js/TypeScript/GraphQL migration

Next.js/TypeScript migration

The approach on the client side was to first setup the project structure for Next.js and TypeScript, while also keeping all the files inside the project. The pages were moved in a temporary directory in order to gradually consume them in the application. TypeScript configuration was used from the start but the components and pages remained in javascript for the first stage. When a component, a file or a function needed to be drastically changed then this new file was written in TypeScript.
Taking into consideration the rendering/generation methods that have been thoroughly presented in the 2nd chapter (Technologies and benefits) it was decided;
The homepage would be generated with ISR (Incremental Static Regeneration) so that it would be regenerated after 30 seconds.
The filtered restaurants page would be rendered on the client side (CSR with SWR hook).
The individual restaurant pages would be generated with ISR so that it would be regenerated after one minute.

GraphQL queries

To make queries to the graphql endpoint and get the appropriate data from the graph, two mechanisms were implemented. In both cases we used methods and providers from a popular GraphQL (React) library named @apollo/client. For the server side queries used to fetch data when generating the appropriate pages a custom hook (gqlQuery) was implemented based on the client.query method by the apollo client. For the client side queries the useQuery hook was used. Using the graphql queries we were able to fetch only the data that was need for the store overview on the homepage (improving overfetching), as well as not having to call two apis on the restaurant page to get the the restaurant data with the catalog (improving underfetching).

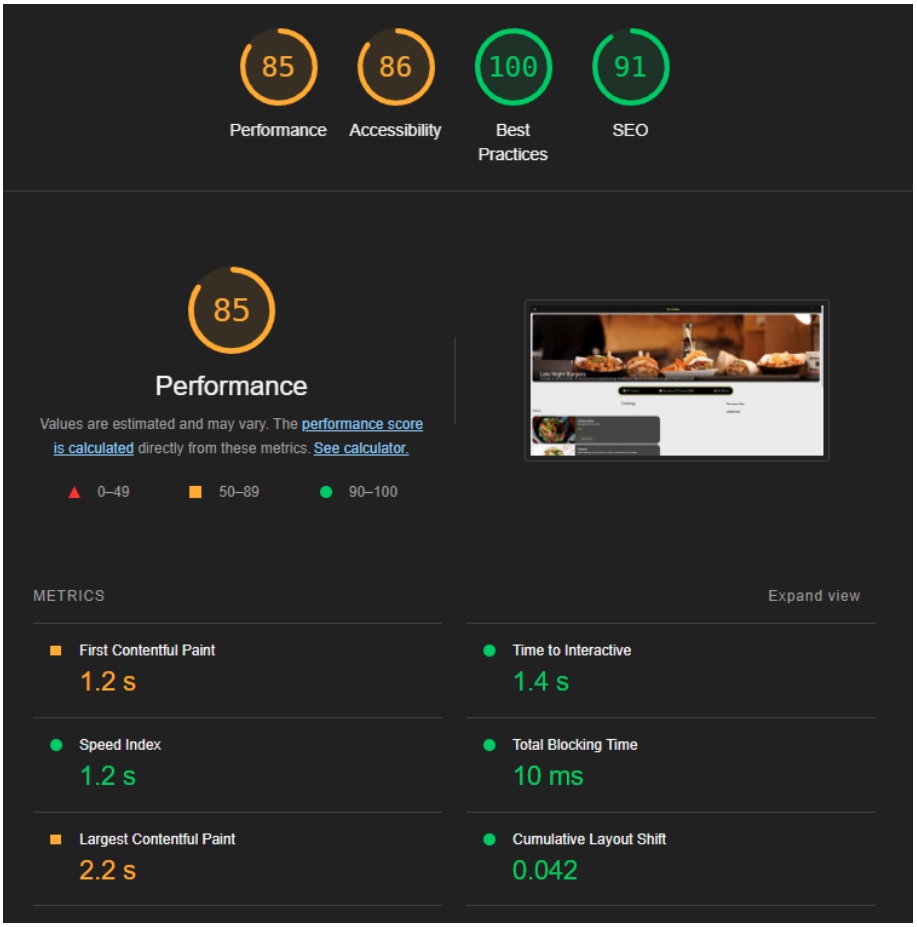
State management

Another point that is worth mentioning is how the state is being handled in this framework and SSR logic. Most of the time it is encouraged to consume and display the data passed as props to the components directly in the components. Sometimes and for better scalability it is needed to store these data as state in a store as usual

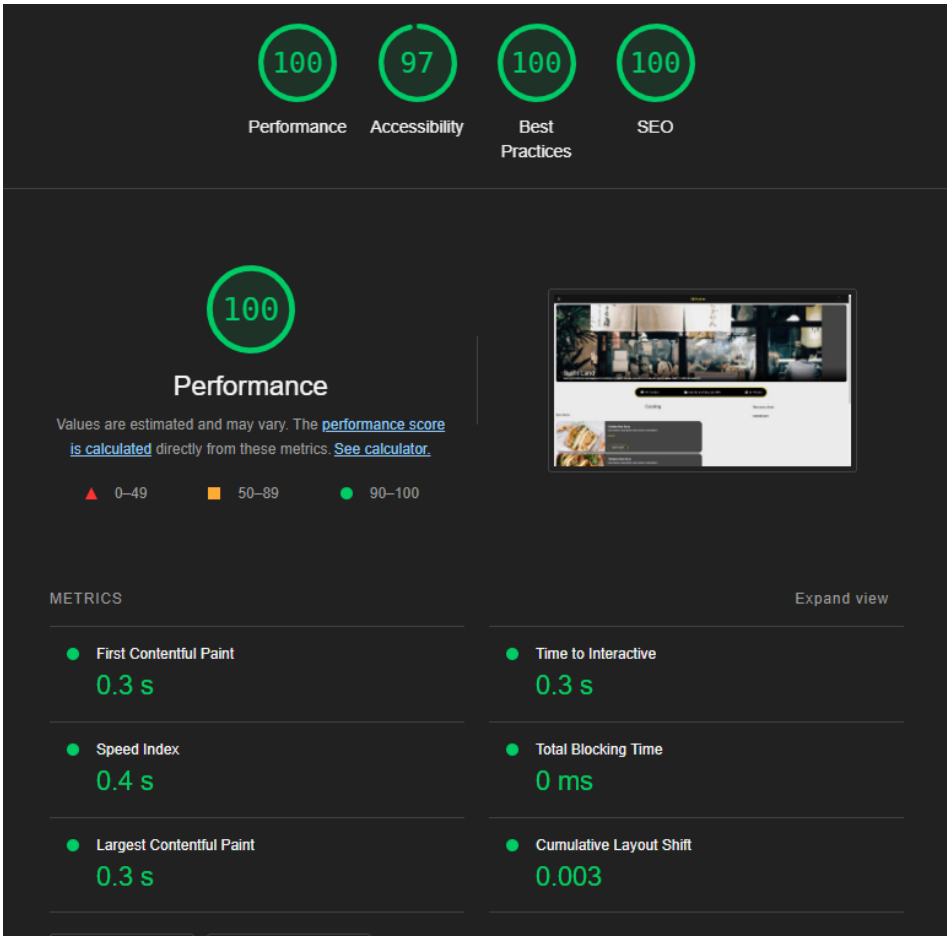
to avoid prop drilling, reusability etc. The issue here is that the data that is being fetched on the server side needs to be synchronized with the state of the store on the client side. Therefore to tackle this problem we essentially create a store on the server when fetching data from an api for example and then sync the client state by hydrating the client store on the first render of the UI. This process is handled by a package named next-redux-wrapper.

Metrics CRA/Next.js comparison

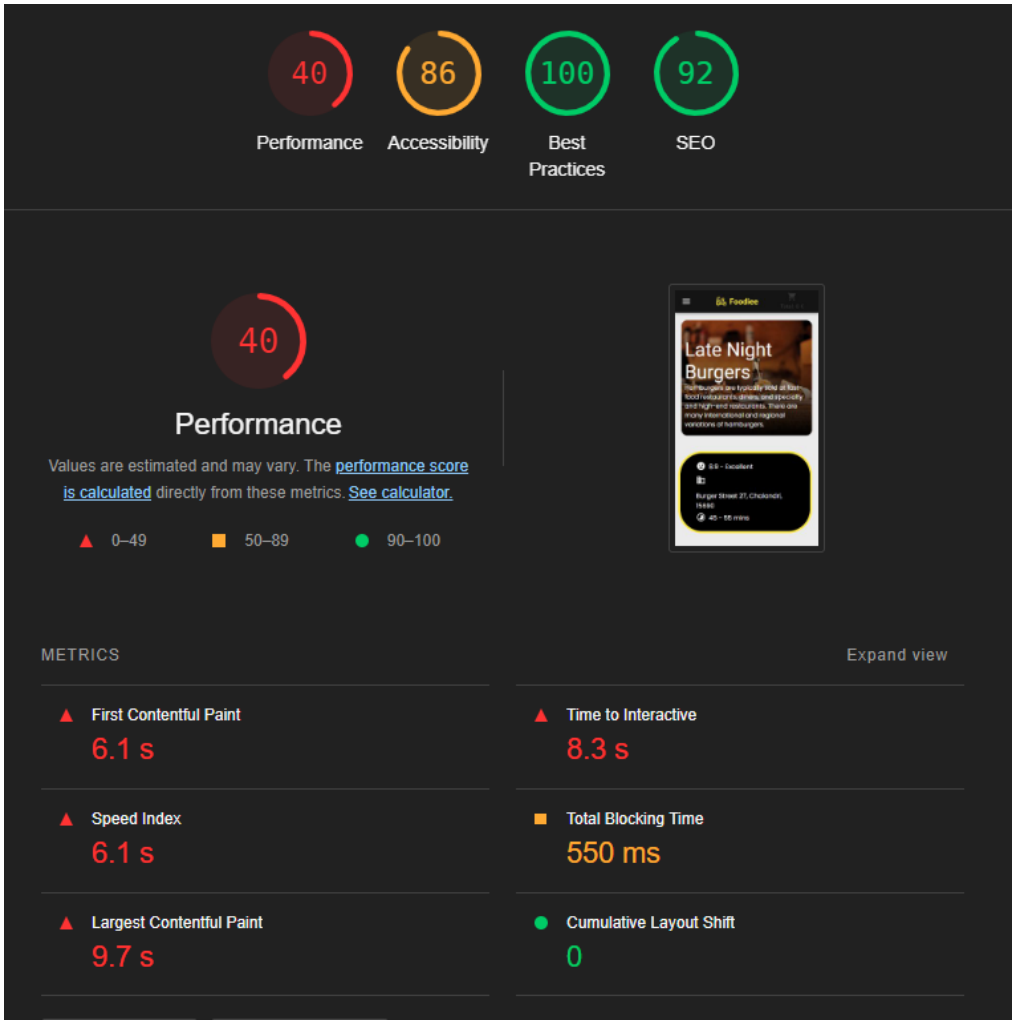
Below can be found screenshots from metrics measured with the Lighthouse tool where the significance of all the above optimizations is profound. Performance of the overall page is vastly improved (mainly due to SSG and Next's component optimizations) and especially on mobile devices.



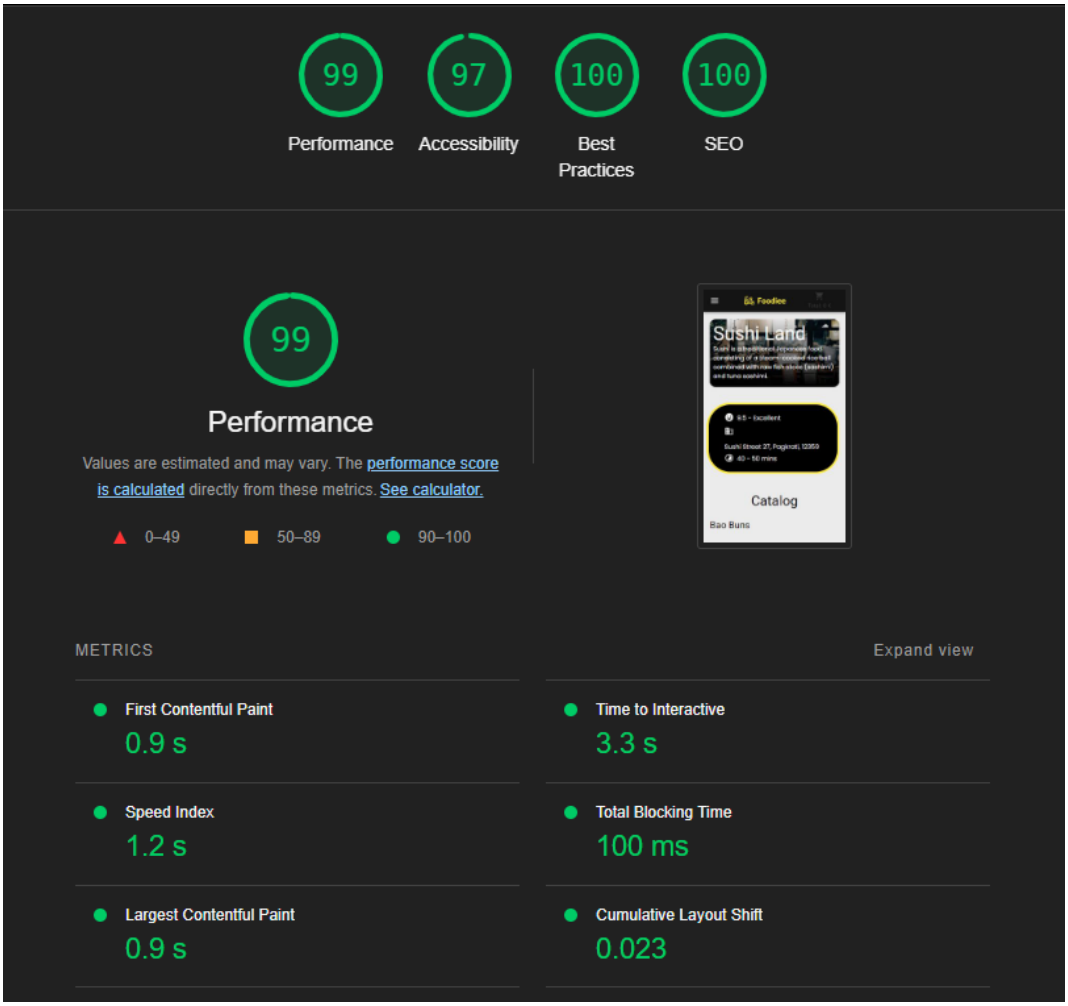
Desktop (CRA/Vite CSR)



Desktop (Next.js)



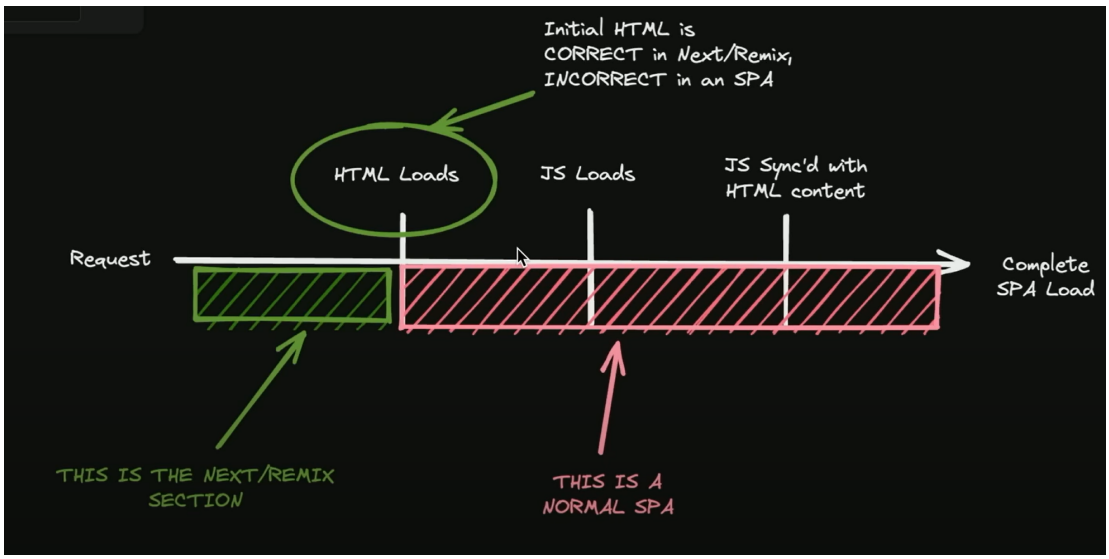
Mobile (CRA/Vite CSR)



Mobile (Next.js)

6. Conclusion

Next.js as a framework seems to apply good practices and optimizations behind the scenes with its own components and functions (generation methods) resulting in overall better performance and SEO without the extra effort from the developer (most of the time at least). It has a great impact on the initial part of the html loading of the page that is being requested, a major disadvantage for CRA or other client side rendering tools (see image below from Theo's Youtube video).



Next.js and React are constantly evolving as Next.js already released v13 and React has announced that from the next update will be server first. Even though Next.js 13 and the async use hook weren't used in this project as they are still in beta and documentation wasn't completely finished, it is definitely worth taking a look at them when they become stable.

7. References

https://www.typescriptlang.org/
https://nextjs.org/
https://graphql.org/
https://theodorusclarence.com/blog/nextjs-fetch-usecase
https://www.conventionalcommits.org/en/v1.0.0-beta.2/#summary
https://leanpub.com/mongodbschemadesign/read
https://www.youtube.com/watch?v=d2yNsZd5PMs