Nhan Nguyen Thien

FAMILY IN MUSIC PLATFORM

NextJS Full Stack Web Application

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Degree Program in Information Technology

ABSTRACT

Author	Nhan Nguyen Thien
Title	Family In Music Platform
Year	2022
Language	English
Pages	47
Name of Supervisor	Mikael Jakas

In this thesis, the objective was to develop the Dashboard page feature with the design and logic for implementation from FAIM's product team to replace current landing page of the platform and the current landing page would become What We Do page.

Subsequently, the Dashboard page was introduced as a current landing page of the platform as well as there was an issue had to be tackled which was the performance of that page when numerous APIs were called in run time when using getServerSideProps from NextJs.

The Dashboard has been released and needs to be maintained whenever some bugs or issue received from customer's feedback. In the future, the product team is having an idea to create new section which called "Wanted Advertisements" section to replace current "Looking fors" and "Services" section in landing page.

Keywords Family In Music Platform, NextJs, Full Stack Web Application

CONTENTS

LIST OF FIGURES

LIST OF CODE SNIPPET

LIST OF ABBREVIATIONS

APIs	Application Programming Interfaces
AWS	Amazon Web Services
CMS	Content Management System
DOM	Document Object Model
EDM	Electronic Dance Music
Env	Environment
FAIM	Family In Music
GraphQL	Graph Query Language
HTML	Hyper Text Markup Language
IDE	Integrated development environment
ISR	Incremental Static Regeneration
JS	JavaScript
MVC	Model-View-Controller
MVCC	Multiversion Concurrency Control
MVP	Minimum viable product
ORM	Object–relational mapper
OP	Operating system
SEO	Search Engine Optimization
SPA	Single Page Application

SSR	Server-side rendering
SSG	Static Site Generation
SQL	Structured query language
SWR	Stale while revalidate
UI	User Interfaces
URL	Uniform resource locator

1 INTRODUCTION

1.1 Background

Every day, more new music is being written and released around the world than ever before in music industry, and it is expected to keep increasing significantly in the future. This has created a rapidly expanding group of new independent creators who often do not fit the existing structures of the industry and whose needs are mostly unmet or underserved. That is why FAIM company is building a new Family In Music Platform for all different kinds of creators as well as people from different roles in music industry such as: Artist, Event Organizer, Music Producer, Advisor as well as with different kinds of genres such as EDM, Pop, Hip hop, Singer-songwriter and some more popular genres. These people from the newbie, and the inexperienced to the seniors or the celebrities have a mission to create connections with others, motivate, educate and elevate. This platform is offering networking, lessons and tools, featuring distribution and release workflows, and with more to come including campaign monitoring, analytics, song title validation, song registration, radio play monitoring, daily payments and royalty advances. Significant features of this platform include:

- Distribution: Distributing music to all leading digital music stores and major streaming services including Apple Music, Spotify, Tik Tok and Instagram which will revolutionise the way these creators work and get paid for what they have contributed to the music industry.

- Knowhow: Lessons from professional and experienced people in music industry who will share their experience, valuable knowledge about how to write, produce and promote music that can help other people, especially new creators just step ping into entertainment industry to gain their own success.

- Workflows: Showing step by step and instructions which help new creators to follow which help their next released albums or songs become smoother. Creators can build their marketing plans or PR campaigns, using FAIM's platform templates, to take all the guesswork out of what should be doing and when.

- Networking: Users can make connections and collaborate with creators and industry professionals as well as discover new opportunities in the platform. They first need to send the requests and waiting for them to accept and then, they can start texting and sharing experience.

- MgNTa: A new industry identifier that supports and empowers the new generation of creators by helping create a pathway to rights, control and revenue. Users will be able to be identified as songwriters and recognised when their music is being used on digital services.

-Wallet: Users can access daily royalty payments and smooth royalty splits for creators. In addition, they can be paid directly on their digital wallet and cash out anytime they want.

For faster development, this MVP platform is built in a monolith architecture. In the case of scalability issues a likely development is to make most popular services independent as they gain traction. Below are some images related to the FAIM platform architecture.

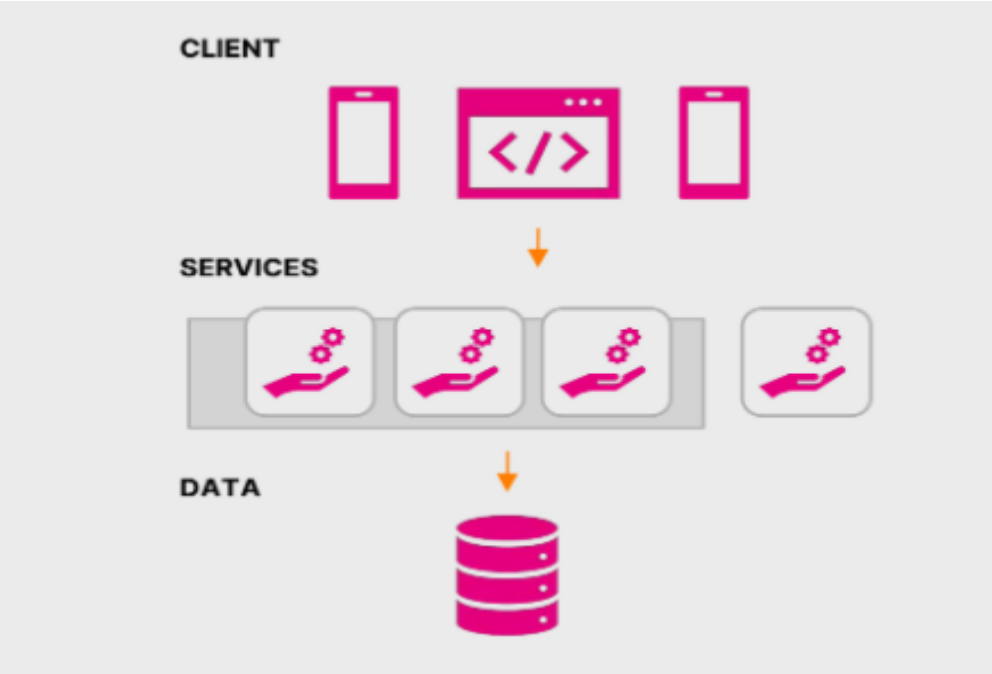


Figure 2. Monolith MVP Platform

In the future, features such as Know How, Workflows and Networking will greatly benefit of mobile clients (iOS, Android). In addition, when going to the global platform, millions of users will eventually require distributed services or microservices infrastructure.

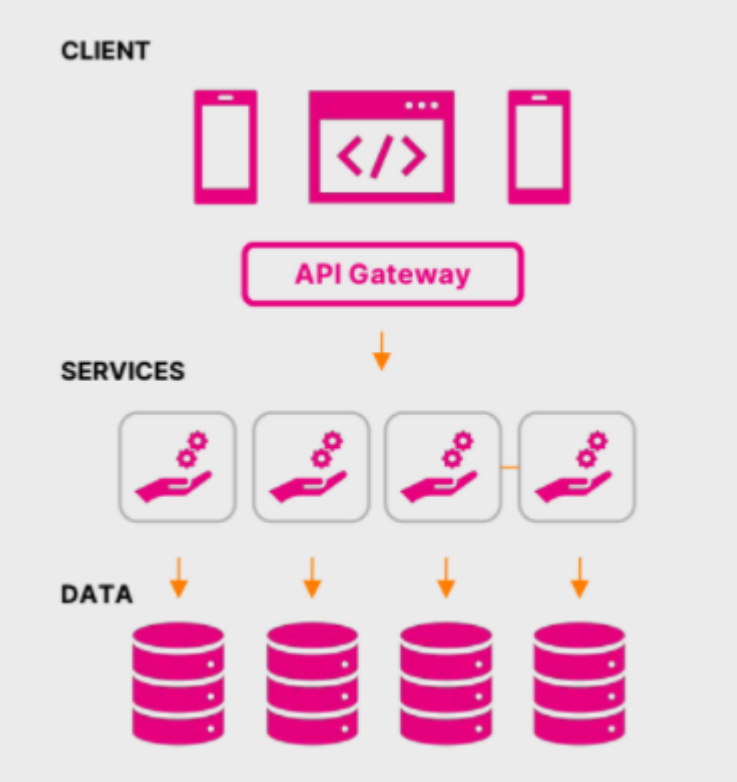


Figure 4. Possible future architecture

This is the data architecture and key software tools which are used for the FAIM platform.

At the moment, this platform is the beta version and new features are constantly being developed and they are willing to hear the feedback to improve their platform to be a better version.

1.2 Motivation

As the platform is developing significantly and a large number of users come to the platform for expanding their connections and gain their success, the motivation to create a Dashboard page to become current landing page and the current landing page will become the What We Do page.

Figma is a collaborative interface design tool which works on any OS and helps design the UI with stylings and exact ratios for each element from designers and developers can rely on it and start developing the UI for website.

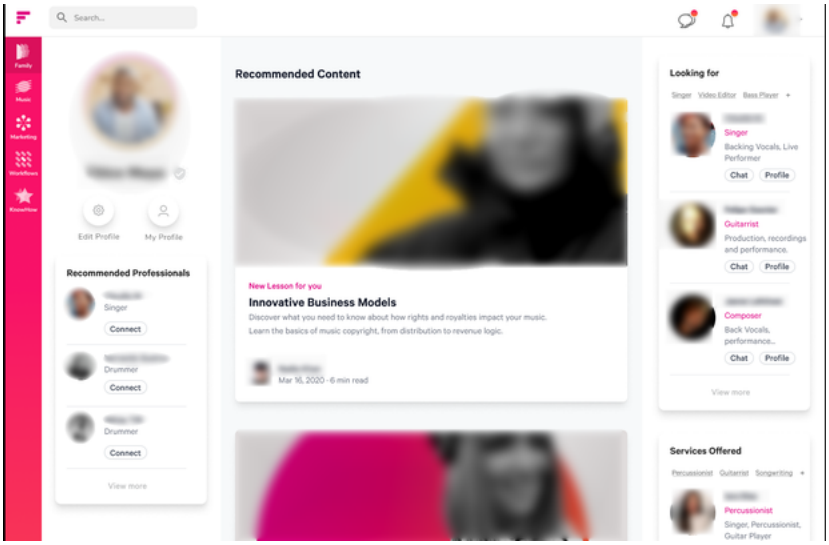


Figure 7. Dashboard page design from Figma

The idea of Dashboard page is as follows:, there are 4-four sections in the page:
+ The first section is about useful and helpful lessons displayed in the center of the page shown as “Recommended Content” for users who could learn about music industry from seniors. These lessons from Knowhow feature of the platform.

+ The second section is on the left side of this page; the current user can view their profile or edit their own profile. Below that section, there is the “Recommended Professionals” section which displays randomly three users each time the Dashboard page is refreshed or reloaded or when navigating back to this page from other pages. This section lets users easily connect with each other.

+ The third section is “Looking for” section on the right of the page which shows three random tags from current user tag lists about roles or services. Below that, there are three random people provided that match the request.

For example, if one user is looking for artists to collaborate with, this section will show random three users who are artists.

+ The fourth section is “Services” section on the right of the page which shows three random tags from the current user tag lists about his/her own roles and services that he/she is working at the moment. Below that, there are three random people who are looking for roles and services that match their current role.

For example, if one user is an artist, this section will display random three users who are looking for artists to collaborate and produce new music.

In addition, the “Recommended Professionals”, “Looking for” and “Services” sections have a “View more” button which can help the user to have more options when finding suitable and appropriate to work and make connections with.

2.TECHNOLOGIES

2.1 ReactJS

ReactJS (known as React) is an open-source front-end JavaScript library which has considerable supports from the developer community. This JavaScript library was first invented on May 29, 2013, by Jordan Walke and is maintained and developed by Meta (formerly known as Facebook). Nowadays, this library has over thousands of contributors according to its popularity and reputation. The most critical purpose of ReactJS is to improve the speed of the applications and require a minimal coding effort when developing web applications. There are numerous websites which have been developed and built with React, such as Netflix, Facebook, PayPal, Twitter and many other web applications. /1/

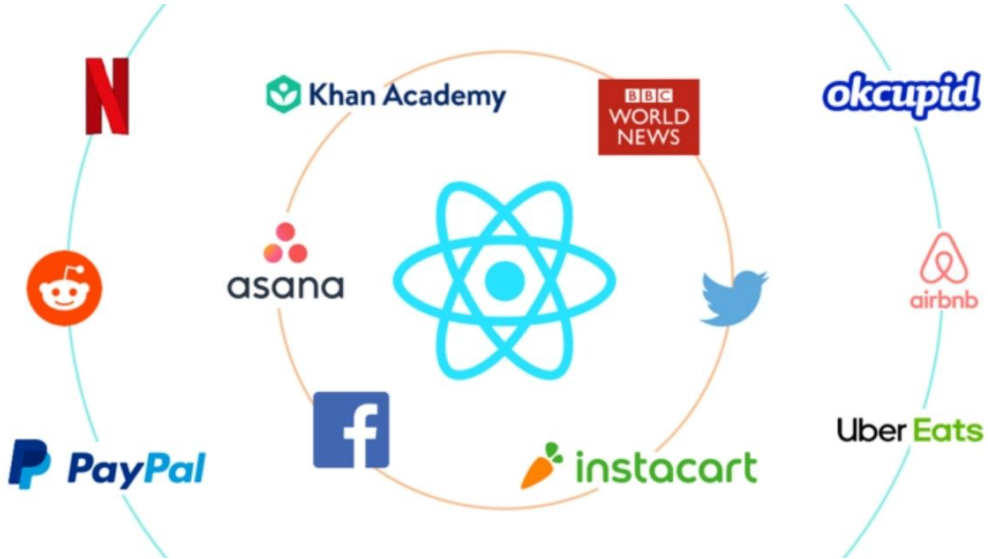


Figure 8. Popular websites built with React /2/

React can be used for building interactive and user-friendly user interfaces by dividing into multiple components and each of them has its own logic, controls and functionalities that developers can reuse effectively.

In addition, React is used to build SPA which can enhance the performance of web applications based on the virtual DOM. When building a SPA by React, for every DOM object, there is corresponding “virtual DOM object”. Basically, the DOM represents the UI of the web application. Additionally, a virtual DOM object is a

representation and lightweight copy of a real DOM object. ReactJS has solved the issue of the slow update of DOM by updating the virtual DOM objects when the state of a component changes first, once the virtual DOM has been fully updated, React updates those objects in real DOM./3/

Furthermore, JavaScript frameworks has an issue with SEO-friendly traditionally. In general, the Google crawlers, which is one of the most popular search engines cannot read JavaScript-heavy applications. With React, there is no such problem because the applications written by this library can run on the server and the virtual DOM will be rendering to the browser as a regular web page, which helps numerous developers to be navigated in different kinds of search engines.

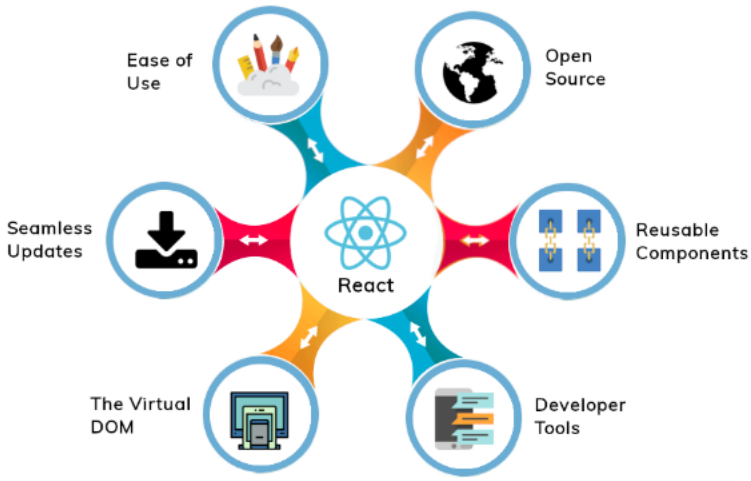


Figure 9. Advantages for websites built with React /4/

In contrast to these significant benefits, there are some drawbacks. A critical drawback of React is that it covers only the UI layers (known as the View part in MVC model) so that we need another framework or library to collaborate with in order to get the complete tools for developing a project. That is why NextJS was invented to serve the purpose of building a full-stack web application from front-end to back-end.

2.2 NextJS

NextJS is an open-source web development framework invented by Guillermo Rauch and first released on October 25, 2016, on GitHub, and currently being

maintained and developed by Vercel developers and open-source community. This is a React framework built on top of NodeJS and it has several remarkable features, including server-side rendering and generating static websites. /5/

Firstly, server-side rendering (SSR) using getServerSideProps function is the ability of a web application to prepare all the content of a page on the server instead of on the client side. When inspecting the source code of a loaded application built with standard React, it is noticed that the page is empty right from the beginning. It only displays a basic HTML skeleton and since React is a client-side Javascript library, all that rendering occurs on the client, so in the browsers, it is not happening on the server. It can be an issue when users must wait for the response of outgoing requests (APIs called for fetching data in client side) because the page we requested did not include any data yet. In addition, the problem can occur with search engine optimization for public-facing pages with a lot of content, which should be found by crawlers or search engines, but it will not be a problem with administration dashboard pages required logging in. These issues can be solved by using server-side rendering in NextJS. In case the page would be prerendered on the server or the data fetching process can be completed on the server. hen users send requests to that page, then the finished page would be served to our users and search engines. Subsequently, search engines crawlers would see the content of the page as well as users would not have to wait for flickering loading state.

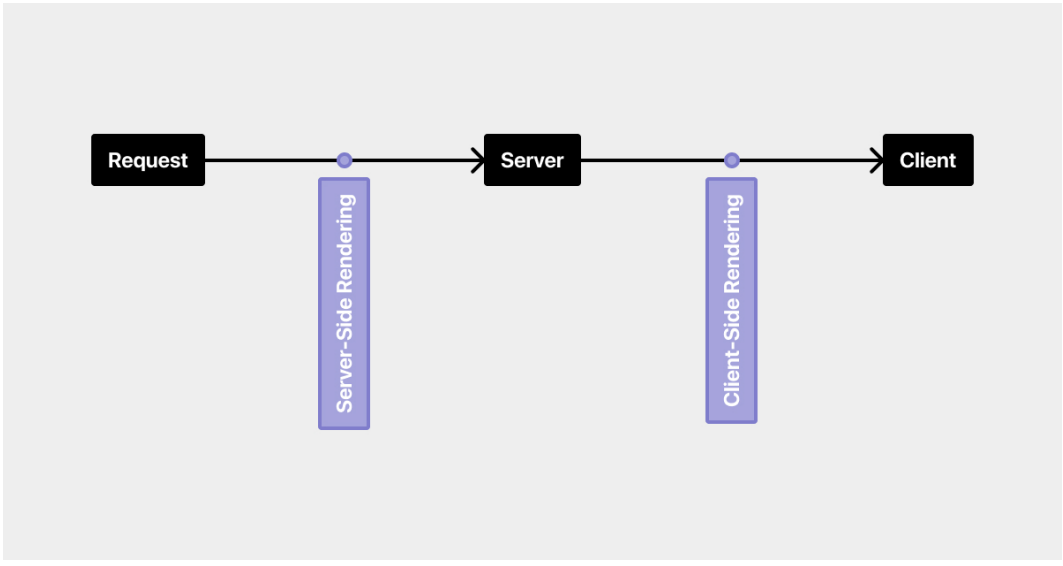


Figure 10. Server-side rendering /6/

An additional feature about NextJS is static page generation using getStaticProps. SSG is a preferred way of rendering the content of websites or applications at build time and the output of it is a set of static files which include the HTML files itself as well as assets such as CSS and JS. When the browser receives the page built with Javascript libraries such as React or scripts at run time in the browser, it is usually simple HTML without much content. This then loads the scripts to pull the content into the page, known as hydration. But with SSG, tools like NextJS, attempt to render that page mostly as it would in the browser, but at compile time, also known as build time. This allows us to serve the whole content on the initial load. During this procedure, the scripts continues to hydrate the page, but with fewer or no changes.

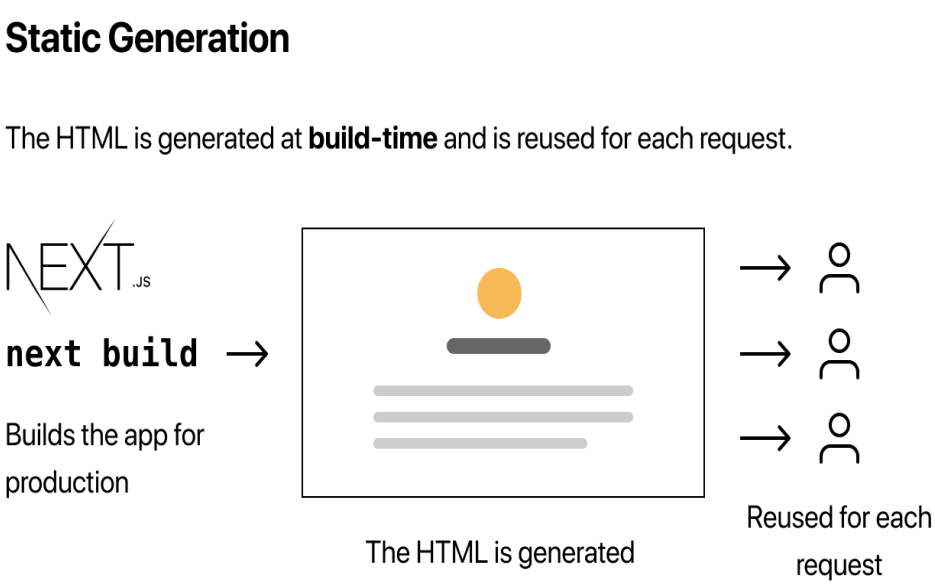


Figure 11. Static site generation /7/

With the updates of NextJS in version 9.5, Incremental Static Regeneration concept allows developers to update existing static pages by doing re-rendering process in the background as requests comes in. It makes the static content also become dynamic. Users' traffic is always delivered from the static storage, although the page content can be revalidated at predefined intervals (in seconds). The regeneration frequency is controlled by the revalidated configuration flag./8/

```
export async function getStaticProps({ query }) {
  try {
    const { data } = await
axios(`https://api.github.com/users/${query.user}`);

    return {
      props: data,
      revalidate: 1, // 👈
    };
  } catch (error) {
    console.error(error);
  }
}
```

Figure 12. SSG with ISR method /6/

2.3 Sanity

The Sanity Studio is a React.js-based open-source CMS which allows for quick configuration as well as free-form customization. Utilizing these toolkits and plugins to design a workflow that is optimized for the way users wish to work with material. /9/

In the FAIM platform, Sanity is containerised separately, working like a micro service. The app container fetches data from there and it builds its own APIs. It is used as the workspace or backend which contains data for lessons and non-developer users can easily add new, edit or delete lessons and do not need to have any knowledge about technical things. In addition, the mission of developers with Sanity is to create additional custom fields by coding so that non-developer users can add the fields they want with the main objective to display the desired UI in the platform.

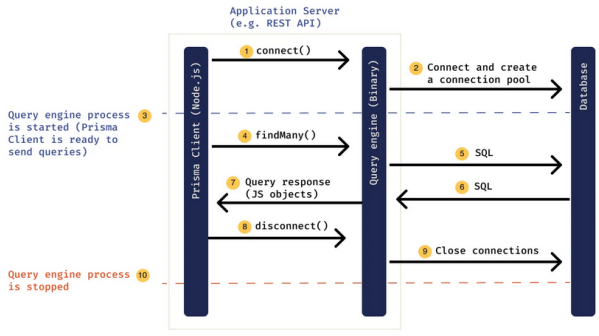
FAIM has 2 Sanity studios, Family in Music Staging and Family in Music Production. Users can add draft versions of lessons on the Staging Studio and display in the Staging server first, then if the lessons are accepted by the product team in the company, they can be deployed to Sanity Production by command: sanity deploy.

2.4 Prisma

Prisma is a next-generation object–relational mapper (ORM) that claims to speed up development and reduce mistakes. In comparison to traditional ORMs, Prisma takes a unique approach to ORMs.It makes use of a Schema Definition Language (SDL) that creates type-safe code and writes migrations automatically. Prisma does not employ classes or decorators for model definition, unlike TypeORM and MikroORM. Instead, it leverages schema-driven code creation, to GraphQL Code Generator tool. For stronger type safety and powerful IDE auto-completion capabilities, the Prisma Client uses this produced TypeScript code.

Compared to Prisma version 1, which required a custom server running in a container to transform calls to database queries, with the update of version 2, Prisma 2 completely changes the approach, using a combined binary. /10/

Prisma manipulates so that, it takes the models written in the schema and produces SQL migrations and types in node modules/.prisma/client using the prisma migrate command. These types are aware of the relationships established in the models and generate code that may be used to query the models. How to develop models and utilize them with the Prisma client will be explained in more detail in the section following. When the client is used to query, it sends the requests to a Query Engine binary, which optimizes them and turns them to database queries.



Describes the Prisma architecture. Source: Prisma Docs

Figure 13. Prisma Architecture

Although the developer will never interact with Prisma's engine when working with it, it is useful to understand how Prisma works. The engine handles all communication with the database layer. The engine's optimization has several advantages, one of which is that it overcomes the N+1 relationship problem, which is frequent in GraphQL applications.

Figure 13 describes the flow of Prisma architecture which shows how the query engine process is started and connects to the query engine which connects directly to the database. After getting into the database, Prisma returns the API and we can get the API from the back end in server-side to display in front-end. /11/

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
  previewFeatures = ["fullTextSearch"]
}
```

Figure 14. Prisma configuration

The schema.prisma file is the most significant since it is the PostgreSQL that we will use to create our models. In figure 14, we configure the datasource with the provider PostgreSQL and URL with the link env. We're specifying our data source and informing Prisma that we'll be utilizing PostgreSQL in these lines. The environment variables will provide the URL for it. Finally, the generator informs Prisma of the language that will be used with the Prisma client. It will be JavaScript in this scenario.

```
62  model Instructor {
63    createdAt           DateTime @default(now())
64    updatedAt           DateTime @updatedAt
65    id                  String   @id @default(uuid())
66    slug                String?  @unique
67    active              Boolean @default(true)
68    description         String?
69
70    user   User?   @relation(fields: [userId], references: [id])
71    userId String?
72  }
```

Figure 15. Prisma Model

We will now create our first model to illustrate this.

The name of a table in the database corresponds to the model. The model's name should follow the following naming conventions:

- Must begin with an alphanumeric letter and continue with numbers, letters, or underscores.
- It must begin with a letter and is usually written in Pascal Case.
- Should be written in the singular (for example, User instead of user, users or Users)

Prisma does not immediately update the database just by writing the model. We will have to build a migration and test it against our database. This may be done in development using the following command:

npx prisma migrate dev --name init

In the Prisma folder, a new folder called migrations will be created. Prisma will generate a PostgreSQL file that will be automatically executed against the database throughout development. Prisma also provides type declarations for all the models that may be utilized with the TypeScript Prisma Client.

2.5 PostgreSQL

PostgreSQL is a sophisticated, open source object-relational database system that employs and extends the SQL language. It is used as a replacement for raw

SQL or other database access tools like SQL query builders (like knex. js) or ORMs (like TypeORM and Sequelize).

Transactions with Atomicity, Consistency, Isolation, and Durability (ACID) qualities, automatically updatable views, materialized views, triggers, foreign keys, and stored procedures are all available in PostgreSQL.

It can manage a wide range of workloads, from single computers to data warehouses or Web services with a large number of concurrent users.

PostgreSQL includes a number of features targeted at assisting developers in the development of applications, administrators in the protection of data integrity and the creation of fault-tolerant settings, and the developer in the management of the data, regardless of the size of the dataset. /12/

Many advanced features are available in PostgreSQL in other enterprise-class database management systems, such as:

- Types defined by the user
- Inheritance in tables
- Locking mechanism with a high level of sophistication
- Referential integrity of foreign keys
- Views, rules, and subquery are all terms that can be used to describe something.
- Transactions nested
- Controlling concurrency across many versions (MVCC)
- Replication that is asynchronous

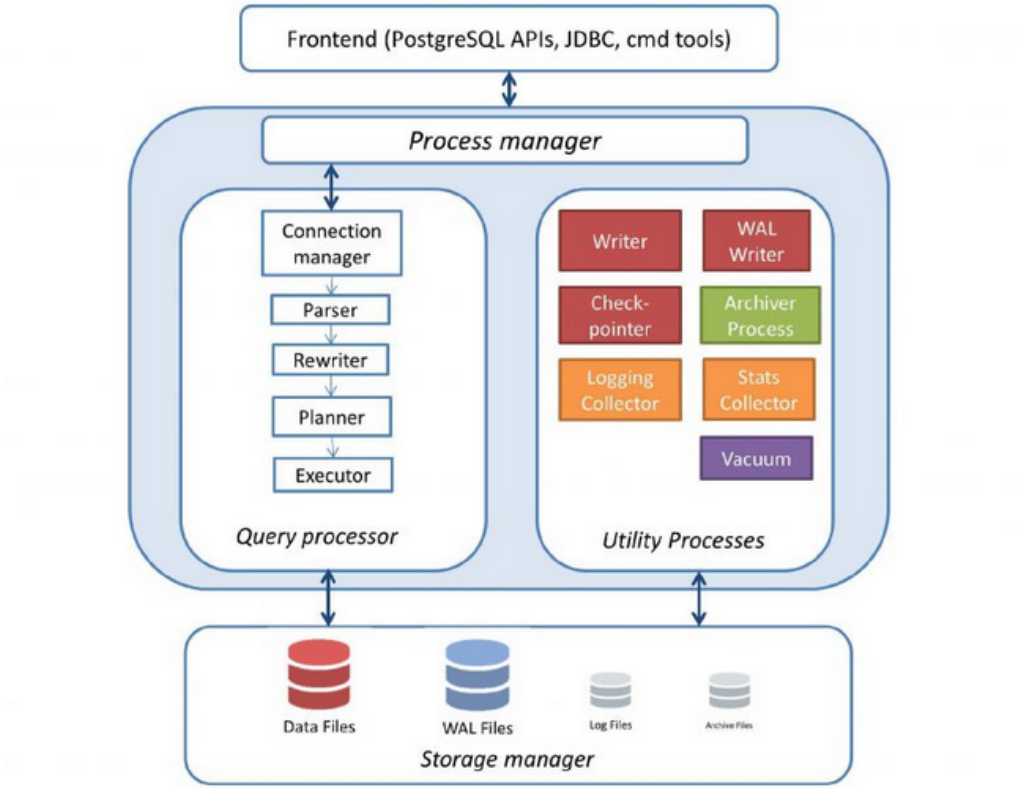


Figure 16. PostgreSQL data structure

For the processes that a PostgreSQL structure, it must go through in order to get results:

- A connection to the PostgreSQL server must be made from an application program.
- The parser stage validates the query sent by the application program for proper syntax before constructing a query tree.
- The rewriting system looks for any rules (stored in the system catalogs) that can be applied to the query tree formed during the parser step.
- The transformations specified in the rule bodies are carried out by it.
- The development of views is one application of the rewriting system.
- When a query is run against a view (i.e., a virtual table), the system re-writes it in order to obtain access to the data table.
- The planner/optimizer builds a query plan from the query tree, which will be passed to the executor. It accomplishes this by first generating all feasible paths that lead to the same outsource.

- The executor traverses the plan tree in a recursive manner, retrieving data in the order specified by the plan. While scanning relations, performing sorts and joins, evaluating qualifications, and finally returning the rows, the executor uses the storage system.

2.6 TailwindCSS

Tailwind CSS is a utility-first CSS framework that allows to create and custom user interfaces quickly. It is a highly configurable, low-level CSS framework that offers all the building blocks needed to create personalized designs.
The advantages of using Tailwind CSS include:

- Fastest UI building process appropriate with ReactJs and NextJS
- It is a utility-first CSS framework which means utility classes can be used to build custom designs without writing CSS as in traditional approach.

There are many ways to use Tailwind CSS to apply in the project, but the easiest way to integrate in the project is installing Tailwind via npm with the command npm install Tailwindcss

After installing package in terminal, Use the @tailwind directive to inject Tailwind's base, components, and utilities styles into your CSS file.
@tailwind base;
@tailwind components;
@tailwind utilities;

Now we can apply it in the project by using class name from tailwind CSS.

In the second method, we can add it through CDN link from Tailwind, adding it directly to the folder public index.js of the project.

```
<link href="https://unpkg.com/tailwindcss@^1.0/dist/tailwind.min.css" rel="stylesheet">
```

Figure 17. Link CDN of Tailwind CSS

But when using with CDN some drawbacks need to be considered:

- Theme of Customize Tailwind default cannot be used.
- Directives like @apply, @variants cannot be used.
- Cannot install third party plug-ins.

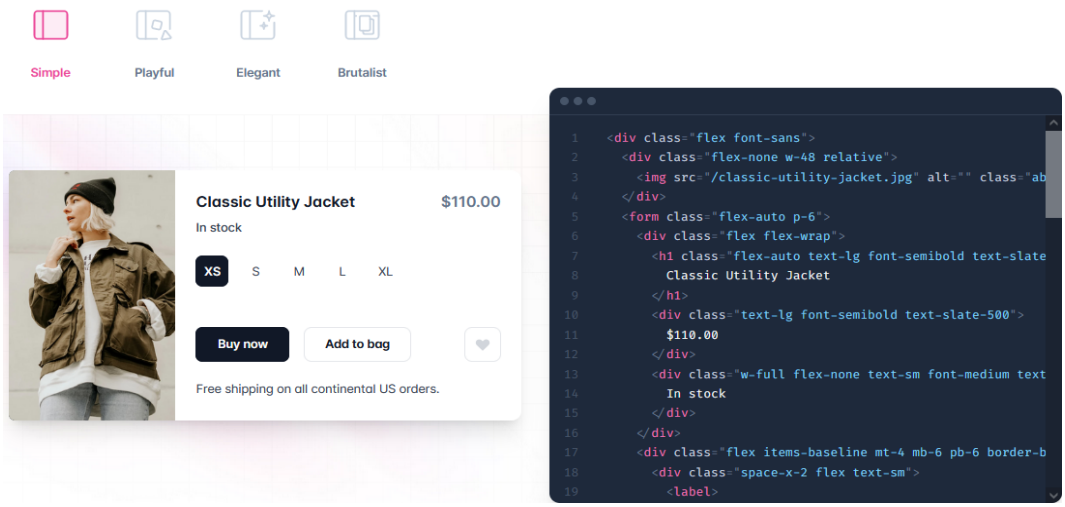


Figure 18. Example of Using Tailwind CSS /9/

Figure 18, shows this is an example of using Tailwind CSS class names to style the component. It is fast and easy to use, the result we get from styling is extremely eye-catching. /13/

2.7 Lodash

Lodash simplifies JavaScript by removing the tedium of dealing with arrays, numbers, objects, texts, and other data types. Great methods for Lodash's modular are iterating arrays, objects, and strings or creating composite functions. /14/

3.ENVIRONMENT SETUP

3.1 Installing Required Tools for Development

For setting up the development environment, some tools must be installed such as:

-Visual Studio Code which is a free coding editor that helps developers code in any programming language. After installing it, some plugins can be added for helping coding process easier: /15/

- ES7+React/Redux/React-Native snippets which support React syntaxes and developers only need to enter the snippets then click Tab button for displaying full syntax for that snippet. /16/

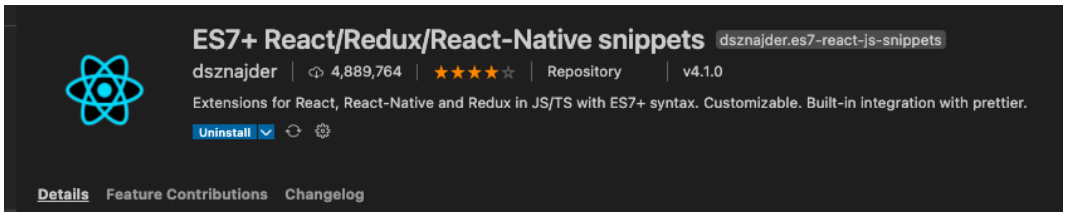


Figure 19. React snippets

- Prettier ESLint snippet which help developers format their code opinionatedly. It enforces a consistent style by analyzing their code and re-printing it according to its own standards, which take into consideration the maximum line length and wrap code as needed. /17/

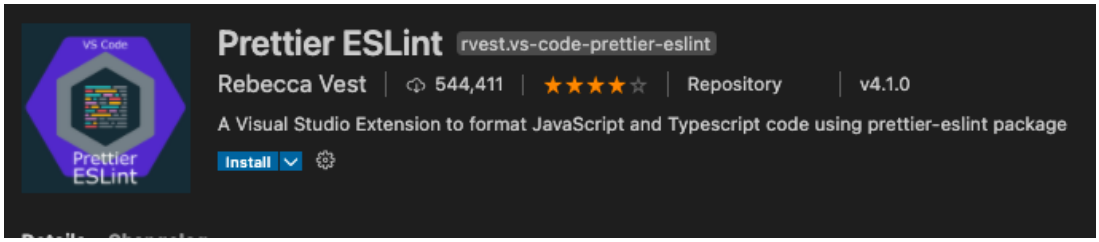


Figure 20. Prettier ESLint snippet

- GitLens is an extension for checking quickly whom, why, and when a line or block of code was changed and how those codes evolved. /18/

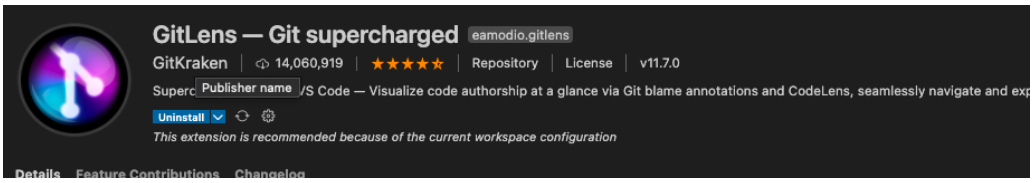


Figure 21. GitLens snippet

- Postman, which is a tool for testing APIs, after creating an API, Postman will support developers to test if that API work or have to make changes something. /19/

- SquirrelSQL which is a tool for connecting to local, Staging or Production database so that developers can test or do some queries directly in database. /20/

3.2 Environment Files for Development from AWS

After installing essential tools for development process, the FAIM platform project needs some .env files for running the project as well as for deployment process. There are three .env files have to be filled:

- . env: This is a file for local development and it can be filled in by asking former developers or from README.md file of the project
- Staging.env: This file is for staging environment and it can be filled by accessing to Staging Amazon Web Services of FAIM and retrieve secret values from Secret Manager section. These values also include keys for deployment to Staging server.
- Production.env: This file is for production environment and the process is similar with staging environment.

3.3 Setting up Database for Local Development

For setting up local database, developer can get access to Confluence website where store all of the documents about the FAIM platform including features, environment setup, wanted features in the future. There are some steps for creating the local database:

- Firstly, get access to postgreSQL server in your local machine
- Then, start creating name of the database and user as instruction.
- After that, granting all privileges for that user.

- Next, add the URL to that database with correct credentials in web/prisma/.env .
- Finally, running prisma commands in order to connect between the project and local database. First command is : “docker-compose run web npm run db: generate” and then, run “docker-compose run web npm run db:migrate”.

Below is the figure about the procedure for creating local database from Confluence website.

Create the database and user:

```
su postgres
psql -p 5433 -h 0.0.0.0
postgres=# CREATE DATABASE FaimDb;
# substitute <PASSWORD> with your own
postgres=# CREATE USER FaimDbUser WITH ENCRYPTED PASSWORD <PASSWORD>;
postgres=# GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO FaimDbUser;
postgres=# GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO FaimDbUser;
postgres=# ALTER USER FaimDbUser CREATEDB;
```

You need to have **web/prisma/.env** file with correct credentials:

`DATABASE_URL="postgresql://FaimDbUser:<PASSWORD>@db:5433/FaimDb?schema=public"`

Run web service:

You'll need an .env file to the project main dir containing keys etc. It will be provided by the team (not through documentation for security reasons)

Depending on your setup, you may need to add *sudo* to docker commands

```
docker-compose run --service-ports web
```

Run migrations:

```
docker-compose run web npm run db:migrate
```

Figure 22. Steps for creating local database

Subsequently, the project can be started locally by running the command: “docker-compose up web”

4.IMPLEMENTATION

4.1 Creating Models for Lookingfors and Services Section from Prisma

The first step of creating the Dashboard page is to design the necessary fields and models in database. For this page, two more models are created which are LookingFor and UserTag.

```
101
102  model UserTag {
103    userId String
104    user   User   @relation(fields: [userId], references: [id])
105    tagId  String
106    tag    Tag    @relation(fields: [tagId], references: [id])
107    score  Int?
108
109    @@id([userId, tagId])
110  }
111
112  model LookingFor {
113    userId String
114    user   User   @relation(fields: [userId], references: [id])
115    tagId  String
116    tag    Tag    @relation(fields: [tagId], references: [id])
117    score  Int?
118
119    @@id([userId, tagId])
120  }
121
```

Code Snippet 2. UserTag and LookingFor model

The UserTag model, is for the connection between the User model and the Tag model and the UserTag is where users's roles and services they are working on are displayed.

The LookingFor model, is the similar connection as UserTag but the main objective is to display roles or services what users are looking for to collaborate and support their own career.

Additionally, these two models must be added inside the User model and the Tag model in order to complete the mapping between the User and Tag model.

```
80
81  model Tag {
82    createdAt       DateTime     @default(now())
83    updatedAt       DateTime     @updatedAt
84    id              String       @id @default(uuid())
85    type            TagType
86    name            String
87    slug            String
88    platformVisible Boolean      @default(false)
89    shortlisted     Boolean      @default(false)
90    userTags        UserTag[]
91    lookingFors     LookingFor[]
92
93    user   User?   @relation(fields: [userId], references: [id])
94    userId String?
95
96    parent   Tag?    @relation(fields: [parentId], references: [id])
97    parentId String?
98
99    children Tag[] @relation("TagToTag")
100 }
```

Code Snippet 3. UserTag and LookingFor added inside Tag model

```
web > prisma > schema.prisma
14  model User {
15    createdAt                 DateTime @default(now())
16    updatedAt                 DateTime @updatedAt
17    id                        String   @id @default(uuid())
18    authSub                   String?  @unique
19    email                     String?  @unique
20    slug                      String?  @unique
21    public                    Boolean  @default(true)
22    firstName                 String?
23    lastName                  String?
24    headline                  String?
25    location                  String?
26    avatarFileKey             String?
27    headerImageFileKey        String?
28    bioTitle                  String?
29    bioText                   String?
30    facebookUrl               String?
31    soundcloudUrl             String?
32    spotifyUrl                String?
33    youtubeUrl                String?
34    instagramUrl              String?
35    twitterUrl                String?
36    bandcampUrl               String?
37    websiteUrl                String?
38
39
40
41
42    networkingEnabled         Boolean  @default(true)
43    tourVirgin                Boolean  @default(true)
44
45    userTags                  UserTag[]
46    tags                      Tag[]
47    chatParticipations        ChatParticipant[]
48    connectionParticipants    ConnectionParticipant[]
49    videos                    Video[]
50    lookingFors               LookingFor[]
51    lessonProgresses          lessonProgress[]
52    campaigns                 Campaign[]
```

Code Snippet 4 . UserTag and LookingFor added inside User model

4.2 Creating Logic for Matching Lookingfors and Services

After completing creating essential models, the next step is to handle Prisma queries to do get, create or delete actions with the database to process and receive expected output.

For the Dashboard page, two queries are needed which are get matching looking fors and get matching services.

The logic with getting matching looking fors, is that after getting ids about logged in user roles and services, a query is written in order to map all users in the platform and find out who are looking for at least one role or service that matches at least one logged in user role or service and then, to select and return needed fields from the User model to improve the performance when getting data instead of returning all fields from the User model.

```
export default async function getMatchingLookingFors(user) {      Kari Leskinen, 4 months ago • FA
  try {
    // Return users where their  "Looking fors" match currently logged in users "Roles & Services"
    const ids = user.userTags.filter(
      (item) => item.tag.type === "SERVICE" ||item.tag.type === "ROLE"
      ).map((item) => item.tag.id)

    const lookingFors = await prisma.user.findMany({
      where: {
        NOT: {
          id: user.id
        },
        public: true,
        lookingFors: { some: { tagId: { in: ids } } }
      },
      select: {
        id: true,
        firstName: true,
        lastName: true,
        location: true,
        headline: true,
        avatarFileKey: true,
        slug: true,
        lookingFors: {
          select: {
            tag: {
              select: {
                id: true,
                name: true,
                slug: true,
                shortlisted: true,
                type: true,
                parentId: true
              }
            }
          }
        },
        connectionParticipants: true
      }
    });

    return lookingFors;
  } catch (e) {
    console.log(e);
    return { error: e.message, lookingFors: [] };
  }
}
```

Code Snippet 5. Get matching Lookingfors query from Prisma

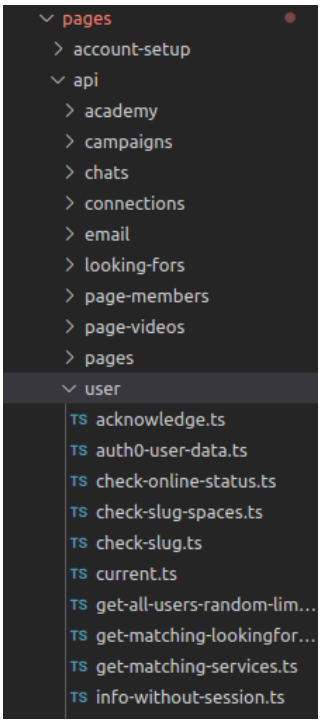
```
export default async function getMatchingServices(user) {
  try {
    // Return users where their "Roles & Services" match currently logged in users "Looking fors"
    const ids = user.lookingFors.filter(
      (item) => item.tag.type === "SERVICE" || item.tag.type === "ROLE"
    ).map((item) => item.tag.id)

    const services = await prisma.user.findMany({
      where: {
        NOT: {
          id: user.id
        },
        public: true,
        userTags: { some: { tagId: { in: ids } } }
      },
      select: {
        id: true,
        firstName: true,
        lastName: true,
        location: true,
        headline: true,
        avatarFileKey: true,
        slug: true,
        userTags: {
          select: {
            tag: {
              select: {
                id: true,
                name: true,
                slug: true,
                shortlisted: true,
                type: true,
                parentId: true
              }
            }
          }
        },
        connectionParticipants: true
      }
    });

    return services;
  } catch (e) {
    console.log(e);
    return { error: e.message, services: [] };
  }
}
```

Code Snippet 6. Get matching Services query from Prisma

After preparing the essential queries, the next phase is to create Apis for these queries. Two new Apis are created inside pages/api folder and because these Apis related to the user so they are put inside the user subfolder.



Code Snippet 7. Folder structure for APIs

In these two APIs, the user object is passed as a body request for each of these two APIs. For getting matching lookingfors, it needs lookingFor ids from the current user, and for getting matching services, it needs userTag ids, so we pass the user object and then when handling queries in Prisma, the user object is accessed to get lookingFor and userTag properties.

In the code snippets, Toni is my English name in the company which help co-workers feel more comfortable when communicating with me in company.

The code snippets below shows the two created APIs:

```
web > pages > api > user > TS get-matching-lookingfors.ts > default
      toni, 2 months ago | 1 author (toni)
1   import { NextApiRequest, NextApiResponse } from "next";
2   import auth0 from "utils/auth0";
3   import getMatchingLookingFors from "actions/looking-fors/get-matching-lookingfors";
4
5   export default auth0.requireAuthentication(
6     async function getMatchingLookingForsApiRoute(req: NextApiRequest, res: NextApiResponse<any>) {
7       const user  = req?.body;
8       const result = await getMatchingLookingFors(user);
9       return res.json( result );
10    }
11  );      toni, 2 months ago • Improve dashboard performance by converting to St…
```

Code Snippet 8. Get matching LookingFors Api

```
web > pages > api > user > TS get-matching-services.ts > default
    You, 1 second ago | 2 authors (toni and others)
1   import { NextApiRequest, NextApiResponse } from "next";
2   import auth0 from "utils/auth0";
3   import getMatchingServices from "actions/looking-fors/get-matching-services";
4
5   export default auth0.requireAuthentication(
6     async function getMatchingServicesApiRoute(req: NextApiRequest, res: NextApiResponse<any>) {
7       const user  = req?.body;
8       const result = await getMatchingServices(user);
9       return res.json( result );
10    }
11  );      toni, 2 months ago • Improve dashboard performance by converting to St…
```

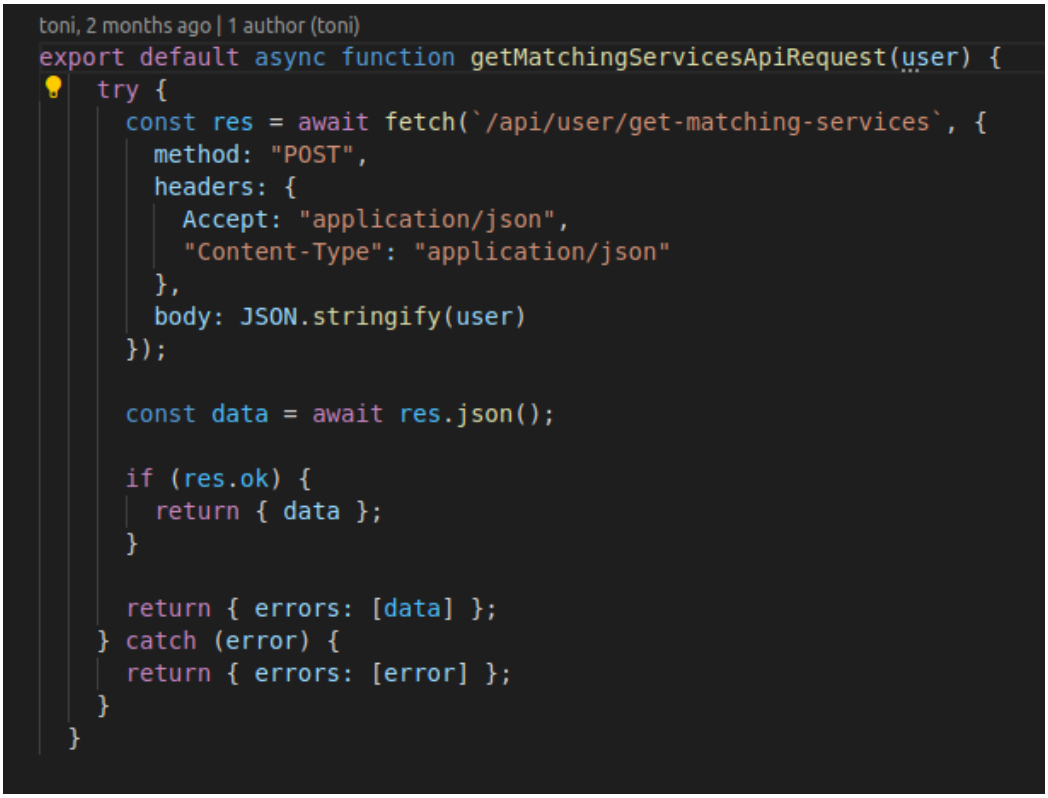
Code Snippet 9. Get matching Services Api

After completing APIs, they can be called and integrated directly inside pages that need to call those actions but in order to make the project structure looks more clearer and cleaner, the APIs should be defined the inside api-request folder where the APIs can be easily reused and found in the future.

Below are the code snippets for those two files.

```
    toni, 2 months ago | 1 author (toni)
1   export default async function getMatchingLookingForsApiRequest(user) {
2     try {
3       const res = await fetch(`/api/user/get-matching-lookingfors`, {
4         method: "POST",
5         headers: {
6           Accept: "application/json",
7           "Content-Type": "application/json"
8         },
9         body: JSON.stringify(user)
10      });
11
12      const data = await res.json();
13
14      if (res.ok) {
15        return { data };
16      }
17
18      return { errors: [data] };
19    } catch (error) {
20      return { errors: [error] };
21    }
22  }
23      toni, 2 months ago • Improve dashboard performance by converting
```

Code Snippet 10. Get matching LookingFors Api Request



```
toni, 2 months ago | 1 author (toni)
export default async function getMatchingServicesApiRequest(user) {
  try {
    const res = await fetch(`/api/user/get-matching-services`, {
      method: "POST",
      headers: {
        Accept: "application/json",
        "Content-Type": "application/json"
      },
      body: JSON.stringify(user)
    });

    const data = await res.json();

    if (res.ok) {
      return { data };
    }

    return { errors: [data] };
  } catch (error) {
    return { errors: [error] };
  }
}
```

Code Snippet 11. Get matching Services Api Request

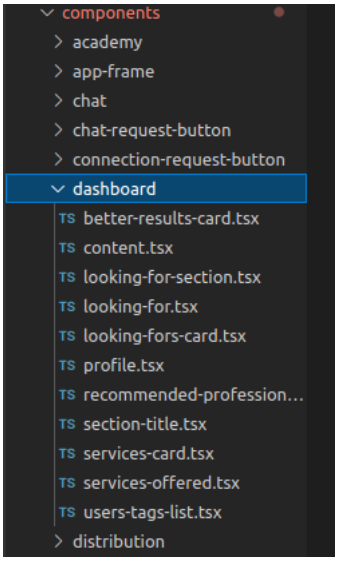
4.3 Creating UI Based on Figma Design

According to the Figma design, the user interfaces of Dashboard page can be divided into three columns:

- The first column for logged in users' information where the user can see their own profile or edit their information. Beneath that, there is a recommended section for some other random users that current user can connect to and chat with. The api for this section was already implemented previously.
- The middle column is used for displaying all the lessons available in the platform. These lessons are from seniors who want to share with newbie artists to enhance and improve their skills as well as knowledge.
- The third column is for LookingFors and Services sections.

This UI including the responsiveness of the page can be implemented faster than before with the support from Tailwind CSS which eliminates long and hard to remember class names as well as easier for maintain and develop.

Code snippet 12 shows the structure of dashboard components inside the project.



Code Snippet 12. Dashboard components

In addition, this is the code snippet for a single card about user information in LookingFors and Services section written in Tailwind CSS.

```
<div id="" className="w-full">
      <div className="flex items-start">
        <div className="mt-2">
          <Link href={`/profiles/${slug}`}>
            <div
              className="w-14 h-14 bg-gray-300 rounded-full bg-cover bg-
center bg-no-repeat cursor-pointer"
              style={{
                backgroundImage: `url(${avatarUrl})`
              }}
            ></div>
          </Link>
        </div>
        <div className="ml-3  mt-2">
          <h3 className="text-base font-bold mb-1">
            <Link href={`/profiles/${slug}`}>
              <a className="text-xl hover:text-brand-pink break-all">
                {firstName} {lastName}
              </a>
            </Link>
          </h3>

          <ul>
```

```
            { !!services && services.filter((item, idx) => idx <
showMaxTags).map(({ tag }) => (
                <li key={tag.id}>{tag.name}</li>
              ))
            }
          </ul>

          <div className="flex flex-row mt-2 space-x-2">
            <p className="text-sm mb-3"></p>
              {currentUserId !== id && <ConnectionRequestButton
                targetId={id}
                currentUserId={currentUserId}
                chatPendingAccept={chatPendingAccept}
                styleName="dashboardConnectionButton"
                chatButton={CHAT_BUTTON.REPLACE}
              />}
              <div className="inline-flex items-center border-brand-text
w-min-20 h-6 text-sm text-brand-text border border-brand-gray-2 rounded-
xl font-semibold tracking-wide hover:opacity-50 focus:outline-none p-2">
                <Link href={`/profiles/${slug}`}>
                  <a className="hover:opacity-50 break-all">
                    Profile
                  </a>
                </Link>
              </div>
          </div>
        </div>
      </div>
      <hr className="mt-2 mb-2"></hr>
    </div>
```

Code Snippet 13. Single user card

For this snippet, there are two buttons for a single user card which can help a logged in user to connect to or visit their profile. The first button is Connection button. There are three values for <ConnectionRequestButton /> which are: Connect, Accept and Chat now:

+ Connect: which means a logged in user and that the user has not connected before

+ Accept: which means that the user sent a connection request and waits for a logged in user to accept that connection

+ Chat now: which means a logged in user and that the users are already connected with each other and ready to chat with each other.

The second button is the Profile button which will link to that the user profile when the logged in user clicks it.

Figure 23 shows the UI.

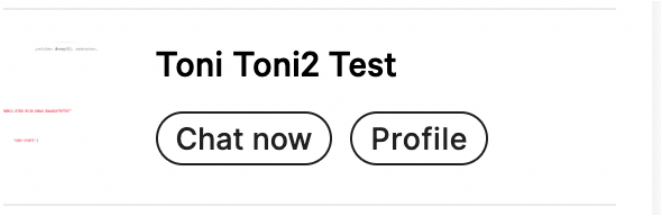


Figure 23. Sample UI for single user card

Additionally, this is the code snippet for UI of one section on the right column of the page and it contain three times component from Code Snippet 13 which can be reused for both LookingFors and Services section.

```
return (
  <div className="flex flex-col bg-white rounded-lg shadow-lg p-4 mb-4">
    <SectionTitle
      headerText={hasMatchingLookingFors ? "Services For You" : "Offering Services"}
      subText={hasMatchingLookingFors ? "": ""}
    />
    {
      hasMatchingLookingFors ? (
        <UsersTagsList
          tags={currentUser.lookingFors}
          showMaxTags={showMaxTags}
          scrollToId={"EditProfileLookingForSection"}
        />
      ):(
        <></>
      )
    }

    { services.filter((item, idx) => idx < showMaxLookingFors).map((user) => (
      <LookingFor
        key={user.slug}
        user={{
          id: user.id,
          location: user.location,
          slug: user.slug,
          firstName: user.firstName,
          lastName: user.lastName,
          headline: user.headline,
          avatarFileKey: user.avatarFileKey,
          services: user.userTags?.filter((item, idx) => item.tag.type !== "GENRE")
        }}
        canChat={checkIfConnected(currentUser, user)}
        chatPendingAccept={checkChatStatus(
          currentUser,
          user,
          "PENDING"
        )}
        onChat={() => {}}
        canConnect={user.id !== currentUser.id}
        onConnect={() => {}}
        currentUserId={currentUser.id}
      />
      ))}

    <Link href={`/discover?q=&tags=${tagsViewMore.join('')}&page=1`}>
      <a className="text-brand-gray-4 text-center mt-4 mb-4 text-sm">
        View more
      </a>
    </Link>
  </div>
)
```

Code Snippet 14. One section on the right column

The UI in this code snippet 14 has the visual similarly with the figure below.

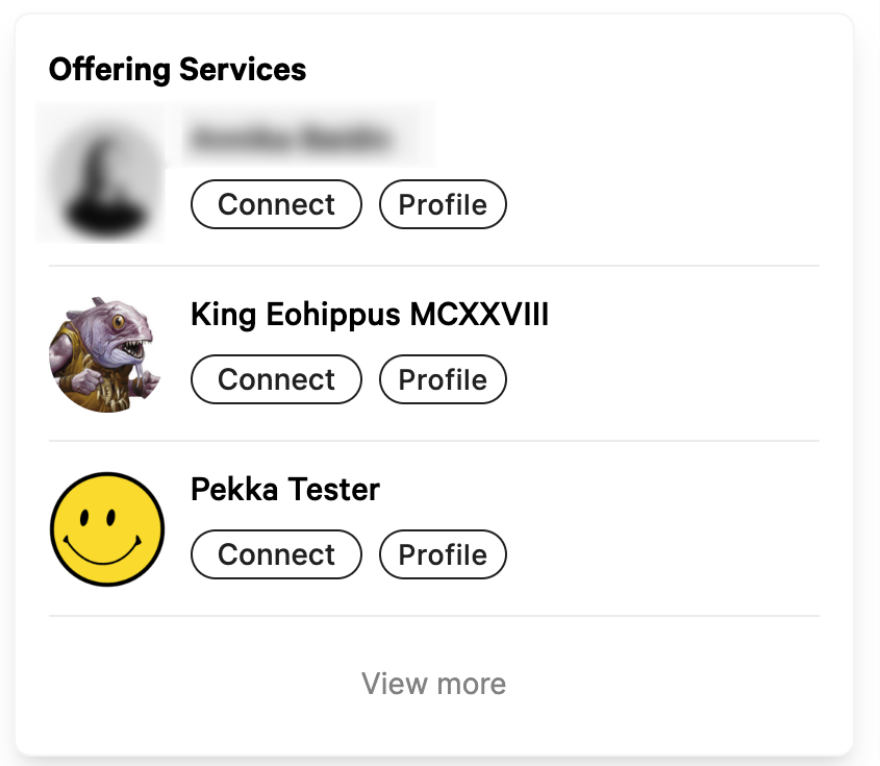


Figure 24. Sample UI for one section on the right column

4.4 Integrating APIs to UI

After completing UI based on the Figma design, the final step is to integrate all created APIs to current UI to render expected results.

At first, because the platform is built with NextJS, getServerSideProps method can be used to improve performance of the page. Because of that, get matching services and get matching lookingfors logic are called directly from the database with Prisma queries on the server side instead of using APIs for calling on the client side. At that time, it worked perfectly in the local and Staging environment but there was an issue since this feature deployed to the Production environment, where thousands of users available compared to only approximately thirty users in the local and hundred users on the Staging. Because it had to load all content, which included lessons data and user data, of the page before rendering to the page, it affected the performance when the loading time was around three to four seconds at that time. We decided to use another feature of NextJS which is getStaticProps, which means that the page is prerendered at build time

instead of at each request or runtime. The approach for this method is to fetch lesson data at build time because these lessons are always the same and with the revalidation method from getStaticProps, the page can be revalidated and fetched new lessons if available.

One more issue is that because when a user log in to the platform, this is running at runtime so that we cannot use get matching LookingFors and Services at build time therefore, they must be fetched on the client side by using created APIs.

Because of that, a function is created with Promise all method which calls all functions at the same time and avoids calling them respectively that affects the performance of the page.

```
const getAllMatchingLookingForsAndServices = async (currentUser) => {
  try {
    const [
      matchingLookingForsRes,
      matchingServicesRes,
      allUsersRandomLimitedWithoutFiltersRes,      toni, 2 months ago •
    ] = await Promise.all([
      getMatchingLookingForsApiRequest(currentUser),
      getMatchingServicesApiRequest(currentUser),
      getAllUsersRandomLimitedApiRequest({
        locations: [],
        tags: [],
        currentUserId: currentUser?.id,
        onlyUsersWithAvatars: true,
        maxResults: 9,
      }),
    ]);

    setMatchingLookingFors(matchingLookingForsRes?.data);
    setMatchingServices(matchingServicesRes?.data);
    setAllUsersRandomLimitedWithoutFilters(
      allUsersRandomLimitedWithoutFiltersRes?.data
    );

    return {
      matchingLookingForsRes,
      matchingServicesRes,
      allUsersRandomLimitedWithoutFiltersRes
    }
  } catch (error) {
    console.log(error)
  }
};
```

Code Snippet 15. Get All Matching LookingFors and Services functions

In this function, except get matching lookingfors and services api requests, there is another api which is get all user random limited. This api is in the first column, used for Recommended section under the current user information which displays random users for the logged in user to connect with. Additionally, one more objective of this api is for the current user who does not set what they are looking for or what their roles and services are in music industry. If they have not

set one of these two or both sections yet, the LookingFors and Services sections will display the user randomly. For LookingFors, it will display three random users who are looking for some roles or services and those roles and services match current user. And For Services, it will display three random users who set their roles and services and maybe current user is looking for them.

After getting data from all random users, two empty arrays have to be set which are lookingForResults and servicesResults in case the current user does not set yet, these arrays will be used to render in landing page.

```
let lookingForsResults = [];
let servicesResults = [];

!!allUsersRandomLimitedWithoutFilters?.length &&
  [...allUsersRandomLimitedWithoutFilters]?.filter((item) => {
    if (item.lookingFors.length > 0) {
      lookingForsResults.push(item);
    }
    if (item.userTags.length > 0) {
      servicesResults.push(item);
    }
  });
```

Code Snippet 16. lookingForsResults and servicesResults

For the condition to use either get matching lookingfors and services apis or get all user random limited, there are two variables for that which are hasMatchingS-ervices and hasMatchingLookingFors.

```
let hasMatchingServices = matchingServices?.length > 0;
let hasMatchingLookingFors = matchingLookingFors?.length > 0;
```

Code Snippet 17. hasMatchingServices and hasMatchingLookingFors

If the logged in user has not set their roles and services or what they are looking for, the data sent back from back-end from getMatchingLookingFors and getMatchingServices apis will be empty array so it can be checked if their length greater than zero.

Finally, there will be three final arrays data which will be displayed in the UI after checking conditions, which are lookingForWithMaxRes for LookingFors section,

servicesWithMaxRes for Services section and networkSuggestions for Recommended section. These arrays will be updated whenever their roles and services are updated based on useMemo hook from React.

In addition, Samplesize function from Lodash is used in this case in order to limit the array data with maxResults variable there. Also, filterAvatars is a helper function for prioritizing and choosing users who already set their own avatars.

```
const lookingForsWithMaxRes = React.useMemo(() => {
  return hasMatchingLookingFors
  ? matchingLookingFors
  : samplesize(filterAvatars(lookingForsResults, true, false), maxResults)
}, [lookingForsResults]);

const servicesWithMaxRes = React.useMemo(() => {
  return hasMatchingServices
  ? matchingServices
  : samplesize(filterAvatars(servicesResults, true, false), maxResults)
}, [servicesResults]);

const networkSuggestions = React.useMemo(() => {
  return !!allUsersRandomLimitedWithoutFilters?.length &&
  samplesize(
    filterAvatars(allUsersRandomLimitedWithoutFilters, true, false),
    maxResults
  );
}, [allUsersRandomLimitedWithoutFilters])
```

Code Snippet 18. LookingFors, Services and NetworkSuggestions data

For updating called apis frequently, useSWR is the best option when using NextJS and this is how those apis are called in SWR after every fifteen seconds which make sure data will not be stale and will be revalidated.

```
useSWR( () =>  currentUser ? 'getAllMatchingLookingForsAndServices' : null , () => {
  getAllMatchingLookingForsAndServices(currentUser)
}, { refreshInterval: 15000 })      You, last month • added refresh interval for fetchi
```

Code Snippet 19. Calling Apis with SWR

After integrating these apis, the final UI for landing page was finished and shown in Figure 25.

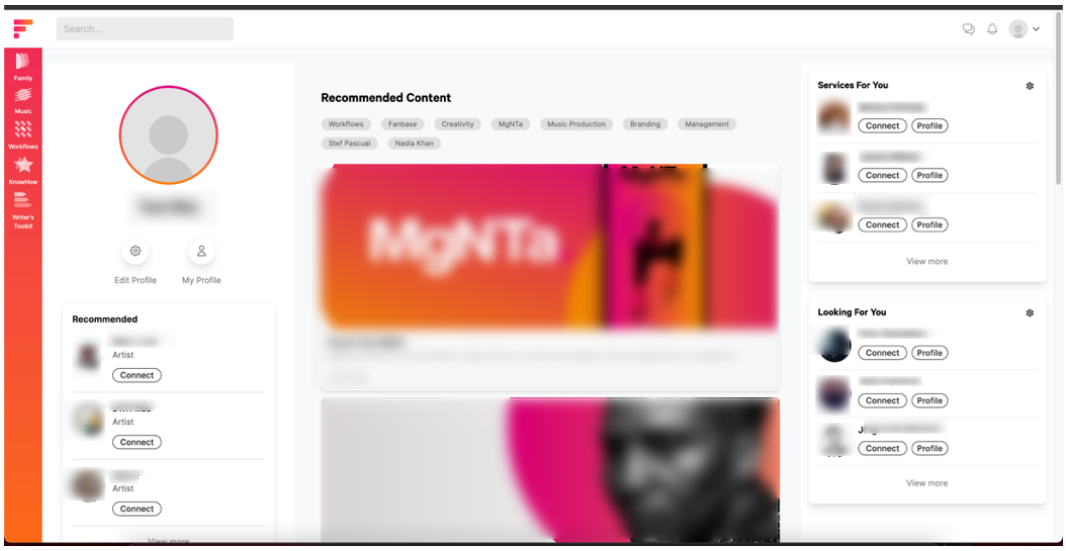


Figure 25. Final landing page UI

5.CONCLUSIONS

This feature is completed and after testing carefully and approved by the product team in Family In Music, it was deployed to Staging and then Production server. This is the link for Production server which can be accessed: https://app.familyinmusic.com/ .

After completing this feature, there is a wide range of new knowledge needed to learn and practice from back-end to front-end side. During the time this feature is on Production server, there are several changes related to UI to make this page look more attractive and cleaner.

The next step for this page, after discussing with the Product team is, to create Wanted Advertisements section beneath LookingFor and Services section so as to attract more new users in music industry. The Figma design is implemented and will be started in the near future.

REFERENCES

/1/ Wikipedia. ReactJS. Accessed 01/03/2022
https://en.wikipedia.org/wiki/React_(JavaScript_library)

/2/ Simform. Accessed 17/04/2022
https://www.simform.com/blog/websites-use-react/

/3/ Peerbits. Advantages of ReactJS. Accessed 01/03/2022
https://www.peerbits.com/blog/reasons-to-choose-reactjs-for-your-web-development-project.html

/4/ Valuecoders. Accessed 17/04/2022
https://www.valuecoders.com/blog/technology-and-apps/business-benefits-of-reactjs-framework-for-modern-web-and-app-development/

/5/ Wikipedia. NextJS. Accessed 01/03/2022
https://en.wikipedia.org/wiki/Next.js

/6/ Grzybek,Pawel.2020. Client-side rendering, serverside rendering and static site generation of Nextjs applications explained. Accessed 17/04/2022
https://pawelgrzybek.com/client-side-rendering-server-side-rendering-and-static-site-generation-of-nextjs-applications-explained/

/7/ Nextjs. Accessed 17/04/2022
https://nextjs.org/learn/basics/data-fetching/two-forms

/8/ NextJS. getServerSIdeProps, getStaticProps, Incremental Static Regeneration. Accessed 01/03/2022
https://nextjs.org/docs/basic-features/data-fetching/overview

/9/ Sanity. Sanity definition. Accessed 02/03/2022
https://www.sanity.io/

/10/ Prisma. Prisma definition. Accessed 02/03/2022
https://www.prisma.io/

/11/ Formidable. Prisma manipulation. Accessed 02/03/2022
https://formidable.com/blog/2021/prisma-orm/#:~:text=Prisma%20is%20a%20next%2Dgeneration,and%20generates%20type%2Dsafe%20code.

/12/ Wikipedia. PostgreSQL. Accessed 02/03/2022
https://en.wikipedia.org/wiki/PostgreSQL

/13/ Tailwindcss. Tailwind CSS. Accessed 03/03/2022
https://tailwindcss.com/

/14/ Lodash. Lodash JavaScript library. Accessed 17/04/2022
https://lodash.com/

/15/ Visual Studio Code. Coding Tool. Accessed 17/04/2022
https://code.visualstudio.com/

/16/ Visual Studio Code Marketplace. ReactJS Snippets. Accessed 17/04/2022
https://marketplace.visualstudio.com/items?itemName=xabikos.ReactSnippets

/17/ Visual Studio Code Marketplace. Prettier ESLint. Accessed 17/04/2022
https://marketplace.visualstudio.com/items?itemName=rvest.vs-code-prettier-eslint

/18/ Visual Studio Code Marketplace. GitLens. Accessed 17/04/2022
https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens

/19/ Postman. APIs platform for building and using APIs. Accessed 17/04/2022
https://www.postman.com/

/20/ Wikipedia. A database administration tool.Accessed 17/04/2022
https://en.wikipedia.org/wiki/SQuirreL_SQL_Client