State of the Art

Introduction

AREA is an ambitious project. Its objective is to create **links of Action-Reaction** between **services** (e.g.: Youtube or Spotify). Our service will detect an **Action** chosen by the user (e.g.: liking a YouTube video), it will then take **data** from this action (e.g.: the title, the genre or the author of the video) and use it while triggering a **Reaction** (e.g.: adding the song linked to the video title to a Spotify Playlist).

To accomplish all of that, we need three extensive parts. Each one will have its own importance in the project. The first one will be an **application server**. It's the part that will handle all the business logic in the background (e.g.: checking for action, triggering reaction). But accessing this without an interface is quite difficult for an average user, that's why we'll have two clients to help customers with the utilization of our service. First, a **web client** that will be used to access our platform from any device connected to the Internet. But also, a **mobile application** to give our mobile users the best experience possible.

Every part of this project needs a **crucial study**. We need to choose the **correct technology** to create the **most successful product** possible while avoiding problems by the due date. That's where this **State of the Art** comes into play. You'll find in this document all the reasons that pushed us into choosing the **technology stack** we decided to approach this project.

Our Technology Stack

After long phases of research and testing, we've come to the conclusion that the best technology we can use for the project are the following and we'll explain why:

Application server

Let's start with the application server. It is the part that requires the most choices but also the most critical ones due to the impact of this part on the project. We needed a programming language coupled with a Framework. With this a choice of Database coupled with an ORM, a tool to simplify the use of databases.

Programming language

We needed to choose a programming language. One that will be the most efficient for our objectives but also the safer one. Multiple options were available to us, PHP with its framework Symfony, Node with either JavaScript or TypeScript and Python with Flask.

Symfony as a Framework isn't really bad, it's the language that comes with it that scared us the most, PHP. It is well-known that despite PHP's popularity, the language lacks a lot of things. We can find a lot of inconsistencies in the code, it lacks restriction and can make programming unpredictable, not a good decision for a project that will last for at least two months. Since no one in our group had a predisposition for PHP, we decided it wasn't a good idea to keep it in our list of choices.

That leaves us with three choices, Python, JavaScript and TypeScript. We rapidly discarded JavaScript, with TypeScript in the list it would be a mistake to choose a language with less tools to scale in a project this big.

We weren’t sure which language to pick between the last two, we had reasons to take TypeScript as we have people in the team already capable of creating API in this language. However this argument doesn’t really stand as Python is famous for being really easy to learn, so we decided to give it a chance by testing the two before making our final decision.

For each language, we created an API capable of routing, making and receiving requests but also interacting with a database. We paired Python with Flask, and TypeScript with Node.
Despite its accessibility, Python had a lot of problems. First of all, coding simple things such as routes were not as accessible as we thought, ending up costing time and space. But it’s not the worst, ORM is a big problem in Python. AREA relies heavily on the database, so the choice of ORM is really important, however Python doesn’t really offer us a lot of possibilities. Most of the famous Python’s ORMs aren’t as popular as Node’s (SQLAlchemy only has 6k Stars on Github where Prisma has 25k for example). But that’s not all, documentation for those isn’t great either, lacks a lot of information and seems outdated. It reaches a point where you start thinking learning SQL is a great alternative, a solution we don’t have the time for.

This testing confirmed the choice we made by instinct, TypeScript and its runtime environment Node will be our choice of language.

Framework

The next step for us was the framework. A framework is a set of tools provided on top of a programming language created to help the developer for certain tasks. In our case we need one that simplifies web development.

The two choices that we pondered upon were Express and Fastify. Both are notorious in the field so we decided to put them to the test. Results were really interesting, despite being more famous than it’s concurrent, Express wasn’t really matching its opponent. Not that it was bad or anything of the like, the framework was originally not really made to support TypeScript, and this resulted in many problems performance-wise. The two weren’t that far from each other but this little detail is the element that gave the edge to Fastify. More than that, our lead developer in back-end has experience in Fastify and really took a liking to it. Making sure that Fastify would be our choice of Framework.

ORM and Database

Two choices to go! ORM and Database, both are linked as the first one is used to help the use of the second, and since data handling and storing is really important in this project we needed to thread carefully.

Starting with ORMs, we had three that caught our attention, Prisma, TypeORM and Sequelize. So we decided to do some research then test them on different applications with the same test as before.

Let’s start with Sequelize, it’s one of the most well known ORM in the field but the problem is that it’s compatibility with TypeScript is way worse than the two others, more than that it’s really complex to understand the documentation for a beginner, making Sequelize not really a tool of choice. TypeORM, on the other hand, has a really strong compatibility with TypeScript and is easy to learn thanks to its intuitivity. The problem resides elsewhere, the maintenance of TypeORM is questionable. Developers responsible for the project are a bit overbooked causing a lot of features to be missing and allowing bugs to be created. Finally, Prisma is a recently created ORM, this can be scary at first. But then we saw that the TypeScript compatibility is on par with TypeORM's one and compared to them their developer team was working full-time on the project. And their documentation is really pertinent and has data on different databases. Making Prisma our ORM for the project.

Then we have Databases, we had four choices for this project. MariaDB, MySQL, PostgreSQL and MangoDB. Four notorious databases that earned their fame. We excluded MySQL from the list since MariaDB is a clear upgrade of MySQL in terms of speed and every other aspect is the same. Leaving us with PostgreSQL, MangoDB and MariaDB. PostgreSQL and MangoDB are both slower than MariaDB, but have the advantage of offering a lot of possibilities the developer is used to the service. However, the possibilities offered are not really interesting for us for this project. It might be too much to commit to on one of those two platforms while knowing that MariaDBwill do the things that we need but faster than the other two. This argument settles MariaDBas our Database for the project.

To conclude this part I will leave you with this image that contain our choices for the application server:

TS node
MariaDB
Prisma
fastify

Web client

The last two parts we have to tackle have the same principle. Help the customer using our service thanks to great interfacing. Firstly, we're going to talk about the Web client. It's a powerful tool capable of running on every platform through a Web Browser and will be the first thing that our new users will see.

In terms of programming language, we decided to stay in coordination with our server application and use TypeScript. It's easier to help each other in the long run and create way less problems this way. All our concentration went into choosing the best Framework to go with it.

We had three possibilities in our minds, Vue, React and Angular, the three most famous Frameworks for front-end in the market. To make up our minds we decided to test them, the test was about creating an application with a button, on the push of this button it should show the result of a GET request.

After testing and doing some research, we saw that React and Angular possess tools that are complicated and need some experience to use optimally while Vue is pretty straightforward and easy to set-up. For Angular, despite the fact that it was made for TypeScript, it is expected to do a really huge project with it, something much bigger than AREA , making using and learning Angular a bit overkill for the project. We were left over with React and Vue. On the first hand, one of the problems of React is that you need a lot of experience to create a good project structure, and AREA is big enough to have a need for a structure in the project. On the other hand, Vue is really intuitive and simple to learn and use, something that we need since not everyone in our group knows how to develop front-end applications. All those reasons convinced us to take Vue over its two concurrents for this project.

But now that we chose Vue, we needed to know which version to take. The problem was between Vue 2 and Vue 3. As the name indicates, Vue 3 is a clear upgrade of Vue 2 and possesses many things that his predecessor doesn't. Moreover, Vue 2 code is also compatible and works on Vue 3 so the decision shouldn't be really complicated, right? Well, we chose Vue 2, for the simple reason that not every package and libraries created for Vue 2 are compatible. And by coming on Vue 3 we probably would have recreated the wheel many times which isn't the objective of the project nor a benefit in terms of time.

With this settled you understand why we chose those for our web client:

TS TypeScript

Vue.js

Mobile application

The last choice we had to make was for the Android application. The objectives are the same as the web client but the target is different, here we create an interface for mobile users so they can be more comfortable with using our service than using it directly in our website.

Three choices of programming language were offering themselves to us Kotlin, Flutter and Vue native. Vue native could be really beneficial for us, since it's the language chosen for our website, we could reuse the code base to gain a lot of time. However, Vue native has been deprecated and is no longer maintained since November 2021, by fear that it would impact our project we decided not to take the risk of taking Vue native into account.

This left us with two choices, Kotlin and Flutter. As usual, to decide we went for testing to be sure of the choice of application. The objective of the application was the same as the website one. On the first hand, achieving the task with flutter was pretty easy, however setting up the project was a bit difficult and there isn't any specialized IDE for Flutter that provides a lot of assistance. Another big advantage of Flutter is the possibility to refresh the application really easily, making compilation time ridiculous. On the other hand, Kotlin took us a bit more time to create the application, but in the background made it more organized and structured. Making it easier to create other requests in the future. This is a thing we need as structure is important for a project of this scale. We need to remind you that everyone in our group is a beginner in mobile development. That's why the fact that Android Studio is an IDE dedicated to Kotlin and has a lot of features that help the developer is a huge bonus, without talking about the documentation and examples that are everywhere since the language is an upgrade of Java.

The easy-access of Kotlin is what made us choose the language. As apprentices mobile developers it was important for us to acquire a solid basis with documentation provided by the best companies like JetBrains or Google.

Kotlin

Conclusion

In conclusion, here is our technology stack. For the application server, TypeScript paired with Node and Fastify as Framework. In terms of ORM and database Prisma paired with MariaDB. Our website will be in TypeScript paired with Vue. And finally our mobile application will be in Kotlin.

Application Server

TS TypeScript

fastify

node

Mobile application

Kotlin

Website Client

Vue.js

TS TypeScript

ORM and Database

Prisma

MariaDB