"ALEXANDRU-IOAN CUZA" UNIVERSITY OF IASI

FACULTY OF COMPUTER SCIENCE

BACHELOR THESIS

Think-In - A virtual conference application

proposed by

Mihai Crisan

Session: june-july, 2021

Scientific coordinator

Prof. Dr. Alboaie Lenuţa

”ALEXANDRU-IOAN CUZA” UNIVERSITY OF IASI

FACULTY OF COMPUTER SCIENCE

Think-In - A virtual conference application

Mihai Crisan

Session: june-july, 2021

Scientific coordinator

Prof. Dr. Alboaie Lenuţa

Avizat,
Îndrumător lucrare de licență,
Prof. Dr. Alboaie Lenuţa.
Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Crisan Mihai** domiciliat în **România, jud. Galaţi, sat. Matca (com. Matca), Strada Morii, nr. 74**, născut la data de **15 october 1999**, identificat prin CNP **1991015226702**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **informatică**, promoția 2021, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Think-In - A virtual conference application** elaborată sub îndrumarea **Prof. Dr. Alboaie Lenuţa**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Think-In - A virtual conference application**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea "Alexandru-Ioan Cuza" din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Mihai Crisan**

Data: ..... Semnătura: .....

Contents

Motivation

Because conferences, meetings, job fairs and any other event based on human interaction could no longer be held because of the pandemic, people have been looking for alternatives in the virtual world. The concept of virtual events is not new, therefore I have analyzed the currently existing platforms in order to understand their advantages and disadvantages. By checking out a dozen of platforms that offer these types of services, and based on the way you interact with the surrounding environment, I have divided them into two large categories: **three-dimensional** and **virtual reality**:

Three-dimensional (3D)

These type of platforms organize their events in a three-dimensional space similar to how most video games are built like nowadays. An example of such a platform is shown in Figure 1.

Figure 1: Example of interface and environment of a 3D event platform.

While this implementation provides a high level of immersion and interaction, it also comes with a steep learning curve and an unpleasant user experience. For instance, learning the 3D space the user is around requires time and users will get frustated if they don't know the place of the booth they want to attend. Additionally, the user might have to use a keyboard and a mouse in order to navigate freely around the world, thus the mobile users will not have the same experience as desktop users. Mobile users will instead have to use touch buttons (notice the lower right corner where the buttons are). A feature many other similar platforms have is 3D avatar creation, which although it provides personalization, it is unnecessarily complex.

Virtual Reality (VR)

If the intent is to permanently move from physical events to virtual events then virtual reality is the way to go. Figure 2 shows such a platform.

Figure 2: Example of environment in a VR event platform.

This platform offers the most immersion there is but it also comes with a steeper learning curve because not many users have interacted with VR headsets and controllers compared to the ubiquitous keyboard and mouse as used in 3D environments, what is even worse is that you are required to own a VR equipment and a machine with enough computing power. Furthermore the users have to go through the process of setting up the device and the chances are low of running setup on mobile.

Conclusion

After examining the platforms that provide virtual events I have come to the conclusion that most of them are not user friendly and require unpleasant learning steps. Additionally, the applications in both categories can share disadvantages such as:

- May require development of distinct desktop and mobile applications (as a consequence the application may not work on certain operating systems).
- May require download of big files and lengthy installation time.

- From the developer's point of view, may assume application maintenance overhead such as updating it, developing patches, and upgrading. The updates then need to be made available to the end users, in case the user doesn't run the update then there can appear inconsistency, usability and security problems.

The solution is to build a virtual event platform that: requires few learning steps, is easily accessible across devices, has few maintenance headaches and is always up-to-date. That is an application which is intuitive to use and it **just works**.

Introduction

The proposed solution to the previously mentioned problems has the name **Think-In**, and is a web application where virtual conferences or events can be held with the purpose of networking. Having mentioned the two below issues, the **general purpose** of the application is to help people connect and network, therefore the users will expand their professional relationships and find new opportunities:

- An issue that can appear in conferences of bigger sizes because of the diversity of the attenders, which other existent applications don't solve, is the **language barrier**. **Think-In** solves this issue as messages sent across the conference are translated in real-time in the user's native language or any other language the user desires. This feature can bring people together from all around the world as there are no communication bounds.

- An inherent problem to real world conferences or events is that you don't know many of the people who attend, therefore the **Think-In** users have the option to provide more information about themselves such as an avatar, their job title and links to their social media accounts so they can be easily discovered and engaged with.

As a response to the disadvantages stated at the end of the previous chapter, **Think-In** is platform agnostic so the user can use the application no matter what device it is accessed from, it was designed to be blazing fast and very easy to use. In contrast to the aforementioned desktop applications, it is a web application, therefore the developers don't have to take care of how updates are brought to the users because everytime the user reloads the page the newest version of the application is used.

From a technical point of view, the application's **frontend** is built using the Next.js[1]

[1]No configuration React framework: https://nextjs.org/

React[2] framework and has a JAMstack[3] architecture, the pages are built at deploy time and they are stored in a S3[4] bucket as a static website, from there on they are served from the CloudFront[5] CDN[6]. This setup ensures a very fast TTFB[7] since there are no round trips to the server in order to render the page and the files required for the web page are fetched from one of the CDN's proxy servers, which, as a consequence of being close to the user, ensure fast file retrieval. The **backend** relies on a Socket.IO[8] server ensuring the fastest communication possible when stage interactions happen or when messages are sent, as well as an AWS API Gateway[9] that centralizes access to Lambda[10] functions, which act as a REST API when they are triggered by HTTP events. TypeScript[11] was the language of choice all around the application stack, this was motivated by the fact that the frontend had to be written in JavaScript and the opportunity to reuse code in the backend was hard not to take. Additionally, TypeScript brings static typing to JavaScript, that has proved to be extremely helpful in eliminating type errors as well as simplifying the developing experience thanks to type hints povided by modern IDEs. In order to simplify the deployment of the application, DevOps practices such as CI/CD[12] and IaC[13] have been adopted.

[2]JavaScript library for building user interfaces: https://reactjs.org/
[3]https://jamstack.org/
[4]AWS' object storage service: https://aws.amazon.com/s3/
[5]AWS' CDN service: https://aws.amazon.com/cloudfront/
[6]Content Distribution Network
[7]Time To First Byte
[8]Real-time communication library which uses WebSockets or HTTP as a fallback: https://socket.io/
[9]API management tool https://aws.amazon.com/api-gateway/
[10]AWS' event-driven serverless compute service: https://aws.amazon.com/lambda/
[11]JavaScript superset language that adds static typing: https://www.typescriptlang.org/
[12]Continuous Integration and Continuous Deployment
[13]Infrastructure as Code

Contributions

While the concept of virtual conferences and events is not new, I have attempted to streamline the user interactions in order to provide quick access to what the attenders want. I have analyzed other existent solutions, borrowed their strongest points and tried to overcome what I thought was lacking. I tried to rethink the business logic so smoother experiences can be had. Apart from thinking of an improved business logic, I also was in the position of choosing the best application architecture that will fit to the proposed business rules.

The combination of the ideas and the used cloud services have brought **Think-In** to an actual tangible product that can be accessed by anyone in the world, additionally, the used technologies and services allow for a very easy extension so that the application can fully scale horizontally. The application can be used by any kind of company since no technical skills are required in order to setup new stages.

I would like to thank my coordinator **Prof. Dr. Alboaie Lenuţa** because even from the very beginning she advised me what technologies I should learn and has been alongside me in the decision of the thesis theme and what features are worth adding.

Chapter 1

Business logic

At its most basic form, conferences are meant to be a formal meeting of people with a shared interest, it is meant to be an area where information is exchanged and people get to know each other. Without much thought, we realize that people at conferences are divided into two groups: organizers and visitors; thus, there must be a slight distinction between the two.

1.1 Attendee user flow

In this subchapter we will analyze what steps an attendee has to go through in order to start using the application, we will start from the very beginning and we will be going into the details of certain features as well as how these can be used. For the purpose of completeness as many steps as possible will be described and images will be used to visualize the said ideas.

1.1.1 Visiting the website and signing up

By visiting the website for the first time we will be greeted with a page where the attendee is prompted to authenticate before using the application.

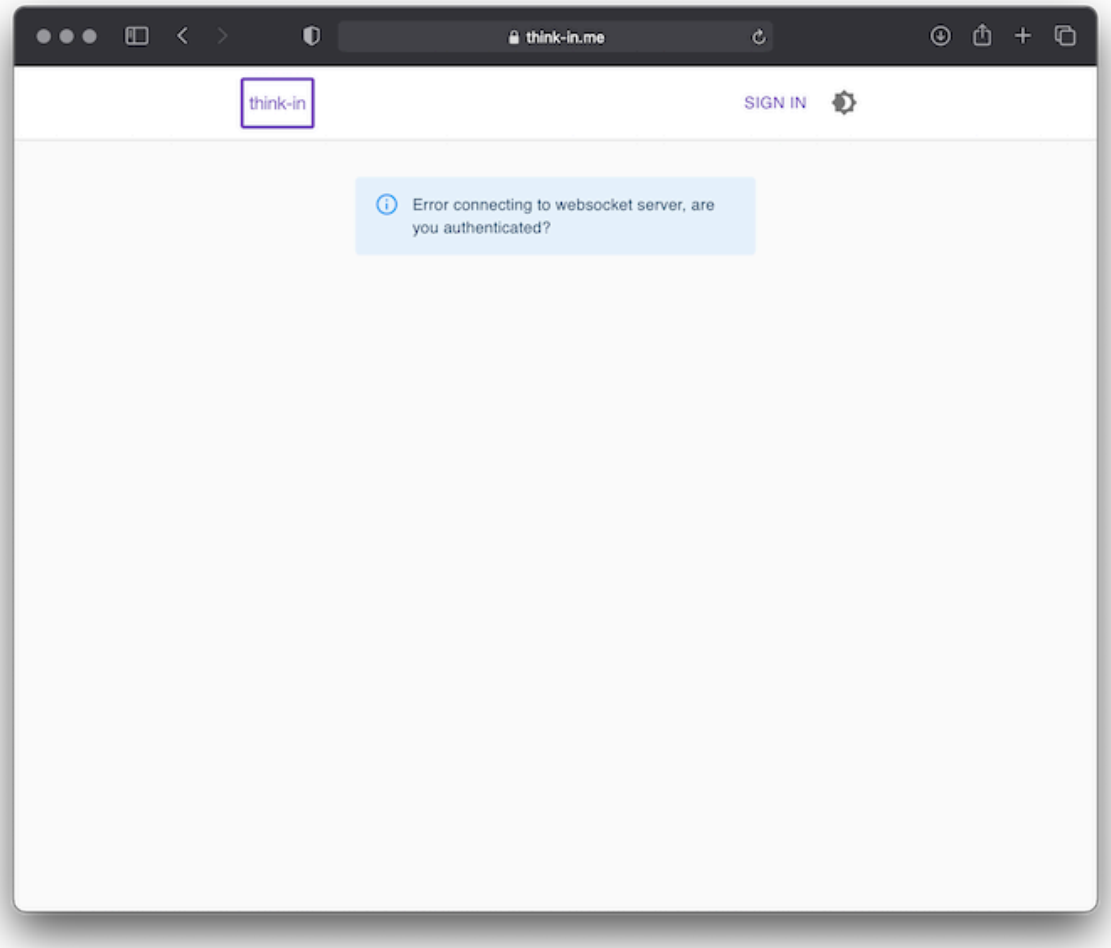

Figure 1.1: Website main page when not signed in.

An attendee has to have an identity, therefore signing in is necessary before using the application. Upon clicking the *Sign In* button located at the right side of the page header the user is presented with the option of signing in through a third party (social sign in with Google in this case) which significantly speeds up the signing up process, or the more traditional option of using an account created specifically for **Think-In**.

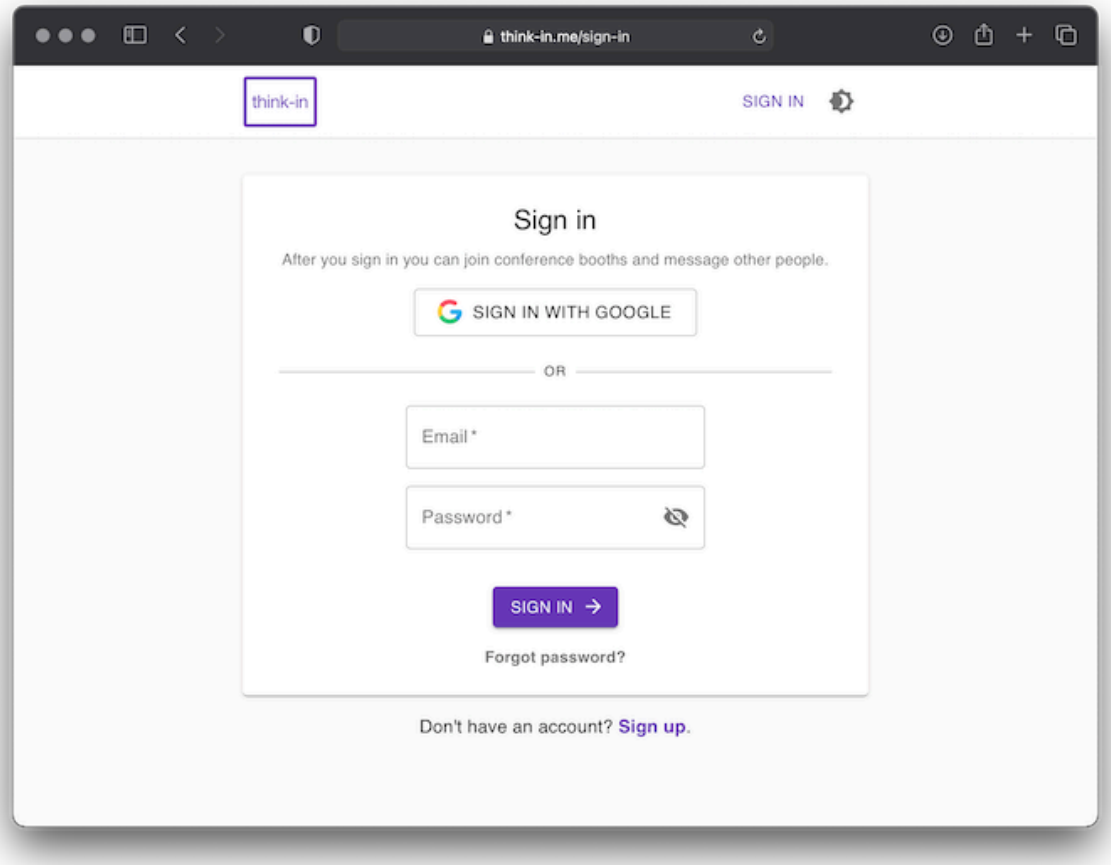

Figure 1.2: Website sign in page.

If the user signs in through an identity provider then he is redirected to the main page (shown in the next subsection), otherwise he has to go to the sign up page by clicking the *Sign Up* button.

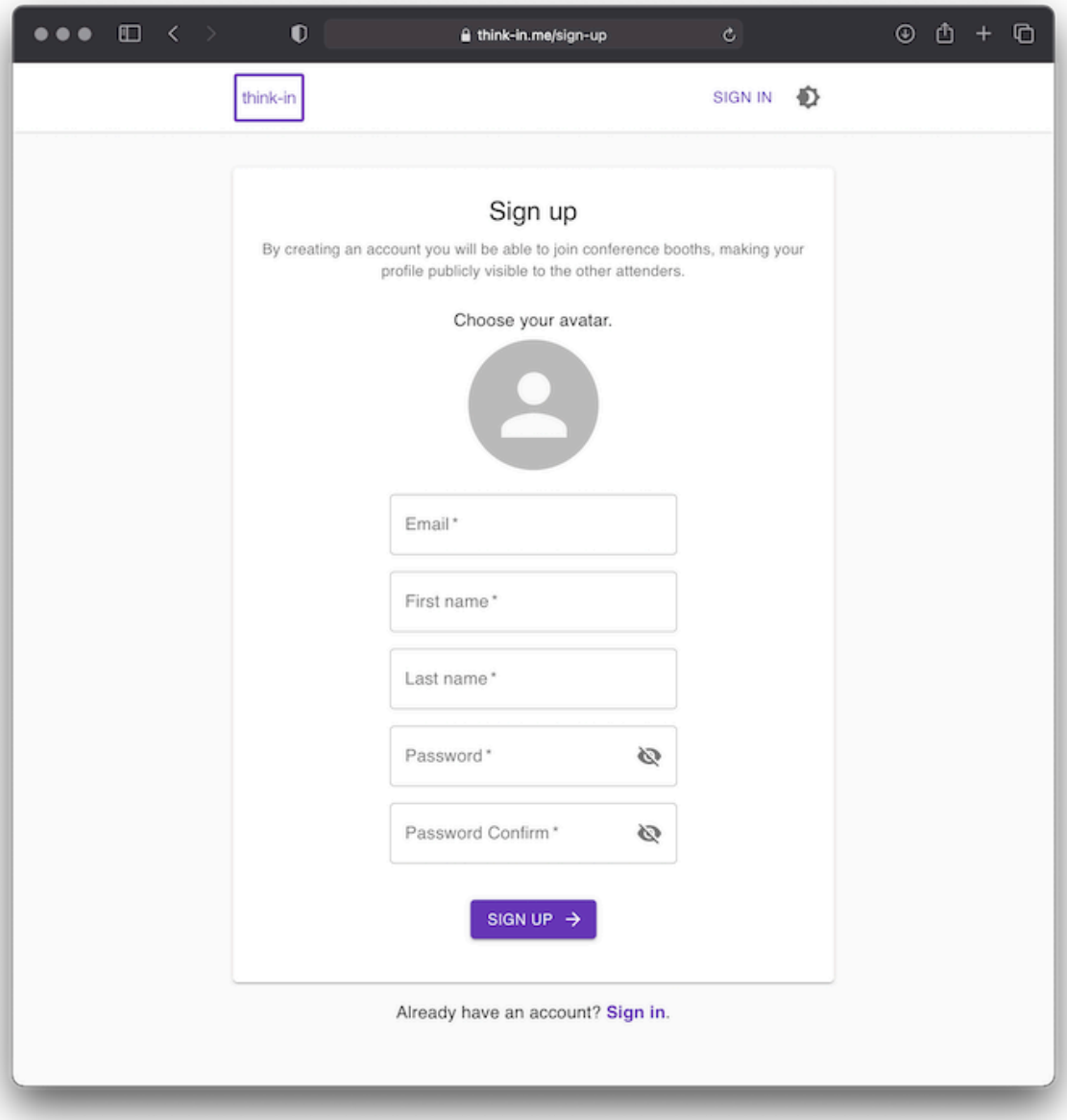

Figure 1.3: Website sign up page.

The user has the choice of selecting an avatar and has to provide the information labeled in the above picture. In case the user did not pick an avatar, an identicon[1] is generated and is used as the avatar. Upon signing up, the user has to confirm the just created account by following a link sent to the entered email address.

[1]https://en.wikipedia.org/wiki/Identicon

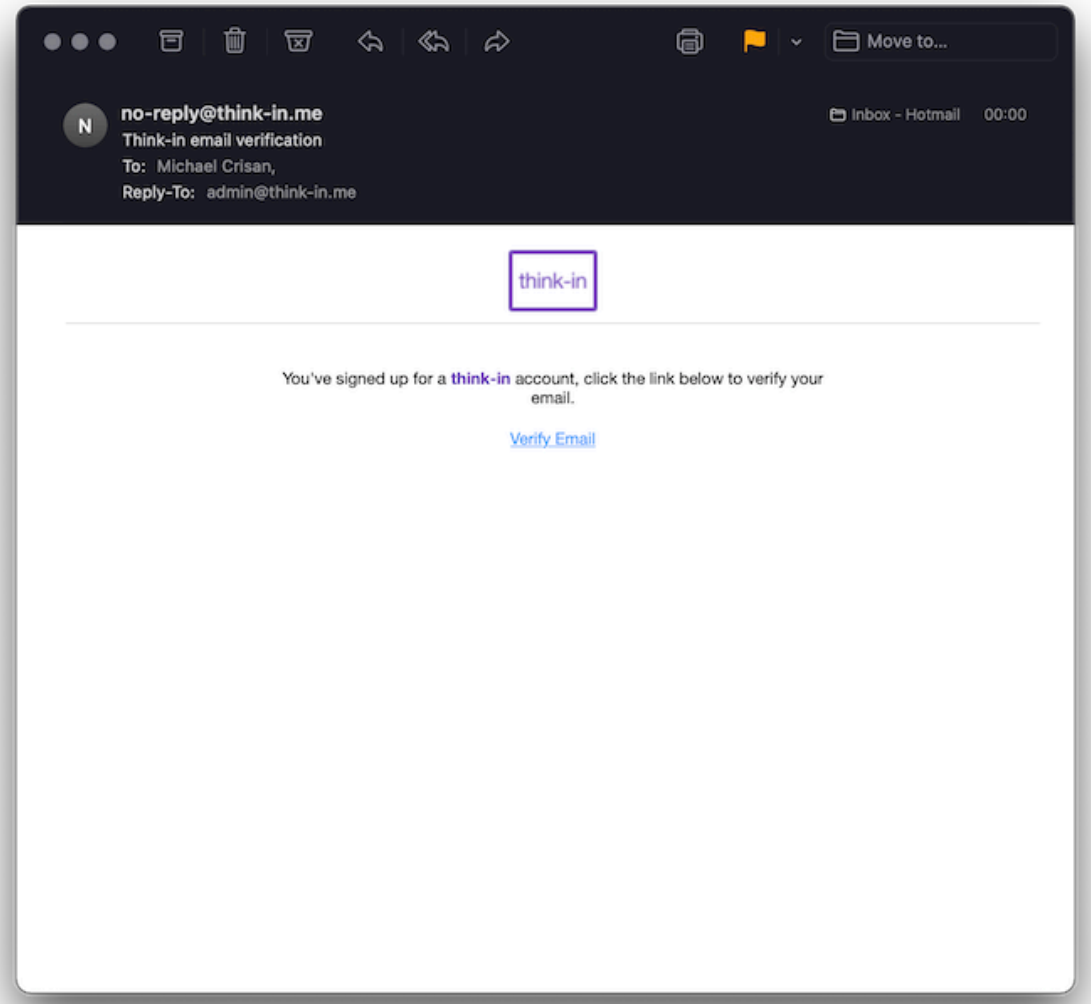

Figure 1.4: Email confirmation showcasing HTML email.

Upon confirming the account and signing in, the user is redirected to the main page.

1.1.2 Main page

The main page is where the users gather and communicate as well as view the media made available by a certain stage.

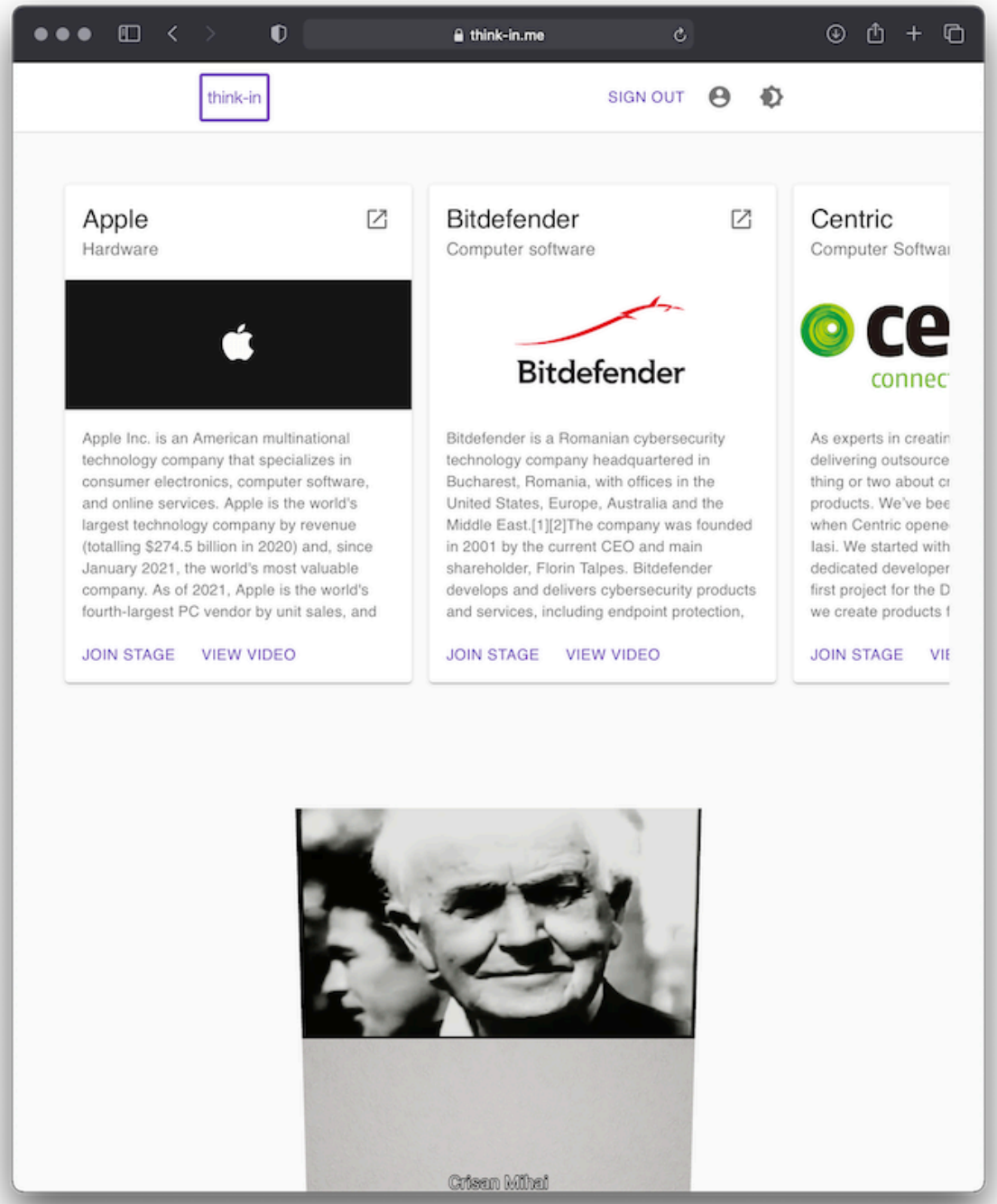

Figure 1.5: Top part of main page when signed in.

In the top part of the main page there is a horizontally scrolling list of stages from which the user can choose to join, every stage has attached a title, subtitle, an external link to the entity representing the stage, a description as well as two actions, *Join Stage* and *View Video*. The first action changes the stage, while the second one gives direct access to the video made available by the entity representing the stage.

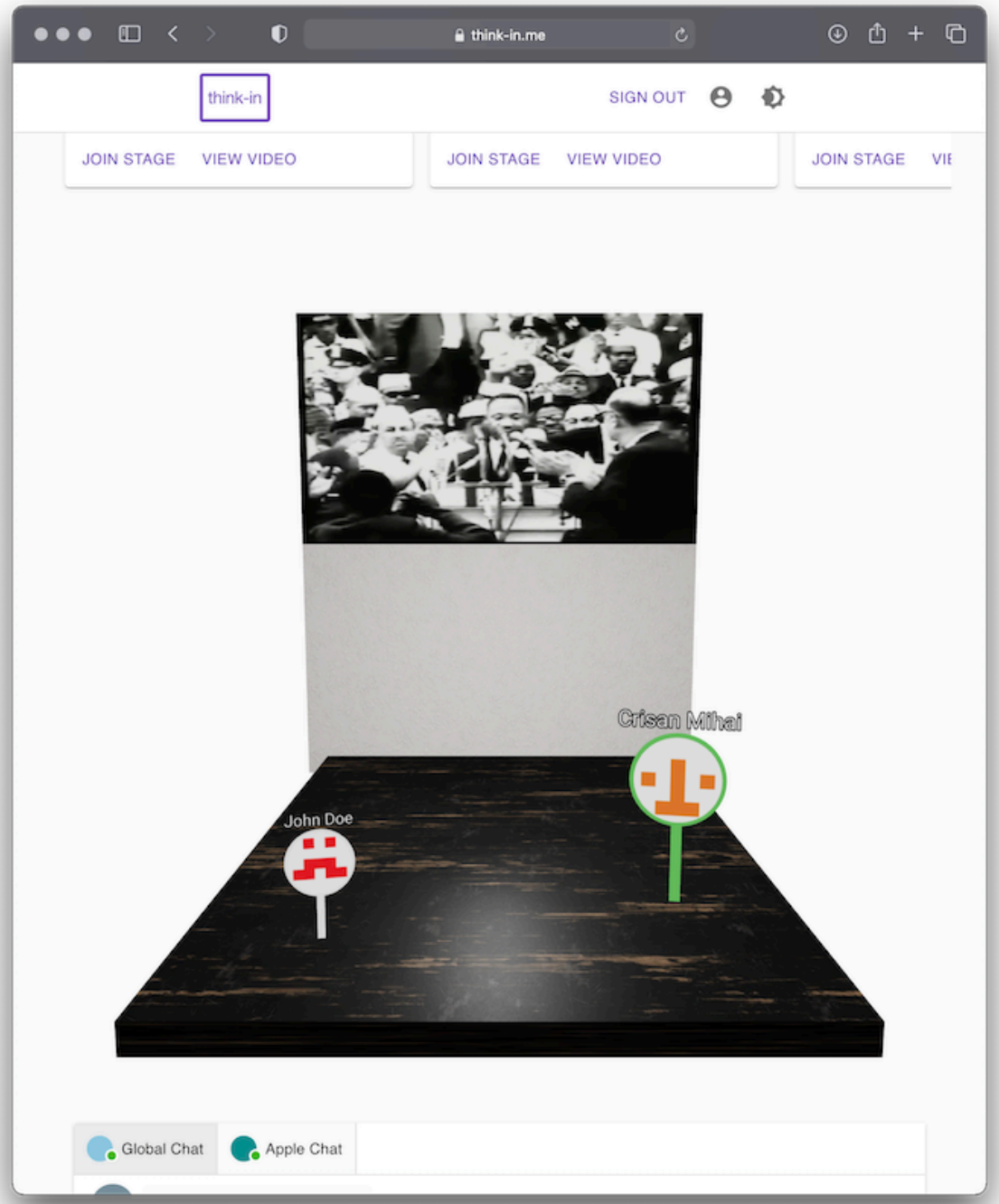

Figure 1.6: Center part of main stage when signed in.

In the center of the main page is the actual stage where users can move their character by simply clicking to the desired location, here the currently present attendees can be seen, in this case *John Doe* and *Crisan Mihai*. As seen in the image above, an attendee is represented by its name and avatar, the current user has a slightly larger character and a green outline instead of grey. The screenshot is taken from the per-

spective of the user *Crisan Mihai*. In the top part of the stage a looped video is playing, upon clicking it a smooth animation will bring the user closer to the playing media.

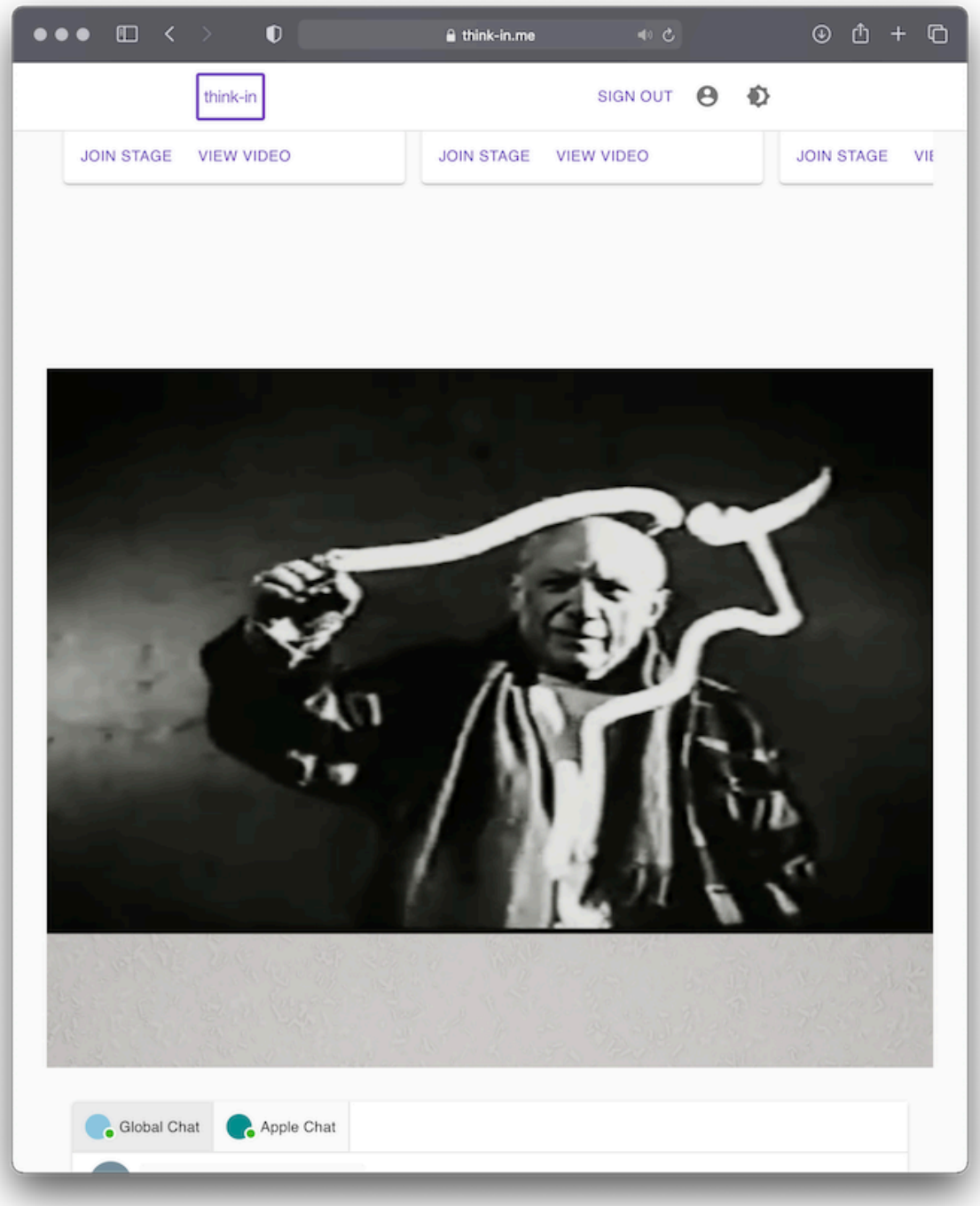

Figure 1.7: Media playing in the stage.

When the user doesn't want to see anymore of the video, he simply clicks again on it and is taken back to the default stage view. In the default stage view, the user can click on an attendee's avatar in order to get more information about him.

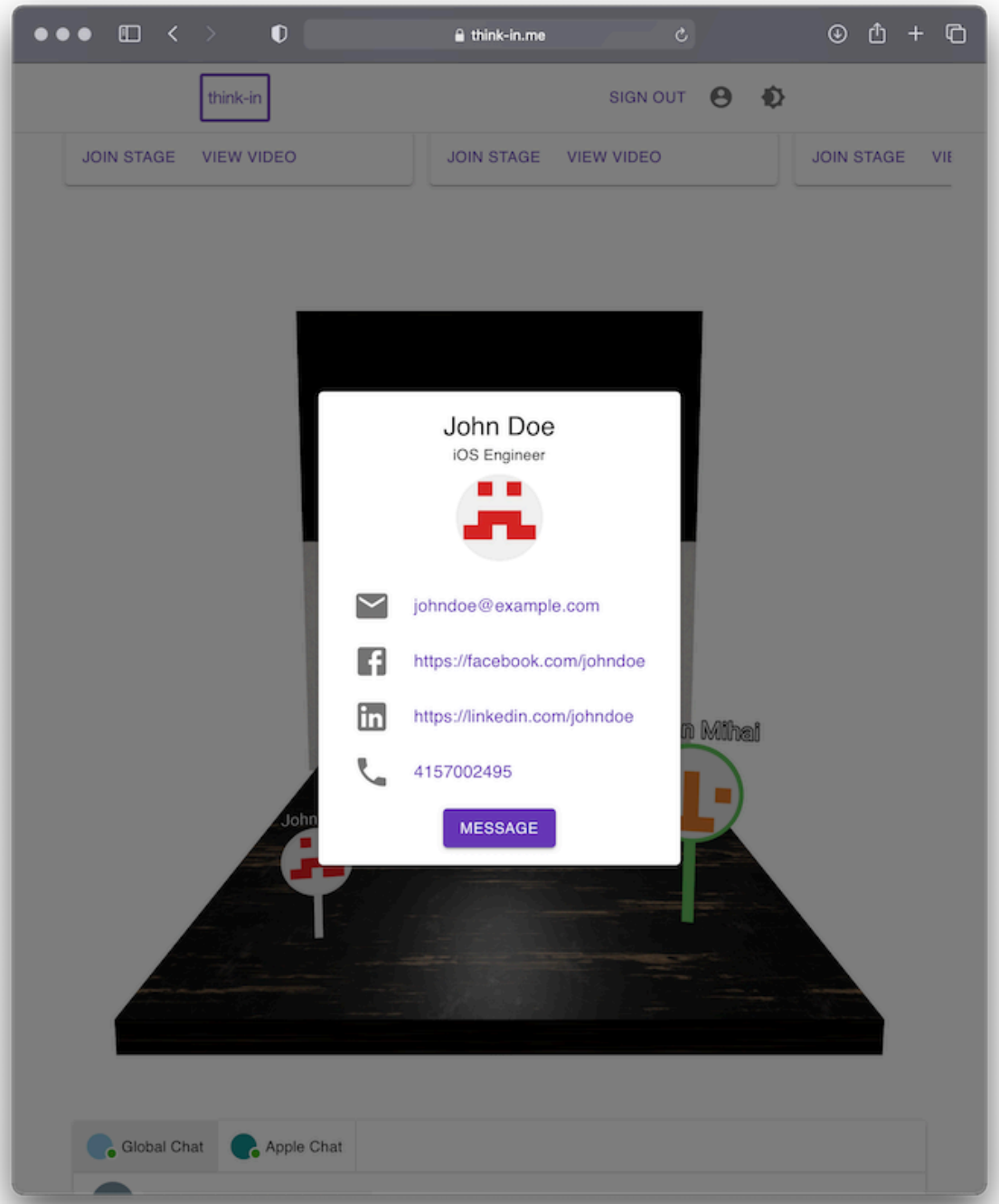

Figure 1.8: Attendee dialog box.

In the above screenshot we have clicked on *John Doe*'s avatar and a dialog has popped up showing us in addition to his name and avatar which we already knew, his job title (iOS Engineer) and various contact links from where we can gather more information about him. Furthermore, there is a *Message* action that allows us to message *John Doe*, by clicking it a new chat is opened and messages can be sent.

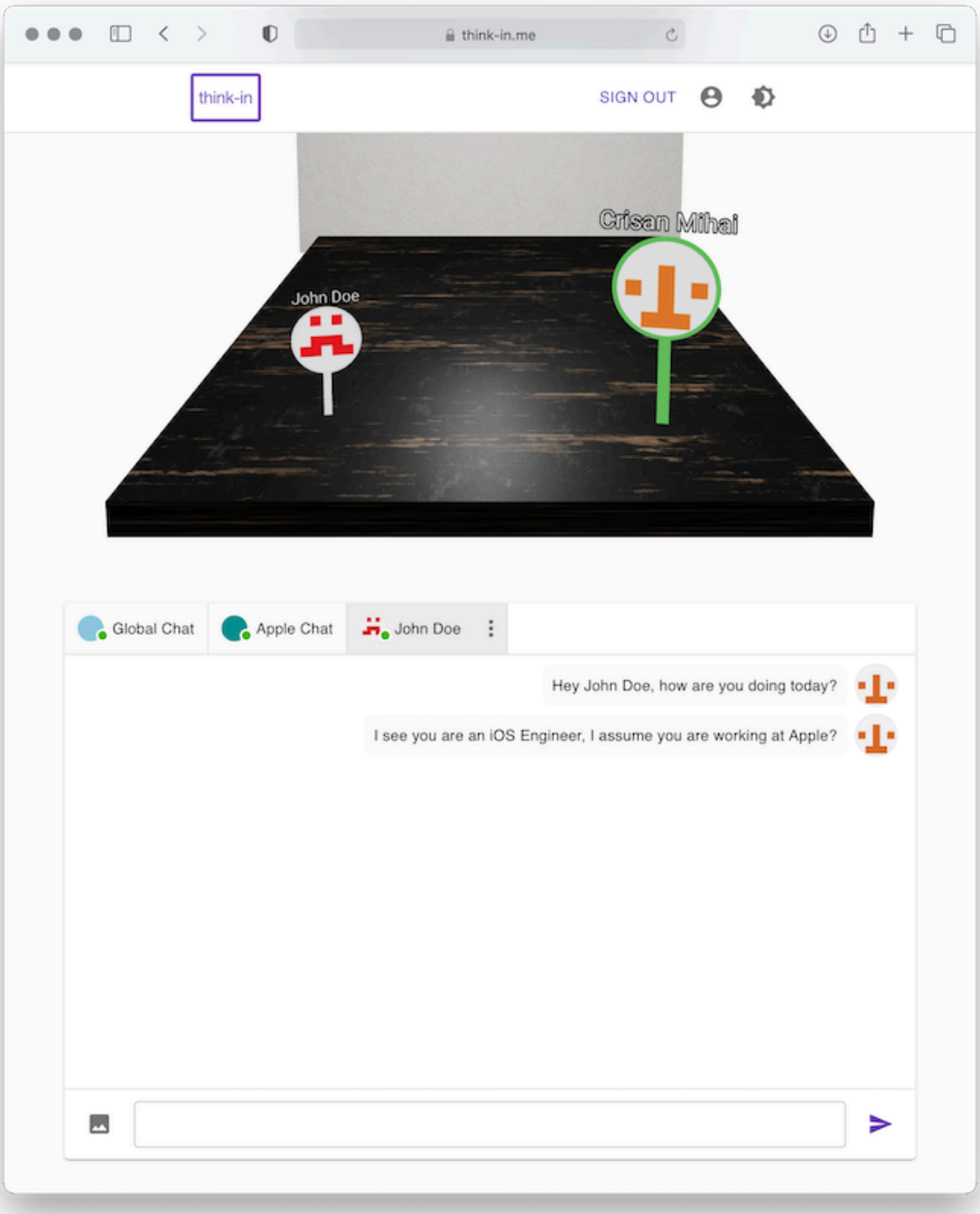

Figure 1.9: Messaging an attendee, perspective of the sender.

The above screenshot is from the perspective of *Crisan Mihai*, what is to be observed is that the messages are translated in your native language, as it is visible in the screenshot below from *John Doe*'s perspective.

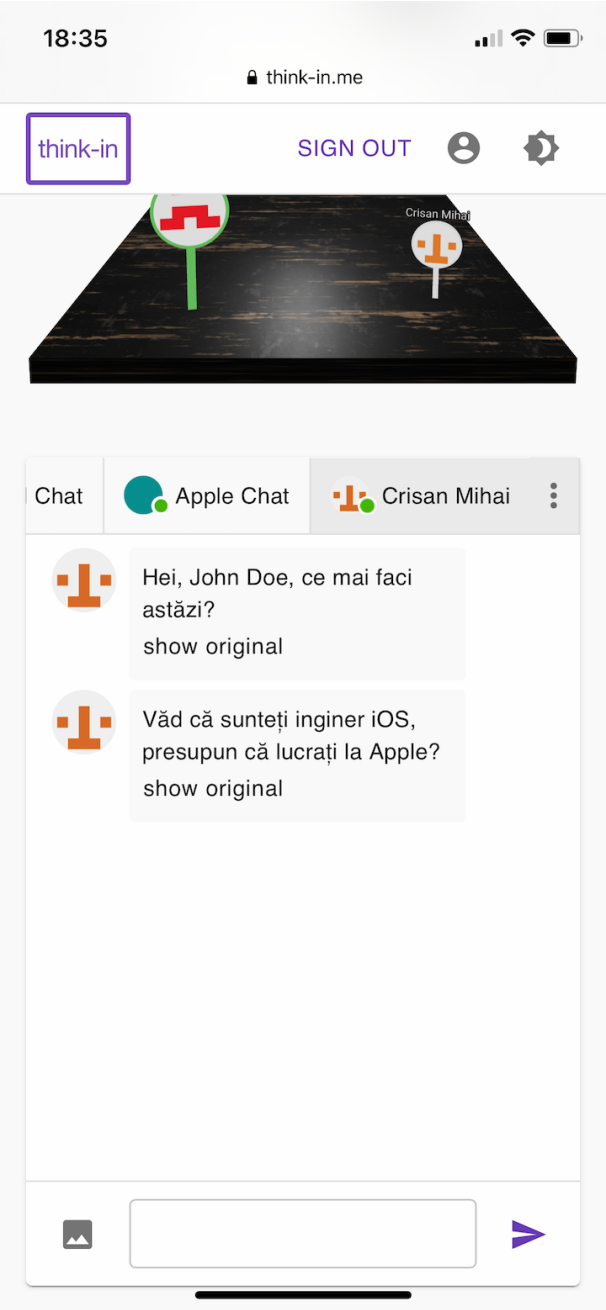

Figure 1.10: Messaging an attendee, perspective of the receiver which has a different language set than the language of the messages.

Because *John Doe* has selected Romanian as his language, all the messages he sees will be translated into Romanian.

Apart from the just described private chat messages that can be sent only between a pair of users, there are two other types of chats:

- Global chat - in this chat the messages are visible to everyone and it is accessible despite of the selected stage
- Stage chat - there is one stage chat for each stage available, i.e. one chat per stage

which is only accessible when the user is in that stage

1.1.3 Profile page

As shown in the previous subsection, *John Doe* has made available links to his social media but nowhere in the sign up process for instance he was asked to provide this information. There is a page where a user can add data about himself, change the avatar or data he has already provided, the language he speaks in and wants to see the messages in. Furthermore, in this same page a user has the option of changing his email or password as well as deleting his account. A screenshot can be seen below.

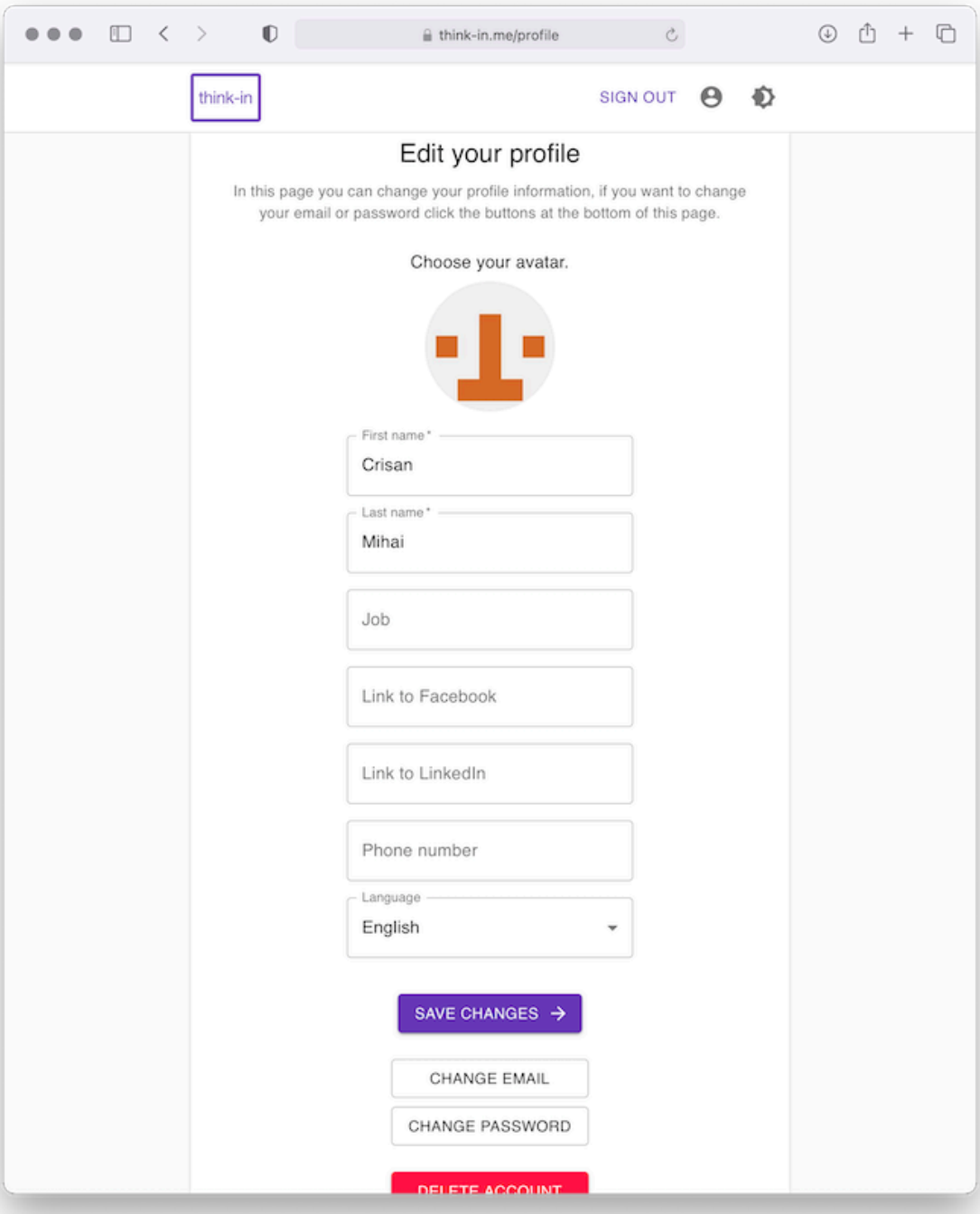

Figure 1.11: Overview of the profile page.

It is worth noting that in the case where the user has signed in through a third party, some of the attributes and actions won't be available, to be more explicit the following are not open to change:

- Avatar, first name and last name - the data used here are provided by the third party identity provider, if they are to be changed, they have to be changed at the

identity provider level

- Change email and password actions - there was no email or password set in the first place, therefore there is nothing that can be changed

1.2 Organizer user flow

The organizer/admin of the application has the same functionalities as a normal attendee plus the capability of adding, updating or deleting a stage. These functionalities are only visible to organizers/admins and are reachable from the Profile page, which was detailed in the previous section's last subsection.

The page where a new stage can be added is shown below, media such as image and video can be drag and dropped into the corresponding areas.

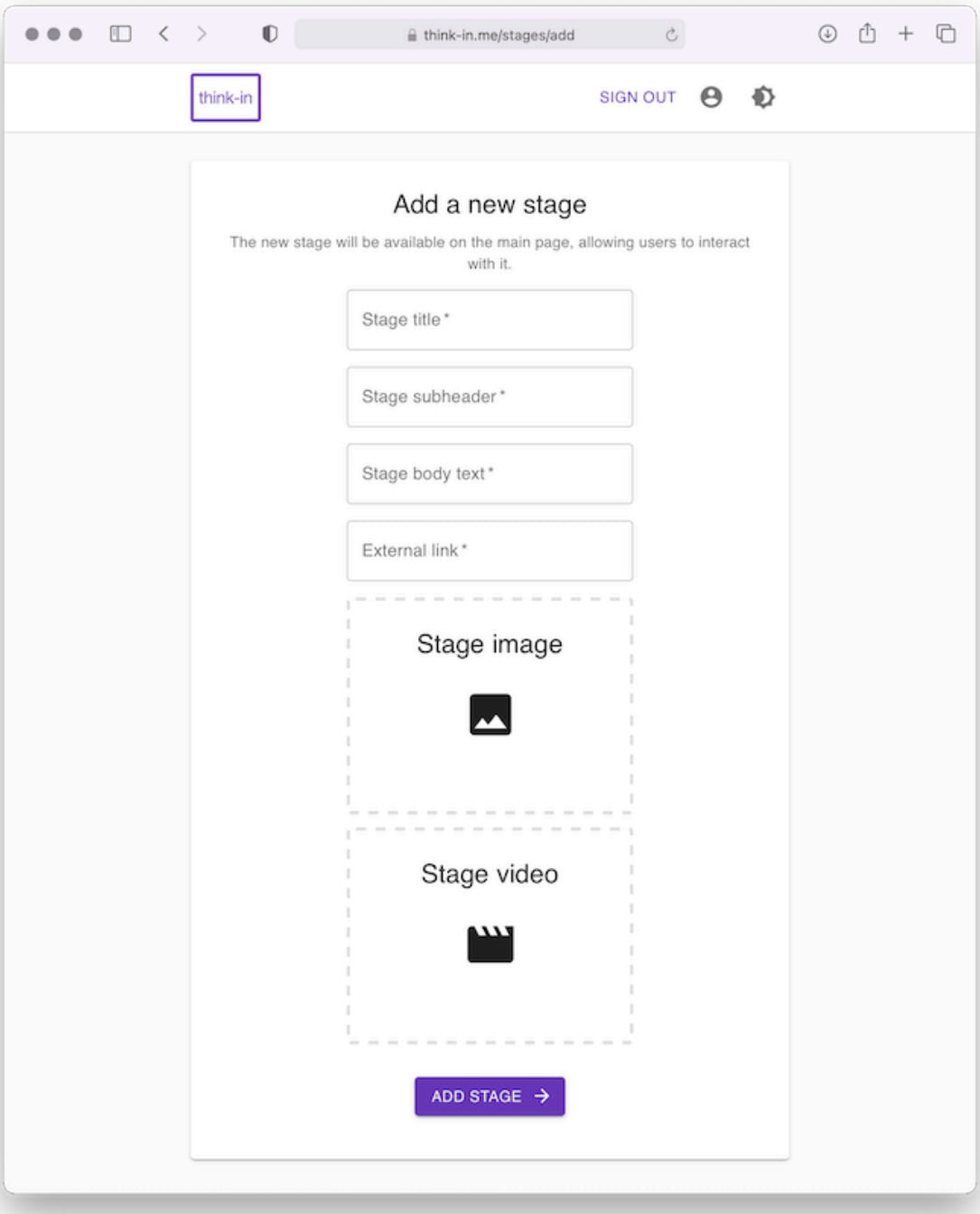

Figure 1.12: Add stage page.

The page where a stage can be edited is very similar to the the one where a stage is added, the difference is that the input fields are filled based on the stage selected for editing. What's more is a delete stage action which is self-explanatory.

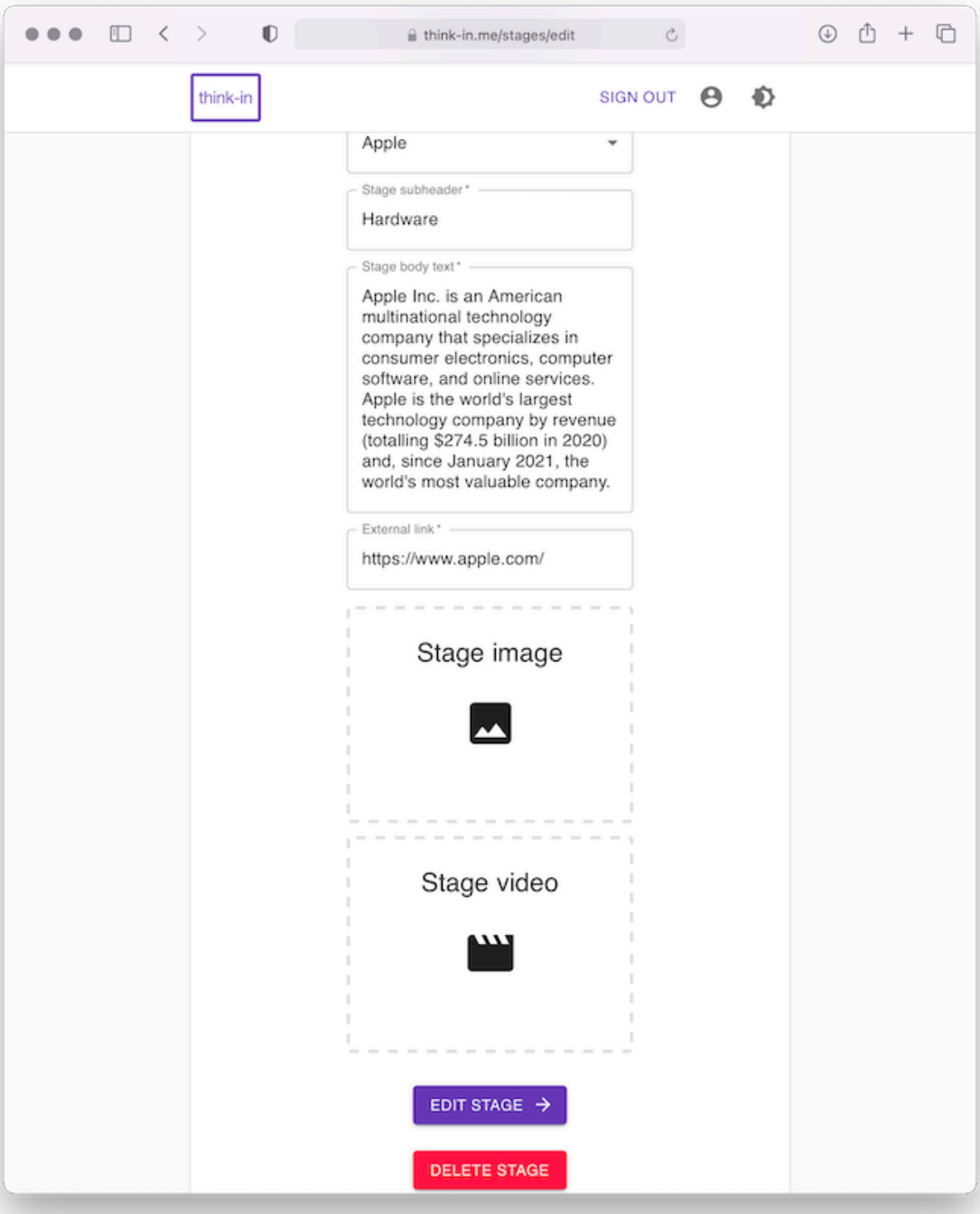

Figure 1.13: Edit stage page.

Chapter 2

Adopted methodology and techniques

Every software application is developed using certain paradigms and methodologies, **Think-In** is no different. In this chapter the choice of the chosen methodology will be explained as well as other used techniques that aided in the development process.

2.1 Analyzing methodologies

Rationally, before making any choice, I analyzed popular software development methodologies and frameworks that have been proven to work in the industry in order to see how they fit to my particular use case:

Waterfall

By definition the waterfall model restricts going back to previous phases and this was very troublesome because for instance it was extremely hard to establish all the requirements beforehand. If for example a new idea appeared in the meantime it would have not been easy to introduce it as a result of how strongly structured this model is. Perhaps a modified waterfall could alleviate this issue but a better model can handle it more effectively.

Scrum

Although extremely popular and effective, the scrum methodology is more destined towards teams that consists of a couple or more members. Since the thesis is intended to be done by a single student this was not the best fit.

Extreme Programming (XP)

The requirements of the application would be changing frequently and extreme programming would accomodate for these changes. The practice of having numerous short releases sounded beneficial because new requirements could be observed once we have a working version of the product.

2.2 Picking a methodology

Considering my case and the above examination it became clear that Extreme Programming was the better option among them. Along the mentioned advantages of XP found above, XP is based on the Agile paradigm and the values of Agile were also found to be appealing. Standout practices of this framework that were used were the following:

- Planning game - Informal explanations such as user stories were used to represent requirements. After writing and estimating the stories, they were sorted by their priority in the sense of the higher the business value they brought the higher the priority. After this operation, the ones with higher priority were chosen to be worked on first. Example of the user stories used:
  - As an attendee I want to move my avatar in the stage by mouse click.
  - In order to get to use the application faster as a user, I can login with a third party instead of creating an account.
  - As a user I want to private message others.
  - In order for other attendees to recognize me, I want to change my avatar.
  - As an attendee I want to switch between stages, so that I can get to know more people and acquire knowledge from the other stages.
  - As a user I want to get more information about another attender.
- Design improvement - There was no hesitance in refactoring code or changing the architecture in favor of a more simple implementation.
- Small releases - By using the released application it was easier to come up with new improvements and suggestions, therefore there were frequent releases.
- Continuous integration - A Continuous Integration and Continuous Deployment pipeline was setup so that the application could be deployed faster and easier.

- Coding standard - Code conventions were established and enforced by tools (i.e. ESLint[1] in this case) in order to eliminate problematic patterns and to keep a consistent style across all the source code. To reduce code comments that are hard to be maintained, an effort was put into writing self-documenting[2] code.
- Simple design - Whenever there was a simpler way to achieve the same functionality the code or architecture was refactored.

As a short conclusion, simplicity was valued throughout the whole project. I think that the conventions I opted for throughout the whole project will make it easier to be maintained over time. Moreover, the conventions used will also make it easier for newcomers to the project to understand the codebase faster.

2.3 Elaborating on requirements

Where additional specifications of a requirement were needed, I have relied on using text based **use cases**. For example, in Table 2.1 we can see the use case I have written before implementing the functionality of changing a user's profile picture.

I also wanted to clarify how an attendee can join a stage because that wasn't clear at first thought either, that use case is visible in Table 2.2.

I wanted to enhance the possibility of getting more information about a user, therefore I made sure there is a way to access more data about an attendee from anywhere he is mentioned. There are three ways in total where a user pops up: from the stage, from an opened chat with that user, or from a message the user wrote. Writing a use case helped in deciding how all of it is going to work in more detail. This use case is shown in Table 2.3.

[1] https://eslint.org/

[2] https://en.wikipedia.org/wiki/Self-documenting_code

Name	Change profile picture.
Description	The user can change its profile picture so its avatar (when browsing the stage) is more recognizable.
Actors	User and system.
Trigger	On the user profile page, the user presses the image indicating his current avatar.
Preconditions	The user is logged in.
Postconditions	The user will have a new profile picture.
Basic flow	1. User goes to profile page. 2. User clicks on picture image. 3. User uploads image from file system. 4. User sees preview of his new profile picture before submitting changes. 5. User clicks *Save Changes*.

Table 2.1: Change profile picture use case.

Name	Join a stage.
Description	An attendee joins a stage.
Actors	User and stage.
Trigger	User clicks the *Join Stage* button of the stage he wants to join.
Preconditions	The user is logged in and at the index page.
Postconditions	The page will refresh, showing the contents of the selected stage.
Basic flow	1. The user is logged in at the index page where there will be a list of stages to choose from. 2. User clicks on the *Join Stage* button of the stage he wants to join. 3. The page will reload with the stage changed.

Table 2.2: Join a stage use case.

Name	Get more information about a user.
Description	While using the application the user sees somebody who he would like to get more information about.
Actors	Current user and target user (the user whose information wants to be seen)
Trigger	The current user clicks on the target user avatar.
Preconditions	The user is logged in and at the index page.
Postconditions	A dialog box will pop up containing more information about the target user.
Basic flow	1. The current user is looking at the stage and a certain user (Target user in this case) attracts his attention. 2. The current user clicks on the target user's avatar. 3. A dialog box pops up with information regarding the target user.
Alternate flow 1	1. The current user has been messaged by the target user and the current user wants to see more information about him. 2. The current user clicks on the three vertical dots button alongside the target user's chat. 3. The current user clicks on *View Profile* 4. A dialog box pops up with information regarding the target user.
Alternate flow 2	1. The current user sees a message in the chat from the target user and wants to see more information about him. 2. The current user clicks on the avatar on the left side of the target user. 3. A dialog box pops up with information regarding the target user.

Table 2.3: Contact a user use case.

Chapter 3

Application infrastructure and architecture

In this chapter I am going to talk about the services that **Think-In** relies on and why I chose them, afterwards I am going to talk about the application's architecture, used technologies and tools.

3.1 Application infrastructure

Think-In is supposed to be an application with people networking in mind, therefore allowing humans far away from each other to connect. Because of this, it had to be available to the public internet. Hosting the application on my own machine would have implied a lot of time spent on maintenance, implementation and let's not even mention the security nightmare that it would have been. The issue of scalability would have been impossible to solve as well. Quickly thereafter I realised that XaaS[1] offerings by cloud computing providers would solve the issues I would face and it would save a lot of time and energy, all while using robust and proven to work software.

3.1.1 Choosing a main cloud platform to use

During my research I have found Gartner[2] to have a great reputation and provide reliable reports on cloud computing service providers. As a result I chose to inspect

[1]Everything as a service: https://en.wikipedia.org/wiki/As_a_service

[2]Research and advisory firm: https://www.gartner.com/en

their famous magic quadrants during which I have found a *Magic Quadrant for Cloud Infrastructure and Platform Services*[3]. Instinctively I would choose a provider with significant age in the field because if I were to face an issue then certainly I wouldn't be first one to do so and I would easily find a fix on the web. Moreover I would expect a cloud provider with experience to have well written documentation about their services, which, because I am a beginner, is essential to me. As a consequence I will consider only the very well known providers as an option.

In the top right quadrant for this specific report (corresponding to the leaders in this domain) I found three providers: Google, Microsoft and Amazon Web Services. All of them have a free offering so I went to research by using each one individually:

Google Cloud Platform

Out of all three providers I found Google to have the best project organization, all created resources are within a logical entity (a project) and that would help me as a newcomer to not confuse resources. Moreover they have the most generous free tier. Unfortunately I found certain services not to work as good as others, for example their API Gateway seems to lack in features compared to the one provided by AWS, it was also hard to read event logs from Cloud Functions.

Microsoft Azure

I have found this platform to be more confusing than others, I also did not like the experience using the web console (e.g. horizontal scrolling), more than this I had trouble using the web console to upload a collection of files under a directory on their object storage service.

Amazon Web Services (AWS)

I "clicked" with the services provided by AWS more than any other cloud provider, they felt simple to use and packed with features. AWS has been available for more time than the other two and is very well established which for sure meant great documentation and lots of resources to learn from. One thing I found inferior compared to others was the organization of resources based on regions.

This being said, I chose Amazon Web Services as the main cloud provider.

[3]https://aws.amazon.com/blogs/aws/aws-named-as-a-cloud-leader-for-the-10th-consecutive-year-in-gartners-infrastructure-platform-services-magic-quadrant/

3.2 Application architecture

Below is attached a simplified overview of the application's architecture, it can be used as a reference when we will go into more detail about the implementation.

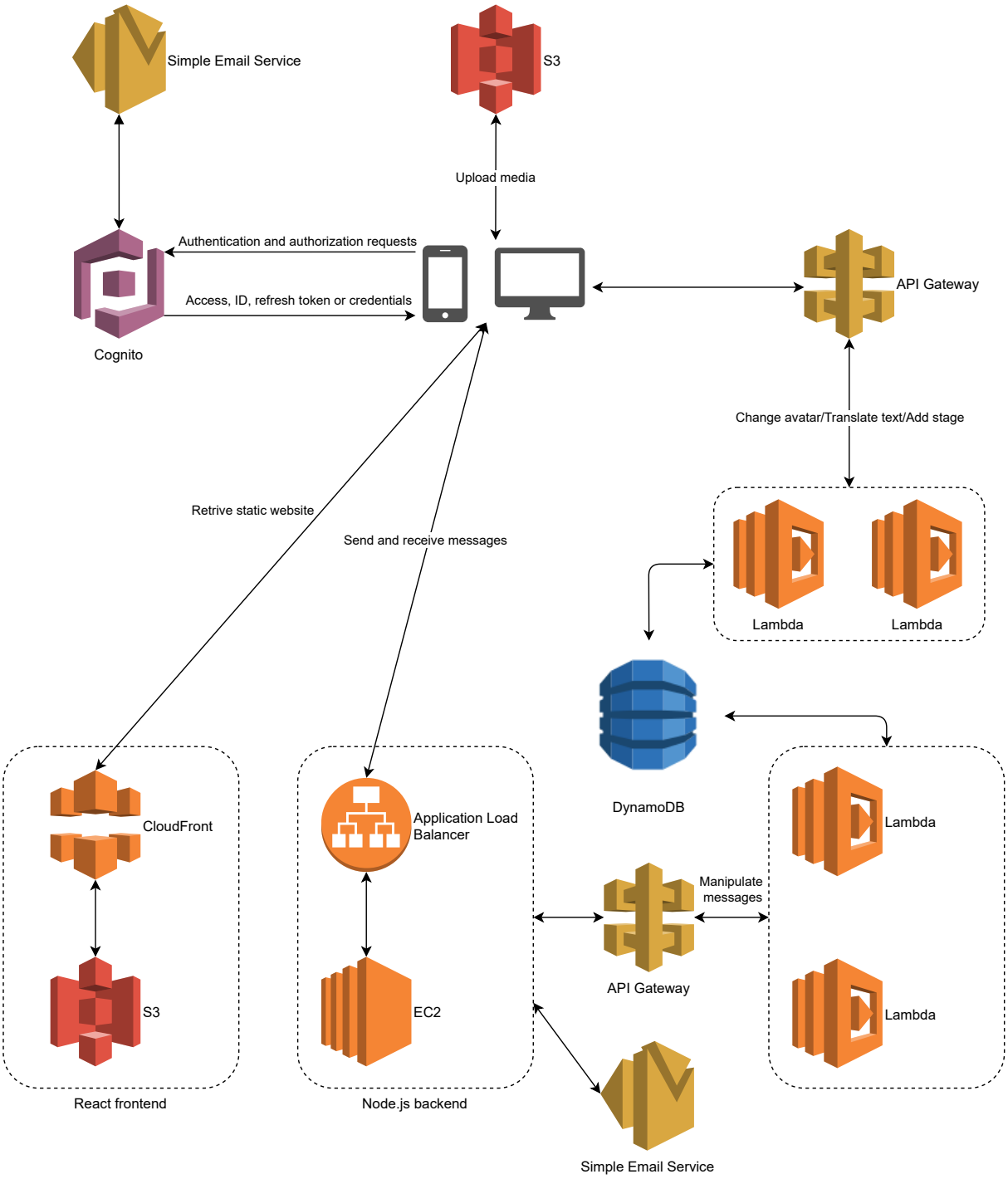

Figure 3.1: Application architecture.

Efforts were put into making the application scale as well as possible, this is why for instance the REST API was created from HTTP triggered Lambda functions, they

scale to ”infinity”. Interesting aspects are that the API is accessed both client-side and server-side, everything that could be done from the client has been done from the client, however as we will see later some actions are only possible on the backend. The backend performs communication using websockets in order to ensure real-time message transmission.

Instead of creating an authentication system I relied that weight on AWS Cognito user pools[4] so that I could easily manage user confirmation, password resets, changing email or other attributes and more without having to build that infrastructure myself. Moreover, Cognito was also used for providing temporary AWS credentials for certain services (e.g S3) so that the client could interact with these services directly.

3.3 Continuous Integration and Continuous Deployment

As talked about in the last chapter, values were found in having short and quick releases, therefore a pipeline that automates the process of deploying the application was set up. Below, in Figure 3.2, is a diagram showcasing the flow of how the application is built.

[4]https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html

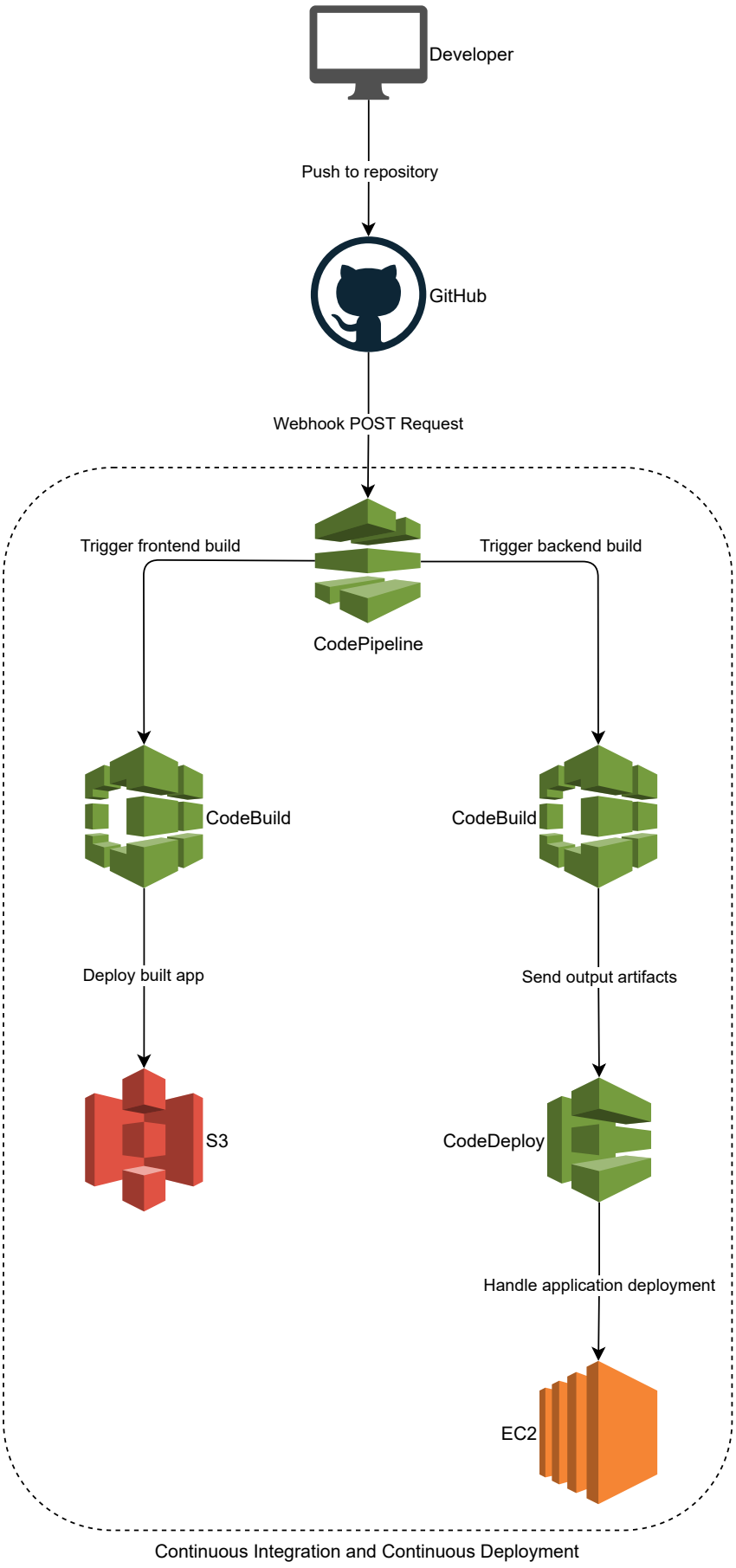

Figure 3.2: Continuous Integration and Continuous Deployment overview.

For those more familiar with CodePipeline, a screenshot of the pipeline from the AWS console is shown in Figure 3.3.

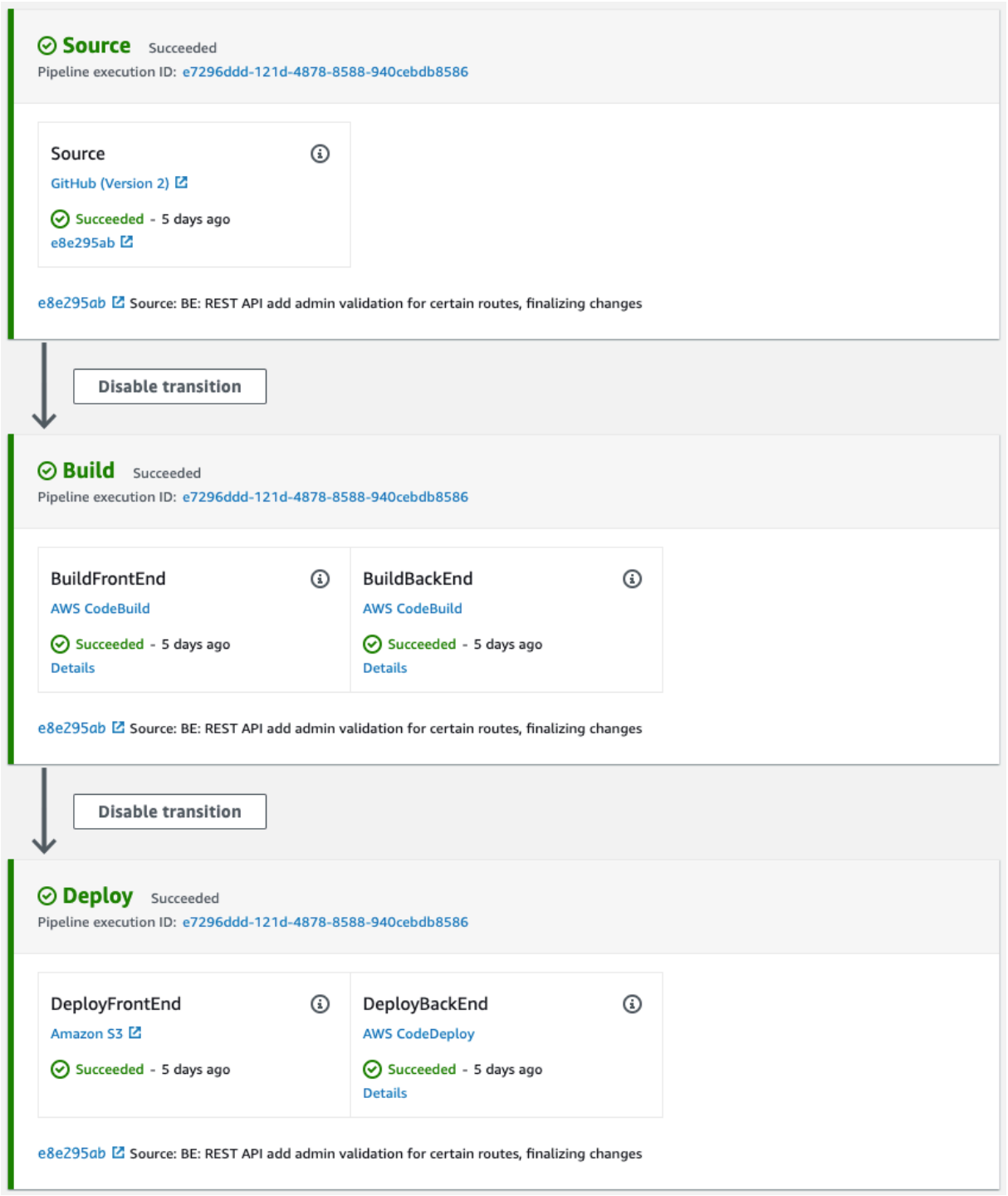

Figure 3.3: CodePipeline from AWS console.

Whenever a push to the GitHub repository is done, a webhook is triggered and GitHub sends a POST request to AWS that starts the pipeline. I set up the pipeline so that two parallel builds are started at once: the **frontend** and **backend** builds.

The code executed for the frontend build can be found in Figure 3.4. In the *install* phase the npm[5] package manager is installed in a non-interactive way (as a result of the `--assume-yes` option) which is needed for installing the application's dependencies. Because all the source code is organized into a monorepo[6], in the *pre_build* phase the working directory has to be changed, in this case to the directory where the frontend application is. Next, on line 14, a clean install of the application's dependencies is done. In the *build* phase I have abstracted the exporting of the application in a single npm script. After the *build* phase is done, the output artifacts corresponding to the built application are put into a S3 bucket from where the CloudFront CDN will serve the static files to clients.

[5]https://www.npmjs.com/

[6]https://en.wikipedia.org/wiki/Monorepo

```yaml
version: 0.2
phases:
  install:
    runtime-versions:
      nodejs: 14.x
    commands:
      - apt --assume-yes update
      - apt --assume-yes install npm
    finally:
      - echo Install phase done.
  pre_build:
    commands:
      - cd app/FE
      - npm ci
    finally:
      - echo Prebuild phase done.
  build:
    commands:
      - npm run export
    finally:
      - echo Build phase done.
artifacts:
  base-directory: 'app/FE/out'
  files:
    - '**/*'
```

Figure 3.4: Frontend CodeBuild specification

On the other side, the code executed by CodeBuild for the backend build can be found in Figure 3.5. The *install* phase is the same as in the frontend build specification. Similar to the frontend, in the *pre_build* phase we change the working directory to where the backend app is and install the dependencies. The first command in the *build* phase creates the environment file on the fly by retrieving the environment variables I

have stored in AWS System Manager Parameter Store[7]. The next command builds the application, after which the output artifacts are passed on to CodeDeploy.

```
1  version: 0.2
2  phases:
3    install:
4      runtime-versions:
5        nodejs: 14.x
6      commands:
7        - apt --assume-yes update
8        - apt --assume-yes install npm
9      finally:
10       - echo Install phase done.
11   pre_build:
12     commands:
13       - cd app/BE/stage_server
14       - npm ci
15     finally:
16       - echo Prebuild phase done.
17   build:
18     commands:
19       - npm run create-env
20       - npm run build
21     finally:
22       - echo Build phase done.
23 artifacts:
24   base-directory: 'app/BE/stage_server'
25   exclude-paths: 'src/**/*'
26   files: '**/*'
```

Figure 3.5: Backend CodeBuild specification

The artifacts are now in CodeDeploy's hands which has the specification shown in Figure 3.6. Lines 3-5 specify the path to where the artifacts will be put on the EC2

[7]Centralized place for keeping secrets/parameters: https://aws.amazon.com/systems-manager/

machine. Lines 7-12 attach shell scripts to different lifecycle events which are self-explanatory, now we will explore each script in detail.

```
1  version: 0.0
2  os: linux
3  files:
4    - source: /
5      destination: /home/ubuntu/websocket-server/
6  hooks:
7    ApplicationStop:
8      - location: ./scripts/application-stop.sh
9    BeforeInstall:
10     - location: ./scripts/before-install.sh
11   ApplicationStart:
12     - location: ./scripts/application-start.sh
```

Figure 3.6: Backend CodeDeploy specification

The *application-stop.sh* script (Figure 3.7) is run for the *ApplicationStop* lifecycle event, here we have to stop the previously running application. We store the process id of the running process in the *pid* variable after which, if there was a process id found, the process is killed. Otherwise, corresponding to the case where there is no process id found, nothing happens.

```
#!/bin/bash
pid=‘ps aux | grep "node␣out/index.js" | grep -v grep | tr -s "␣" "␣"
 | cut -d "␣" -f2‘
# if string is not empty
if [ -n "${pid}" ]
then
    kill -9 $pid
    echo Killed process with pid $pid
else
    echo Found no process with pid $pid
fi
```

Figure 3.7: Backend CodeDeploy application stop script.

Next up is the *before-install.sh* script (Figure 3.8). This script simply clears the directory where the application is.

```
#!/bin/bash
rm -rf /home/ubuntu/websocket-server
mkdir /home/ubuntu/websocket-server
```

Figure 3.8: Backend CodeDeploy before install script.

Before this script is ran, the built files have been copied to the */home/ubuntu/websocket-server* directory. Finally we have the *application-start.sh* script (Figure 3.9). We first navigate to the directory where the application is, we install its dependencies non-interactively and we create the environment file on the fly just like in the build phase. After this everything has been finished and we can start the application by running *yarn start* which corresponds to the command below:

```
nohup node out/index.js > stdout.log 2> stderr.log < /dev/null &
```

```
1 #!/bin/bash
2 cd /home/ubuntu/websocket-server
3 yarn install --non-interactive
4 node `dirname "$BASH_SOURCE"`/application-start-create-env.js
5 yarn start
```

Figure 3.9: Backend CodeDeploy application start script.

3.4 REST API Deployment

For managing the deployment of the REST API I have relied on Serverless[8] framework, it has allowed me to easily deploy AWS resources such as Lambda functions, API Gateway and DynamoDB[9]. For this I have used DevOps practices such as Infrastructure as Code for configuring the resources that are created.

Going further we're going to analyze the configuration code (Figure 3.10) for a Lambda function as an example. On line 2, we specify the path to the function that will be executed when this Lambda function is triggered. Line 4 to 11 indicate that this function is HTTP triggered whenever a `GET` request is sent to `/chats/{chatId}`, moreover it attaches an authorizer so only tokens created by AWS Cognito are approved. Lines 12-17 define the roles this function has, in this case it allows the `dynamodb:Query` action only on a specific DynamoDB table. As observed I have tried to respect the least privilege principle.

[8]https://www.serverless.com/

[9]Fast and flexible NoSQL database: https://aws.amazon.com/dynamodb/

```yaml
get-chats:
    handler: chats/get-chats.getChats
    events:
      - http:
          path: /chats/{chatId}
          method: get
          cors: true
          authorizer:
            type: COGNITO_USER_POOLS
            authorizerId:
              Ref: thinkInApiGatewayAuthorizer
    iamRoleStatements:
      - Effect: 'Allow'
        Action:
          - dynamodb:Query
        Resource: arn:aws:dynamodb:${self:provider.region}:*:table/
        ${self:provider.environment.DYNAMODB_TABLE_NAME}
```

Figure 3.10: Example of configuration code for a Lambda function.

Chapter 4

Frontend

While I had the option of writing pure JavaScript, because this is an application of a larger scale than I am used to I knew that with a dynamically typed language such as JavaScript the project will be harder to maintain as it grows, as a result I much more preferred TypeScript. I wrote type annotations everywhere I could. In retrospective this was one of the best decisions I made, I have avoided type errors that would have taken a long time to discover and the development experience was great thanks to code hinting tools available in IDEs[1].

Observant people might have noticed that the application respects the Material Design[2] guidelines, this is not because I created the components myself to respect the design system but because I chose to use a component library (React Material-UI[3]) that has built-in styling and much more. As a consequence a consistent user interface was achieved, and more importantly a significant amount of developing time was saved due to not having to fiddle with styling or writing boilerplate code, all while reusing robust components that respect semantic markup.

Wherever I had to write components I tried my best to organise them into two categories:

- Smart components - these components maintain state, perform operations on data and they don't have anything to do with view logic, they simply pass the necessary data to the dumb components
- Dumb components - in contrast, these components' sole purpose is to present the data

[1]Integrated development environments
[2]https://material.io/design
[3]https://www.npmjs.com/package/@material-ui/core

Going further I am going to detail the organization of the files and directories, then I will talk about used conventions and dive deep into some details.

4.1 Project structure

Figure 4.1 illustrates the files and directories hierarchy. *buildspec.yml* includes the build specifications, *tsconfig.json* contains the rules regarding transpilation, *.eslintrc.json* is the configuration file for ESLint that enforces code conventions and consistent styling and *.prettierrc.yaml* is the configuration file for the Prettier[4] code formatter. The application's dependencies are installed under the *node_modules* directory and as for any web application the static files are found in the *public* directory. In the pages directory each file corresponds to a page in the web application. Looking at the *src* and *pages* directory we observe that they are mirrored in some way, that is because I have opted to organize page components by features instead of by type. This was caused by a refactoring midway through the project for the reason that components organized by type didn't scale well at all. The opted for file structure brought components used in the same context closer together, thus higher cohesion was achieved, for instance components required by the index page belong to the *components/index* directory. The *components/shared* directory is intended for components that can be reused.

Going into more detail, the *contexts* and *hooks* directories are React oriented. Among the choices I had for managing application state I chose the React Context API because it was the built-in solution for this and so contexts live in the *contexts* directory. Hooks are a way to achieve code reusability and similar to contexts they too exist in their own directory.

[4]https://prettier.io/

```
FE
|- buildspec.yml
|- tsconfig.json
|- .eslintrc.json
|- .prettierrc.yaml
|- node_modules
|- public/
|- src/
   |- components/
      |- profile/
      |- stages/
      |- forgot-password/
      |- index/
      |- sign-in/
      |- sign-up/
      |- shared/
   |- contexts/
   |- hooks/
   |- pages/
      |- profile/
         |- change-email.tsx
         |- change-password.tsx
         |- index.tsx
      |- stages/
         |- add.tsx
         |- edit.tsx
         |- index.tsx
      |- forgot-password.tsx
      |- index.tsx
      |- sign-in.tsx
      |- sign-up.tsx
   |- types/
   |- utils/
```

Figure 4.1: Frontend project structure.

4.2 Used conventions

As a result of writing dozens of files and components I have established some conventions that helped the arrangement of code inside a file.

4.2.1 Absolute imports

Thanks to a feature available in TypeScript that allows to change module resolution, I could use absolute imports therefore avoiding relative import hell. If it wasn't for this feature there would've been non-comprehensible imports like the following:

```
import { component } from "../../../../../components";
```

I have configured the following mapping for imports:

```
#public/*      --> public/*
#components/* --> src/components/*
#utils         --> src/utils
#types/*       --> src/types/*
#shared        --> src/shared
#contexts      --> src/contexts
#hooks         --> src/hooks
```

Therefore, if I had to import a component I would simply type something like:

```
import { component } from "#components";
```

As seen above, I chose absolute imports to be prefixed with the # symbol. At first I chose @ to be the symbol but not too soon after I realized there are npm packages using @ as a prefix and that could lead to potential confusion. The change from @ to # happened due to a refactor.

4.2.2 Order of imports and exports

In some files there is a long list of imports so I decided to structure them in some way, this can be seen for an example file in Figure 4.2. External dependencies are first, followed by absolute imports in the project and relative imports. By examining a significant amount of files I found the relative imports to have depth at most two,

so relative import hell is avoided. I think that this import order convention greatly improves legibility.

```
1  import React, { useContext } from 'react';
2  import { CognitoUserSession } from 'amazon-cognito-identity-js';
3  import { Socket } from 'socket.io-client';
4  import { Emitter } from 'mitt';
5  import axios from 'axios';
6  import { Container, Paper } from '@material-ui/core';
7  import { translateMessage } from '#utils';
8  import { AccountContext, SocketContext } from '#contexts';
9  import type { MessageType } from './shared';
10 import ChatHeader from './chatHeader';
11 import ChatBody from './chatBody';
12 import ChatFooter from './chatFooter';
13 import { AttenderType } from '../stage/shared';
```

Figure 4.2: Frontend import order convention.

I have established a convention about exports as well, they are always placed at the end of the file so whenever somebody wants to see what a file exports they only have to look at the end of it.

4.3 Deep dive

In think section we will go in-depth about certain functionalities that I consider to be important.

4.3.1 Forms with validation

While building the pages required for authentication I found value in building reusable components that performed input validation. In doing so my tools of choice were Formik[5] and Yup[6] which I have augmented in order to abstract and generalize input fields and their validation. In figures 4.3 and 4.4 are snippets of code that showcase

[5]Easier React form validation: https://www.npmjs.com/package/formik
[6]Schema builder: https://www.npmjs.com/package/yup

how the fields in the sign up page are handled, namely Figure 4.3 is the object schema the fields in the form have to respect and Figure 4.4 is the code for the form without boilerplate code in order to increase brevity.

```
1  const validationSchema = Yup.object().shape({
2    firstName: Yup.string().required('Your␣first␣name␣is␣required.'),
3    lastName: Yup.string().required('Your␣last␣name␣is␣required.'),
4    email: Yup.string()
5      .email('Invalid␣email␣format.')
6      .required('Your␣email␣is␣required.'),
7    password: Yup.string()
8      .min(8, '8␣characters␣minimum.')
9      .matches(/\d/, 'A␣number␣is␣required.')
10     .required('A␣password␣is␣required.'),
11   passwordConfirm: Yup.string()
12     .oneOf([Yup.ref('password'), null], 'Passwords␣must␣match.')
13     .required('Password␣confirm␣is␣required.'),
14 });
```

Figure 4.3: Frontend validation schema example.

Notice in Figure 4.4 on line 1 I pass the validation schema object created earlier. The `FormikField` component I built is responsible for displaying errors in case there are any. With this highly reusable setup I created the forms for the profile page in just a couple hours.

```
<Formik validationSchema={validationSchema}>
      {(props): { props: FormikProps<typeof initialValues> } => (
        <>
          <ChooseAvatar
            email={props.values.email}
            avatarSrc={avatarSrc}
            setAvatarSrc={setAvatarSrc}
          />
          <FormikField type="email" name="email"
             label="Email" required />
          <FormikField type="text" name="firstName"
             label="First name" required />
          <FormikField type="text" name="lastName"
             label="Last name" required />
          <FormikField type="password" name="password"
             label="Password" required />
          <FormikField type="password" name="passwordConfirm"
            label="Password Confirm" required />

          <Button
            endIcon={<ArrowForward />}
            variant="contained"
            color="primary"
            type="submit"
            disabled={props.isSubmitting}
          >
            Sign up
          </Button>
        </>
      )}
    </Formik>
```

Figure 4.4: Frontend simplified sign-up form code.

4.3.2 3D stage

One of the more interesting areas of the application is the 3D stage available at the index page. Instead of using the WebGL[7] API, which would have meant going extremely in detail with 3D graphics, I picked a popular JavaScript library named Three.js[8]. One issue I had to deal with was the imperative programming nature of this library and its incompatibility with React, thankfully though multiple people in the same situation as me were inspired to build a wrapper-like library (@react-three/fiber[9]) that provides both the declarative aspect of React and the abstractness of working with 3D of Three.js. The advantage was that code could be organized in React components, unfortunately in some situations imperative code had to be written because there was no other choice to achieve certain functionality.

Because the code that contains the logic for rendering the 3D scene is filled with intricacies of the libraries I used I am going to give a high level explanation instead. I will describe each component of the 3D scene:

- `StageFloor` component - consists of the ”walking area” where the attendees are moving on, its geometry is a box and its material has attached a dark wood-like texture. It is responsible for intercepting clicks by the use of ray casting and for adding the arrow object as a visual feedback for clicking. It also transmits the coordinates of where the casted ray intersected via an event to the `AttenderManager` component.

- `Wall` component - this is the simple background sitting behind the playing video, it has a box geometry and a white material texture.

- `Screen` component - the plane where the media is seen, the component receives a link to a video or image and fallbacks to a generic image in case none of the provided links point to a valid source. It is responsible for muting and unmuting the video sound when it is clicked.

- `AttenderManager` - this component deals with showing the avatars of the attendees in a stage, it communicates with the backend server in order to receive position changes and other user data.

[7]Web Graphics Library: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
[8]https://threejs.org/
[9]https://www.npmjs.com/package/@react-three/fiber

4.3.3 Chat component

The `ChatManager` component handles the chat functionality. It communicates with the backend using websockets and with S3 directly when photos are sent. It reacts to twelve events in total which can be split into two categories: **local emitted** events and **server emitted** events. A short description will be given to each one of the events.

The local emitted events are emitted and received by the client, this was used in order to decouple components. The local events are the following:

- `close-chat` - handles the logic when a chat is closed
- `opened-chat` - triggered by the *Message* action within a user pop-up, adds and opens the new chat
- `change-chat` - triggered when a new chat is selected
- `private-message` - triggered when a message is sent, it appends a message and sends it to the backend so the message gets delivered to the other chat participant(s)
- `file-upload` - triggered when a photo is selected to be uploaded from the file system, it queries the backend for a signed URL with which the client can directly upload the photo to S3
- `chat-messages` - triggered when the top of the messages has been reached, it queries the backend for the next page of messages if available

The server emitted events are emitted by the backend server and received by the client. The server emitted events are the following:

- `put-file-signed-url` - this event occurs only after a locally emitted `file-upload` event, the content of this event is a signed S3 URL which allows data upload
- `notifications` - emitted when the user connects and had pending notifications that now need to be shown
- `chat-messages` - first the client emits this event to the server, after which the server responds with the event of the same name containing messages of a newly selected chat

- `private-message` - emitted by the server when the client has received a new message
- `connected-users` - emitted when the user has connected to the chat, receives data about which users are online
- `user-state-change` - emitted by the server when a client has connected/disconnected

It is important to specify what data are used here, each chat respects the contract specified by the TypeScript `interface` in Figure 4.5. The `UserAttributes` interface is shown in Figure 4.6. The fields suffixed by a question mark are optional, some fields were eliminated in order to increase brevity.

So, each chat contains how many notifications there are from a user, whether that user is online or not, whether that chat is the current selected one and a lot of information about the user in question. The reason for holding all the information specified by the `UserAttributes` contract about a user is in order to get fast access to more information about that user, so no additional lengthy network requests are made.

```
1 interface HeaderChatType {
2   user: UserAttributes;
3   notifications: number;
4   online: boolean;
5   selected: boolean;
6 }
```

Figure 4.5: Format of the information of a chat.

```
interface UserAttributes {
  picture: string;
  id: string;
  email: string;
  emailVerified: boolean;
  givenName: string;
  familyName: string;
  groups?: string[];
  roles?: string[];
  preferredRole?: string;
  customFacebook?: string;
  customLinkedin?: string;
  customPhone?: string;
  customJob?: string;
  customLanguage?: string;
}
```

Figure 4.6: Format of the information of a user.

For the actual messages exchanged in the chat, the formats of the data are those shown in Figure 4.7. Here we don't store so much information about the user and a network request has to be done if somebody wants to gather more information about the user that sent a message. The justification here is obvious, there would have been simply too much redundant data if all the user information was stored alongside a message.

So, in total, for a message we have the originator's ID so we can obtain more information about him, his name and picture to show in the chat, the MIME type of the sent message so we know what to show on the frontend, the actual data of the message which can be plain text or a link to S3 of the media file uploaded and the time when the message was sent. Additionally, if the message type is plain text, an `ISO-639-1` language code will be attached to the message in order to accomplish the text translation functionality, or if the message contains media it will contain an alternative text describing the image.

```
1  type UserInformationType = {
2    id: string;
3    name: string;
4    picture: string;
5  };
6
7  interface BaseMessage {
8    userInformation: UserInformationType;
9    type: string; // MIME type of messages
10   data: string;
11   time: number;
12 }
13
14 interface TextMessageType extends BaseMessage {
15   // ISO-639-1 language code
16   language: string;
17 }
18
19 interface MediaMessageType extends BaseMessage {
20   // alternative text description of the media
21   alt: string;
22 }
```

Figure 4.7: Format of a chat message.

Sharp-eyed readers might have noticed that both the `AttenderManager` and `ChatManager` components communicate with the backend, however they are not coupled by any means.

4.3.4 Restricting access to protected routes

Just like a lot of other applications, some routes requires the user to be authenticated. For this I have created a reusable component that wraps the protected component, it allows access if the user is authenticated and it redirects the user to the index

page if not. Figure 4.8 shows legible code that can be understood even by those unfamiliar with the used tools and technologies.

```
1  const RequireAuthentication: React.FunctionComponent =
2    ({ children }) => {
3    const router = useRouter();
4    const { isLoggingIn, isLoggedIn } = useUser();
5
6    if (isLoggingIn) {
7      // loading user state
8      return <></>;
9    }
10
11   if (!isLoggedIn) {
12     // if user is not logged in, redirect to index page
13     router.push('/');
14     return <></>;
15   }
16
17   return <>{children}</>;
18 };
```

Figure 4.8: Require authentication reusable component.

4.3.5 Computing language to translate messages to

Not every user will go to the profile page and change their preferred language, therefore fallbacks had to be implemented in this regard. The function that performs this computation is shown in Figure 4.9:

```
1 const getUserLanguage = (user) => {
2   if (user.customLanguage) return user.customLanguage;
3   if (window.navigator.language) return window.navigator
4                                          .language.split('-')[0];
5   return 'en';
6 };
```

Figure 4.9: Function that computes the user language.

The return value has to be a ISO-639-1 language code, hence the splitting at line 4. The `window.navigator.language` value should be a string representing the preferred language of the user, in case that doesn't exist as the last fallback we choose the english language.

4.3.6 Accessibility features

When clicking on the playing media in the stage, an animation brings the camera closer to the plane whose texture is that of the media. Depending on the device the camera position will cut more or less of that plane, therefore I have applied principles similar to responsive web design by adjusting the camera position based on the width of the page.

Another accessibility feature is dark mode so during night time (or low light conditions in general) the user's eye strain is reduced. The toggle for light/dark mode is found in the rightmost upper right button in the page. Example of how the page looks with it toggled on is shown in Figure 4.10.

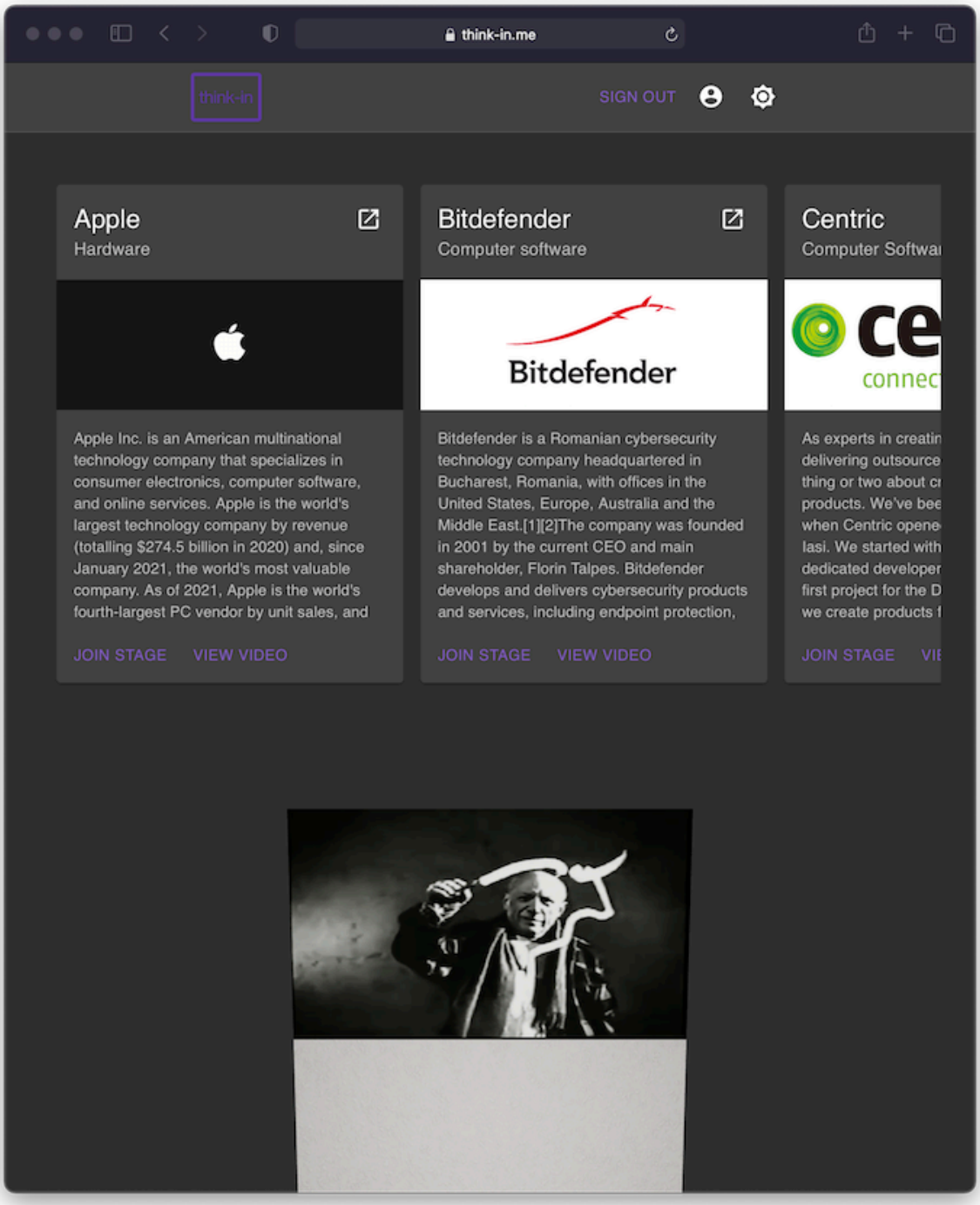

Figure 4.10: Frontend dark mode.

Moreover, I have attempted to fix the case where two attendee's avatars are overlapping such as shown in Figure 4.11, here we would have trouble clicking on *User Two*'s avatar because *User One* is sitting right in front of him. One could imagine this could become a real problem when a large amount of people are connected.

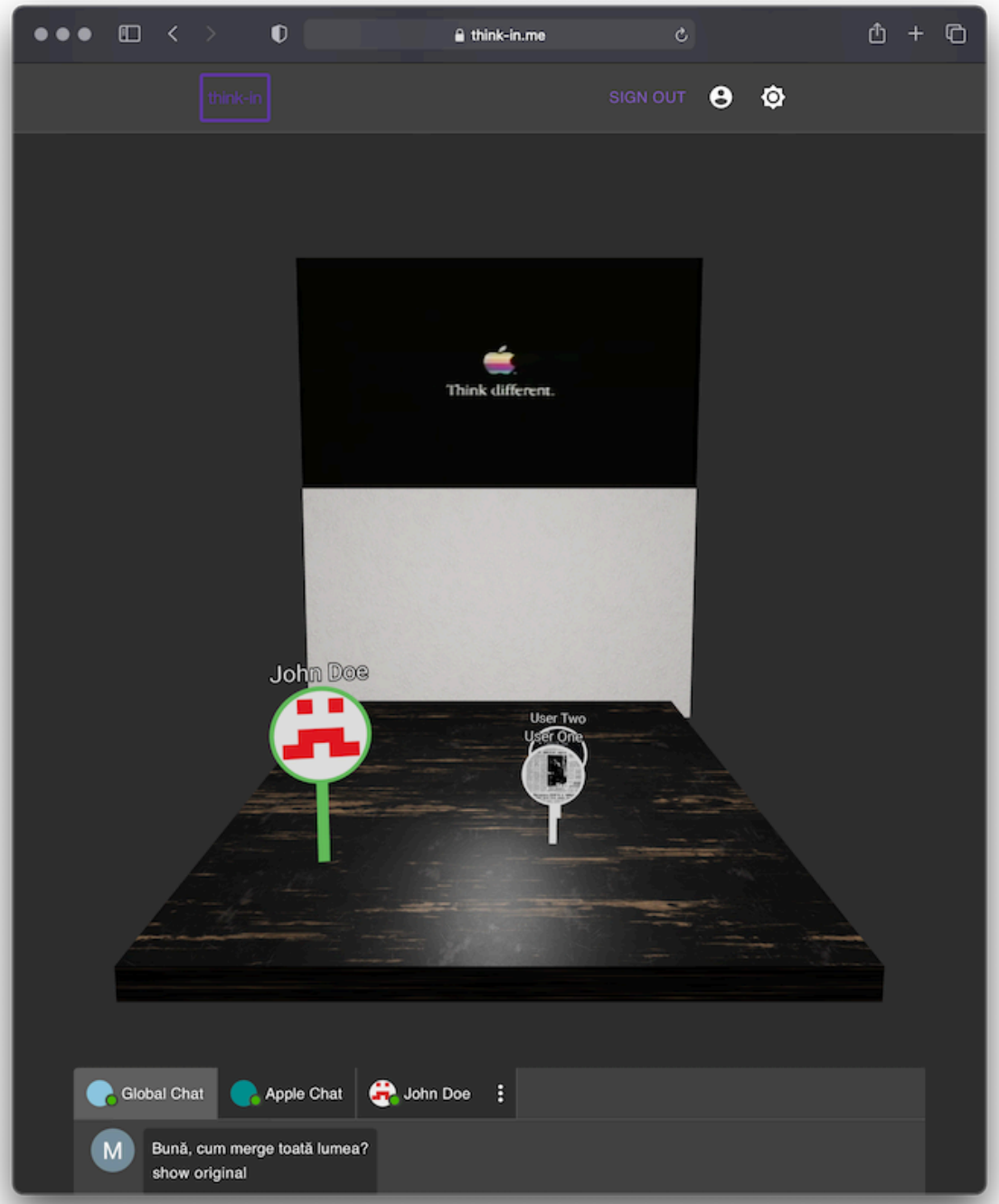

Figure 4.11: Overlapping original camera view.

As an attempt to fix the issue just described I allow the users to control the camera by moving it around the origin as well as zooming in. An example of when a user has moved the camera is shown in Figure 4.12, here the user has moved the camera just enough in order to see *User One* and *User Two* not overlap anymore.

Figure 4.12: Non-overlapping camera view.

Chapter 5

Backend

The backend of **Think-In** is logically split into two pieces and so this chapter will be split into two subsections where each piece will be talked about in more detail. One of the logical entities of the backend is the EC2 virtual machine running a HTTP server where communication is done through the WebSocket protocol. The other logical entity is where the HTTP triggered Lambda functions reside, this entity serves as the API of the application.

5.1 The server that handles in real-time communication

This server handles stage movements and chat messages in an event-driven and very quick way thanks to the WebSocket[1] communication protocol and the Socket.IO[2] library. Justification for choosing the Socket.IO library is that it can fallback to HTTP polling in case communication through websockets can't be done, as well as providing abstractions that help in accelerating the development process such as: liveness detection and maintenance, reconnection in case of errors, encoding sent data as JSON and rooms.

5.1.1 Project structure

Just like in the previous chapter where I have talked about the frontend I will talk about the organization of files and directories and detail certain files. The file hierarchy is shown in Figure 5.6. We have met the *buildspec.yml*, *tsconfig.json*, *.eslintrc.json*

[1]https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[2]https://socket.io/

and *.prettierrc.yaml* files in the frontend, they serve the same purpose here so talking about them again is unnecessary. The *appspec.yml* file contains configurations for CodeDeploy and the *scripts* directory contains the scripts ran by CodeDeploy, these have been detailed in the infrastructure chapter. Exploring the files in *src/*, the *socketio* directory contains files whose logic handles chat and stage interactions, more precisely code in *chats.ts* manages chat messages and the code in *stages.ts* handles stage interactions. Eagle-eyed readers will find a correlation between these two files and the *ChatManager* and *AttenderManager* components in the frontend, that's no coincidence because code in these two files emit the events handled by the two components. *socketIOServer.ts* aggregates the code in these two files and creates a HTTP server to be used in the *src/index.ts* file. The *types/* directory contains type declarations that simply augment the dependency type declarations. In the *utils/sesEmail* directory we find one HTML email template and one plain text email template as well as code using the AWS SES[3] service. The *src/index.ts* serves as the entry point of the application and the *src/shared.ts* file contains common code utilized throughout the application.

[3]https://aws.amazon.com/ses/

```
STAGE_SERVER
|- buildspec.yml
|- appspec.yml
|- tsconfig.json
|- .eslintrc.json
|- .prettierrc.yaml
|- node_modules/
|- scripts/
|- src/
   |- socketio/
      |- chats.ts
      |- stages.ts
      |- socketIOServer.ts
   |- types/
   |- utils/
      |- sesEmail/
         |- email.txt
         |- email.html
         |- sesEmail.ts
   |- index.ts
   |- shared.ts
```

Figure 5.1: Backend server project structure.

5.1.2 Decoupled logic

Going forward I would like to detail the *src/socketIOServer.ts* file. I have purposely stripped code in order to improve readability. Here an implementation of the Chain of Responsibility[4] pattern was used, which is made visible between lines 4-20 in Figure 5.2 where the authorization of a user is done based on whether or not the provided JWT upon socket connection is valid or not. In the *createSocketIOServer* function we simply attach the middleware and listeners to the server instace (represented by the *io* variable here. Observe how the stage and chat listeners are in no way dependent

[4]https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern

upon each other, this can be useful when only one of these functionalities is wanted. Even more, these functionalities can be split among servers, therefore achieving easier scalability on an exact feature.

```
1  import registerStageListeners from './stages';
2  import registerChatListeners from './chats';
3
4  const registerGeneralMiddleware = (io: SocketIOServer) => {
5    ['/stages', '/chats'].forEach((namespace) => {
6      io.of(namespace).use(async (socket, next) => {
7        try {
8          const idTokenDecoded =
9            await validateJWT(socket.handshake.auth.idToken as string);
10         socket.idToken = socket.handshake.auth.idToken;
11         socket.idTokenDecoded = idTokenDecoded;
12         socket.attender =
13           JSON.parse(socket.handshake.query.attender as string);
14         next();
15       } catch (e) {
16         next(new Error('Not␣authenticated.'));
17       }
18     });
19   });
20 };
21
22 const createSocketIOServer: CreateSocketIOServer = (httpServer) => {
23   const io = new SocketIOServer(httpServer, socketIOOptions);
24
25   registerGeneralMiddleware(io);
26   registerStageListeners(io);
27   registerChatListeners(io);
28
29   return io;
30 };
```

Figure 5.2: Backend *socketIOServer.ts* file

5.1.3 Handling messages

Sending the message when both the originator and the receiver of the message are online is trivial, the server simply relays the message to the receiver. However, this logic gets more complex when the receiver is offline. The code that is executed whenever a message is sent is shown in Figure 5.3, changes regarding comments, formatting and removing requests headers were made to it in order to fit in the page. Lines 2-6 deals with messages sent to the Global/Stage chat, these two chats are perceived as special users and are always considered online. Lines 6-24 manage messages sent to users in the following way: the condition at line 8 is equivalent to checking if the user is online, if so simply relay the message, however if the user is offline we wish to inform him of the message via email, as well as notify him next time he opens the application. In case he receives more messages from a single user, we want to avoid sending multiple emails to the user and instead send a single email informing the user of multiple potential messages. This logic is implemented between lines 12-25. Finally, lines 29-31 persist the message to the database so they can be retrieved even through client sessions.

```
// toUser, message, isGlobalOrStageChat are sent by the client
if (isGlobalOrStageChat) {
  // message was meant for the global or stage chat
  io.of('/chats').in(toUser.id).except(socket.id)
    .emit('private-message', { message, fromUser: toUser });
} else {
  // message was meant for a user, check if the user is online
  if (userIdToSocketIdMap.has(toUser.id)) {
    // toUser is online, directly send message
    socket.to(userIdToSocketIdMap.get(toUser.id) as string)
          .emit('private-message', { message, fromUser: socket.attender });
  } else {
    // toUser is offline

    // send email to user regarding this message only if this is the first
    // unread message
    // verify there are no other unread messages before sending email
    const response = await axios.get(`${API_URL}/notifications/${toUser.id}`);
    if (response.data.data.items.length === 0)
        await sendEmail({ destinationEmail: toUser.email,
          fromUser: socket.attender, message });

    // save notification to DB
    await axios.post(`${API_URL}/notifications/${toUser.id}`, socket.attender)
  }
}

// save message to DB
const url = new URL(`${API_URL}/chats/
    ${computeChatId(toUser.id, socket.attender.id)}`);
await axios.post(url.toString(), message);
```

Figure 5.3: Handling messages.

The above logic applies whether the message is plain text or media. Now let's see how sending plain text messages differs from sending media messages. Sending a

plain text message is straightforward both for the user and also from the developer's point of view, an event is emitted from the client with the data of the plain text message. Sending a media message however implies more than one single round trip, a sequence diagram[5] is shown in Figure 5.4 which shows the interactions needed in order to send one such message.

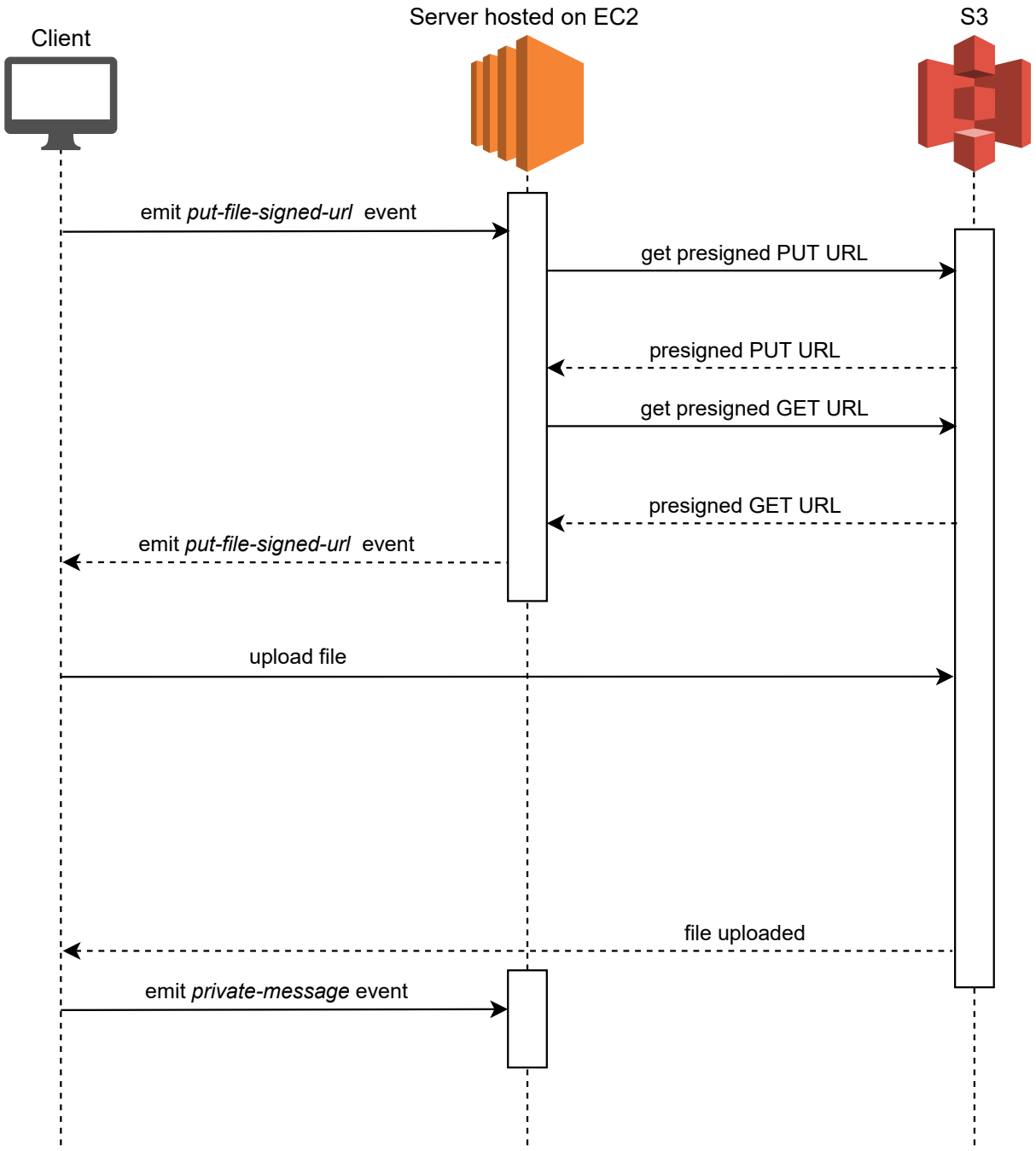

Figure 5.4: Interactions necessary for uploading media.

[5]https://en.wikipedia.org/wiki/Sequence_diagram

As seen from the diagram, the client uploads the file directly to S3. In order to do so, the client first asks the backend server for a presigned PUT URL, used for uploading the photo, and a presigned GET URL, which is valid only for a day and used later in order to send the message in the chat. Had I learned about Cognito Identity before doing this it could have been done differently, the client could get authorization credentials straight from Cognito and the entire communication with the server before uploading the file would be omitted.

5.1.4 Securing messages

For securing plain text messages, access to the API is granted only to those who are participants of a chat. First of all the JWT is verified and if valid then it is checked whether or not the participants of the chat include that user, if so then the user is authorized.

The media messages are stored in a private S3 bucket and they are not accessible to the public. For retrieving messages from a chat the server is asked to do so, the server then gets them from the API and presigns media messages' URLs after which the links to media are valid for one day (can be changed to any amount of time). Advantages are that, if the user wishes so, he can share the link to the media to non-participants of the chat. A disadvantage to this approach is that the server is involved, by the time I realized this could be done with a request to a HTTP triggered function the logic was already implemented.

5.1.5 Using HTTPS

Because there can be private data transported the server had to be secured. The way I achieved the server to run over HTTPS is somewhat unexpected. Another way I think this could have been done is by getting a certificate authority[6] such as Let's Encrypt[7] to issue a digital certificate, use it to host a HTTPS server instead and add a DNS CNAME record to point to the EC2 instance hostname. However, this was achieved by using an Application Load Balancer[8] as AWS provides communication over HTTPS to the load balancer. The load balancer essentially acts as a proxy server,

[6]https://en.wikipedia.org/wiki/Certificate_authority

[7]https://letsencrypt.org/

[8]https://aws.amazon.com/elasticloadbalancing/application-load-balancer/

as seen in Figure 5.5. An advantage of doing this was that a load balancer needed to be set up for greater scalability anyway.

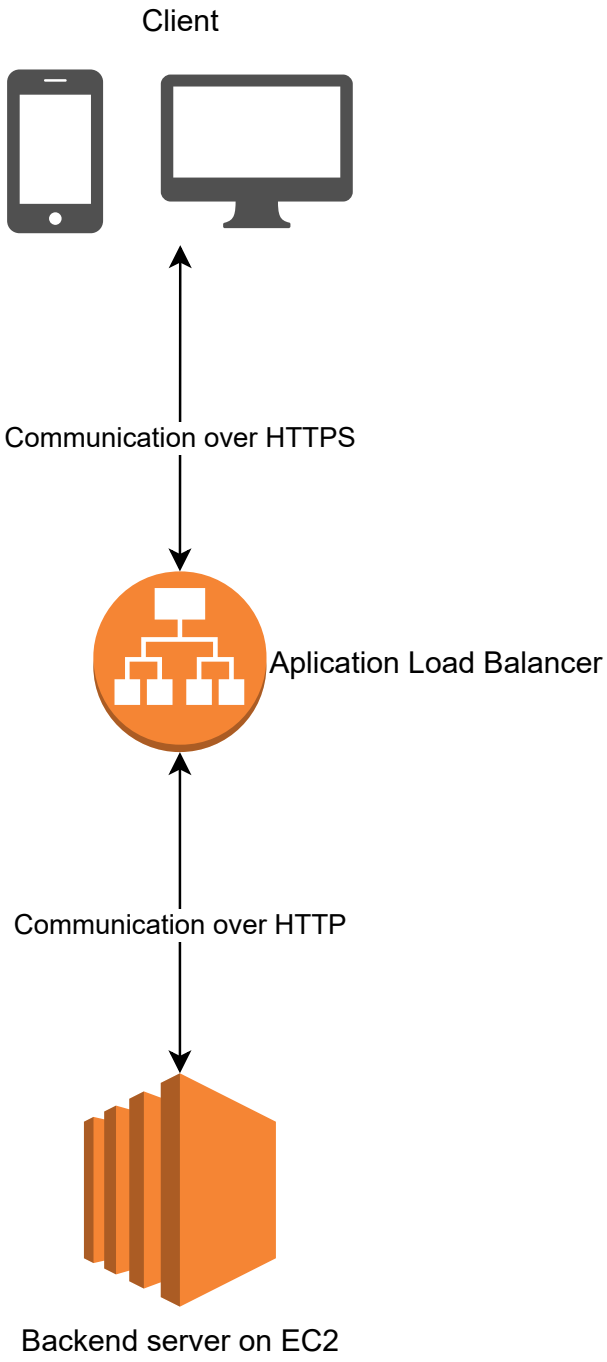

Figure 5.5: How the ALB[9] acts a TLS termination proxy[10]

5.2 REST API

As seen and talked about in the architecture chapter the REST API is created from HTTP triggered AWS Lambda functions whose access are centralized by an AWS API Gateway, which essentially acts as a reverse proxy. For persisting the data I chose AWS

DynamoDB[11] as that is a managed solution as well.

5.2.1 Project structure

The file and directories hierarchy here is the easiest to understand compared to the other ones. The first four files/directories have been explained before. The *serverless.yaml* file is Serverless[12] specific and contains IaC[13] that deploys all the necessary resources with their configurations, as talked about in the infrastructure chapter. The *OPENAPI.yaml* file contains an API specification that makes it easier to describes and visualize the services providen by the **Think-In** API, it is written according to the OpenAPI Specification[14]. The *shared.ts* file contains code that is common across this application. In the last six directories every file corresponds to a HTTP triggered Lambda function. In the *avatars* directory the *post-avatar-token.ts* file contains logic that resizes and changes the user avatar when the user passes a token, i.e. when the user is logged in and changes his avatar from the profile page. The *post-avatar-id.ts* file contains logic for handling the use case when the user picks an avatar when registering, at this point the user can't be logged in and therefore can't use a token to request to the function previously explained. The hacky solution I went through is that a user can change his avatar as long as he is unconfirmed and knows his ID. The *translations* directory contains logic that deals with, of course, translation. For this I have relied the burden of translating between languages to Google Translate[15] because it has always been my go-to language translation app. The function in the *get-languages.ts* file simply returns the languages supported by Google Translate, the function in *post-translate.ts* deals with actual language translation. The last four directories contain code for functions that respect the REST architectural style constraints.

[11] https://aws.amazon.com/dynamodb/
[12] https://serverless.com/
[13] Infrastructure as Code
[14] https://swagger.io/specification/
[15] https://en.wikipedia.org/wiki/Google_Translate

```
REST_API
|- tsconfig.json
|- .eslintrc.json
|- .prettierrc.yaml
|- node_modules/
|- serverless.yaml
|- OPENAPI.yaml
|- shared.ts
|- avatars/
   |- post-avatar-id.ts
   |- post-avatar-token.ts
|- translations/
   |- get-languages.ts
   |- post-translate.ts
|- chats/
|- notifications/
|- stages/
|- users/
```

Figure 5.6: Backend server project structure.

5.2.2 DynamoDB design

Before further explaining we need to have a slight idea of how DynamoDB works. It is a NoSQL database that stores items in a table, each item containing attributes, this results in schemaless items very alike JSON[16] objects. DynamoDB supports two types of primary keys:

- **Partition key** - items are uniquely identified by a single attribute, based on the attribute the item will be assigned to a physical partition
- **Partition key and sort key** (composite key) - items are uniquely identified by a combination of two attributes, the items are assigned to a physical partition by the primary key, and are sorted in that partition by the sort key.

[16]https://www.json.org/json-en.html

Having this knowledge I had to study the data access patterns of the application. One particular use case that stands out is fast message retrieval, most certainly if people used **Think-In** they exchanged only a couple messages and most importantly other contact data so further talking will be done through another medium. For this reason I don't want people to wait a long amount of time to retrieve that meaningful information. In order to achieve this I didn't want time to be lost on sorting the messages between users, so the fact that DynamoDB holds the data in memory naturally sorted helped, therefore I chose to have a **partition key** and a **sort key** where a message will have the following format:

- Partition key of the format `chat_{chatId}`
- Sort key of the format `message_{timestamp}_{id}`.

In doing so, if we search by the partition key we get to the physical partition where the messages are stored sorted by timestamp, that sounds fast! The justification for an additional ID at the end of the sort key is to account for the rare case where multiple messages have the same timestamp. Knowing that DynamoDB and the Lambda function are hosted in the Frankfurt region, I got an average response time of 281 milliseconds over 15 requests. The function was initialised (i.e. it had no cold start[17]) before performing the requests. In case there was any doubt, the data identified by the partition key and sort key respects the contract of the message described in the frontend chapter.

Another data that needed to persist was notifications, that is who messaged a user while he was offline, next time he logs in he should be notified of the users who left him messages. In this case a notification has the partition key of format `user_{userId}` and sort key `notification_{id}`. An access pattern here would be to simply get all items that have the partition key equal to that of a user's ID in order to get all his notifications, whether or not the notifications are sorted doesn't matter. The contents stored for a notification is the message sender's information so next time the receiver logs in a new chat with the sender user will be loaded.

What is stored in the database is also information about stages, the information that can be seen in the horizontally scrolling area of the index page. In this case the partition key is `stages` and the sort key is `{stageId}`. The information stored about a stage is the following:

[17]https://www.serverless.com/blog/keep-your-lambdas-warm

- title - the stage title
- subheader - the text shown below the stage title
- body - description about the entity behind the stage
- externalLink - an external link to more information about the entity behind the stage
- imageLink - a link to an image provided by the entity behind the stage
- videoLink - a link to a video provided by the entity behind the stage.

In retrospective, after using other managed database solutions provided by other cloud platforms (e.g. Cloud Firestore, Firebase Realtime Database), interacting with DynamoDB hasn't been the most pleasant experience from a developer's point of view. For instance having to specify query expressions using strings did not feel modern to me. Moreover, performing pagination was a headache, the traditional limit and offset parameters have been substituted by the partition key and sort key, so everytime I wanted to obtain another page I had to specify the last retrieved item's partition key and sort key.

5.2.3 Changing avatar functionality

At the heart of changing a user's avatar sits the function in Figure 5.7. As observed by lines 2-3, in order to save bandwidth and improve loading times the image is processed so that it is resized, furthermore it's brought to a common format, in this case `image/jpeg`. Between lines 5-12 the new avatar is uploaded to S3, and lines 14-20 change the *picture* attribute of the user so that it now points to the newest location.

```
1  const uploadAvatarToS3AndUpdateUserAttribute = async (userId, avatarBuffer) =>
2  {
3    const processedAvatar = await sharp(avatarBuffer)
4                                    .resize(256, 256).jpeg().toBuffer();
5
6    const { Location: avatarURI } = await s3
7      .upload({
8        Bucket: process.env.S3_CHATS_BUCKET!,
9        Key: `avatars/${userId}.jpg`,
10       Body: processedAvatar,
11       ContentType: 'image/jpeg',
12     })
13     .promise();
14
15   await cognitoIdentityServiceProvider
16     .adminUpdateUserAttributes({
17       UserPoolId: process.env.COGNITO_USER_POOL_ID!,
18       Username: userId,
19       UserAttributes: [{ Name: 'picture', Value: avatarURI }],
20     })
21     .promise();
22
23   return avatarURI;
24 };
```

Figure 5.7: Logic of changing an avatar.

Chapter 6

Limitations and further improvements.

Further development that can be done is towards scalability, for instance the application load balancer can be used together with Socket.IO Redis Adapter[1] and multiple EC2 instances in order to accommodate for a larger amount of people using the application. More interactivity can be brought to the 3D stage, the fact that the attendee's can move their avatar around the stage serves a good basis for that.

If group chats are desired then a refactoring of the way chats are stored has to be made, currently a chat ID is calculated by hashing the ID of the user participants, which unless the chat is the global/stage chat the number of participants is always two.

An issue that can be more thoroughly analyzed is why sometimes requests to S3 from the frontend are unsuccessful. Several browsers claim it's a CORS[2] issue but I have configured the bucket such that CORS requests are allowed from any origin. What's weird is the inconsistency of the error, sometimes it works and sometimes it doesn't, on certain resources it works and on others it doesn't.

[1] https://socket.io/docs/v4/redis-adapter/

[2] Cross-Origin Resource Sharing: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Conclusions

As overall conclusions I have:

- examined other implementations and studied their weak and strong points by using the applications that were free, after which I borrowed their strong points and improved their weak ones
- chosen the best available services and tools that fit to my use case in order to provide the best developer and user experience
- created an application that is straightforward to use and brought it into a working state into the real world, available to everyone on the internet, that includes acquiring and setting up a domain, setting up services and making them work together

Bibliography

- https://www.typescriptlang.org/docs/
- https://nextjs.org/
- https://socket.io/
- https://jamstack.org/
- https://agilemanifesto.org/
- http://www.extremeprogramming.org/values.html
- https://en.wikipedia.org/wiki/Extreme_programming
- https://en.wikipedia.org/wiki/Software_development
- https://en.wikipedia.org/wiki/Waterfall_model
- https://en.wikipedia.org/wiki/Infrastructure_as_code
- https://en.wikipedia.org/wiki/Event-driven_programming
- https://en.wikipedia.org/wiki/Representational_state_transfer
- https://reactjs.org/docs/faq-structure.html
- https://levelup.gitconnected.com/get-rid-of-relative-import-path-hell-in-your-typescript-project-9952adec2e84
- https://www.coreycleary.me/escaping-relative-path-hell
- https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0
- https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
- https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.ht

- https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
- https://docs.aws.amazon.com/ses/latest/dg/Welcome.html
- https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-user-identity-pools.html
- https://docs.aws.amazon.com/cognito/latest/developerguide/cognito-identity.html
- https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html
- https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do
- https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html
- https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction