Sabanci University

ENS492 Graduation Project (Implementation)

Final Report

Project Title: Meeting Scheduler Chatbot

Group Members:

Cavit Çakır

Kaya Kapağan

Supervisors:

Duygu Karaoğlan Altop

Reyyan Yeniterzi

June 16, 2021

Contents

1 PROBLEM STATEMENT **4**
- 1.1 Objectives/Tasks . . . 6
- 1.2 Realistic Constraints . . . 8

2 METHODOLOGY, RESULTS and DISCUSSION **9**
- 2.1 UML Diagram . . . 9
- 2.2 Database . . . 9
  - 2.2.1 Company Model . . . 9
  - 2.2.2 Group Model . . . 10
  - 2.2.3 User Model . . . 11
- 2.3 Website . . . 11
  - 2.3.1 Front-End . . . 11
  - 2.3.2 Back-End . . . 22
- 2.4 Chatbot . . . 34
  - 2.4.1 Selecting Framework . . . 34
  - 2.4.2 Intents . . . 36
  - 2.4.3 Stories . . . 37
  - 2.4.4 Rules . . . 38
  - 2.4.5 Regex for Extraction . . . 39
  - 2.4.6 Actions . . . 39
  - 2.4.7 NLU Pipeline . . . 41
- 2.5 Integration . . . 41
  - 2.5.1 Front-end - Back-end . . . 41
  - 2.5.2 Back-End - Google Authentication . . . 42
  - 2.5.3 Back-end - Database . . . 42
  - 2.5.4 Back-end - Chatbot . . . 42
  - 2.5.5 Chatbot - Scheduling Algorithm . . . 42
  - 2.5.6 Chatbot - Duckling . . . 43
- 2.6 Deployment . . . 43
  - 2.6.1 Database . . . 43
  - 2.6.2 Back-end and Front-end . . . 43
  - 2.6.3 Rasa Actions and Rasa Bot . . . 43
  - 2.6.4 Duckling . . . 44

3 IMPACT **44**

4 ETHICAL ISSUES **44**

5 PROJECT MANAGEMENT **45**
- 5.1 Initial Plan . . . 45
- 5.2 Plan in the Progress Report 1 . . . 45
- 5.3 Plan in the Progress Report 2 . . . 47
- 5.4 Plan in the Final Report . . . 49

6 CONCLUSION AND FUTURE WORK **50**
6.1 Results 50
6.2 Future Work 51

7 APPENDIX **52**

8 REFERENCES **57**

Abstract

Scheduling a meeting is a really time-consuming task. We are planning to make this process easier by developing a chatbot with a special scheduling algorithm which is based on participants calendar information that we will access by authenticating users calendars. This way users will not suffer because of the looping mail traffic while arranging meetings and they can focus on their tasks. We decided to develop a chatbot since we can integrate it with the tools that are used in the industry, such as Slack[1] or Microsoft Teams[2]. To develop a chatbot we should, we decided to use Natural Language Understanding (NLU) methods instead of the state machine approach, in order to increase user experience. We are going to use NLU to understand the intent of the user to make the conversation more likely to human-human conversations. Our goal is to give the feeling to the user that they are talking with a real assistant. Besides the chatbot part of this project, we also need create database in order to manage user information. Also, a website is required to register users, authenticate their calendars, and manage their meeting groups. Therefore, a lot of information transaction happens between database, website, and chatbot. Since we will store the user's personal data and authenticate keys, we will manage security between database and chatbot by analyzing possible vulnerabilities of system. In the end of the project, a flow is created such that a user can register the website, authenticate his/her calendar, create a meeting group, and then go to Google's Hangout Chat, from app market can add the bot and arrange a meeting. After that, the participants of the meeting are receives an email and can see the event in their calendars with the related name and duration.

1 PROBLEM STATEMENT

Scheduling a meeting is a time-consuming task. We intended to make this process easier by developing a chatbot with a special scheduling algorithm that is based on participants' calendar information that we access by authenticating users' calendars. This way users will not suffer because of the looping mail traffic while arranging meetings and they can focus on their tasks. We decided to develop a chatbot since we can integrate it with the tools that are used in the industry, such as Slack or Google Hangouts [3].

Conversational agents are artificial intelligence (AI) systems that can handle a conversation with a user in natural language, mostly through messaging platforms or websites. We could consider conversational agents in two sections: Chatbots and Dialog Agents [4]. Chatbots are rule or corpus-based systems: they cannot learn from the user and adapt to the conversation. Chatbots are usually used in areas like clinical therapy and chitchat. Dialog Agents are goal-based systems. They are more often used in areas that require some information from the user and process that information to achieve a goal, such as; booking a flight or banking operations. The dialog agent model is more suitable for our requirements, so we decided to design our conversational agent as a goal-based dialog agent. When a user opens our chatbot, they will write their intent about what they want. The chatbot will understand the intent and help the user with scheduling/updating a meeting or creating a group for further meetings to be scheduled and many more actions according to the users' intentions.

To increase user experience, we decided to use Natural Language Understanding (NLU) methods instead of the state machine approach since our chatbot's conversations could be more likely to human-human conversations. Our goal is to give the feeling to the user that they are talking with a real assistant. Besides the chatbot part of this project, we also need to create a database and website to manage and authenticate user information. Therefore, a lot of information transfer occurs between the database, our back-end server, our front-end client, and chatbot. Since we are storing the users' data and authentication keys, we also manage security between the database, website, and chatbot by analyzing possible vulnerabilities of the system.

Some applications already provide a solution to our problem. Firstly, Kono.ai [5] is an AI scheduling assistant for teams and enterprises. Users could chat with Kono through email and Slack. Kono syncs with Google [6] and Outlook [7] calendars. It supports integrating multiple calendars into your Kono account. It also considers holidays while scheduling a meeting. They have ISO/IEC 27001, ISO/IEC 27018, and SAP certifications. In the industry market, the share of Microsoft Teams is worth considering and Kono does not support Microsoft Teams and some other platforms like Teams. Another example is Doodle's Chatbot [8], it also integrates with Slack, there are some options like voting for meeting slots. Doodle automatically sends reminders before meetings and also if any of the participants is not voted for appropriate scheduled meeting time. It also considers the time zones of different participants. Lastly, Calendly [9] is the most used scheduling application with over 5 million users. They provide integration with all messaging platforms and Calendly can sync to all the calendar tools. Also, it supports participants from other companies. It connects other apps via Zapier [10] but it cannot take input from other sources like Slack. This system requires users to go to the Calendly site, which is not a good practice since users need to leave their communication environment.

Compared with these tools, we planned to have an integration with Google Calendar, Slack, and Hangouts. We planned to have a website for users to authenticate their Google Calendars and create groups to use in third-party applications such as Hangouts to which our bot is integrated. Since we provide integration with Hangouts, users can stay in their chatting environments and can schedule meetings easily and fast. Also, we have an NLU based chatbot that can provide a better user experience compared with other tools' bots.

Our main motivations for this project:

- Developing a NLU chatbot is motivating us because in the industry there are several chatbots but generally, they have strict rules and have an artificial feeling.

- It is really hard to arrange the meetings since finding a suitable time for everyone requires many repetitive conversations between team members. Thus, solving this problem helps a lot of people.

- Learning industry standard frameworks and tools and using them in the real project is also motivating us because in most of the courses we used outdated tools.
- As we all aware, privacy and security are problematic topics, so learning how to develop a secure tool is motivating us.

1.1 Objectives/Tasks

Our objectives for this project were below and while working on those objectives, security is our first concern. Detailed explanations of those tasks and objectives will be given in other sections.

- **Choosing Bot Framework** (✓): Choosing the one that best suits for our goal, among several bot frameworks by comparing ecosystem maturity, NLU features, scalability, licensing, and usage cost.
  - We decided to implement the bot by using Rasa Machine Learning Framework because RASA is an open-source tool[11]; so it has a large community and supports a lot of NLU features.
- **Learning Bot Framework** (✓): Understanding the core features of the selected bot framework; Learning how to integrate with other platforms and how to deploy the bot on a server.
- **Implementing bot** (✓): Creating intents and their example sentences, creating entities and mentioning them in the example sentences; Creating chat flows and write them as stories; Creating necessary rules that can be driven from the chat flow; Creating all actions that are needed for the flow, combining the written actions and bot together.
- **Creating NLU Pipeline for**
  English (✓) — **Turkish** (X): Selecting the best NLU methods for the pipeline by analyzing outcomes of many approaches.
- **Data Acquisition** — **Labeling** (X): A lot of English/Turkish speakers to generate conversations by chatting with bot and labeling them later on.

- **Connect Bot to Chat Environments** (**Slack** ✓— **Hangouts** ✓): Setup the bot such that many users can be able to use it simultaneously over Slack, Hangouts.

- **Creating Database** (✓): Designing the database, deciding which models are required to meet our needs. Creating the database from a cloud service. Creating necessary fields of the models and storing the necessary data, managing database.

- **Front-end of Website** (✓): Design the pages for each required feature. Integration with back-end server to create the necessary data for bot in the database. Doing state management correctly in the front-end so that development can be easier while the code base is expanding and becoming more complex.

- **Back-end of Website** (✓): Deciding the features of the project and the API needs of those features. Required to manage the needed data for both our chatbot and our front-end. Checking the security vulnerabilities of our APIs. Creating, editing and deleting necessary fields in the database. Extracting the needed data from sources such as calendar events and calculating necessary information from extracted data.

- **Calendar Integration** (✓): From Google Developer Console, creating a project and getting the necessary access and by using front-end and back-end together, showing the Google allowance screens and getting the allowance of the integration of the user's calendar.

- **Develop Scheduling Algorithm** (✓): Developing an efficient Calendar Scheduling Algorithm to shorten the respond time of the bot.

- **Deployment** (✓): Need to transfer all code base which is related to bot, bot's actions server, front-end, back-end. To deploy any server without any problems, dockerize the front-end and back-end. Additionally there is a duckling server that need to be pulled from docker hub[12]. Run the necessary commands after transferring the codes and need to open the necessary tcp ports to the world.

1.2 Realistic Constraints

In this section, we will go over the possible constraints of the project. The most important realistic constrain is the social one, since our data should be having rights and permissions for personal privacy.

- **Environmental:** We need to limit the consumption of the energy in order to make less effect to global warming. To achieve this goal, we are going to choose the correct hardware and only spend computational resource(energy) for the required cases.
- **Social:** Our data should be having rights and permissions for personal privacy. We are planning to acquire data, from only well-known sources which have required permits to use.
- **Health:** No known health danger during manufacturing or later.
- **Manufacturability:** Our bot and server should support each other, and our data also should be suitable for training. Moreover, our model should be suitable for native language conversations. In our design, we will consider each component and its relationship with other components.
- **Sustainability:** The integration of calendars is a constraint. If Google or other calendar providers restrict their calendar authentication API for developers, we may need to reconsider this process to sustain our bot. Our bot can last potentially operate as long as the server operates, which is depending on the server providers. Our bot could be outdated with any other NLU model or framework, in case of a better model or framework is required. In such cases re-design and development are required.

2 METHODOLOGY, RESULTS and DISCUSSION

You can find technical details and all other details about methodology below.

2.1 UML Diagram

You can see project's UML diagram here. It shows each core element of each part of the project. Our 3 database model is connected to database from our back-end. Those models are connected to our back-end APIs. Those APIs are connected to our front-end and the actions of Rasa. Lastly, Rasa actions are connected to Rasa and users can communicate with our bot over Hangouts. You can the elements more clearly from the diagram and below each part is explained in a more detailed way.

2.2 Database

Our database is stored in MongoDB Atlas [13], and it is a NoSQL [14] database that stores JSON [15] objects.

Collection Name	Documents	Documents Size	Documents Avg	Indexes	Index Size	Index Avg
companies	1	77B	77B	1	32KB	32KB
groups	11	2.59KB	242B	1	32KB	32KB
users	5	2.49KB	511B	1	32KB	32KB

Figure 1

2.2.1 Company Model

Company model is for storing:

- The company name - type: String
- Creator - type: ObjectId

It is also used as a referenced key in the User model. You can see an example company instance from our database below in Figure 2.

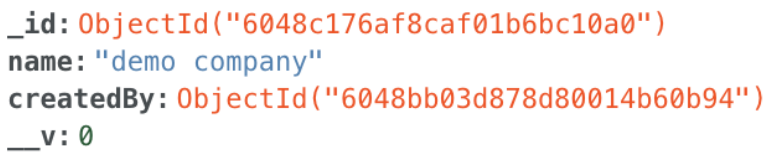


Figure 2

2.2.2 Group Model

Group model is for storing:

- Creator - type: ObjectId
- Which company that group belongs to - type: ObjectId
- The group name - type: String
- List of group members - type: Array

It is also used as a referenced key in the User model. You can see an example company instance from our database below in Figure 3.

Figure 3

2.2.3 User Model

User model is for storing:

- Which company that user belongs to - type: ObjectId
- Email of the user - type: String
- Password of the user which is hashed for security reasons - type: String
- Type of the user - type: String
- List of groups that member belongs to - type: Array
- Refresh token for google calendar - type: String

This model uses the Company model and Group model as its reference keys.

Figure 4

2.3 Website

This section explains the front-end and back-end of the website. We used React.js[16] as front-end language, and Node.js[17] back-end language. Details are given below.

2.3.1 Front-End

For the front-end of the project, we decided to use React and we also used some of the Material-UI [18] components in order to make things faster since our time is really limited. To manage the states of the front-end, we decided to use Redux [19] so that it makes the code more readable. To get more benefit from React, we write out components first and then combine them in containers (pages). We also have a request.js file in our code

directory inside the utils folder so that we write all the requests inside this folder in order to keep the code even more readable and easy to manage.

- **Sign up Page:**

  This page is for individual users to sign up and use the bot and website. Below you can find 2 images as Figure 5 and Figure 6, the first one is the one that we have designed in the previous step and the second one is the real web page that we wrote using React.

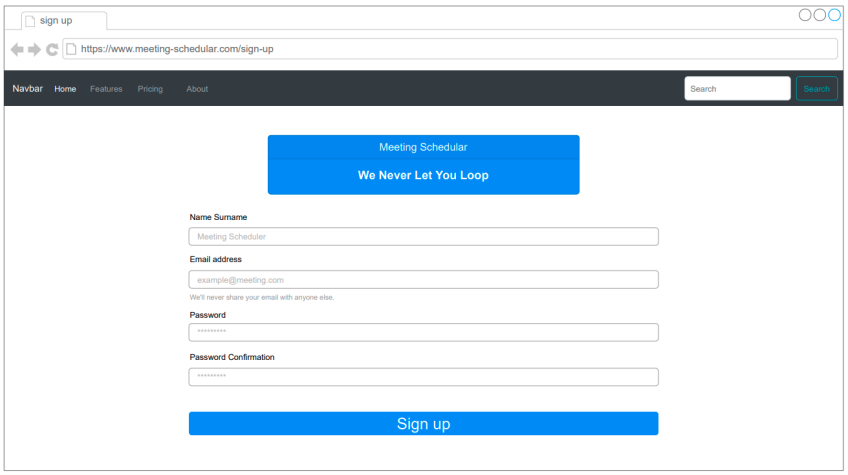


Figure 5

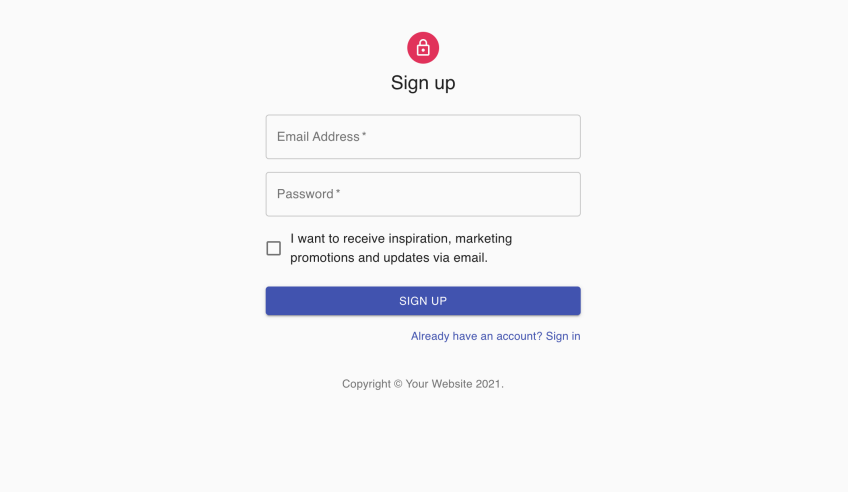


Figure 6

- **Sign in Page:**

  This page is for all signed-up users in some way to sign in to the website. Below you can find 2 images named as Figure 7 and Figure 8, the first one is the one that we have designed in the previous step and the second one is the real web page that we wrote using React.

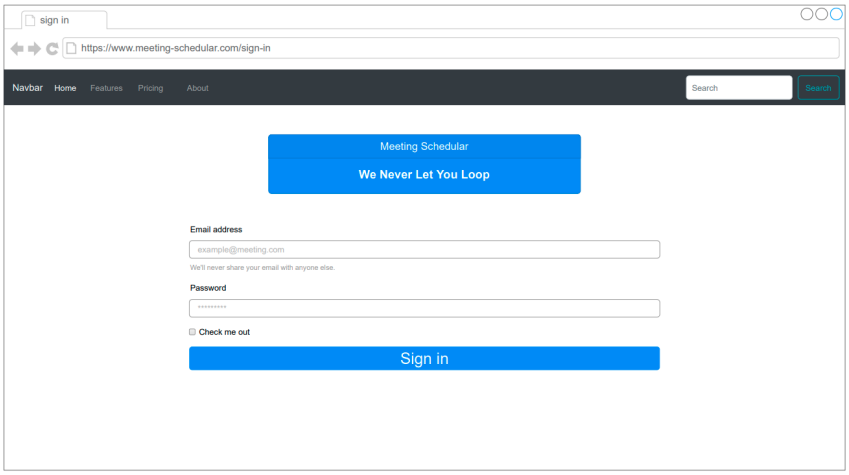


Figure 7

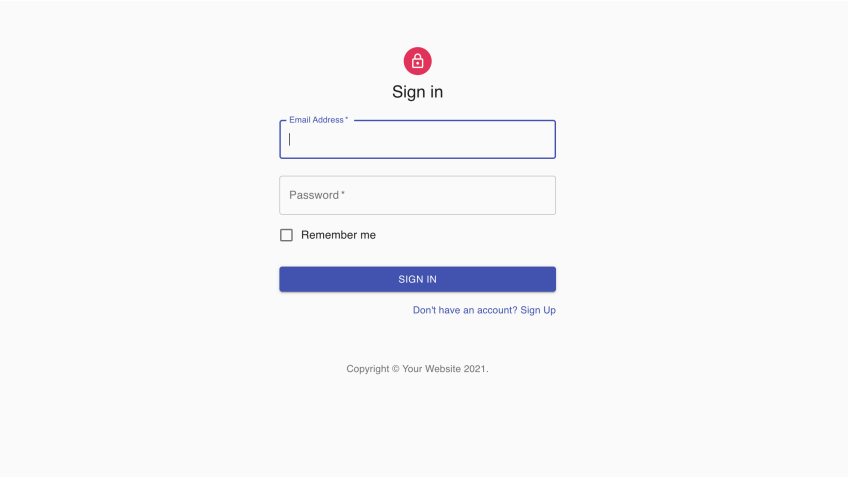


Figure 8

- **Profile (as a wrapper to redirect correctly):**

  This component is a wrapper for other pages except sign in, sign up, and not found. It sends a get request to the server to get the users' information. Until the data loads, it shows a loading component to the user. If the user is not signed in and does not have access to that page, redirects the user to the sign-in page.

- **Drawer**

  The drawer is also a wrapper-like Profile component. Similarly, it wraps all pages except sign in, sign up, and not found. It helps users to navigate more easily. You can find the image of the drawer below as Figure 9 and under it, you can find detailed explanations of the parts of the drawer which are drawer header and drawer content.

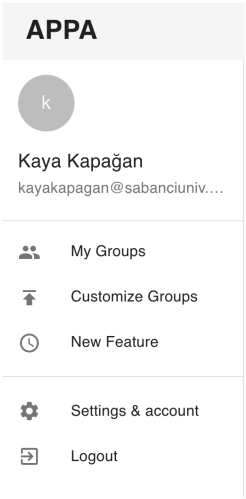


Figure 9

  - **Drawer Header:** This part sends a get request to get the information of the user and shows the full name of the user, email, and an avatar that contains the first letter of the user's name.
  - **Drawer Content**

    This part has the following sections:

    * My Groups - redirects to my groups' page
    * Customize Groups - redirects to edit group page
    * Settings & account - Settings and also a page for authenticating google calendar
    * Logout - when clicked, logged user out and redirect to the sign-in page

- **My Groups Page**

  This page is for users to see all their groups which are created by themselves or just added in. Below you can find 2 images as Figure 10 and Figure 11, the first one is the one that we have designed in the previous step and the second one is the real web page that we wrote using React. After those images, you can see the detailed explanations of the inner components created for this page.

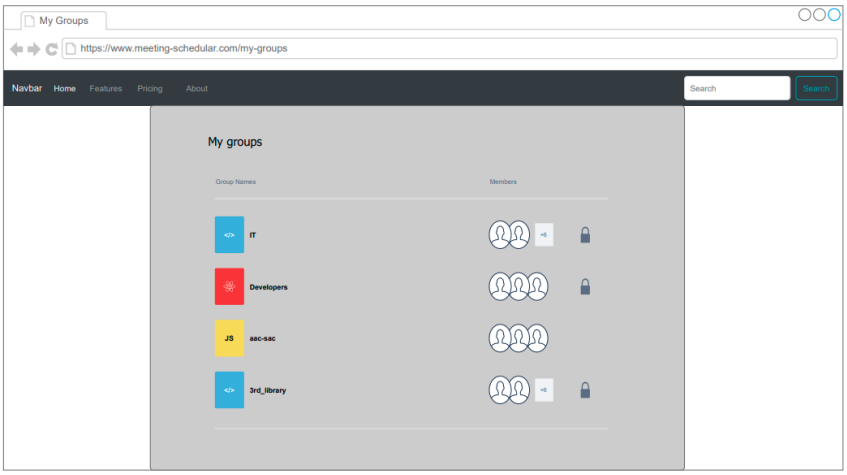


Figure 10

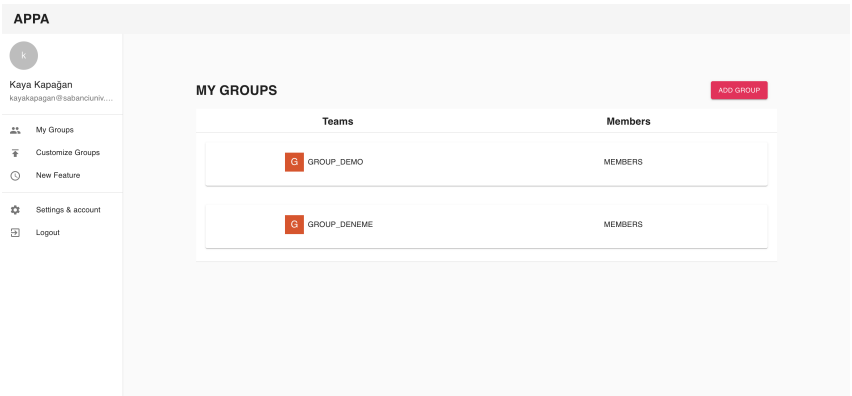


Figure 11

  - **Group List Header:** This component is the header part of the list, Teams, and Members are the words used for the representation while the component is used for this list.
  - **Group List Card:** This component is the card component for the list. It is given in the List Item component of the Material-UI while generating the List.

– **Group List:** This component is the combination of the Group List Header component and the List of Group List Card components generated with the map function.

- **Edit Group Page**

  On this page, one can see all groups which (s)he is the creator of. One can create a new group, edit an already created group's name, delete any group (s)he created before. One can see all company members that can be addable to the selected group, can add new members to a group, and delete members from the group. Below you can find 2 images Figure 12 and Figure 13, the first one is the one that we have designed in the previous step and the second one is the real web page that we wrote using React. After those images, you can see the detailed explanations of the inner components created for this page with their detailed images.

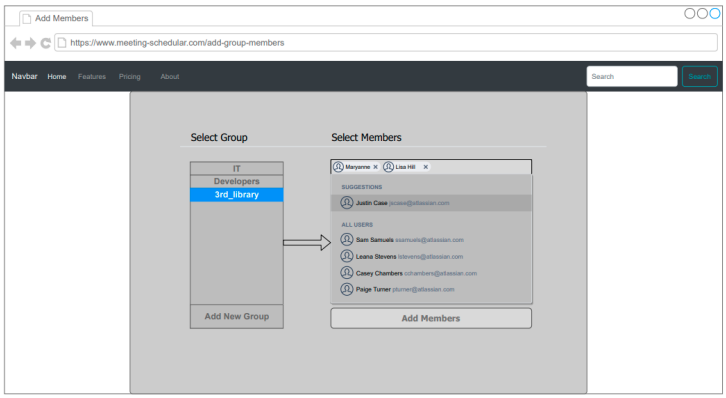


Figure 12

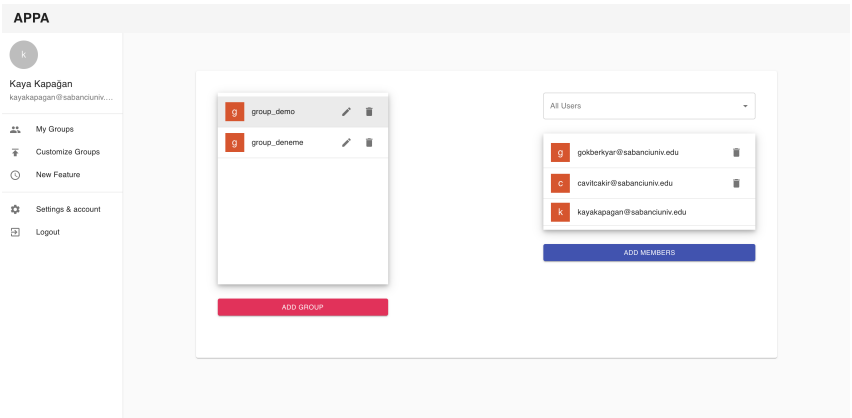


Figure 13

– **Narrow Group List:** This component is for a user to see and edit groups, (s)he can see the groups, can trigger the modal for edit group by clicking the pencil icon, can trigger the modal for add group by clicking the add group button or even delete a group by clicking the trash icon.

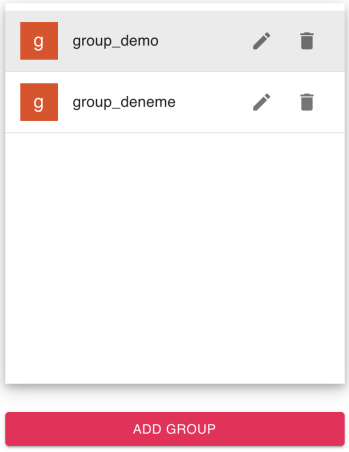


Figure 14

– **Modal for Add Group:** This component is triggered by the add group button. In the text, the field user needs to starts with the "group_" prefix in order to click the add group button in the modal. Otherwise, the button should not be clickable.

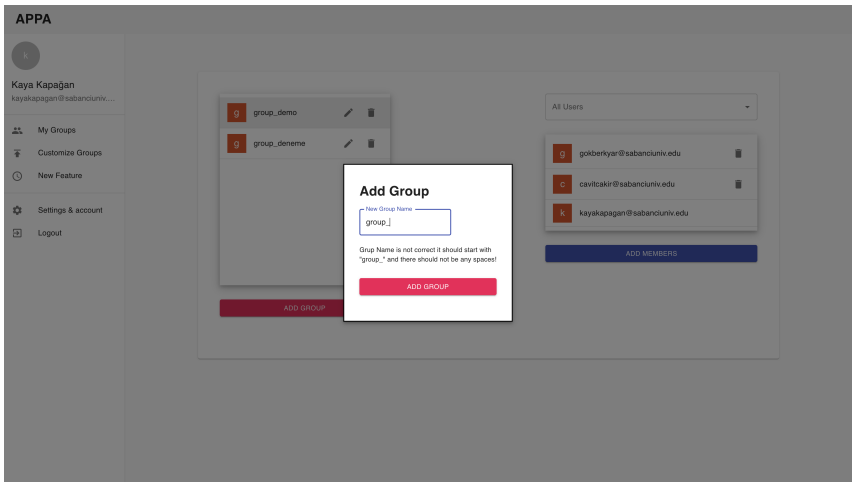


Figure 15

– **Modal for Edit Group:** This component is triggered by the pencil icon next to the group name. In the text field, the group's name can be seen as a translucent way. The rules for add group modal also exists with the edit group modal.

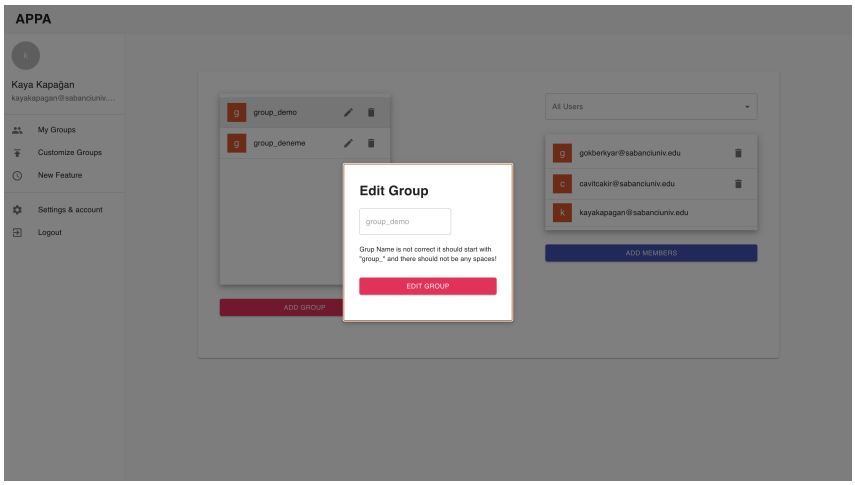


Figure 16

– **See and Add Members:** This component gets all the users of the company that the current user belongs to, except the users that are already in the group. You can select multiple users and add multiple users at once.

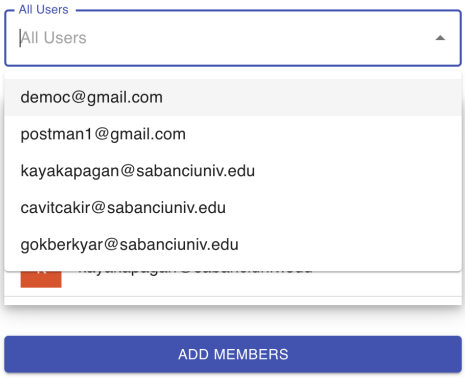


Figure 17

– **Narrow Member List:** In this component, the user can see all users of the selected group and delete them if (s)he wants, but can not delete himself/herself from the group.

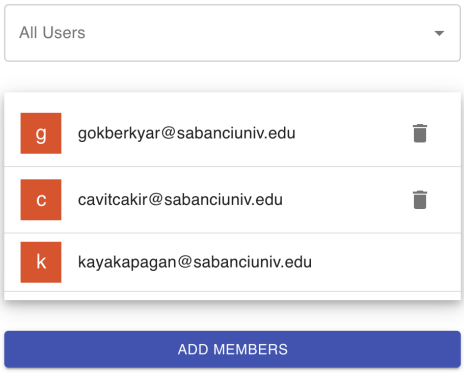


Figure 18

- **Settings Page**

  On this page, the user can authenticate his/her Google Calendar. After clicking the authenticate Google Calendar button, our website will redirect the user to Google's authentication pages. You can see the settings page and Google's authentication pages, as images below in Figure 19, Figure 20, and Figure 21.

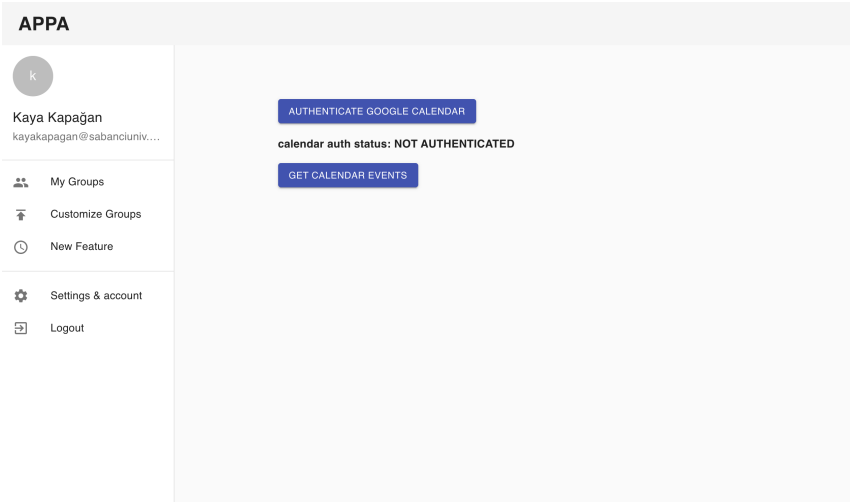


Figure 19

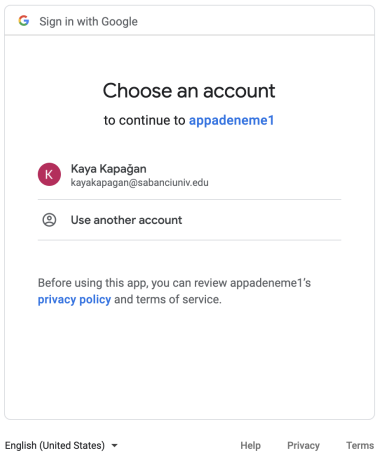


Figure 20

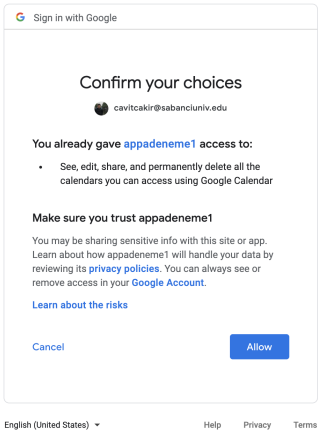


Figure 21

- **Page Not Found**

  This page shown in Figure 22 is for the links that are not found in our routes.

Figure 22

- **Redux**

  Redux is a state management tool for Javascript [20] applications that is known for its predictability and offers a great experience to the developers. Since we are using React and developing our front-end component by component, we need to pass some states and functions for several components which are not close in the component tree.

  All the states that are kept track with Redux have a getter function so that we can use them in our components when needed by using Redux's built-in 'useSelector' function which calls that state's getter function from Redux.

  - **Authentication:**
    - We store the state "token" of the user at this part.
    - We have 2 helper functions which are to set a token and to clear token, and we write this to the local storage since we will need it for every page and we need it although the user refreshes the page. It is the reason that we are storing the token at local storage with the help of our Redux helper functions we wrote so that it will be persistent.
  - **Calendar:**
    - We store states "calendar status" which shows if the user is authenticated his/her Google Calendar to our application or not and "calendar authentication URL" which comes from the "Get URL for Calendar Authentication" API which will be explained below in the Back-end's "Google Related APIs" section.
    - We both have set functions for those states mentioned in the above item.
  - **Group:**

    At this Redux reducer, we store states that are listed below:
    - **Groups** — list of groups for that specific user belongs to
    - **Group Members** — list of group members for a specifically selected group

    * **Company Members** — list of company members for that specific user's company
    * **Group ID** — selected group's ID to find its members for another component
    * **Is Loading Needed for Groups** — boolean parameter which is set to true when something changes in the "Narrow Group List" component mentioned above.
    * **Is Loading Needed for Company Members** — boolean parameter which is set to true when something changes in the "See and Add Members" component mentioned above.
  - **Drawer:**
    * We store state "email" which can be seen in the drawer at the "Drawer Header" component.
    * We have a set function for this state mentioned in the above item.

2.3.2 Back-End

Under each API, you can find an image from our API documentation to increase the understanding of API more deeply with its example parameters.

- **Authentication Related APIs**
  - **Sign-up:** It accepts email and password parameters. As password arrives to the server, it is encrypted by bcrypt. With the scheme of User model, a user is created.
  - **Login:** It accepts email and password parameters. As password arrives to the server it is encrypted by bcrypt [21] and the user information is checked with a database query. If it is a valid login attempt, JWT token[22] is created and returned to the client.

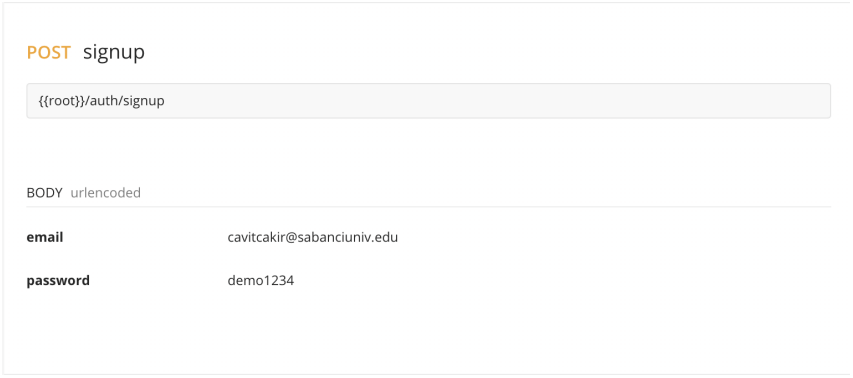


Figure 23

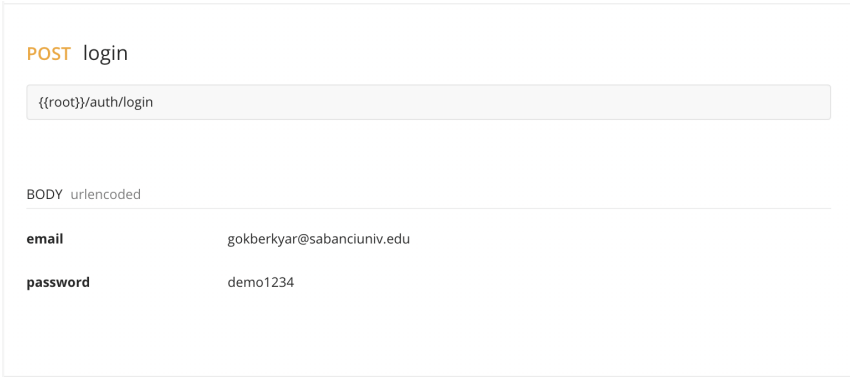


Figure 24

- **Company Related APIs**
  - **Create Company:** It requires JWT token authentication to process the request securely. The API endpoint accepts the company name as a parameter. A company with the entered name is created on a database with a Company model scheme.

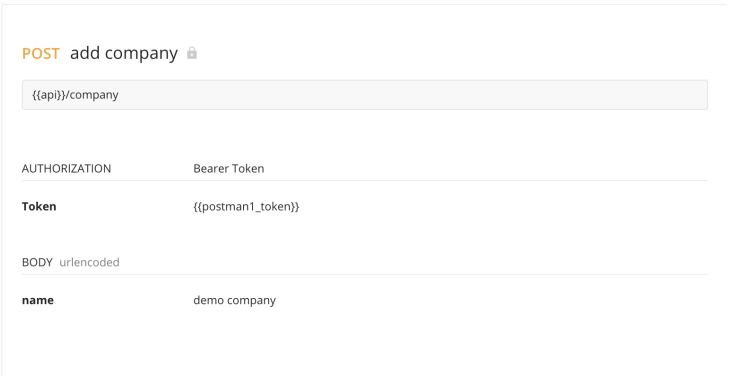


Figure 25

– **Get Members of the Company Which is Not Included in the Specific Group:** It requires JWT token authentication to process the request in a secure way. The API endpoint requires group ID as a parameter and returns all the members in the company except ones that are not included in the specific group.

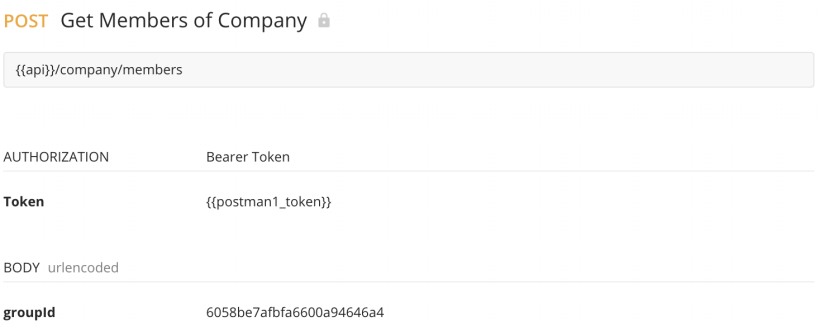


Figure 26

– **Get All Members of the Company:** It requires JWT token authentication to process the request securely. The API endpoint returns all the members of the company.

Figure 27

– **Add Member to a Company:** It requires JWT token authentication to process the request in a secure way. The API endpoint requires company ID and adds that ID into the user's belongCompany field in the database.

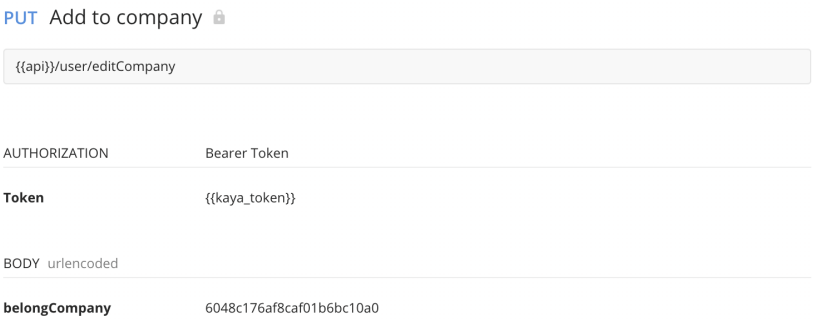


Figure 28

- **Google Related APIs**

  – **Get URL for Calendar Authentication:** It requires JWT token authentication to process the request securely. It generates the URL by using our application's ID, application's secret, and calendar redirects URI. It returns the URL to the front-end and in the front-end, we redirect the user to this URL. The user must allow necessary permissions to authenticate its calendar.

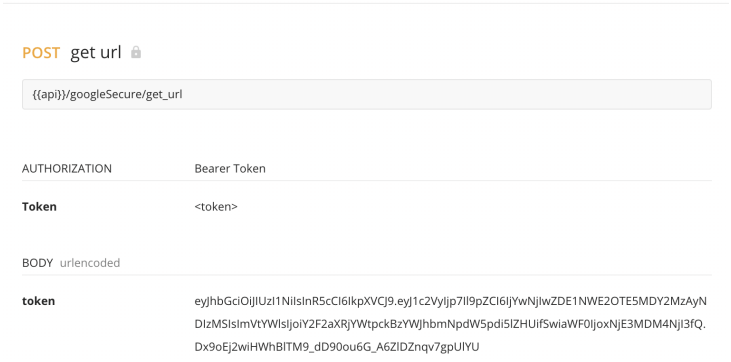


Figure 29

– **Insert Event to the Calendar of Users of Scheduled Meeting:** It requires JWT token authentication to process the request securely. The API requires attendees' emails, start & end times of the event, and calendar tokens to create a calendar event by sending a request to Google's APIs [23]. This API is called from the server of the bot and after this call, the user and the others in the meeting can see the event in their calendars'.

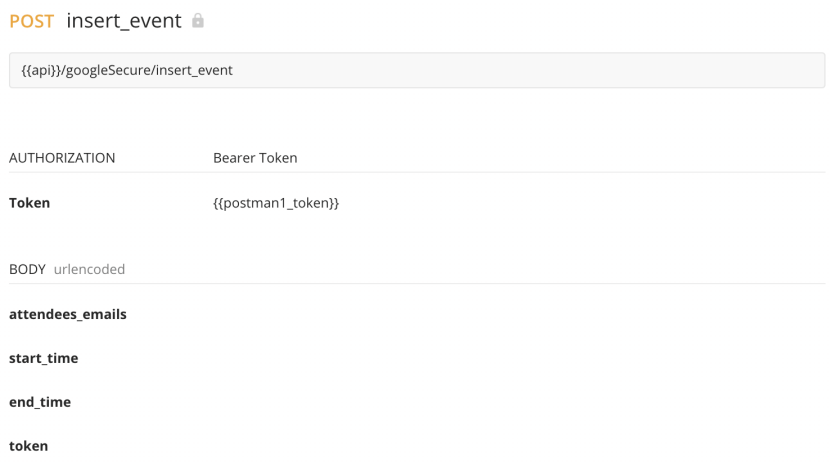


Figure 30

– **Get Calendar Events of a Specific User:** It requires JWT token authentication to process the request securely. The API gets the calendar token of the client from the database and sends a request to Google's APIs to get the following events of the client, and then returns them.

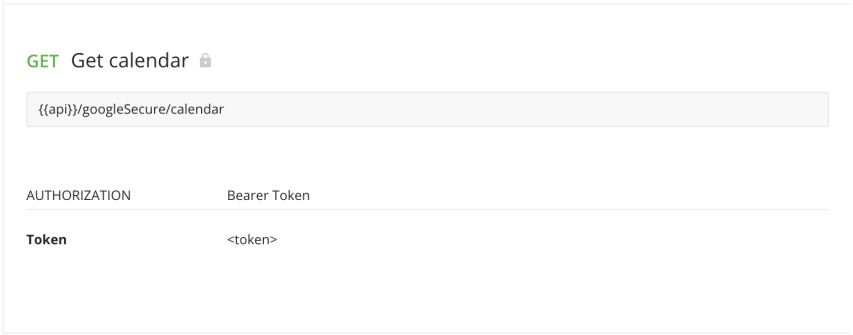


Figure 31

– **Get Calendar Events of a Group of Users:** It requires JWT token authentication in request in a secure way. The API gets the calendar tokens of the clients from the request body and sends a request to Google's APIs to get the following events of the clients then returns them. While doing so we use the promise concept of Javascript to have parallelism while sending requests to Google's API and not to wait for each one, one after and other, instead wait much less.

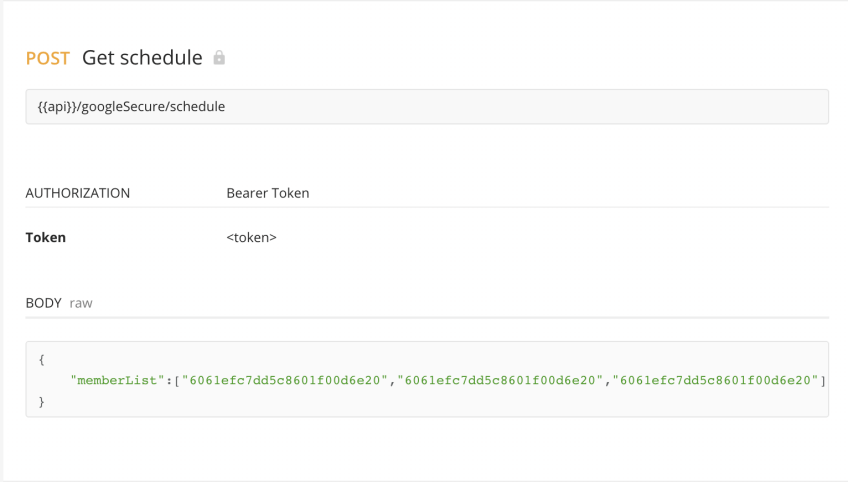


Figure 32

– **Create Refresh Token:** It requires JWT token authentication to process the request securely. After calendar authentication, Google creates the redirect URL. This API handles Google's redirect URL and gets the user's calendar token.

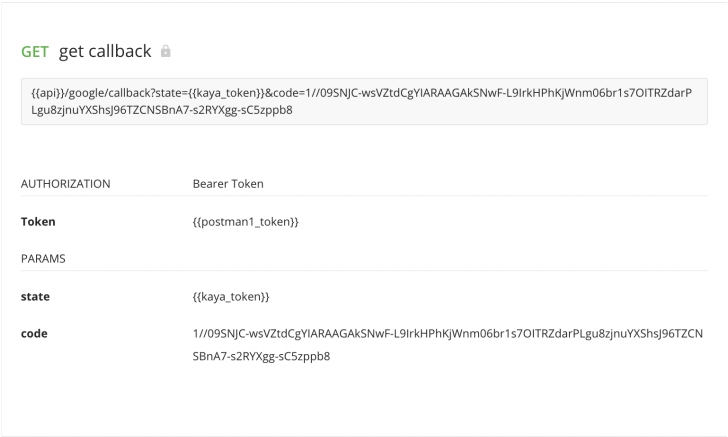


Figure 33

- **Group Related APIs**
  - **Get Groups of a Specific Company:** It requires JWT token authentication to process the request securely. This API returns all the groups of the user's company.

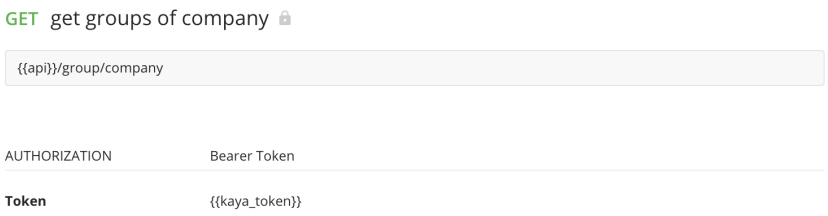


Figure 34

  - **Create Group for Users Company:** It requires JWT token authentication to process the request securely. This API accepts name input and creates a group by group model schema. Automatically adds group creators to the members of the group.

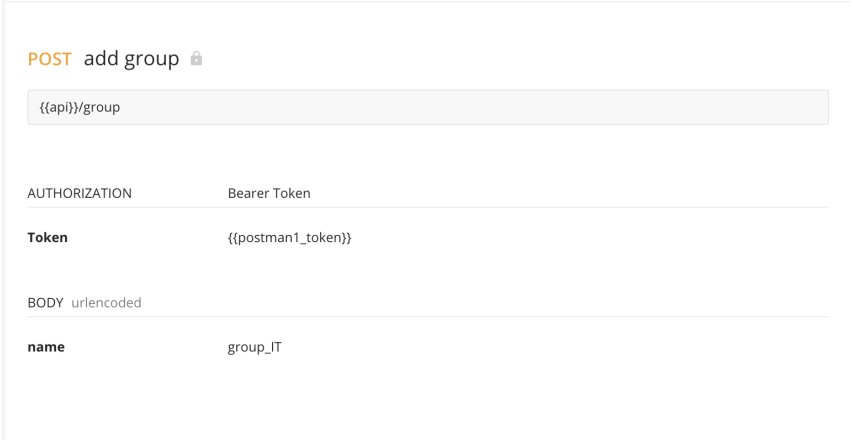


Figure 35

– **Get Groups that User Belongs to:** It requires JWT token authentication to process the request securely. This API returns all the groups of the user.

Figure 36

– **Get Members of a Specific Group:** It requires JWT token authentication to process the request securely. The API requires a group ID as input. It returns all the members in the specific group.

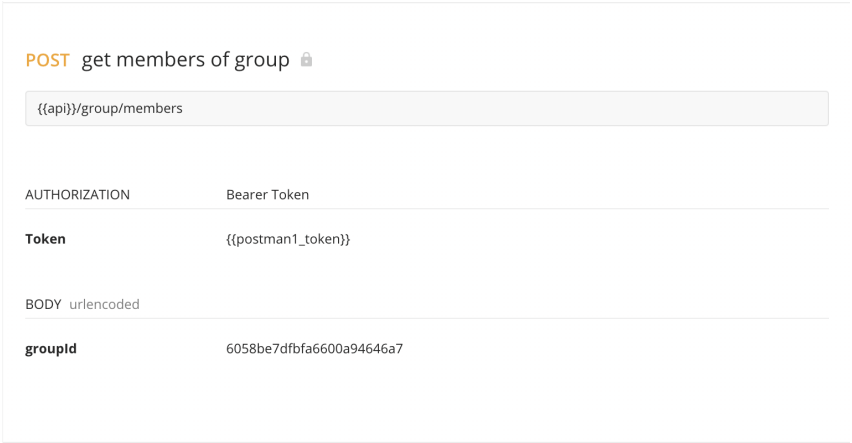


Figure 37

– **Remove Member from a Specific Group:** It requires JWT token authentication to process the request securely. It requires the ID of the member and the ID of the group in the request body. The back-end server updates the group's database table by removing specific user IDs from members and also updates the user database table by removing specific group IDs from user's attributes.

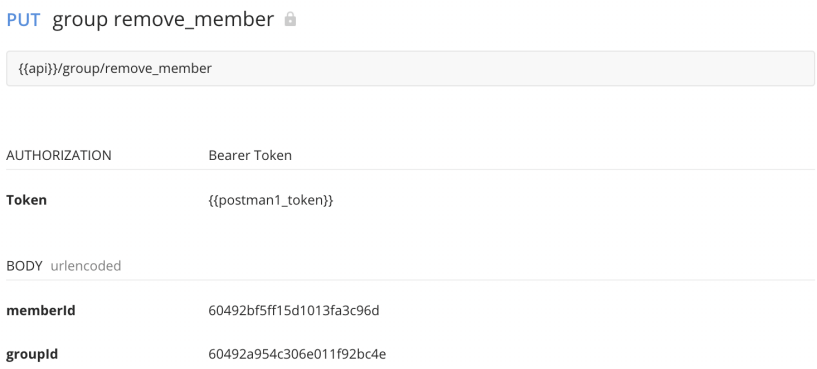


Figure 38

– **Add Member to a Specific Group:** It requires JWT token authentication to process the request securely. It requires the ID of the member and the ID of the group in the request body. The back-end server updates the group's database table by adding specific user-ID into members and also updates the user database table by adding specific group ID into user's attributes.

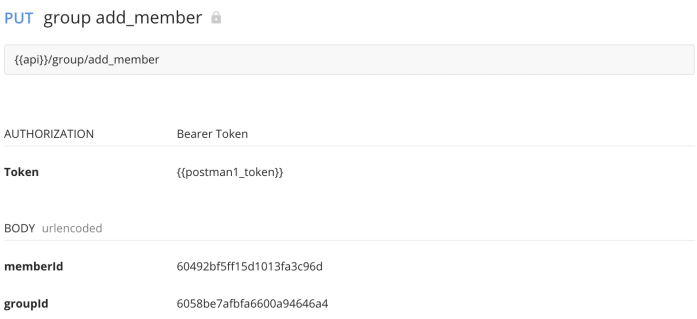


Figure 39

– **Add Multiple Users Efficiently to Group (bulk add):** It requires JWT token authentication to process the request securely. This API same as the previous endpoint but the only difference is it accepts an array of users.

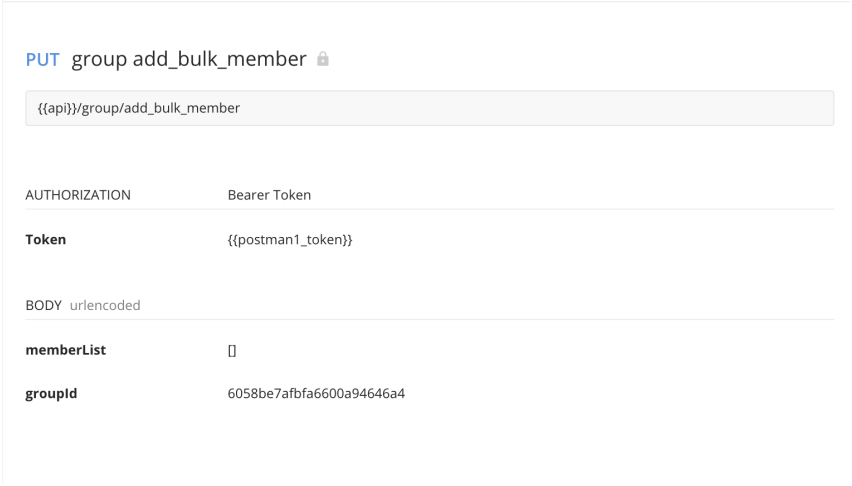


Figure 40

– **Delete Group:** It requires JWT token authentication to process the request securely. It takes group ID as an input parameter and removes all the users from the group and updates each users' attributes. After updating the user database table, deletes a specific group from the database.

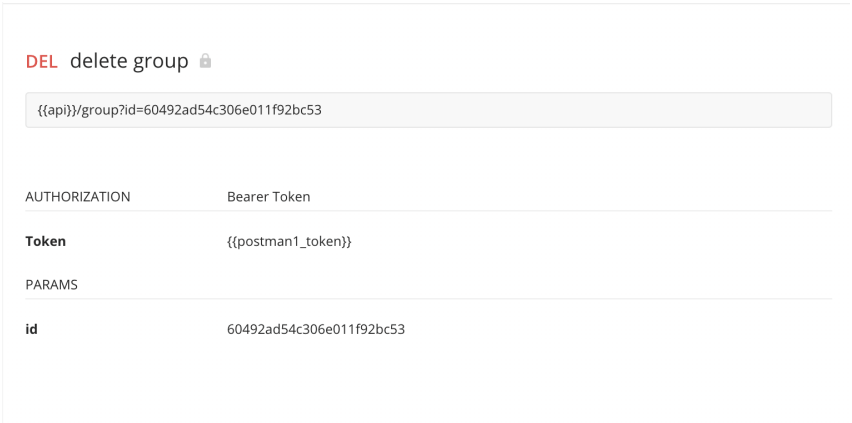


Figure 41

- **User Related Information APIs**
  - **Get Information About a User from the User's ID:** It requires JWT token authentication to process the request securely. This API gets information of request sender from user database table and returns the data.

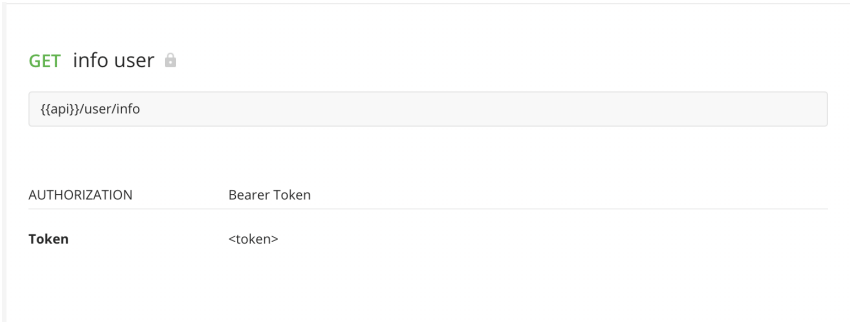


Figure 42

  - **Get Information About a User from User's Email:** It requires JWT token authentication to process the request securely. It accepts email parameter and gets information of request sender by filtering users with an email from user database table and returns the data.

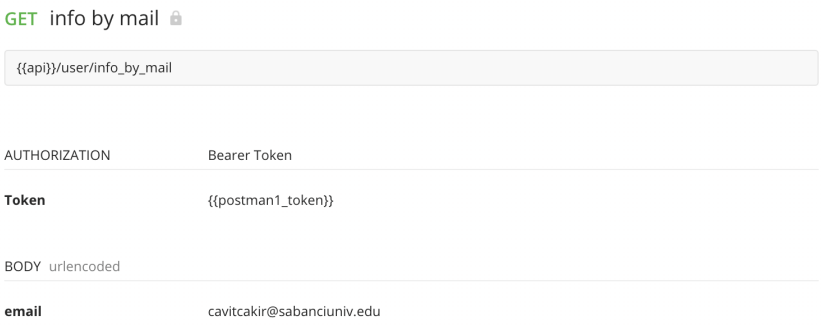


Figure 43

- **Source Code Analyzer:** In order to analyze the security issues of back-end, we used an open-source source code analyzer tool named njsscan[23]. Njsscan is a static application testing (SAST) tool that can find insecure code patterns in node.js applications. First of all, we pull the docker of njsscan and run in on port 9090, then from GUI, we zipped our project and upload it to the tool. Njsscan, scanned

and analyzed 67 files, and with 17 different types, it found 37 issues which can be seen in the Figure 44. With the suggestions of the tool we eliminated each of the possible security issues. We used Helmet.js[25] to set various HTTP headers and we re-write some of the api codes to eliminate possible attacks. You can see the end version of scan in Figure 45.

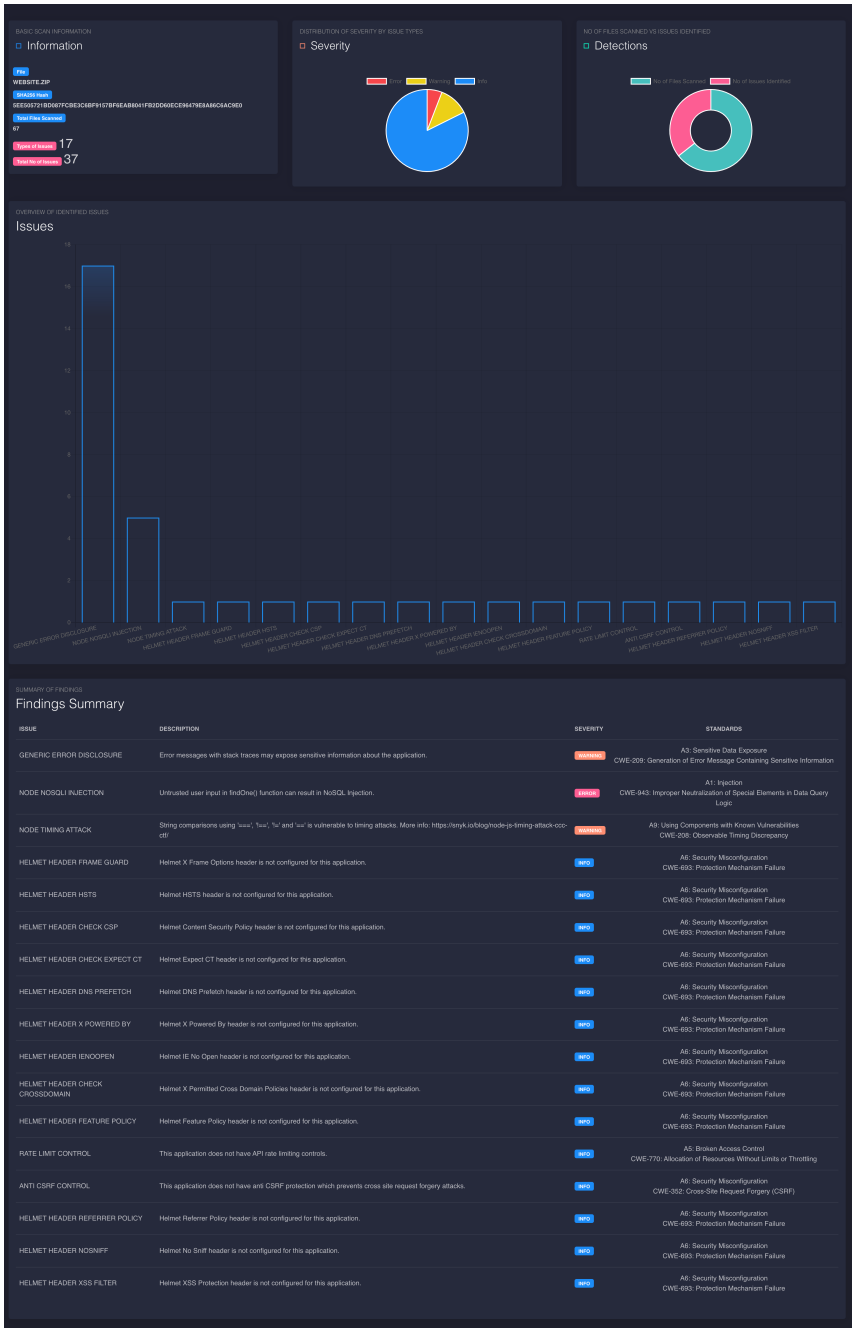


Figure 44

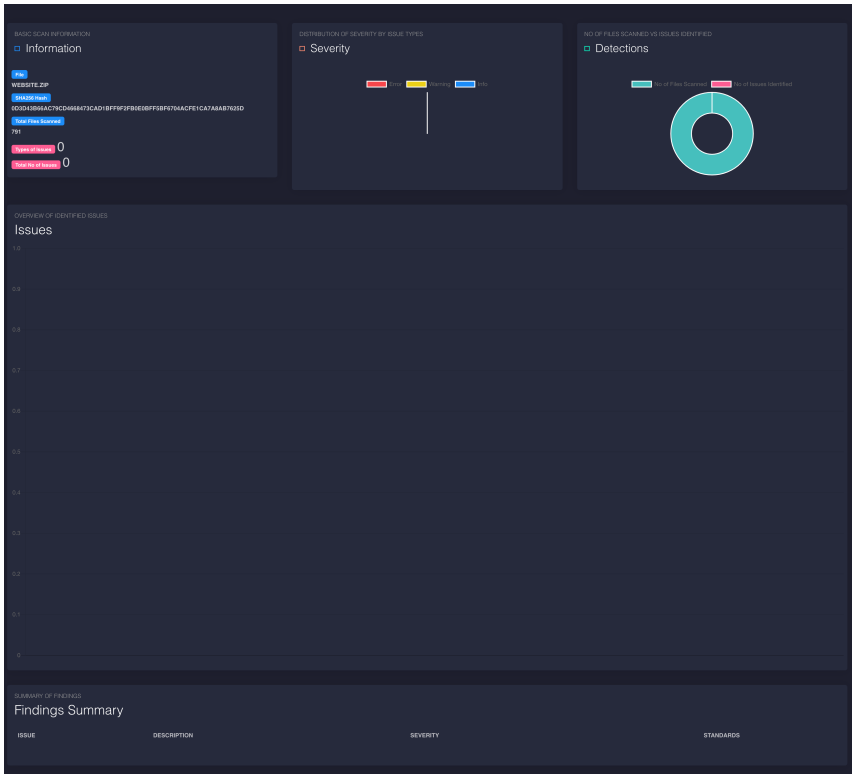


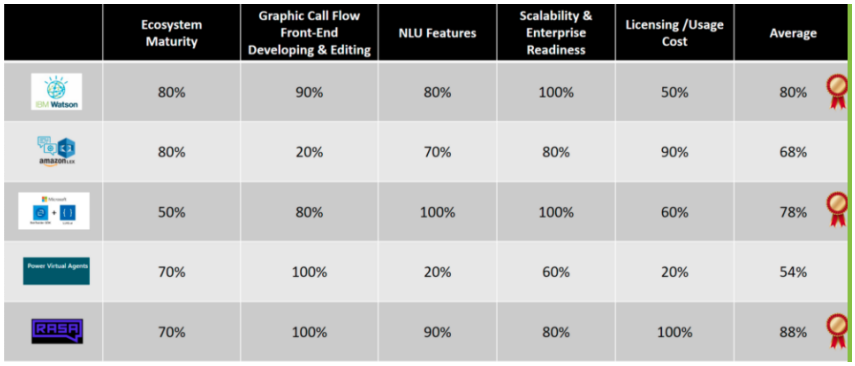
Figure 45

2.4 Chatbot

In this section, we explained how we selected the framework, designed the intents, the stories, and the rules of the bot.

2.4.1 Selecting Framework

To implement the solution that we propose, we searched for different bot frameworks used in the industry to decide which one is the most suitable for our purpose. The top 3 most used frameworks can be seen in Figure 46 [26] were: IBM Watson [27], Microsoft Bot Framework [28], and RASA. We compared each of them by NLU features, usage costs, and user community. In a conclusion, we decided to implement the bot by using Rasa Machine Learning Framework because RASA is an open-source tool; so it has a large community and supports a lot of NLU features.



	Ecosystem Maturity	Graphic Call Flow Front-End Developing & Editing	NLU Features	Scalability & Enterprise Readiness	Licensing /Usage Cost	Average
IBM Watson	80%	90%	80%	100%	50%	80%
amazon lex	80%	20%	70%	80%	90%	68%
Microsoft Bot Framework	50%	80%	100%	100%	60%	78%
Power Virtual Agents	70%	100%	20%	60%	20%	54%
RASA	70%	100%	90%	80%	100%	88%

Figure 46

Rasa has its own NLU pipeline components.

- **Language Models:** Rasa provides us several choices about Language Models, we could develop our own Language Model from scratch or use pre-trained models such as;
  - BERT [29]
  - Spacy [30]
  - GPT [31]
- **Tokenizers:** Rasa provides a good amount of tokenizers, which will be used in the NLU pipeline. For example, there is WhitespaceTokenizer to tokenizes whitespaces.
- **Intent Classifiers:** There are built-in intention classifiers, which help to understand what the intent of the user is.
- **Entity Extractors:** Rasa has its entity extractors to fill the entity slots of the intent of the user input.

In end of the project, from our DIETClassifier[32] we got 94.4% accuracy while training as f1 score over 100 epochs which took 45 seconds to train.

2.4.2 Intents

In Rasa, we define our intents and write several example sentences to those intents, we also mentioned entities while writing those sentences for our extractors to understand what may need to be extracted from given inputs.

- **Greet:** Greet intent is to understand that the user is saying "hello" or similar things. We understand user is trying to talk with us and we reply accordingly.
- **Goodbye:** Goodbye intent is to understand that the user is leaving the conversation and the chatbot writes back "goodbye" to the user.
- **Thank You:** Thank you intent is for understanding that the user is saying something similar like "thanks" and chatbots reply is "No problem :)".
- **Affirm:** Affirm intent is to understand that the user is approving something and the chatbot takes necessary actions accordingly.
- **Deny:** Deny intent is to understand that the user is not approving something and the chatbot takes necessary actions accordingly.
- **Out of Scope:** Out of Scope intent is for the understanding user is saying irrelevant things, and chatbot takes necessary actions accordingly.
- **Bot Challenge:** When the user tries to challenge the bot with questions similar to "Are you a human?" or "Who are you?", our chatbot replies with "I am a bot.".
- **Inform:** Inform event is for understanding any information given by the user like the meeting name, participant group(s), and/or participant mail(s).
- **Show Upcoming Events:** Show upcoming events intent is for understanding that user wants to see his/her schedule and chatbot takes necessary actions accordingly.
- **Schedule Meeting:** Schedule meeting intent is for understanding that the user wants to arrange a meeting and the chatbot executes the form to collect the necessary information.

- **Add New People or Groups to Meeting:** Add new people or groups to the meeting is for understanding that new user(s) and/or group(s) are included in the meeting except for the user(s) and/or group(s) given before.

- **Remove People or Groups from Meeting:** Remove people or groups from the meeting is for understanding that user(s) and/or group(s) are excluded from the meeting that is included as a participant before.

2.4.3 Stories

In Rasa, we define our stories/paths so that in our NLU pipeline, the machine learning algorithm can learn how it needs to act and can augment by cropping and combining paths to cover many more cases given by ourselves. Our chatbot is for scheduling meetings, so we define its story.

- **Schedule Meeting Happy Path:** This is the happy path of the scheduling meeting story, in this path, the user gives logical inputs to the chatbot, and the chatbot schedules the meeting by creating a form from the inputs that the user gives to it and in the end schedules the meeting.

- **Schedule Meeting Continue:** In this path user is not always giving logical inputs to the chatbot and the chatbot understands this by catching those inputs classifying them as out of scope intent and asks the user if (s)he wants to continue to arrange a meeting or wants the stop. If the user wants to continue, the chatbot understands it by classifying his/her intent as affirm and continue asking questions to get the needed information to schedule a meeting.

- **Schedule Meeting Stop:** This story is similar to the continuous path until users reply to the question if (s)he wants to continue or not, in this path user refuses to continue and the chatbot classifies his/her intent as deny. After that, it stops asking questions and says "goodbye" to the user.

2.4.4 Rules

Rules in Rasa is to describe short pieces of conversations that should always follow the same path.

- **Say goodbye anytime the user says goodbye:** When the 'goodbye' intent is understood by Rasa NLU, the goodbye rule invokes and the bot immediately says "goodbye".
- **Say 'I am a bot' anytime the user challenges:** When the 'bot challenge' intent is understood by Rasa NLU, bot challenge rule invokes and says "I am a bot.".
- **submit schedule meeting form:** This rule manages the last part of schedule a meeting intent. When the 'Schedule Meeting' form is filled and the chatbot extracts the last form entry, the rule is invoked and 'action schedule event' is called to schedule a meeting with the given information.
- **Show upcoming events when user intents** When the user asks for his/her upcoming events, Rasa NLU understands it and responds with the most recent upcoming event.
- **Add new people or groups to mail list** While schedule a meeting form is active, which means that the bot is asking for form inputs to the user. After the user enters attendees, if he/she wants to add more people to the meeting, this rule invokes and calls the 'action add new people or groups to meeting' action.
- **Remove people or groups from mail list** While schedule a meeting form is active, which means that the bot is asking for form inputs to the user. After the user enters attendees, if he/she wants to remove people from the meeting, this rule invokes and calls the 'action remove new people or groups from meeting' action.

2.4.5 Regex for Extraction

Rasa also allows a developer to define any regular expressions near its NLU pipeline extractions. If both extractors catch the entity you can use any one of them by deciding them in the actions.

- **user info:** We give examples so that our extractors in the pipeline can extract the related entries but for meeting participants, we mostly know the format of them so we can write regular expressions to it. Below is our regular expressions:
  - ([a-zA-Z0-9 \-\.]+)@([a-zA-Z0-9 \-\.]+)\.([a-zA-Z]{2,5})
  - group (\w+|\d+)

2.4.6 Actions

In Rasa, Actions is a standalone server that manages the needed computations by receiving requests from the bot and responding with related information.

- **Action Add New People or Groups to Meeting:** In the actions server, we keep the attendee list as an array. If the meeting manager wants to add more people to the meeting that currently scheduling, this action takes the mail information and adds the corresponding users to the array.
- **Action Remove New People or Groups from Meeting:** In the actions server, we keep the attendee list as an array. If the meeting manager wants to remove people from the meeting that currently scheduling, this action takes the mail information and removes the corresponding users from the array.
- **Action Validate User Info:** This validation is triggered automatically after extracting a user info entity. We check if the group(s) and/or email(s) are can be valid participants for the user that arranging the meeting.
- **Action Validate Duration:** This validation is triggered automatically after extracting the duration entity. We extract by using the duckling[33] server and we convert it to minutes every time since it returns us as seconds.

- **Action Schedule Event:** When the scheduling form is fulfilled, then this action invokes and passes the following information to the Scheduling Algorithm: Attendee emails, meeting time, and meeting name.

  - **Scheduling Algorithm:** First of all we send a request to the back-end to learn participants' calendar events for the future 5 days. Then, we create duration-long and 1 minute separated chunks. These chunk slots are initially set to 0. Afterward, for each chunk, we check if the user is free or not. And if (s)he is free, the slot is filled by 1. After this filling step is done for each attendee, columns of the matrix are summed up and checked if the summation is equal to the number of attendees. If a time slot is found, then it sends a request to the back-end to insert the event and send informative emails to the attendees. In the end, chatbot shows the meeting details to the user.

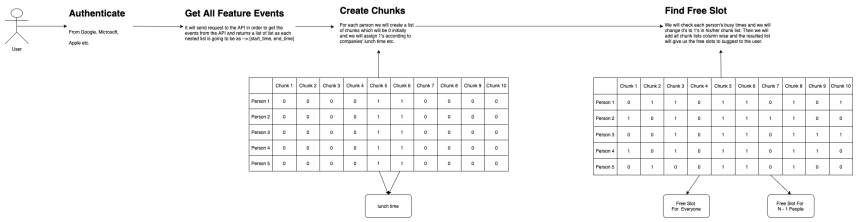


Figure 47

From the graphs, we can see that request time is not depend on participant count since we parallelize that process (see **Appendix** for graphs). All requests are sent to Google API in a parallel fashion. When we look at the time that scheduling algorithm takes, we can see that is is increasing while the participant count is increasing with a constant duration. However this increase is not exponential or linear, it is more likely to logarithmic which is a really good solution. Also when duration is increasing, the scheduling time decreases since the algorithm creates less chunks and it takes less time compared to a shorter duration meeting scheduling.

2.4.7 NLU Pipeline

In Rasa, NLU pipeline is to understand and classify the inputs of the users.

- **SpacyNLP:** We used Spacy's tokenizer and featurizer. Language model is the pre-trained model en_core_web_md.
- **RegexFeaturizer:** The featurizer is explained in section 2.4.5.
- **LexicalSyntacticFeaturizer:** Creates lexical and syntactic features for a user message to support entity extraction.
- **CountVectorsFeaturizer:** Creates bag-of-words representation of user messages, intents, and responses.
- **DucklingHTTPExtractor:** This extractor is explained in section 2.5.6.
- **DIETClassifier:** Dual Intent Entity Transformer (DIET) used for intent classification and entity extraction
- **EntitySynonymMapper:** Maps synonymous entity values to the same value.
- **ResponseSelector:** As the last process of the pipeline, it selects the response.

2.5 Integration

We combine all parts of the project and as an outcome, we successfully created the flow of our application from beginning to end in a basic way. Below, you can find the integration steps part by part and at the end, a flow diagram is created by us for further explanation.

2.5.1 Front-end - Back-end

Front-end and Back-end of the website are connected with HTTP [20] requests. To increase security, requests should contain a JWT token. Flow as follows: Client sends a request to a specific endpoint with requirements of that endpoint. Then server processes input, and if required, sends queries to the database. After all, it responds user with the data or an error.

2.5.2 Back-End - Google Authentication

At first, we created an account in the Google Developers Platform and got the necessary keys. Then with the use of the "Get URL for calendar Authentication" API which is detailed above in the "Google Related APIs" section, we create a Google Authentication URL for each user. When a user proceeds to the link, several permissions about Google Calendar are asked to the user. If the user accepts them, Google redirects the user into our "Create Refresh Token" endpoint. When these steps are completed, we wrote the google calendar refresh token of the user to the database for future needs.

2.5.3 Back-end - Database

Since our back-end framework is Node.js, we are using Mongoose [34] database managing framework. It makes database queries easier with its predefined functions. We store MongoDB secret keys on the environment file. When we initially run the server, those keys are accessed from the environment file and used to connect our back-end server to MongoDB.

2.5.4 Back-end - Chatbot

Rasa chatbot has its server which called "actions". Actions are written in Python [35], so we used Python's requests framework to communicate with the back-end server with HTTP requests. For instance, when an intent is given a group of users or a specific user to the chatbot. The chatbot also needs to check those groups and/or users so that, we can continue with the scheduling algorithm without any problem. To do these checks, our bot connects with our back-end and first gets the groups or company users when needed.

2.5.5 Chatbot - Scheduling Algorithm

Also in Rasa's actions server, we implemented our scheduling algorithm. When the bot needs to find an optimal slot for the meeting, it sends a request to the back-end server to get attendees' calendar information. Afterward, runs scheduling algorithm on actions server. Then sends another request to the back-end to create a new event on Google Calendar.

2.5.6 Chatbot - Duckling

We add duckling entity extractor to the pipeline and told the port to the Rasa Bot where Duckling is running. After that Rasa Bot can reach and use that Duckling server.

2.6 Deployment

Deployment is the last part of this project, after creating all part and combine them together. For world to use this tool it need to be deployed to a cloud (server) with a public IP address so that others can reach the website and chat with the bot. We used Digital Ocean [36] while deploying our project. We first create a droplet from its website and then access the server by ssh access. Then transferred necessary files and download necessary packet that project codes needs.

2.6.1 Database

This project is using the database service MongoDB Atlassian. So, it is already reachable on a server. It is a proved, secure, NoSQL database service so we did not deploy anything to our server, we just used our credentials in our ".env" file and pass them to our back-end in order to reach our database.

2.6.2 Back-end and Front-end

This projects back-end and front-end is Dockerized. So we are running them by just running and detaching their production dockers inside the Ubuntu terminal. We also expose the port 8080 for back-end and port 3000 for front-end, so that they are reachable by users.

2.6.3 Rasa Actions and Rasa Bot

We run Rasa Actions by running the command "rasa run actions" and we expose port 5505 for Rasa Bot to reach and we run Rasa Bot with the command "rasa run" and we expose port 5005 for users to reach it over Hangouts. But we both run them as a back process by using nohup [37].

2.6.4 Duckling

We did docker pull to download the Duckling server from Docker Hub and run and detached it and expose the port 8000 for Rasa Actions to reach it.

3 IMPACT

This project can help workers and companies to gain lots of time in total. Each white collar employee spends their 20-25 minutes [38] to arrange a meeting and attends 60 meetings per month [39] in average which contains 5 participants in average. This means each participant arranges 12 meetings per month in average. When we calculate, an employee spends 48 hours to arrange meetings per year in average. With this tool we are planning to decrease it to 2 minutes per day which decreases the time to 4,8 hours per year. We can save 12.960 hours of work each year from a company which has 300 employees. It may help employees to get less bored since they do not need to spend with a boring job like arranging meetings, instead our bot can handle it for them easily. Also, in the aspect of the companies, this a lot of wasted time is costs huge amount of money. This project can also used in commercial use and can be converted to a start-up project.

4 ETHICAL ISSUES

In the project, only relevant and sufficient resources are used. There are no not ethical or illegal consequences in the solution.

- We used open source products and frameworks.
- We store data without personal information and we are using user's calendars with permission.

5 PROJECT MANAGEMENT

5.1 Initial Plan

Initial gannt chart of timeline can be seen in Figure 48 below.

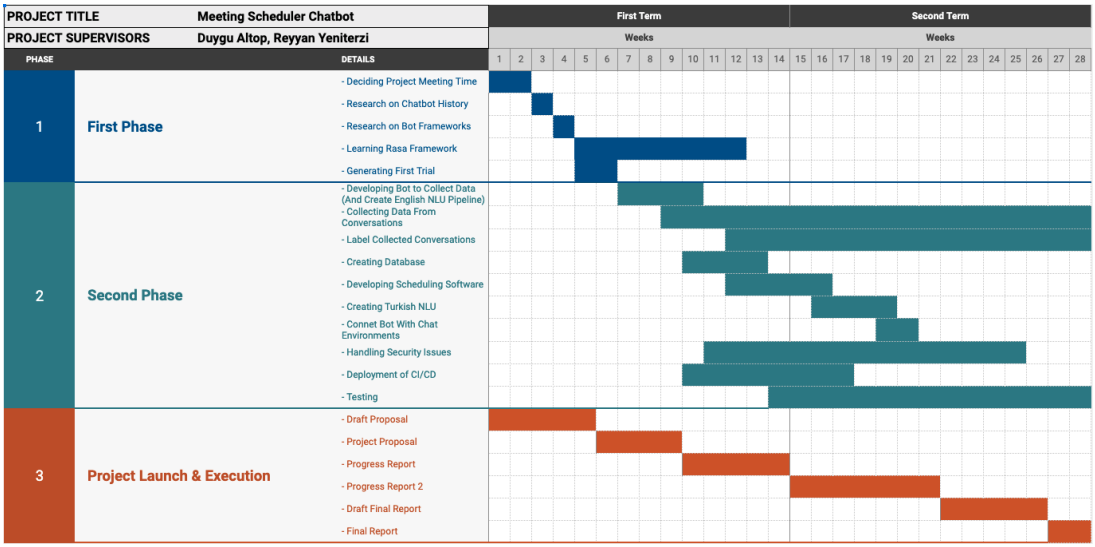


Figure 48

5.2 Plan in the Progress Report 1

After the timeline that we created at the proposal, we did some changes until the progress report 1.

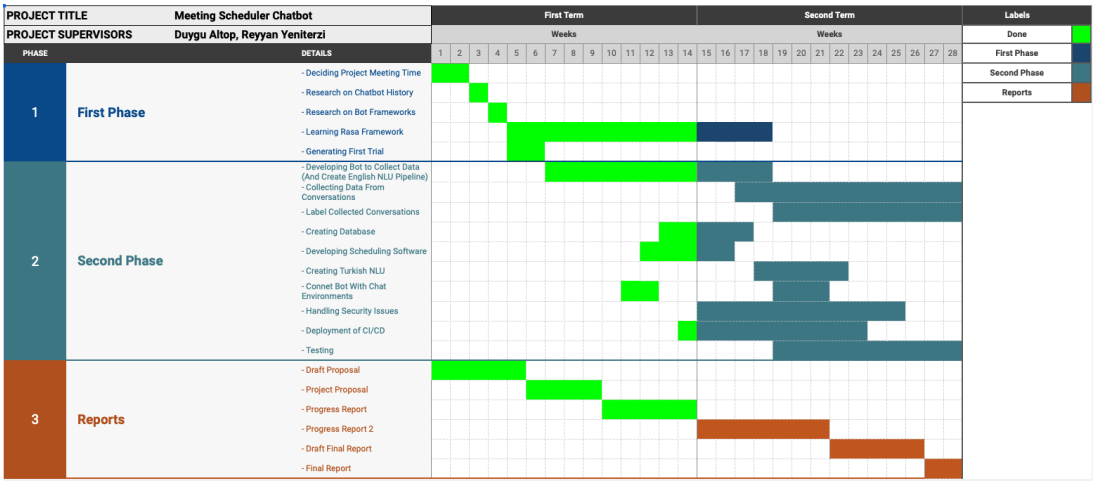


Figure 49

- **Group Intents Removed:** When we shared our ideas about the chatbot with people and received their feedback, we decided to migrate the group creation and managing part to the website from chatbot to increase the user experience.

There are some objectives that we were behind the timeline. The reasons we were back from the schedule were, doing some parts of the project earlier than planned such as connecting chatbot to chat environments like slack and adding some new intents that realized later parts of the design process that we needed to implement those intents too. Some of the following objectives were in mid-process, and some of them were not started yet.

- **Learning Rasa Framework:** We learned the basics and some advanced aspects of Rasa Framework but, we were still needed to learn the advanced parts of the framework. So we needed to continue learning as we were developing the bot.
- **Developing Bot:** As the end of Progress Report I, our chatbot can talk and schedule a meeting but still it is not product level, we needed to continue developing so we extend the deadline for this task.
- **Develop Scheduling:** We developed a scheduling algorithm but still some improvements could be done. Our algorithm was able to schedule a meeting with five people in less than a second. Besides, we also planned to add the rescheduling part of this algorithm until the end of the project, for that reason we extended this deadline but since other parts of the project are more essential we needed to pass this feature.
- **Collecting data:** We did not collect any data because our chatbot was not ready at the planned time.
- **Label Collected Data:** As we did not collect any data, we cannot label it.

5.3 Plan in the Progress Report 2

We made some changes after the last report, we have done those changes due to time constraints and for the efficiency of the code.

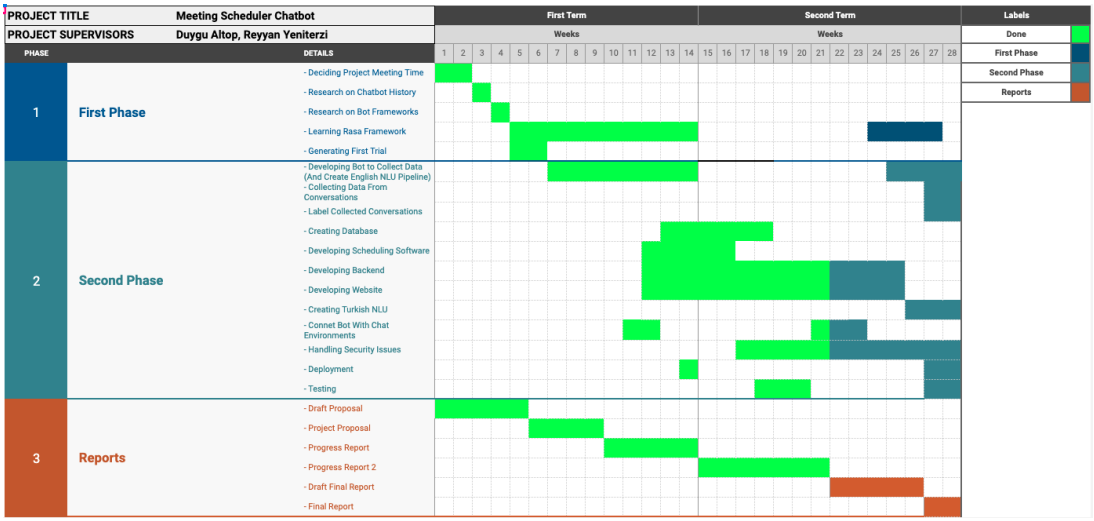


Figure 50

- **Google Authentication:** Authentication of the Google Calendars was managed in Python by low-level HTTP requests with the help of the requests library [40]. Since we chanced the back-end server to Node.js, we migrated our authentication service to JavaScript. While migrating, the separate Python code that we have written, to the back-end server, as an API, we also did some more research on it and found out that Google has a library for its APIs and we used it while writing the calendar API at our back-end. By using Google Calendar APIs, we also strengthen the security of the authentication process. However, while this change made our code better and faster, it also slowed us compared with our planned timeline.

There were some objectives that we were behind the timeline. The reasons we were back from the schedule were; doing some changes, fastening the project with the technologies we learned during the term break, doing some changes to make our codebase more secure as we mentioned at the beginning part of this section. There was one other reason that causes the project to stay back from the schedule is some learning process of other technologies that we were lack of knowledge like Redux and Docker for the back-end.

- **Learning Redux:** While creating the front-end as we mentioned in the second section, we needed to use Redux to handle state management. But this process required us to learn the Redux extension for React projects, and it took our time more than we expected since it was a more complex tool than we thought.

- **Learning Docker for Front-end:** Although we had an experience with Dockerizing the back-end, it took time for us to learn it for our front-end. We went over several tutorials and tried to solve the problems that we encountered, learning the necessary parts and solving those problems slowed us down.

Due to those problems that we encounter about time constraints, there were some topics that we still could not start covering such as data collection and labeling those collected data.

- **Collecting Data:** We did not collect any data due to time constraints and unpredicted problems that we encountered and mentioned above. Despite, our bot was mostly developed, we had to deploy the bot to the server and due to timing, we could not do it at that time. Also, after combining the scheduling algorithm and back-end to the bot, we achieved doing some intents but still, we needed to manage some other intents for the bot to be fully ready for the data collection process.

- **Label Collected Data:** As we did not collect any data until that time, we could not label it. So this task was also postponed since the previous one needed to be handled first.

5.4 Plan in the Final Report

After the timeline that we created at the Progress Report 2, We did not make changes until the Final Report. However, there are some objectives that we could not finish due to time constraints.

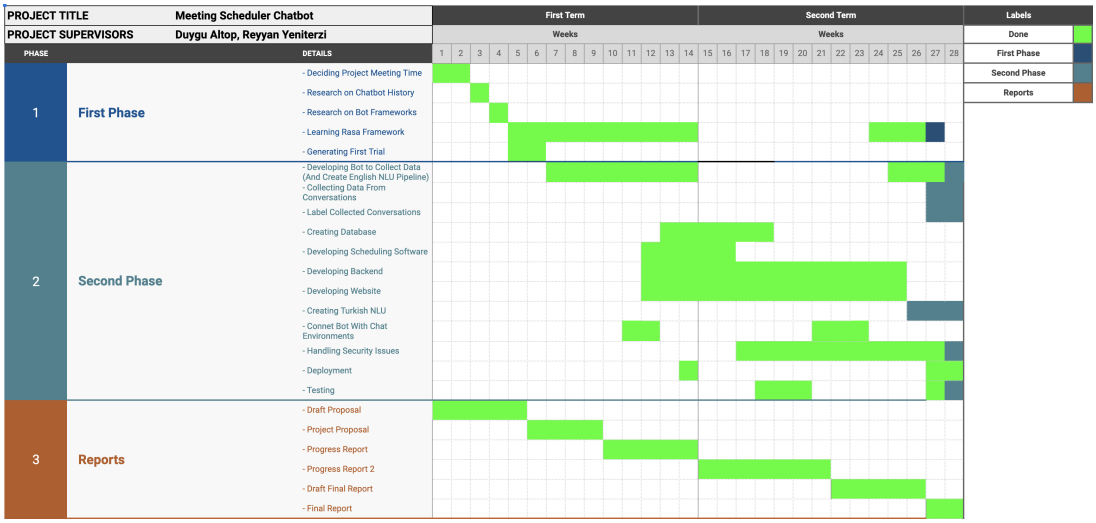


Figure 51

- **Learning Rasa Framework:** Rasa is a complex and advanced tool that makes it hard to be mastered. Also, it constantly gets updates which push developers to periodically learn. Thus, we did not learn Rasa completely.
- **Collect Data:** To collect data from real users, there are several tasks to be done before. Firstly, the bot, front-end, and back-end should be developed and deployed on a server. Secondly, each of the modules of the project should be stable and easily monitored in the case of any errors or bugs. We managed to deploy our project at the very end of the Ens492 project, which makes it impossible to collect data from real users during the project.
- **Label Collected Data:** As stated above, we did not collect any real-world data so it is not possible to label any data.
- **Turkish NLU:** We did not have time to develop Turkish NLU.
- **Handling Security Issues:** We tested our static code with a source code analyzer tool, but we did not try to attack each of the APIs.

- **Testing:** We could write unit tests for each API, but we did not have time to do so, but we tried by our hands over Postman.

The most important thing that we learned about project management is; There can always problems that can occur and a project can bring to a standstill. So for those times, while managing the project and doing the schedule we need to give extra time for the deadlines and consider the project in that way.

6 CONCLUSION AND FUTURE WORK

In this projects, many components are used together to come up with a solution. Although most of the planned tasks are done, still there are objectives to be completed. We started the project to create a chatbot for scheduling meetings which able to chat in both Turkish and English. However due to limitations, we finished English version only and the project also evolved in a different way due to requirements and we have a website beside our chatbot in order to handle those requirements.

6.1 Results

We learned Rasa framework in order to develop our chatbot and by using our knowledge we implemented the bot. We created the NLU pipeline for our bot in English. We tried several different pipelines and we decided that using Spacy extractors are giving us the best experience while talking with chatbot and best training results. Then we connected our chatbot to different chat environments which are Slack and Hangouts. One can go to their Gmail account, open hangouts chat, and can find our bot to chat. We created our website's back-end, front-end and we integrated it with Google Calender to access their calender information. So that we can use it while scheduling their meetings, with the efficient scheduling algorithm that we developed for that purpose. In the end we integrate all of those components with docker and deployed them to the Digital Ocean Cloud Platform. In Figure 52 below, a user's experience from the beginning to the end can be found.

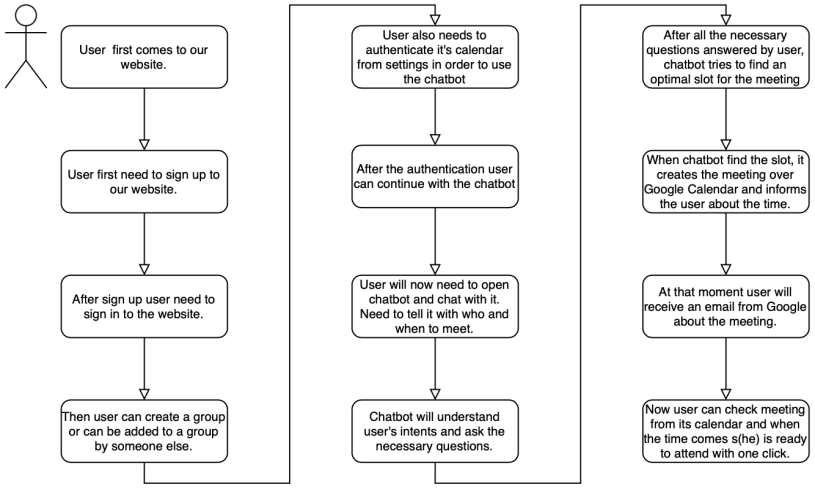


Figure 52

6.2 Future Work

As future work, creating Turkish NLU pipeline, data acquisition and labeling can be done. We planned to handle them in this project but due to time constrains and encountered problems, it was not possible for us to finished them all on time. Beside those, the UI and UX of the website can also be improved because we mostly created the website without considering those concepts much since we created those website as a proof of concept.

7 APPENDIX

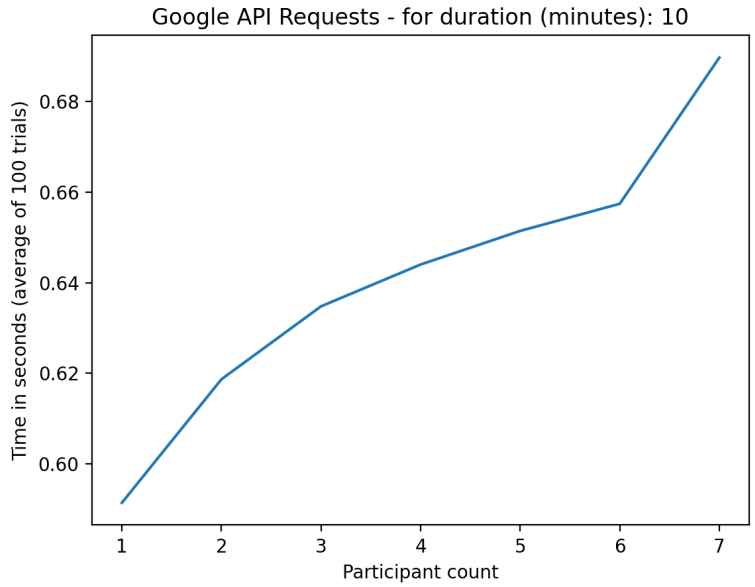


Figure 53

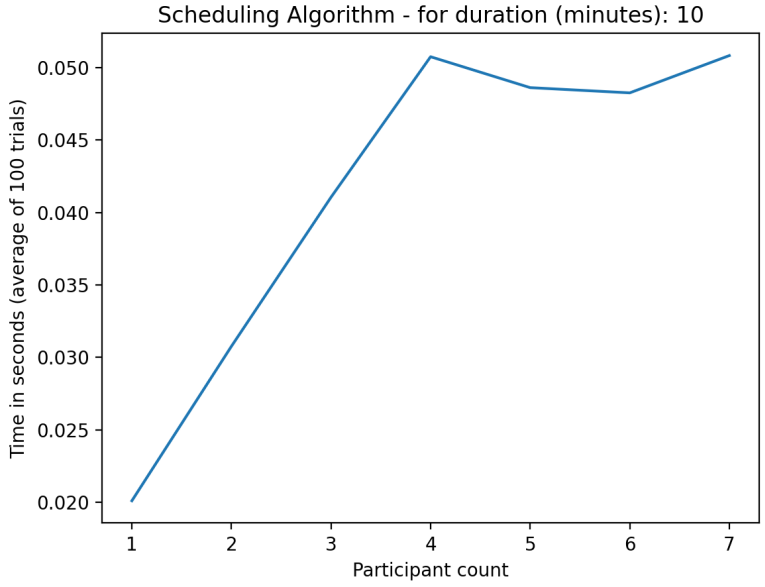


Figure 54

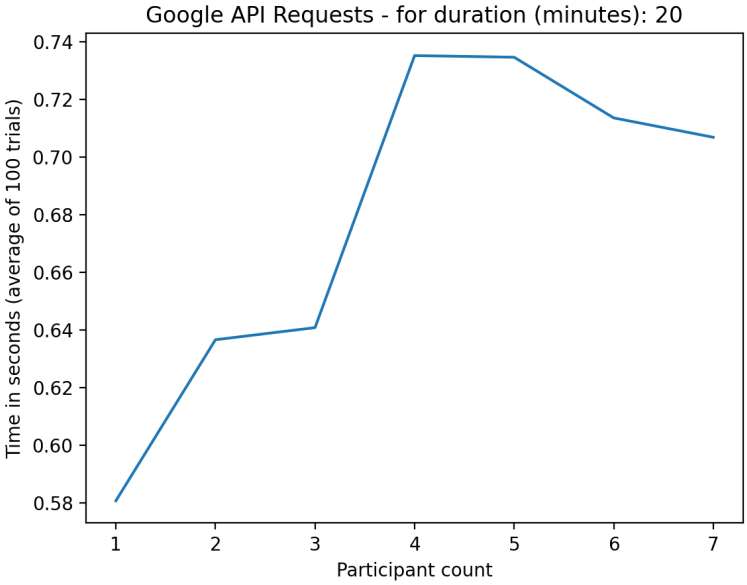


Figure 55

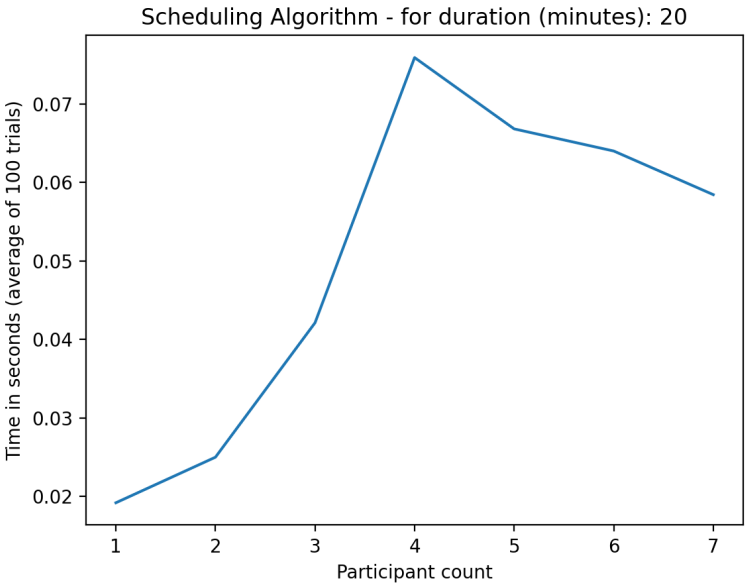


Figure 56

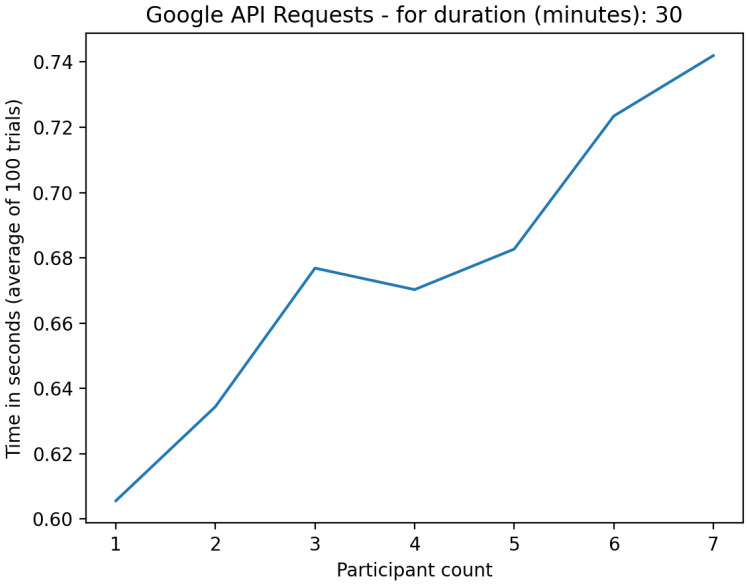


Figure 57

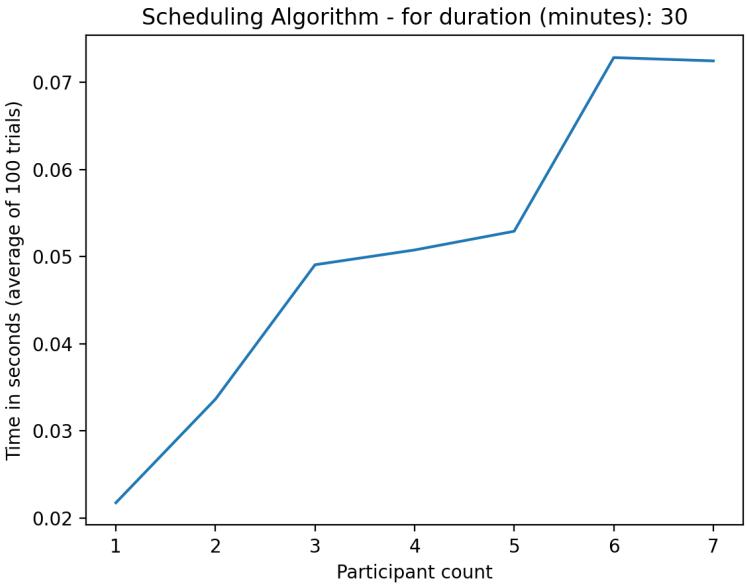


Figure 58

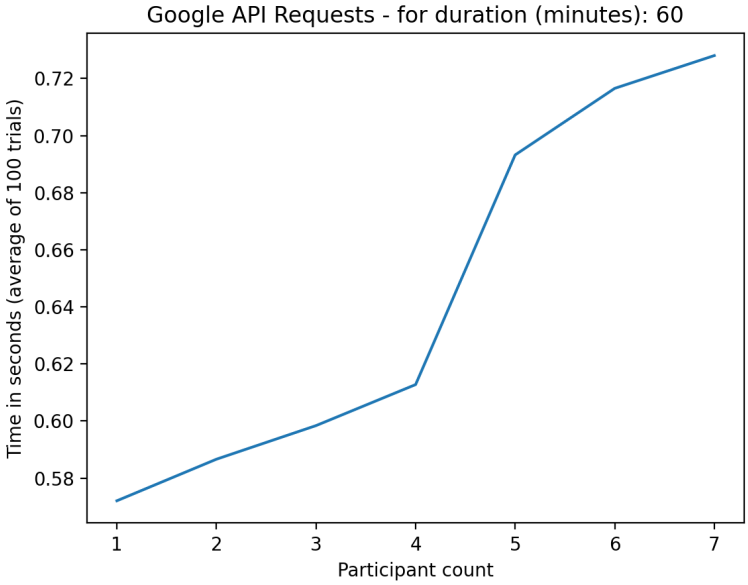


Figure 59

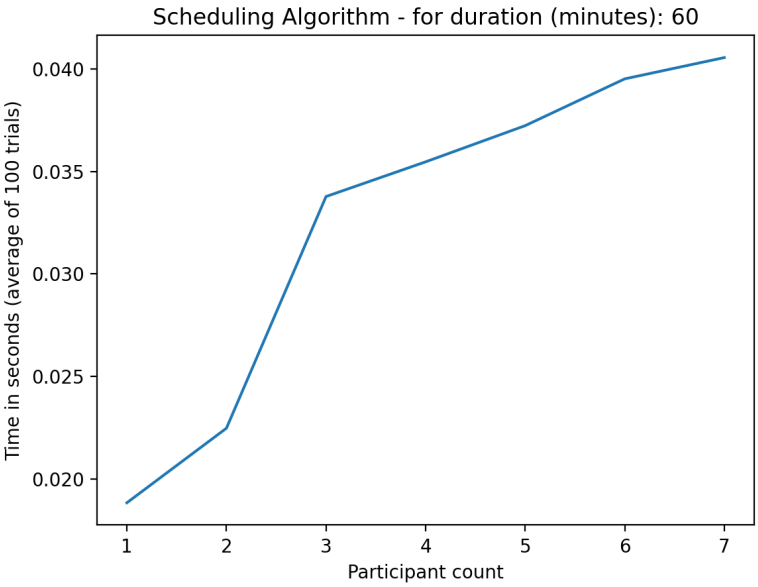


Figure 60

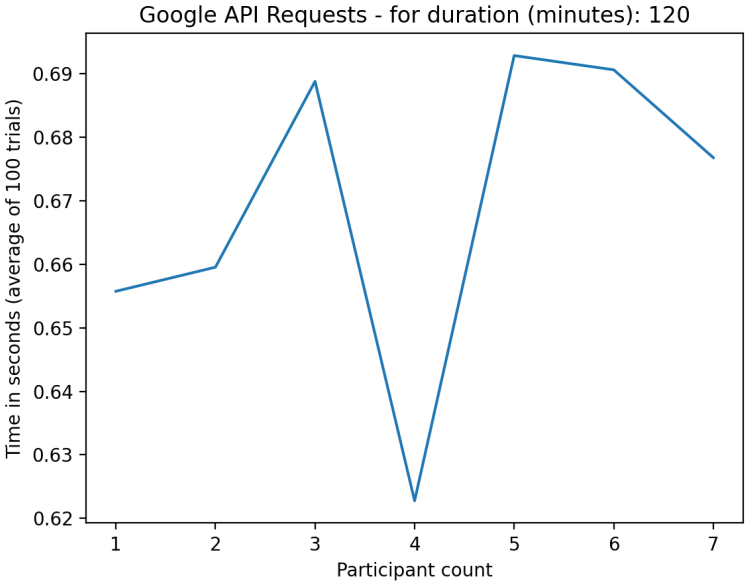


Figure 61

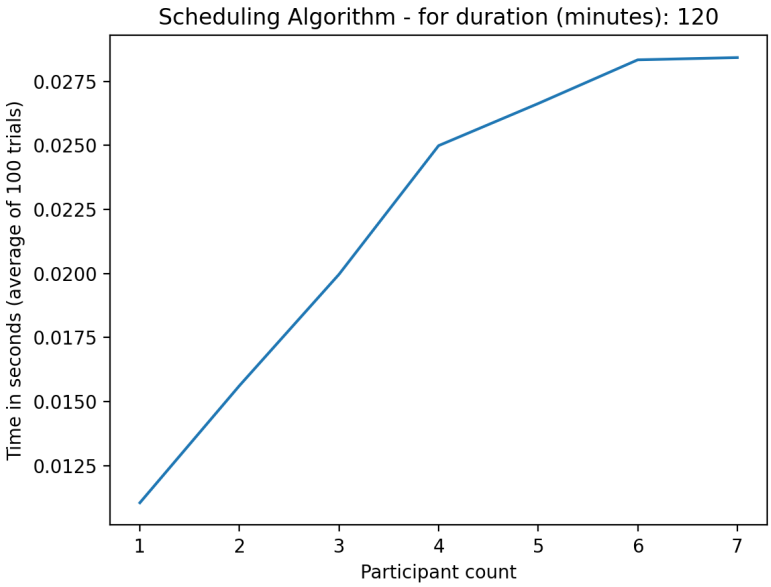


Figure 62

8 REFERENCES

1. Slack. Welcome to your new HQ. Slack. https://slack.com/intl/en-tr/
2. Chat, Meetings, Calling, Collaboration — Microsoft Teams. www.microsoft.com. https://www.microsoft.com/en-ww/microsoft-teams/group-chat-software
3. Google Hangouts. hangouts.google.com. https://hangouts.google.com
4. Jurafsky, Dan. CS 124/LINGUIST 180 From Languages to Information Conversational Agents.
5. Schedule Simple, Get saved! — Kono.ai. Kono.ai : AI Scheduling Assistant for Enterprises. Published September 1, 2017. Accessed April 10, 2021. https://about.kono.ai/
6. Google Calendar: Free Calendar App for Personal Use. www.google.com. https://www.google.com/calendar/about/
7. Calendar. office.live.com. Accessed April 10, 2021. https://office.live.com/start/Calendar.aspx?ui=en
8. Doodle Bot for Slack. Doodle's Content Pages. https://doodle.com/content/doodle-bot
9. Free Online Appointment Scheduling Software - Calendly. calendly.com. https://calendly.com
10. Zapier. The easiest way to automate your work. Zapier. Published 2011. https://zapier.com/
11. Rasa: Open source conversational AI. Rasa.com. Published 2019. https://rasa.com/
12. Enterprise Application Container Platform — Docker. Docker. Published 2018. https://www.docker.com/
13. Fully Managed MongoDB, hosted on AWS, Azure, and GCP. MongoDB. Published 2019. https://www.mongodb.com/cloud/atlas

14. MongoDB. NoSQL Databases Explained. MongoDB. Published 2019. https://www.mongodb.com/nosql-explained

15. JSON. www.json.org. Accessed April 10, 2021. https://www.json.org

16. React – A JavaScript library for building user interfaces. Reactjs.org. Published 2019. https://reactjs.org/

17. Node.js Foundation. Node.js. Node.js. Published 2019. https://nodejs.org/en/

18. Material-UI: A popular React UI framework. material-ui.com. https://material-ui.com

19. Redux - A predictable state container for JavaScript apps. Js.org. Published 2015. https://redux.js.org/

20. Free JavaScript training, resources and examples for the community. www.javascript.com. https://www.javascript.com

21. How To Safely Store A Password. codahale.com. Published January 31, 2010. https://codahale.com/how-to-safely-store-a-password/

22. auth0.com. JWT.IO. jwt.io. https://jwt.io

23. "Google APIs Explorer." Google Developers, developers.google.com/apis-explorer.

24. Abraham, Ajin. "Ajinabraham/Njsscan." GitHub, 17 May 2021, github.com/ajinabraham/njsscan.

25. "Helmet." Helmetjs.github.io, helmetjs.github.io. Accessed 24 May 2021.

26. Greyling, Cobus. "Updated: A Comparison Of Eight Chatbot Environments." Medium, 31 May 2020,https://cobusgreyling.medium.com/updated-a-comparison-of-eight-chatbot-environments-7f57d4e2dc09. Accessed 8 Nov. 2020.

27. "IBM Watson." Ibm.com, 2019, www.ibm.com/watson.

28. "Microsoft Bot Framework." Dev.botframework.com, dev.botframework.com. Accessed 24 May 2021.

29. Devlin, Jacob, et al. “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.” ArXiv.org, 2018, arxiv.org/abs/1810.04805.

30. spaCy · Industrial-strength Natural Language Processing in Python. Published 2015. https://spacy.io/

31. Brown, Tom B., et al. “Language Models Are Few-Shot Learners.” Arxiv.org, 28 May 2020, arxiv.org/abs/2005.14165.

32. Bunk, Tanja, et al. DIET: Lightweight Language Understanding for Dialogue Systems.

33. “Duckling.” Duckling.wit.ai, duckling.wit.ai/. Accessed 24 May 2021.

34. Mongoose ODM v5.12.3. mongoosejs.com. Accessed April 10, 2021. https://mongoosejs.com

35. Python. Welcome to Python.org. Python.org. Published May 29, 2019. https://www.python.org/

36. “DigitalOcean – the Developer Cloud.” DigitalOcean, www.digitalocean.com/.

37. “Nohup Command.” Www.ibm.com, www.ibm.com/docs/en/aix/7.1?topic=n-nohup-command. Accessed 24 May 2021.

38. Doodle. “Study Reveals Time Spent with Scheduling.” Doodle Blog, 8 July 2013, en.blog.doodle.com/2013/07/08/study-reveals-time-spent-with-scheduling/. Accessed 24 May 2021.

39. “Verizon Conferencing - Audio, Web, and Video Conferencing Services.” E-Meetings.verizonbusiness.com, e-meetings.verizonbusiness.com/global/en/meetingsinamerica/uswhitepaper.php.

40. Requests: HTTP for HumansTM — Requests 2.25.1 documentation. docs.python-requests.org. Accessed April 10, 2021. https://docs.python-requests.org/en/master/