App Support Chatbot

Team Members:

Meetkumar Patel
meetpatel0963@gmail.com

Devansh Suthar
devanshsuthar0612@gmail.com

Kartik Bhanderi
kartikbhanderi1151@gmail.com

Index

Introduction to Chatbots

What is a Chatbot ?

A chatbot is a computer program that simulates and processes human conversation, allowing humans to interact with digital devices as if they were communicating with a real person. Chatbots can be as simple as rudimentary programs that answer a simple query with a single-line response, or as sophisticated as digital assistants that learn and evolve to deliver increasing levels of personalization as they gather and process information.

Types of Chatbot

Chatbots process data to deliver quick responses to all kinds of user requests with pre-defined rules and AI based chatbots. There are two types of chatbots.

- Rule based chatbots
    - Rule-based chatbots also referred to as decision-tree bots, use a series of defined rules. These rules are the basis for the types of problems the chatbot is familiar with and can deliver solutions for.
    - These bots follow predetermined rules. So it becomes easy to use the bot for simpler scenarios.
    - Interactions with rule based chatbots are highly structured and are most applicable to customer support functions.
    - Rule based bots are ideally suitable for answering common queries such as an inquiry about business hours, delivery status, or tracking details.
- Conversational AI chatbots
    - These chatbots generate their own answers to more complicated questions using natural-language responses. The more you use and train these bots, the more they learn and the better they operate with the user.
    - It can further be classified into two types.
        - **Retrieval-based system**: When it's the machine's turn to respond, the model reaches into its repository of predefined responses and picks the most relevant answer. This system can rely on machine learning to retrieve the best response. Retrieval systems can only use the text they are given and can't generate new answers.
        - **Generative-based system**: In this case, responses are actually generated from scratch depending on the given conversation history.

This system can handle both common and unforeseen questions, making it appear more human-like and better for longer conversations. However, this savvy response system also increases implementation complexity.

Technology Stack

1. Chatbot Model
   Python, ML, DL, NLP
2. Chatbot UI
   React.js

Domain of Implemented ChatBot

Our Chatbot can answer user queries from CONFLUENCE, JIRA and BITBUCKET. Given Username and Password (Token), Users can ask any type of queries whose answer resides under chosen domain(Confluence, Jira or bitbucket). For more visual information and to understand other aspects of UI, one should visit the *Frontend of Chatbot* section.

Confluence

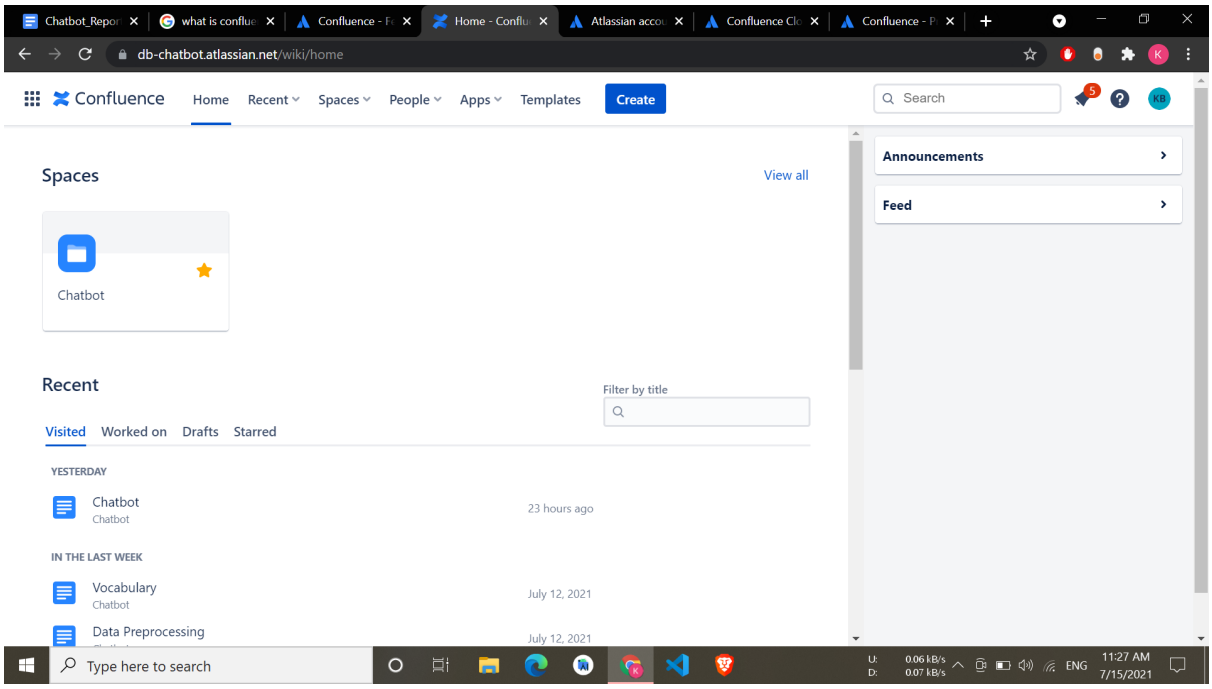
Confluence is a collaboration wiki tool used to help teams to collaborate and share knowledge efficiently. It helps to Create, collaborate, and organize all the work in one place. Confluence is a team workspace where knowledge and collaboration meet. Dynamic pages give the team a place to create, capture, organize and collaborate on any project or idea. Intuitive structure makes setup, creation, and discovery easy to use. To know more about Confluence, visit Link.

Everything in Confluence is organized in pages and spaces.

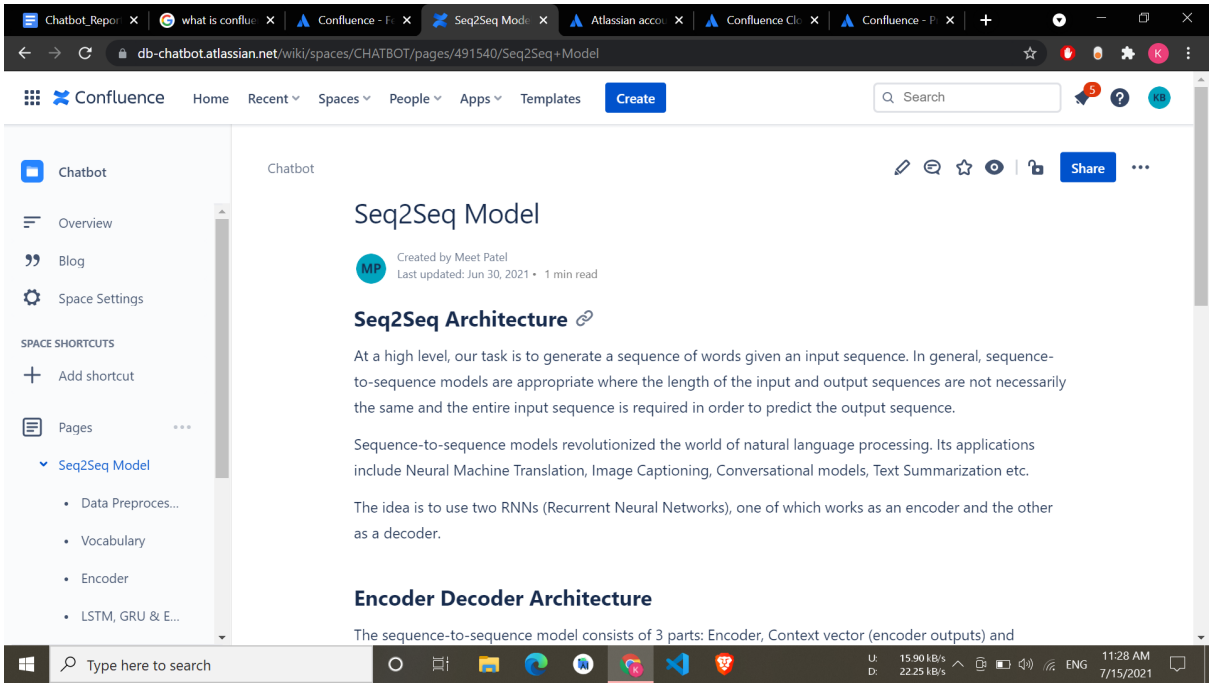
- Pages are documents where people create, edit, and discuss their work.
- Spaces are areas that contain pages for individuals, teams, and strategic projects.

Every type of user can create unlimited spaces and unlimited pages inside them. Only Enterprise users can create unlimited domains/sites.

This is how the homepage of any domain/site looks like.



This is how any particular space looks like. Space contains multiple pages which can have subpages. We can provide a description of space in the *overview* page.



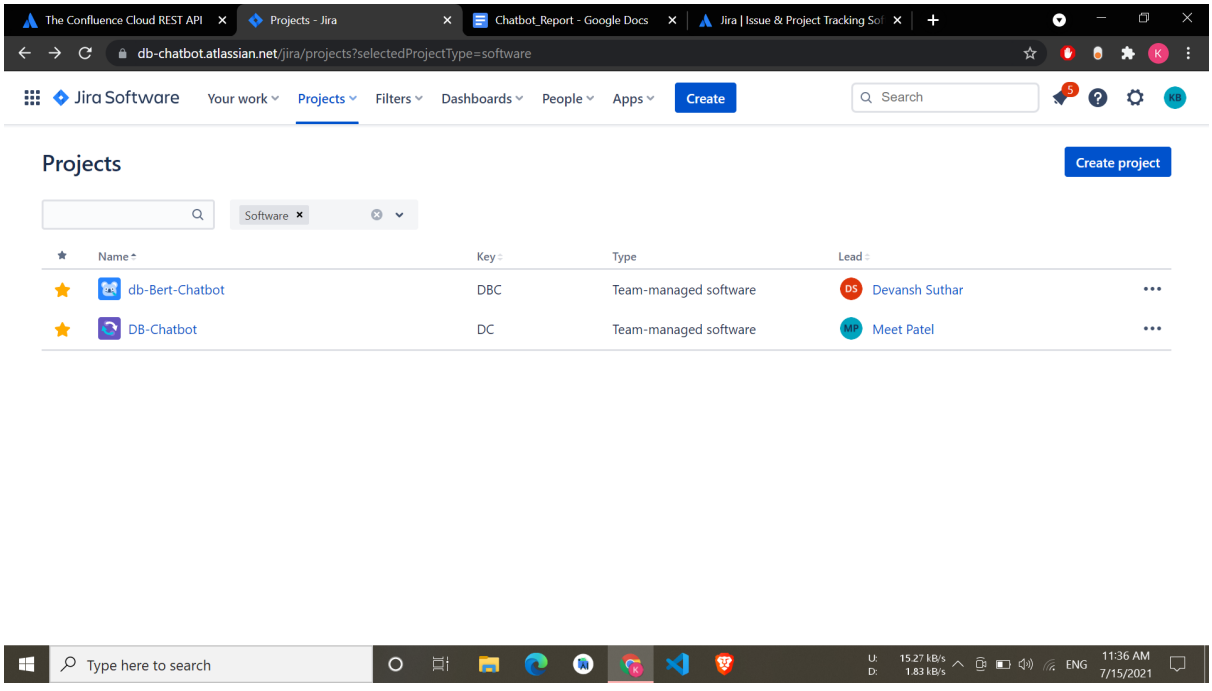
We can fetch details about all spaces and pages of any user using Confluence REST APIs. To know more about REST APIs, visit Link.

JIRA

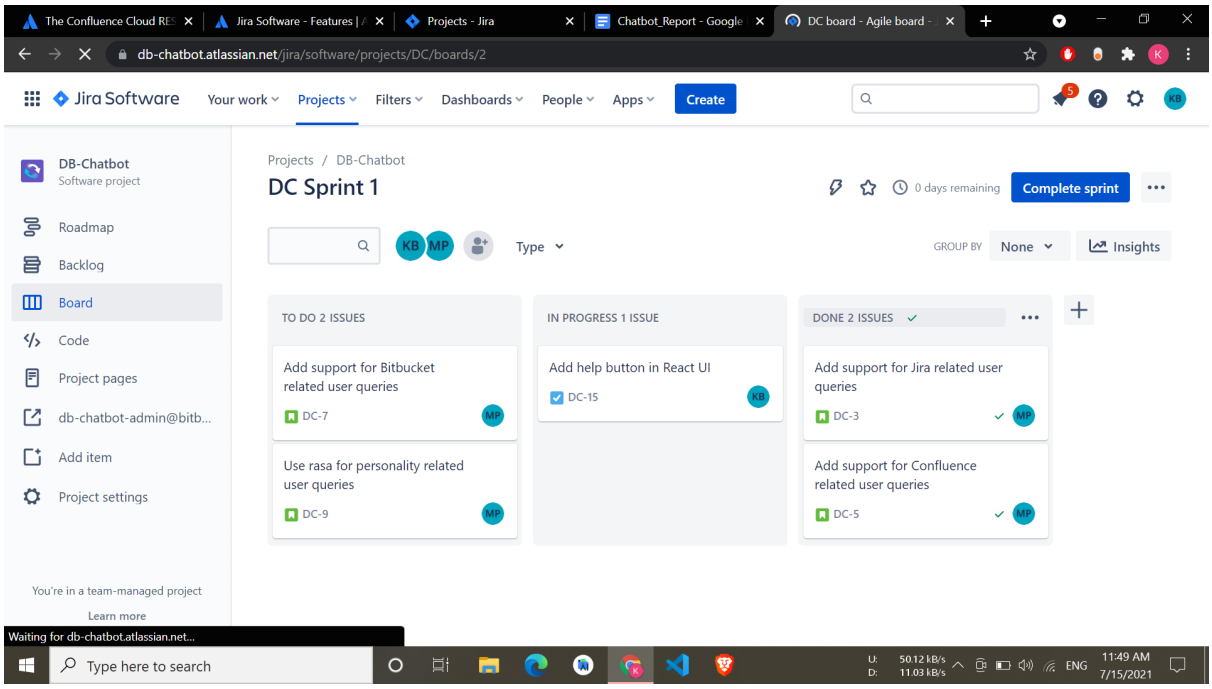
JIRA is a tool developed by Australian Company Atlassian. This software is used for bug tracking, issue tracking, and project management. The basic use of this tool is to track issues and bugs related to your software and Mobile apps. It is also used for project management. To know more about JIRA, visit Link.

Any project consists of Sprints which is a fixed time period in a continuous development cycle where teams complete work from their product backlog. At the end of the sprint, a team will typically have built and implemented a working product increment. Each sprint consists of **Issues, Bugs or Stories.** Each one of them can have properties like status, assignee, reporter, comments, description, parent etc. Each of these can have child issues/bugs/stories as well. Users can create unlimited sprints, issues, bugs or stories according to their need.

This is how the JIRA homepage looks like. It displays all the projects you are part of.



This is how the Dashboard of any project looks like.



We can fetch details about all sprints and its tasks of any user using JIRA REST APIs. To know more about REST APIs, visit Link.

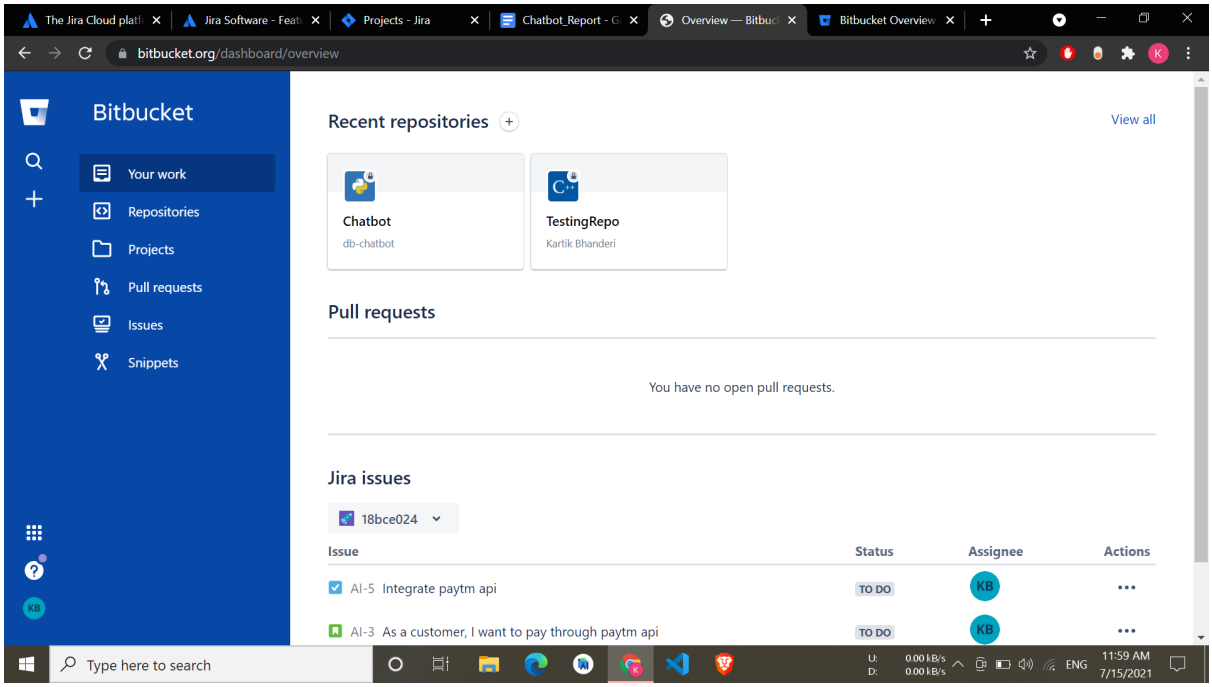
BitBucket

Bitbucket Cloud is a Git based code hosting and collaboration tool, built for teams. It provides one place for your team to collaborate on code from concept to Cloud, build quality code through automated testing, and deploy code with confidence. To know more about Bitbucket, visit Link.

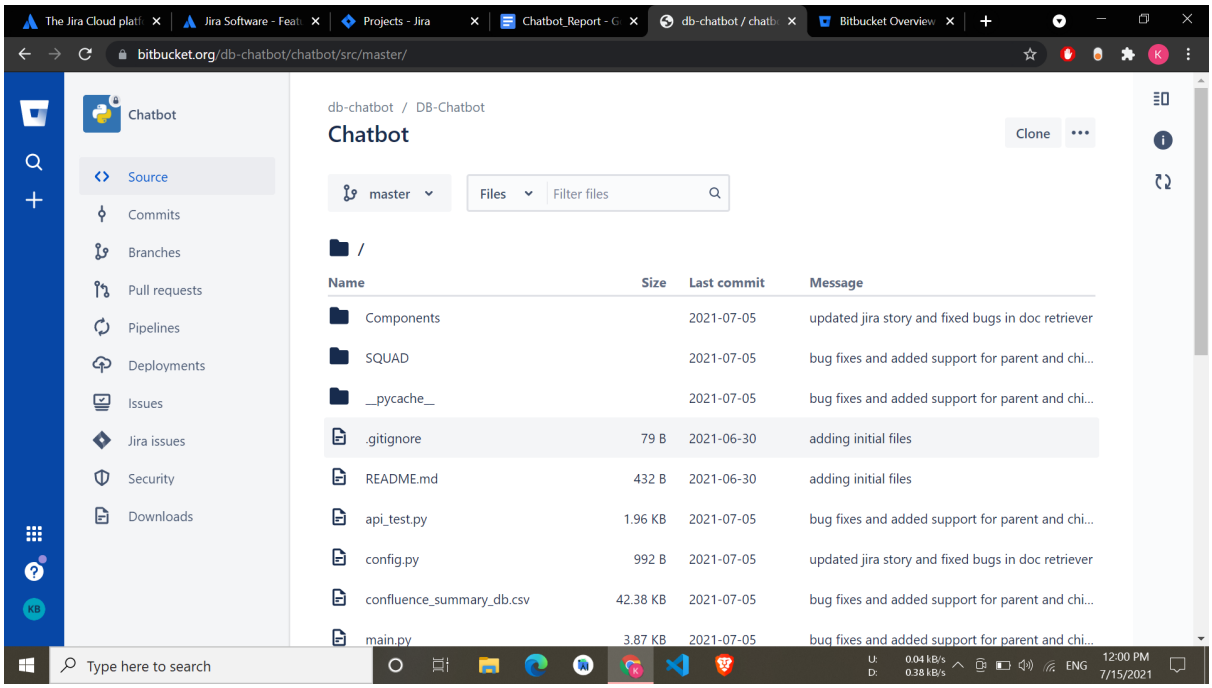
In Bitbucket, Users can have unlimited Workspaces. Each workspace consists of related Repositories. In each workspace, related Repositories can be grouped together.

Each repository consists of Code files of the project, All the details about commits, Number of branches in the current repository, All the pull requests, Issues and their statuses etc.

Bitbucket's homepage looks like this.



Particular Repository looks like this.



We can fetch details about all repositories of any user using bitbucket REST APIs. To know more about REST APIs, visit Link.

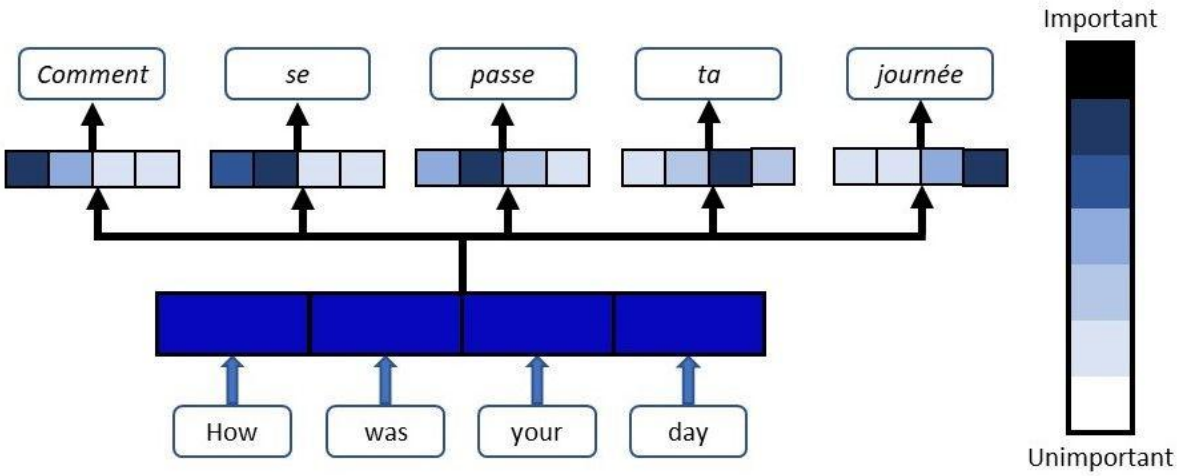
Attention Mechanism

Why Attention Mechanism ?

In our seq2seq model, the encoder reads a sequence of words and represents it with a high-dimensional real-valued vector, often called as a context vector, which is given to the decoder, which then generates another sequence of words in the target language. If the input sequence is very long, a single vector from the encoder doesn't give enough information for the decoder. A critical and apparent disadvantage of this fixed-length context vector design is incapability of remembering long sentences. Often it has forgotten the first part once it completes processing the whole input. The attention mechanism was born to resolve this problem.

Analogy

Say we have the sentence “How was your day”, which we would like to translate to the French version - “Comment se passe ta journée”. What the Attention component of the network will do for each word in the output sentence is map the important and relevant words from the input sentence and assign higher weights to these words, enhancing the accuracy of the output prediction.



Weights are assigned to input words at each step of the translation

Introduction

Attention is about giving more contextual information to the decoder. At every decoding step, the decoder is informed how much "attention" it should give to each input word.

While attention started this way in sequence-to-sequence modelling, it was later applied to words within the same sequence, giving rise to self-attention and transformer architecture.

While the context vector has access to the entire input sequence, we don't need to worry about forgetting. The alignment between the source and target is learned and controlled by the context vector. Essentially the context vector consumes three pieces of information:

- encoder hidden states;
- decoder hidden states;
- alignment between source and target.

An alignment score quantifies how well output at position *i* is aligned to the input at position *j*. The context vector that goes to the decoder is based on the weighted sum of the encoder's RNN hidden states h_j. These weights come from the alignment.

There are different alignment scoring functions available. We have defined three of them : dot, general and concat. These scoring functions make use of the encoder outputs and the decoder hidden state produced in the previous step to calculate the alignment scores.

- Dot
  The first one is the dot scoring function. This is the simplest of the functions; to produce the alignment score we only need to take the hidden states of the encoder and multiply them by the hidden state of the decoder.

$$score_{alignment} = H_{encoder} \cdot H_{decoder}$$

- General
  The second type is called general and is similar to the dot function, except that a weight matrix is added into the equation as well.

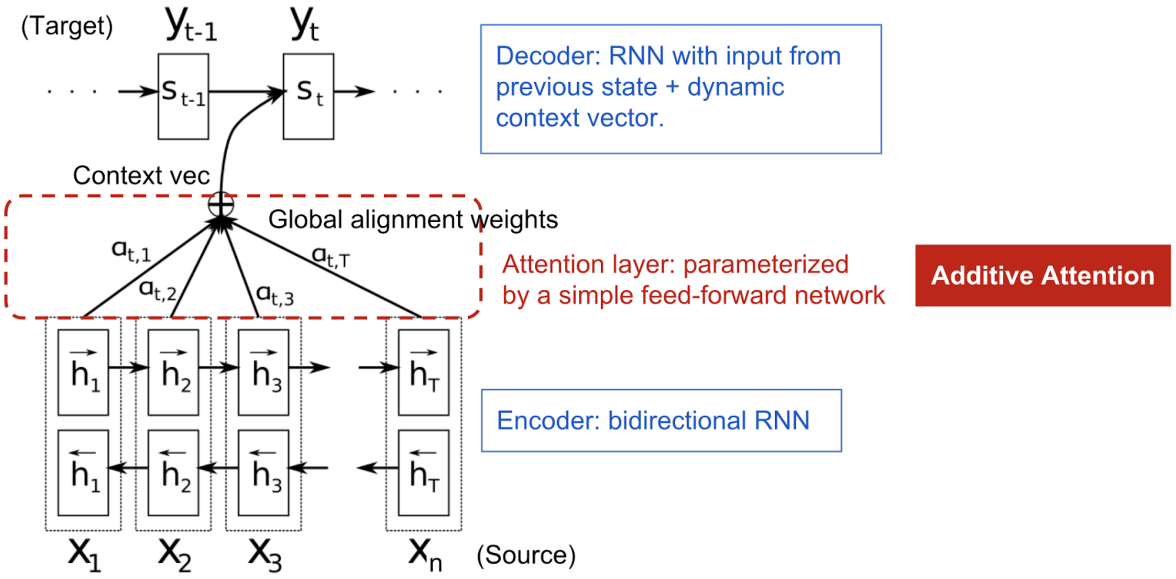
$$score_{alignment} = W(H_{encoder} \cdot H_{decoder})$$

- Concat
  This function is slightly similar to the way that alignment scores are calculated in Bahdanau Attention, whereby the decoder hidden state is added to the encoder hidden states.

$$score_{alignment} = W \cdot tanh(W_{combined}(H_{encoder} + H_{decoder}))$$

Now let's define the attention mechanism in a scientific way. Say, we have a source sequence x of length n and try to output a target sequence y of length m:

$$x = [x1, x2, ..., xn]$$
$$y = [y1, y2, ..., ym]$$



The encoder-decoder model with additive attention mechanism

The encoder is a bidirectional RNN with a forward hidden state $h_i\rightarrow$ and a backward one $h_i\leftarrow$. A simple concatenation of two represents the encoder state. The motivation is to include both the preceding and following words in the annotation of one word.

$$h_i = [h_i\rightarrow; h_i\leftarrow]^T, i = 1, ..., n$$

The decoder network has hidden state $s_t = f(s_{t-1}, y_{t-1}, c_t)$for the output word at position t, t=1,…,m, where the context vector c_t is a sum of hidden states of the input sequence, weighted by alignment scores:

$$\begin{aligned}\mathbf{c}_t &= \sum_{i=1}^{n} \alpha_{t,i} \boldsymbol{h}_i \\ \alpha_{t,i} &= \text{align}(y_t, x_i) \\ &= \frac{\exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i))}{\sum_{i'=1}^{n} \exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_{i'}))}\end{aligned}$$

The alignment model assigns a score $\alpha_{t,i}$ to the pair of input at position i and output at position t, (y_t, x_i), based on how well they match. The set of $\{\alpha_{t,i}\}$ are weights defining how much of each source hidden state should be considered for output at timestep t.

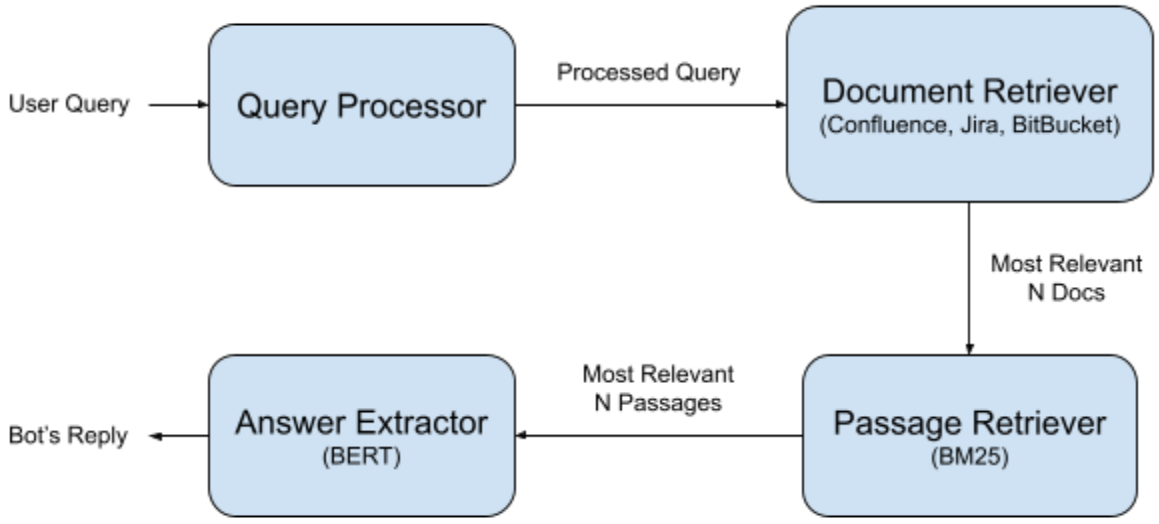
General Process

1. Producing the Encoder Hidden States - Encoder produces hidden states of each element in the input sequence
2. Calculating Alignment Scores between the previous decoder hidden state and each of the encoder's hidden states are calculated
3. Softmaxing the Alignment Scores - the alignment scores for each encoder hidden state are combined and represented in a single vector and subsequently softmaxed
4. Calculating the Context Vector - the encoder hidden states and their respective alignment scores are multiplied to form the context vector
5. Decoding the Output - the context vector is concatenated with the previous decoder output and fed into the Decoder RNN for that time step along with the previous decoder hidden state to produce a new output
6. The process (steps 2-5) repeats itself for each time step of the decoder until an token is produced or output is past the specified maximum length

Components of the Model

There are 4 components as follows:

1. Query Processor
2. Document Retriever
3. Passage Retriever
4. Answer Extractor



Query Processor

Goal: To preprocess the user query and extract important information from the query. Given a user query, this function will keep only the relevant data from which context of the question can be understood i.e. Nouns, Proper Nouns, Numerical Values, Verbs and Adjectives.

Document Retriever

Goal: To retrieve documents from the given platform - Confluence, Jira, Bitbucket or all. There are 3 separate classes in this module, one for each of the platforms.

1. Confluence Document Retriever

This class is for retrieving documents from atlassian confluence API and returning the pages in the form of list.

In the constructor, all the necessary attributes such as username, password/token, domain and projectKey are initialized. Apart from this, it initializes the path to the summary file and GPT2 model for generating summaries.

Optimization using Page Summary

Getting all the pages from each domain and each space is time consuming. Thus, we are performing specialized searches based on these attributes. Also, we are keeping track of the summaries of each page on the confluence platform in a csv file.

The idea is to update the summary file from time to time. We can just set a flag in the configuration file to update the summary csv, it will add the summaries and passages for a newly created page and also delete the data of a page which was deleted from the confluence.

This way we only need to process the page once when it is created.

Now, when users perform a search for a query, we can just use these summaries which are at max 200 words for each page. These small summaries allow faster search for the relevant docs. We can apply the BM25 (Best Matching) algorithm between the summary of each page and the user query and get the top N relevant summaries. Since we are searching on the summaries, it is much faster to extract relevant pages as compared to searching on the entire page data.

We are using the GPT2 model provided by the bert-extractive-summarizer library to generate summaries because GPT2 is the most accurate model for text generation tasks.

Functions:

validate: To validate the credentials.

search: Search the pages from an API and return passages and tables in the form of a list of strings.

- First, It will get all the pages for a user.
- If the summary of some page is not present in the summary file, then it will generate and store the summary for that page.
- Apply BM25 between the user's question (query) and summary of docs and get TOP N docs to search using API.
- Search the most relevant N pages (extracted in previous) step through API.
- Extract text and tables from these pages and convert it into a list of passages.

generate_summary: Given a page ID, generate summary of that page content using GPT-2 Model.
get_relavant_pages: Extract the most relevant N pages using BM25 between query and summary of the docs.
search_pages: Get metadata of all the pages of a user and return a list of page ids.
search_page: Given a page id, get html of that page using an API.
extract_text: To extract text data from given html data of a page.
extract_passages: To extract passages from the text data of a page.
extract_codes: To extract codes using a tag and append a CODE_PREFIX at the front of this string to recognize it later. Each code snippet on the confluence page is under the "ac:structured-macro" tag with "ac:name = code" property.
extract tables: To extract tables from given html data of a page in below format:

col1: val1, col2: val2, ... , colN: valN
Rows within a table are separated by " | ".

update_confluence_summary_db: To update the confluence summary csv - delete the summary of deleted docs and add the summary for newly created docs.

2. Jira Document Retriever

This class is for retrieving Documents from JIRA API and returning the data (objects - tasks, issues etc.) in the form of stories.

In the constructor, all the necessary attributes are initialized such as username, password, domain and projectKey, fields (to be fetched using an API call).

Jira API returns an array of objects containing many fields. We are passing a query parameter to fetch only the required fields from the object. Now, we cannot create a passage by converting this object to string directly. Since it is in unstructured format, our model cannot understand the context accurately. Also, there are around 40 fields out of which we are fetching 20 fields that are required to answer user's questions. Our model cannot keep track of long dependencies among these many fields represented in unstructured format. Thus, we have created a story (paragraph) in which we can just replace the field values from the given object. This passage is then passed through our model. Since this passage is structured with no long distance dependencies, our model can easily find an answer for the given question.

Functions:
validate: To validate the credentials.
search: Search for items (issues, tasks, stories etc.) for a given Project using Jira API.
get_total_items: Get total Number of items for a given project.
search_items: Get metadata about each item for a given project.
extract_passages: Extract required fields from the object, parse it and return a list of strings.
create_stroy: To create a story from a given object.
create_stories: To loop over each object and create a story.

3. BitBucket Retriever

This class is for retrieving Documents from BitBuckets API and returning the data (objects - commits, issues, branches, workspaces, repositories etc.) in the form of stories.

Similar to Jira API, BitBucket API returns an array of objects. To make it easy and accurate for our model to extract the correct answer from this data, we are creating a story (passage) in which we can fill in the blanks by using fields from the objects returned by an API. We are fetching all the issues and commits from a given repository.

Functions:
validate: To validate the credentials.
search: Search for items (issues, commits, workspaces, repositories etc.) for a given Project using BitBucket API.
search_issues: Search for all the issues from a given repository of a given workspace and create a story from each issue object. These stories will be returned as a list of strings.
search_commits: Search for all the commits from a given repository of a given workspace and create a story from each issue object. These stories will be returned as a list of strings.
search_all_repositories: To get all the repositories for a given workspace.
search_all_workspaces: To get all the workspaces for a given user.

Passage Retriever

To retrieve the top N passages from the given corpus.

After fetching the documents and applying all the necessary preprocessing, these documents are passed to the Passage Retriever. Here, we apply BM25 (Best Matching) between the user query and each passage of each document in order to get the most relevant N passages from the data.

Optimization using BM25

Since the BERT (or any other) Answer Extractor Model contains a lot of parameters, passing each passage from the answer extractor is a time consuming process. Instead, we can get the top N passages that are most relevant to the user's query and pass only these N passages from the answer extractor model. This is the same as using the summaries of pages to get the top N relevant pages.

Each of these retrieved passages will be given as a context along with the user query into the AnswerExtractor Model.

Retrieval Algorithm: BM25 (Best Matching)

Given a query Q containing keywords q1, q2, ... , qn, the BM25 score of a document D is given by:

score(D, Q) : Total_IDF * [{ f(qi, D) * (k1 + 1) } / { f(qi, D) + k1 * (1 - b + b * (|D| / avgDL) }])

Where,

Total_IDF = SUM (FOR EACH TERM IN THE QUERY => IDF(qi),

IDF(qi) = Inverse document frequency weight of the query term qi,

f(qi, D) = qi's term frequency in document D,

|D| = length of the document D in words,

avgDL = average document length in text collection

k1 and b are free parameters : In absence of advanced optimization => k1 : [1.2, 2] and b : 0.75

IDF(qi) = ln(({ N - n(qi) + 0.5 } / { n(qi) + 0.5 }) + 1)

where, N = total number of docs in the collection,

n(qi) = number of docs containing qi.

Answer Extractor

Goal: To get an answer for a user's query.

Each of the top N passages returned by Passage Retriever is passed through the Answer Extractor Model along with the question (query) of the user.

In the constructor, we are initializing the tokenizer and model as per the MODEL_PATH argument in the configuration file. Then, we pass this tokenizer and model to the QuestionAnsweringPipeline from Huggingface Transformers.

Now, we can pass a tuple (passage, question) for each passage and get the extracted answer and the score - confidence of our model. At the end, we sort these answers by decreasing the order of their confidence and give the answer with the highest confidence as an output to the user.

Model API

To avoid loading the model for each query of the user, we are creating the instances of Query Processor, Passage Retriever and Answer Extractor only once when the model-api.py file is executed first time to start the server.

We are using FastAPI because it is the most efficient, fast (high performance) web framework to build APIs with python (significant improvement in the response time as compared to Flask).

We have only one route "/query". It takes 6 required arguments: username, password/token, user's query (question), platform to retrieve the documents from, domain and projectKey (in case of Confluence and Jira) or workspace and repository (in case of BitBucket).

Now, we create an instance of Document Retriever based on the search platform selected by the user, get the relevant documents from that platform, get relevant passages from the fetched documents and get the most appropriate answer by passing these passages to the Answer Extractor Model. The final response of the API contains two fields: answer and error - one of these fields will be Null depending on the successful execution.

BERT (Bi-directional Encoder Representations from Transformers)

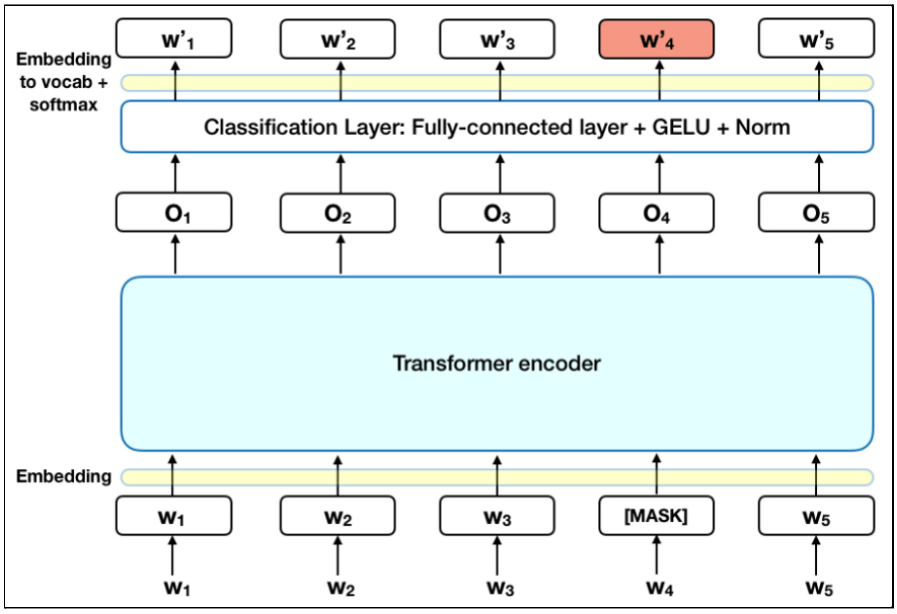
BERT (Bidirectional Encoder Representations from Transformers) is a model developed by researchers at Google AI Language. It has caused a stir in the Machine Learning community by presenting state-of-the-art results.

BERT's key technical innovation is applying the bidirectional training of Transformer, a popular attention model, to language modelling. This is in contrast to previous efforts which looked at a text sequence either from left to right or combined left-to-right and right-to-left training. The paper's results show that a language model which is bidirectionally trained can have a deeper sense of language context and flow than single-direction language models.

Masked LM (MLM)

Before feeding word sequences into BERT, 15% of the words in each sequence are replaced with a [MASK] token. The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence. In technical terms, the prediction of the output words requires:

1. Adding a classification layer on top of the encoder output.
2. Multiplying the output vectors by the embedding matrix, transforming them into the vocabulary dimension.
3. Calculating the probability of each word in the vocabulary with softmax.



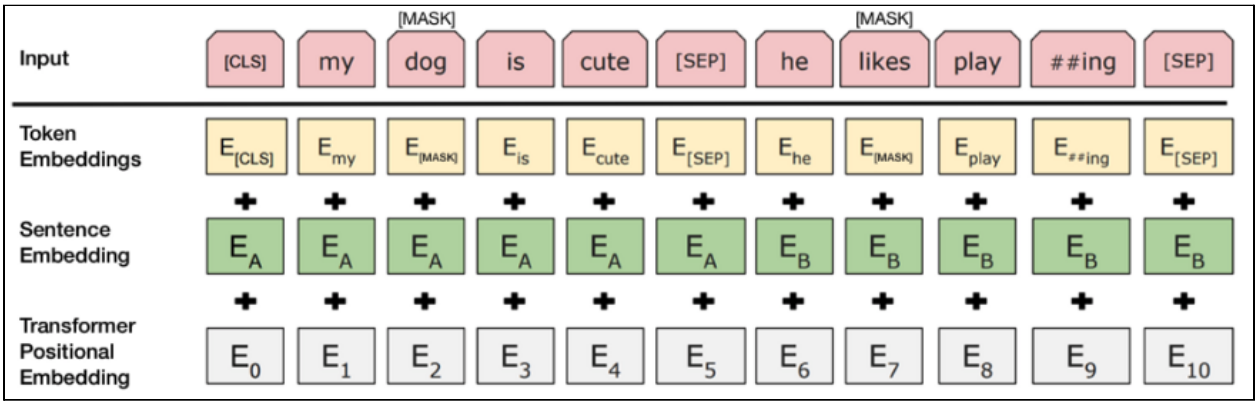
The BERT loss function takes into consideration only the prediction of the masked values and ignores the prediction of the non-masked words. As a consequence, the model converges slower than directional models, a characteristic which is offset by its increased context awareness.

Next Sentence Prediction (NSP)

In the BERT training process, the model receives pairs of sentences as input and learns to predict if the second sentence in the pair is the subsequent sentence in the original document. During training, 50% of the inputs are a pair in which the second sentence is the subsequent sentence in the original document, while in the other 50% a random sentence from the corpus is chosen as the second sentence. The assumption is that the random sentence will be disconnected from the first sentence.

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

1. A [CLS] token is inserted at the beginning of the first sentence and a [SEP] token is inserted at the end of each sentence.
2. A sentence embedding indicating Sentence A or Sentence B is added to each token. Sentence embeddings are similar in concept to token embeddings with a vocabulary of 2.
3. A positional embedding is added to each token to indicate its position in the sequence. The concept and implementation of positional embedding are presented in the Transformer paper.



To predict if the second sentence is indeed connected to the first, the following steps are performed:

1. The entire input sequence goes through the Transformer model.
2. The output of the [CLS] token is transformed into a 2×1 shaped vector, using a simple classification layer (learned matrices of weights and biases).
3. Calculating the probability of IsNextSequence with softmax.

When training the BERT model, Masked LM and Next Sentence Prediction are trained together, with the goal of minimizing the combined loss function of the two strategies.

Fine Tuning BERT Model

We have taken a pre-trained BERT model from Huggingface Transformers and fine tuned this model on SQUAD v2.0 dataset. The version of BERT that we have used is "bert-large-uncased-whole-word-masking". This model has 24 layers, 1024 hidden dimensions (units) in RNN, 16 attention heads and a total of **336M** parameters.

Stanford **Qu**estion **A**nswering **D**ataset (SQuAD) is a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text, or *span*, from the corresponding reading passage, or the question might be unanswerable.
SQuAD2.0 combines the 100,000 questions in SQuAD1.1 with over 50,000 unanswerable questions written adversarially by crowdworkers to look similar to answerable ones.

We are getting **85% f1-score for SQUAD v2.0** and **91% f1-score on SQUAD v1.0** dataset by fine tuning this model. It takes on average 8-10 seconds to answer a user query. We can improve this response time by reducing the number of layers, hidden dimensions and attention heads in the BERT model but it can reduce the accuracy of our model. Thus, we have chosen the best possible parameters to achieve reasonable accuracy as well as response time.

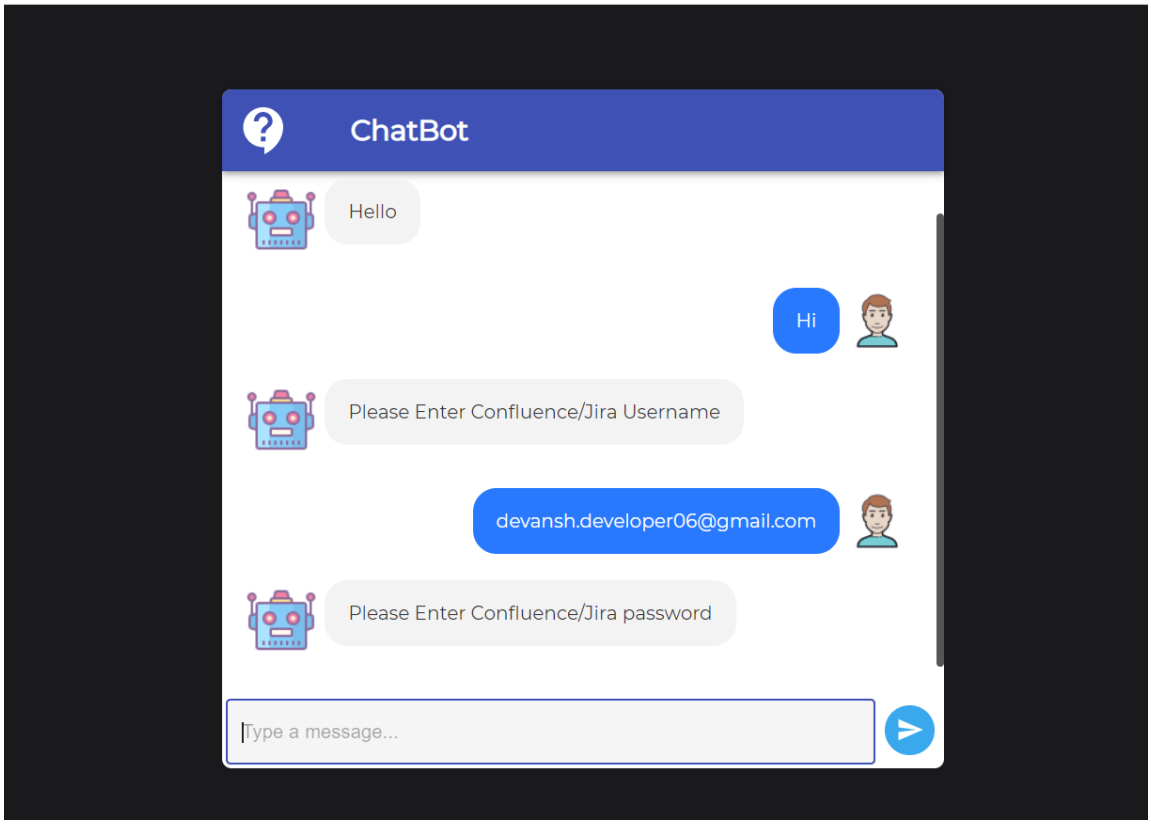
Frontend of Chatbot

The frontend of chatbot is built using react. We have used material-ui for UI.

Steps to run the frontend project:

1. Run the command **npm i** in the root directory (directory containing the package.json file) to install necessary dependencies for react app.
2. After installation of node modules is completed run **npm start** and the project will run at port 3000.

UI-Image:



Technical Details of react app:

Components:

Chat: This component is the main component of our app as it renders all the components like infobar,messages and input. It fetches answers from the backend for a question.

Important States of Chat component:

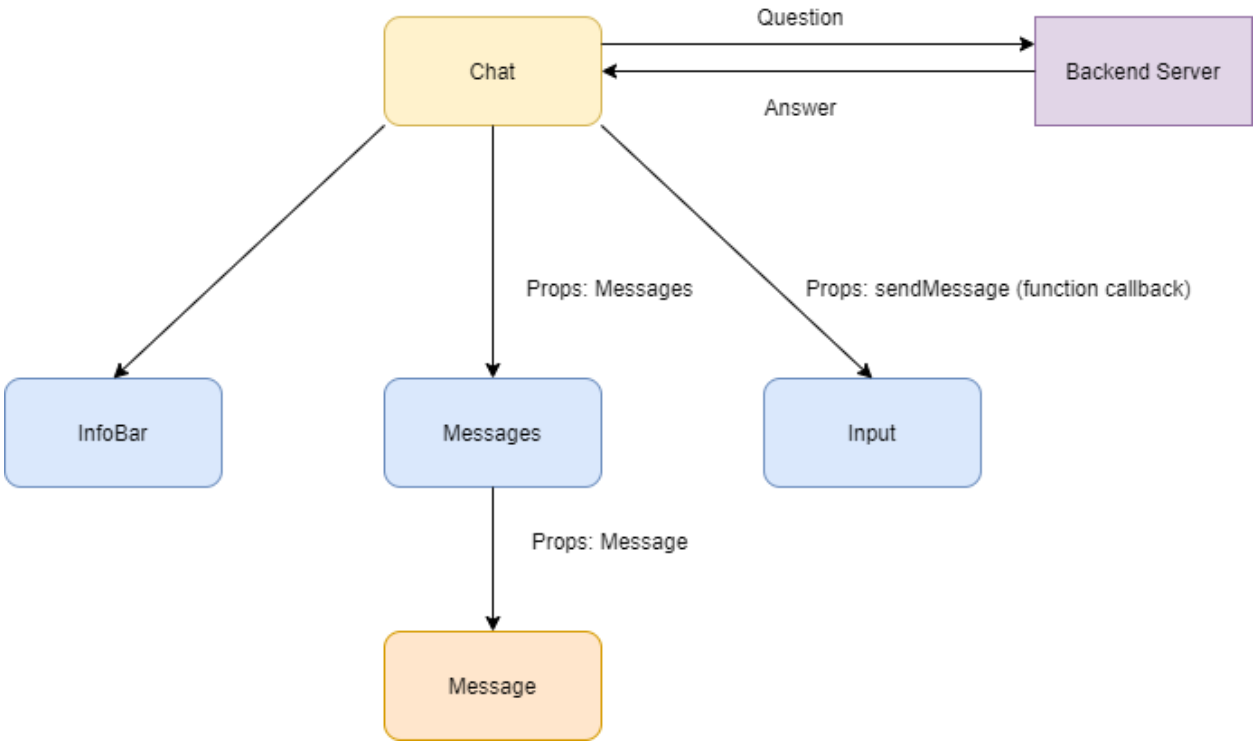
Messages - It is an array of all messages.
SearchType - It is the type of search user wants to make.
Domain - Domain of Confluence,Jira or BitBucket
Project - Project key of project

InfoBar: It contains the UI for the header of our app.

Messages: It renders all the messages which it receives as the props. It makes use of Message for rendering individual messages.

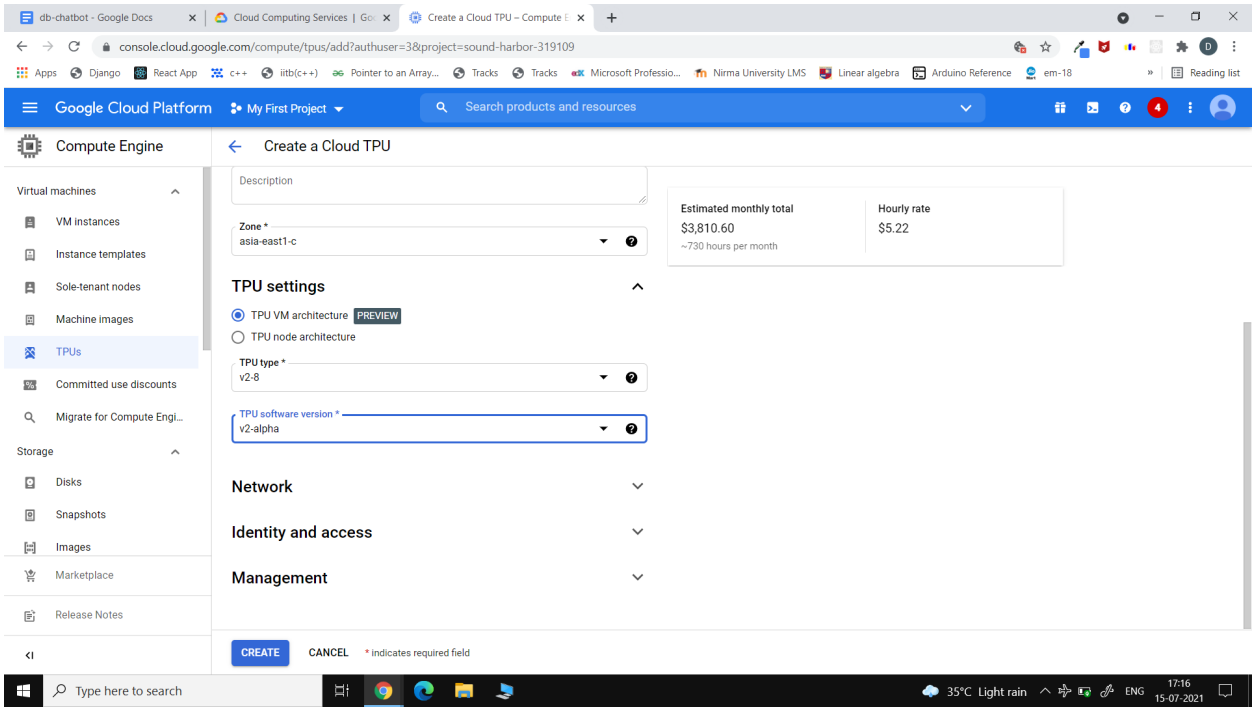
Message: It renders the message according to the sender. If the message is sent by the user the message is rendered on right and if sent by bot it is rendered on left.

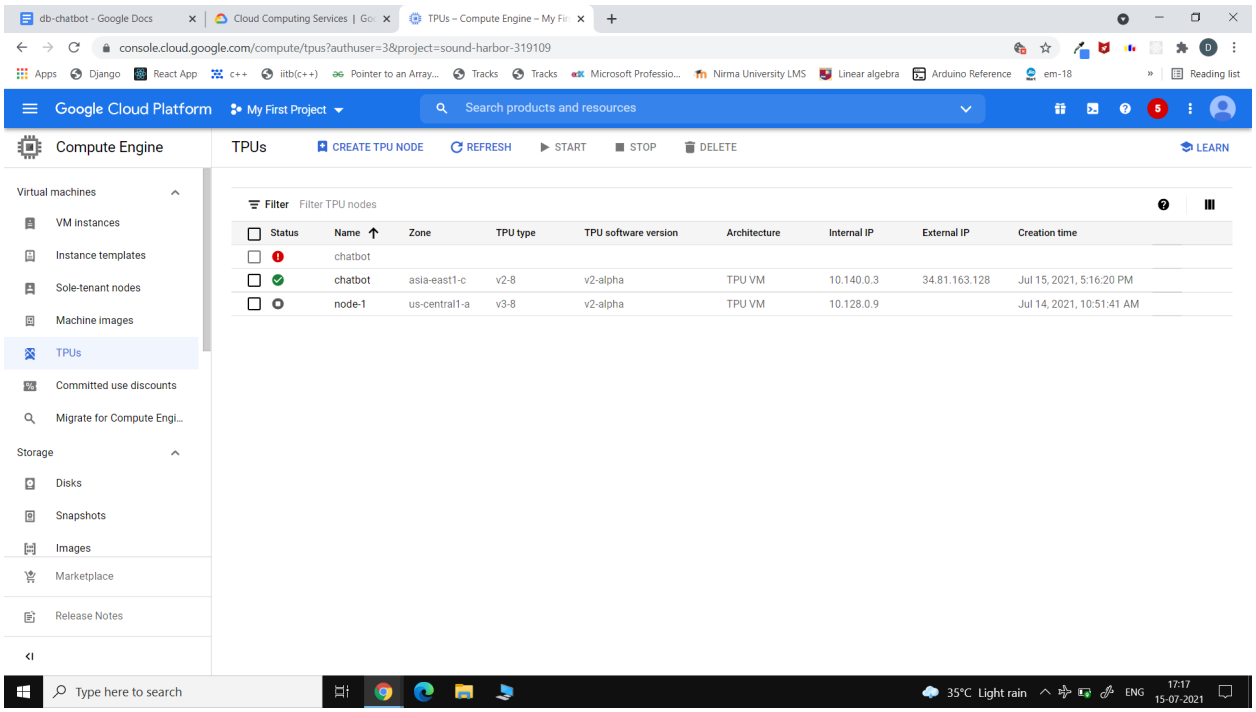
Component Diagram:



Deployment on GCP

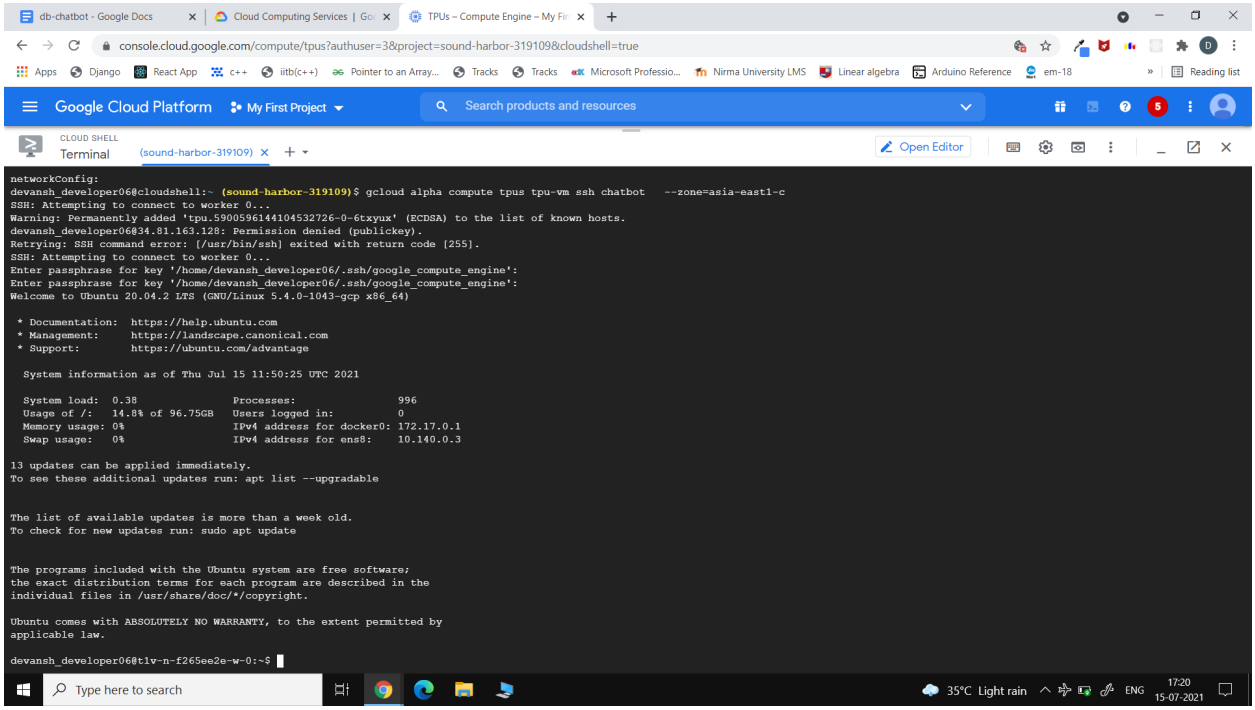
For deploying our BERT model and frontend we used google cloud's TPU-VM instance.





Then we connected to our instance with ssh using google cloud shell. For that we used following command:

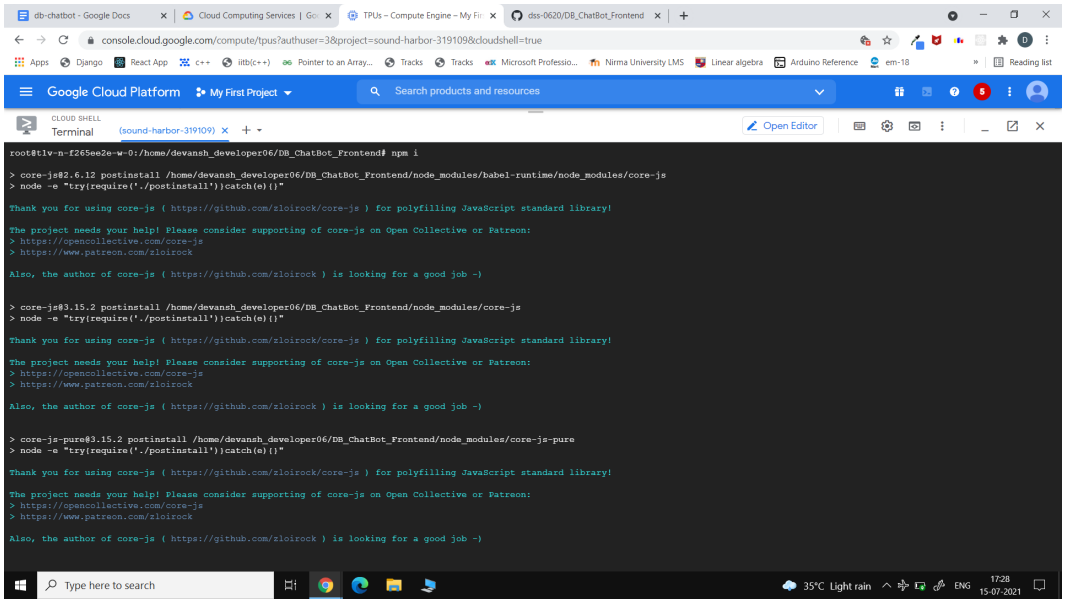
gcloud alpha compute tpus tpu-vm ssh chatbot --zone=asia-east1-c



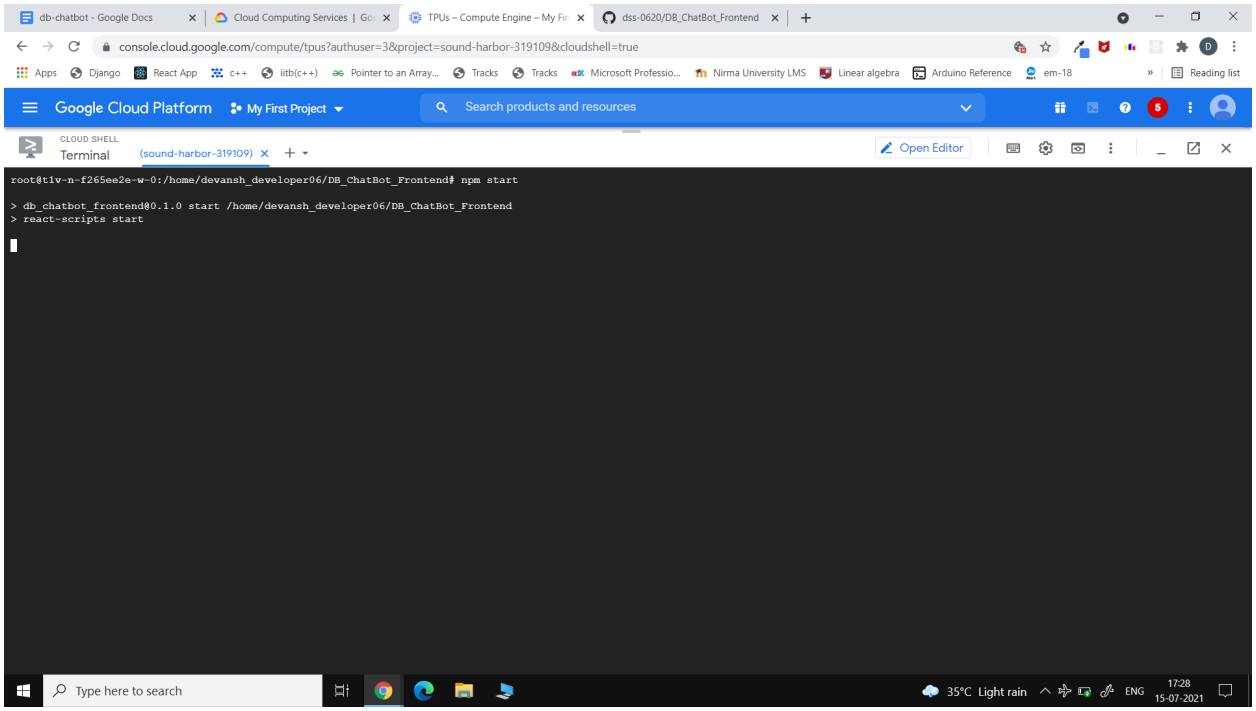
Then we cloned our code of frontend and backend from our github repository.

Frontend Deployment

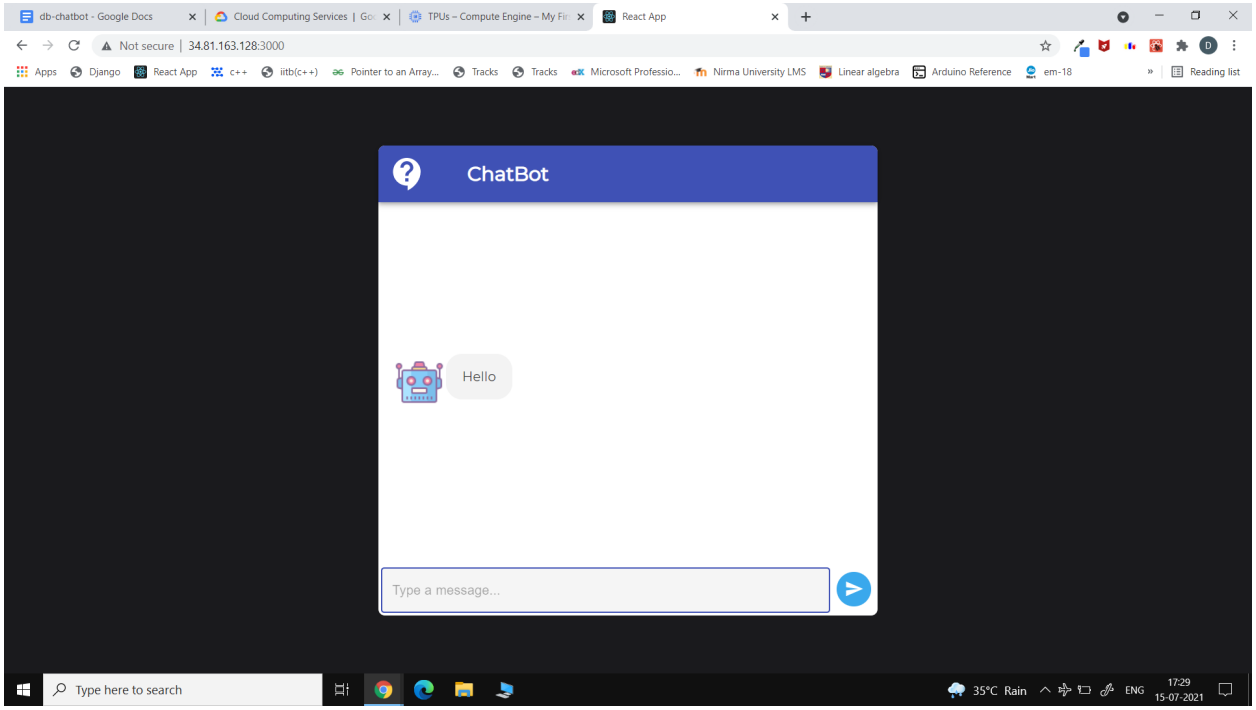
We installed React's necessary dependencies.



After installing it we started server using npm start



Hence now our app is deployed at http://public_ip:3000/



For production purposes we used pm2 with express to deploy production build.

Installation Guide

You can refer to this Link to BitBucket for the Installation Guide and Code for Chatbot Model and UI.