Technical Specification

Project Title:	Infer-Read
Student One:	Ryan McQuillan 19452414
Student Two:	Ethan Hall 19436514
Supervisor:	Dr Jennifer Foster
Start Date:	09/04/2023
Completion Date:	06/05/2023

Abstract

In recent years, the internet has revolutionized the way people learn languages. With the availability of online resources, language learning has become more accessible, convenient, and flexible. As a result, individuals can now learn a new language from the comfort of their own homes. However, despite this, there is a lack of online reading tools for moderate to advanced language learners in this space. Ideally, there would exist accessible tools tailored to the needs of intermediate and advanced language learners.

This project's objective is to further assist intermediate language learners by using language prediction to identify words and phrases within an uploaded text. The system highlights previously unseen and unknown words for the learner to engage and disambiguate, promoting learning via inference. Each learner will have a unique word bank, initially established through a simple examination, and updated during each reading session. This word bank is used for future reading sessions but also to provide statistics and to show the user their achievements in their usage.

Table of Contents

1. Introduction

1.1 Overview

This project makes use of language prediction to identify words and phrases within an incoming, user-uploaded text to provide assistance in language learning. The web app uses a user profile to store the learner's known words and performs frequency analysis to rank a text's difficulty in comparison to the learner's word bank. It highlights previously unseen and unknown words for the learner to engage and try to disambiguate.

The goal is for learners to learn via inference, similar to how one learns a language as a native. By fetching synonyms and other instances of a word's usage, the learner can pick up an intuition of meaning via context before providing an explicit translation.
Each learner has a unique word bank, which is initially tested by examining their awareness of common words and phrases. During each "reading session," the backend is updated through a POST, updating the user's word bank if they declare a novel word as known or indicate difficulty with another word.

1.2 Motivation

As two students with a passion for language learning, we found ourselves struggling to find an application that was a convenient and accessible way to read in a non-native language. We are both intermediate language learners of Irish and French individually, additionally avid readers. We wanted an application that allowed us to improve our language learning through reading. After conducting research, we found that most language learning applications on the market are beginner-centric and focus on introducing basic vocabulary and grammar. We believed that there was a gap in the market for an application that catered to intermediate to advanced learners who desired a more diverse and engaging reading experience. This led us to develop our own application, which we believe can fill this gap and provide a helpful tool for learners.

1.3 Business Context

We believe there are several scenarios where this project could be utilized in a business context. We believe that translators could make use of the application. A translator could potentially fasten their workflow using the application as an assistant. Language schools could provide the application to their students.

Students could then study novels and literature at an earlier stage, enabling language immersion.
Companies such as Babbel or Duolingo could integrate the Infer-Read Language Learning tool within their respective applications, replacing the existing reading functionality, which as of 2023, is human-curated and in short supply.

1.4 Glossary

API	Application programming interface
HTML	HyperText Markup Language
SCSS	Sassy CSS, a preprocessor scripting language that is used to write stylesheets for web applications. It is a more powerful version of CSS that includes features like variables, nesting, and functions.
OCR	Optical Character Recognition
SPA	Single Page Application
GET	A HTTP method used to request data from a server. It retrieves a representation of a resource without modifying it. It is commonly used to read or fetch data from a server.
PUT	A HTTP method used to update an entire resource on a server. It replaces the existing resource with the new one provided in the request. It is commonly used to update an entire record, such as a user profile or blog post.
POST	A HTTP method used to submit data to a server to create or update a resource. It is commonly used to create new resources, such as a new user account or blog post.
PATCH	A HTTP method used to update a portion of an existing resource on a server. It is commonly used to make minor modifications to a resource, such as updating a user's password or email address.

NLP	Natural language processing
Angular / NG	A popular open-source framework for building web applications. It uses TypeScript and provides a structure for organizing code and building user interfaces.
BERT	Bidirectional Encoders Representations from Transformers - BERT is trained on a large corpus of text by randomly masking certain words in a sentence and then predicting them based on the context provided by the surrounding words.
SpaCy	Open-source Python library for natural language processing (NLP) tasks using deep learning.

2. Research

2.1 Machine Learning Model

Utilizing and training machine learning models was undoubtedly the most crucial aspect of our project. Neither of us had much experience utilizing machine learning models. Consequently, we invested a substantial amount of time researching the best practices and approaches for creating an effective model that met our project's requirements.

2.1.1 Irish Model

Irish modelling was achieved using a pre-trained BERT model for Irish produced by DCU insight. [1] Which was then further trained with the wikipedia dataset, which contains cleaned articles from the Irish part of the encyclopedia. Torch was then used for the manipulation of incoming data to be used in the prediction of viable synonyms.

1: Barry, J. et al. (2022) 'gaBERT --- an Irish Language Model', in Proceedings of the Thirteenth Language Resources and Evaluation Conference. Marseille, France: European Language Resources Association, pp. 4774–4788. Available at: https://aclanthology.org/2022.lrec-1.511.

2.1.2 French Model

French modelling was achieved with the help of the pre-trained CamemBERT model - implemented using TensorFlow. This involved familiarising myself with both the BERT architecture, masked language model and the TensorFlow ML library.
SpaCy was used originally for word vector similarity but later used for parts of speech tagging and morphological analysis of words using the medium-sized model trained on French news.

2.2 Angular

To design our web application, we circled a number of frontend frameworks for various languages. We decided on Angular as one of the devs had considerable experience from INTRA, in addition we decided to use FontAwesome & DevExtreme for icons and complex components respectively. Specifically, we used DevExtreme for complex components that were low priority to speed up development. DevExtreme has free licensing for non-commercial usage, which we believe this project falls under.

Angular is written in TypeScript, a superset of JavaScript. Angular's component-based architecture allowed us to break code into reusable components with TS source code, HTML templates, stylesheets, and unit tests. This allowed us to easily break code into atoms of functionality, eliminating the majority of copied code. We found Angular encouraged code reusability, maintainability, and testability, as each component can be developed and tested in isolation.

2.3 FE Design Heuristics

Design heuristics are guidelines that help developers create user interfaces that are intuitive, easy to use, and aesthetically pleasing. In our project, we focused on three key heuristics:

- Visibility of status
- Consistency & standards
- User control.

Visibility of status ensures that users are always aware of what is happening, which describes giving clear feedback and indicators to let users know that their actions are being processed, e.g. loading bars. The idea is that informed users are not confused or frustrated.

Consistency and standards promote familiarity and predictability. By copying established design patterns and conventions, we can create an interface with minimal cognitive stress.

User control empowers users to explore and interact with the application in a way that makes sense to them. This means providing clear navigation and menu options, allowing users to undo and redo actions.

2.4 Database Design

We opted toward a NoSQL database for the project; utilising MongoDB. Both a SQL and NoSQL database design were suitable solutions for the given project, but MongoDB was chosen on the basis of it being unfamiliar to the developers and giving an added dimension to the challenge of the project.

The data model is user-centric; each sub-model is embedded into the User model. MongoDB is designed to model relationships between data via embeddings as opposed to the SQL pattern of normalised, referential relationships. This provides an advantage in that if we frequently need one data entity in the context of another, we don’t need to resolve the reference with multiple queries but instead can capture the data in a single query. There are quite a few data models in Infer-Read which are encapsulated in embeddings.

Almost any operation in the application involves the handling of one of these models, and each model is almost inextricably bound to each other; a user has a word bank (vocabulary) and a collection of documents, a document belongs to a user and also possesses its own vocabulary.

2.4.1 The User Model

The User model can essentially be viewed as the user's collection of documents and their word bank (if we abstract away implementation details like id, email, password and userconfig etc.) The user model has a one to many relationship with the Document sub-model, where the latter is embedded within the User model.

The model also has a 1 to 1 relationship with the Word Bank model, where a user's known words and words they're currently learning are stored.

2.4.2 The Document Model

This model encapsulates the documents the user chooses to upload; the title, language and textual content of the document segmented into pages.

2.4.3 The Word Bank Model

The Word Bank model divides its data (words) into two sections: known and learning. Known words are simple strings, while words that are being learned are stored with more data about the word and its semantic relationships. These sections are also one to one, and embedded within the Word Bank.

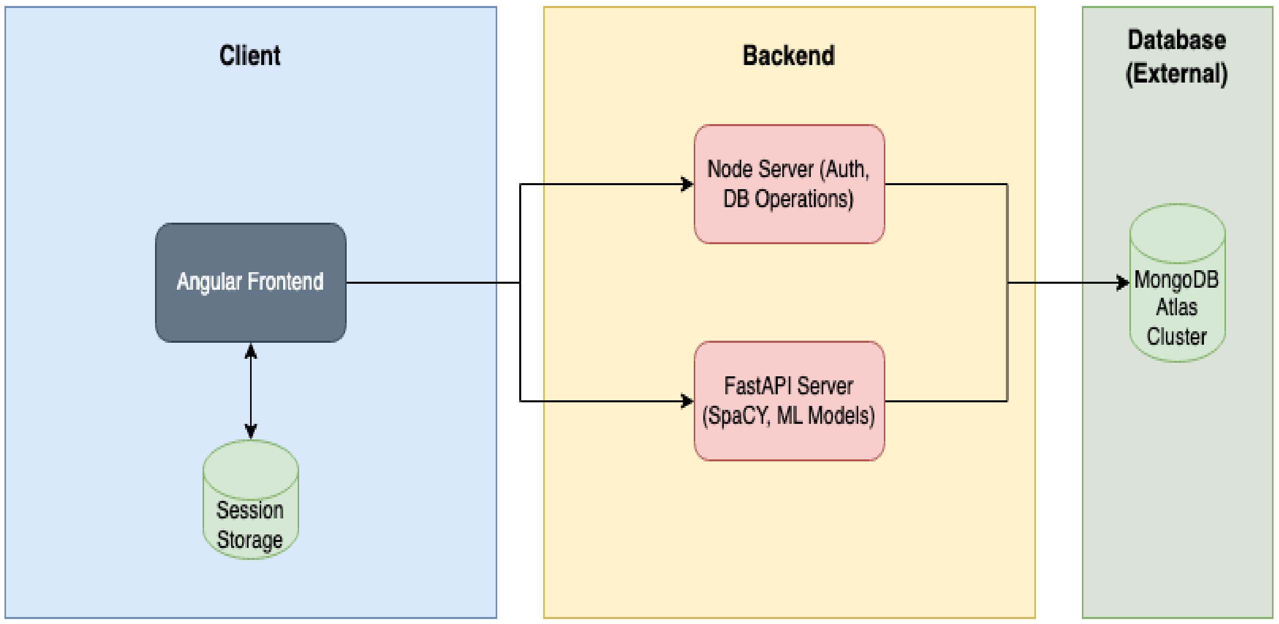
2.4.4 The Word Model

A word can also be thought about as a model; either atomic in the case of known words or having more features; part of speech, plurality, tense etc. A word has a many to one relationship with each of the other models.

2.5 Testing

To ensure the quality and reliability of our project, we dedicated time before development to research how we execute our testing. Using Angular, we were given a native method of unit testing the frontend, via Jasmine. Our testing plan included a variety of testing methods, such as unit tests, integration tests, and acceptance tests, to ensure that our software performed as expected under various scenarios and conditions. We also established a set of testing standards and guidelines to ensure consistency and accuracy across all testing activities.

3. Design



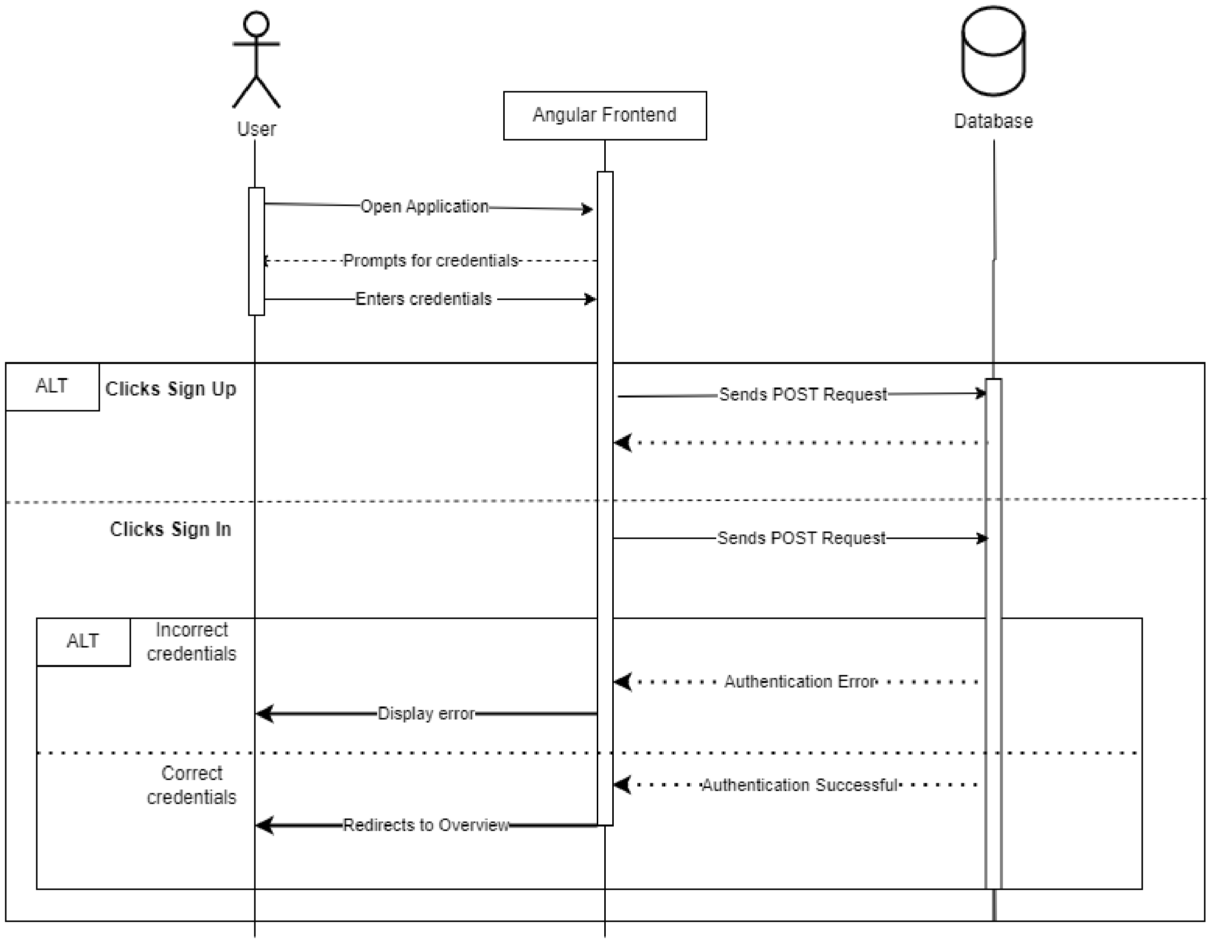
3.1 System Architecture

The system architecture shows at a high level how the system is built. Users input and interact with the system on the client side through the Angular application. The Angular frontend serves as a client and communicates with the FastAPI API. The client will make requests to the API, which will then interact with the MongoDB database to retrieve or store data. Our MongoDB is where user credentials are stored, on log in & sign up the frontend communicates with the API to retrieve a token for the session when authenticated. MongoDB handles the encryption of the passwords.

Any document uploaded is sent through the backend where it is processed into a format that is desirable for having a language model used on it. When a user begins a reading session on the frontend, the document is retrieved via API request. Further, when the previous or next page is requested that is another request to the API. Whenever a word or sentence is highlighted, the word & its context are sent to the API which then utilizes the machine learning models for the appropriate language. Upon completion of a session, a request updating the users word bank is sent, tracking knowledge of previously highlighted words and sentences.

3.2 Components

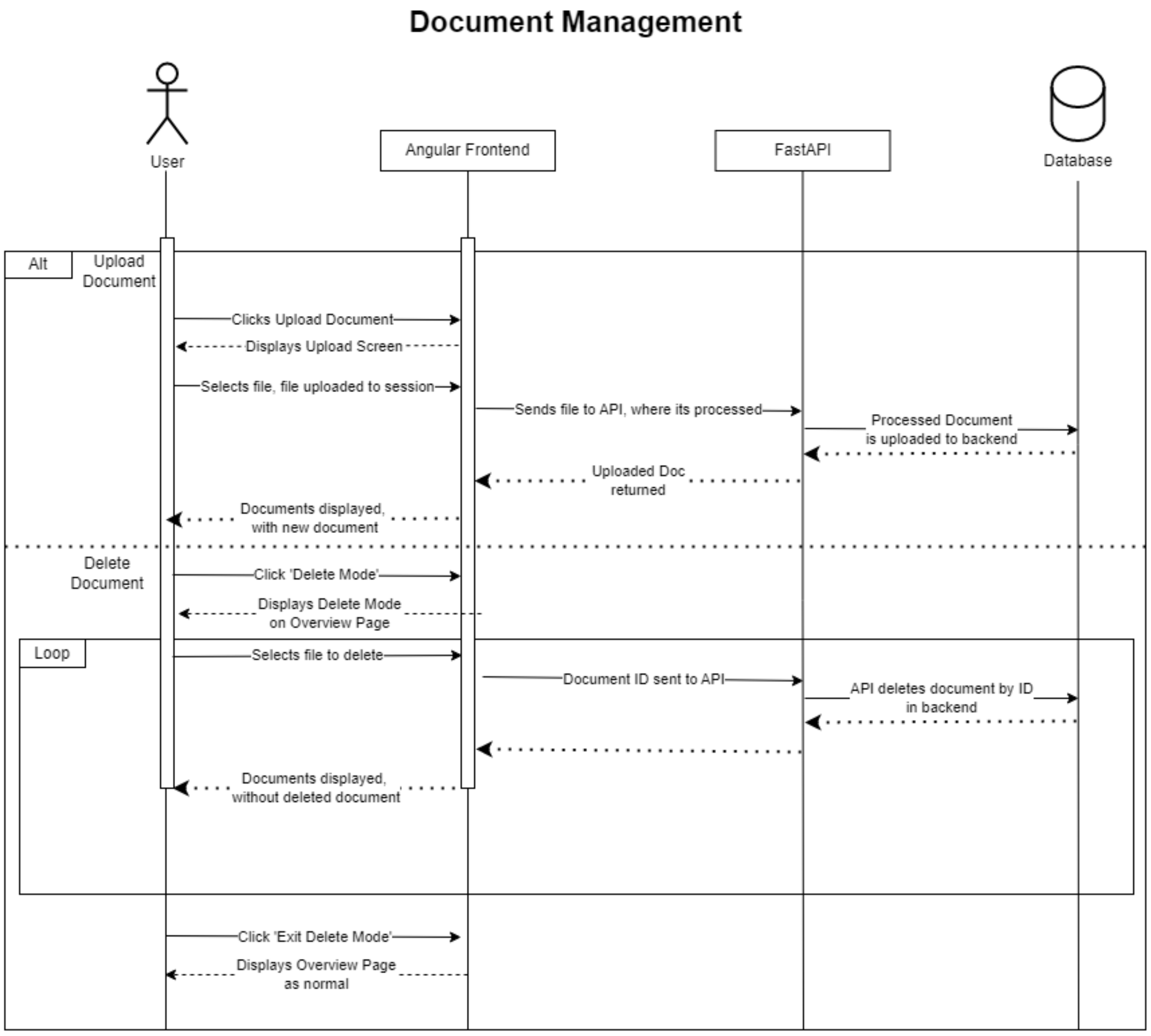
3.2.1 User Sign In and Authentication



The sequence diagram above shows what happens when a user tries to sign in. When a user initially lands on the website, they will be presented with a login page, where they can enter their login credentials. If the user does not have an existing account, they should click the ‘Sign Up’ button, which will trigger a process that creates a new account and stores the user's details in the database via Mongoose.

If the user has already created an account, they should sign in by clicking the ‘Sign In’ button. This will initiate a POST request to the server, which will query the database for the user's credentials. If the entered credentials are incorrect, the server will respond with an authentication error, which will be displayed to the user by the Angular frontend. If the entered credentials are correct, the application will redirect the user to the overview page, allowing them to access the features and functionality of the system. It is important to note that the security and confidentiality of user data is of utmost importance, and appropriate measures such as encryption and secure storage must be implemented to safeguard user information.

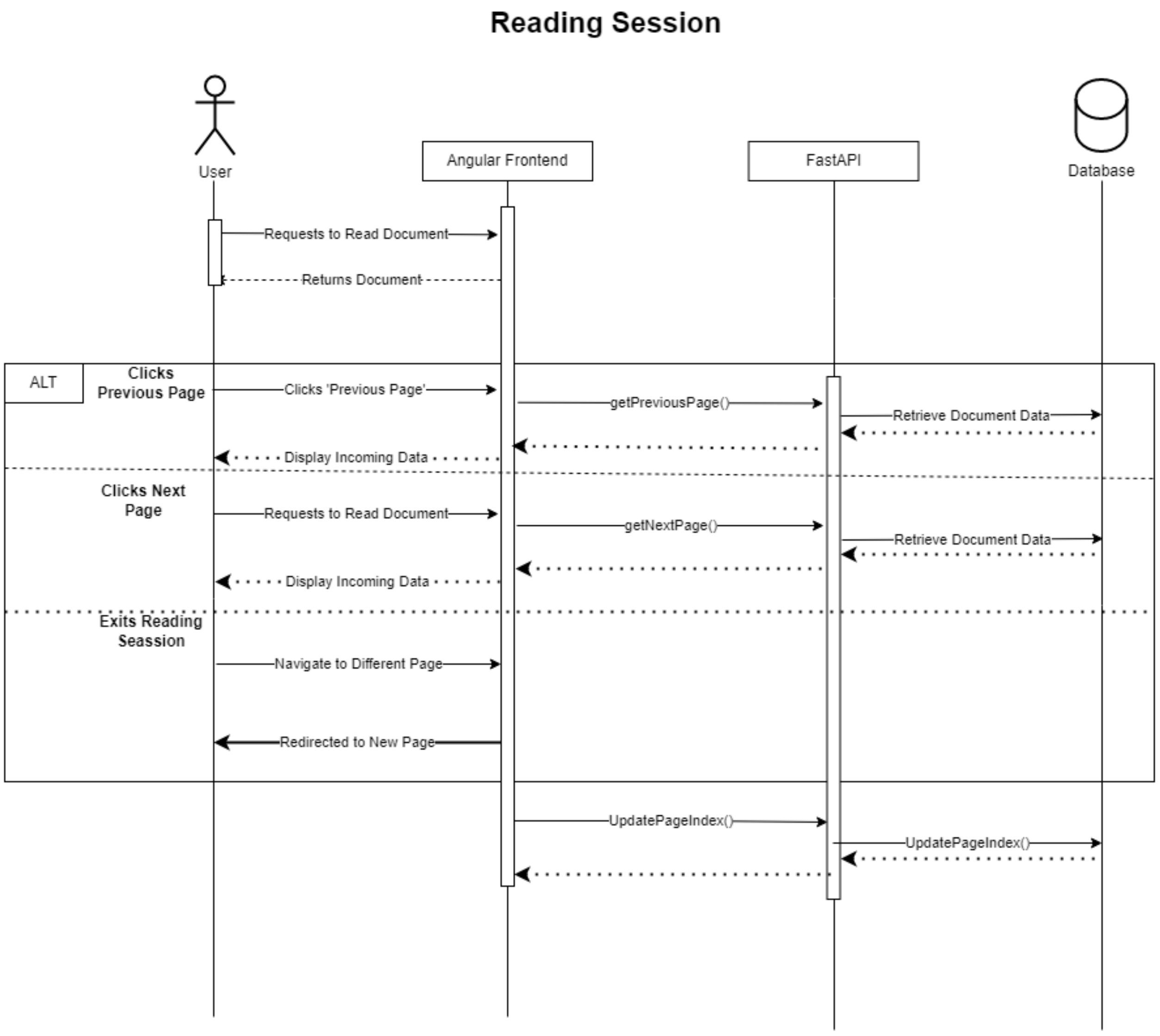
3.2.3 Document Upload


Document Management
User
Angular Frontend
FastAPI
Database
Alt
Upload Document
Clicks Upload Document
Displays Upload Screen
Selects file, file uploaded to session
Sends file to API, where its processed
Processed Document is uploaded to backend
Uploaded Doc returned
Documents displayed, with new document
Delete Document
Click 'Delete Mode'
Displays Delete Mode on Overview Page
Loop
Selects file to delete
Document ID sent to API
API deletes document by ID in backend
Documents displayed, without deleted document
Click 'Exit Delete Mode'
Displays Overview Page as normal

The sequence diagram above shows the communication between user, frontend, api and backend during the process of document manipulation.The two options are to either upload documents, or delete documents.
First is Uploading, the user starts on overview and has to click the upload button which is located on the floating action button. This brings up the ‘Upload Document’ popover, the user clicks a document from their system. This file is stored on the local session on the frontend, which is then sent to the API. The API performs preprocessing on the document. After this, the document is sent to the backend to be stored, response is returned to API, which then instructs the frontend to retrieve the updated document data.

The other option is document deletion. The user clicks ‘Deletion Mode’, the frontend then displays deletion mode, where delete icons are displayed beside each document. Then we enter a loop where the following actions are repeated 1+ times. A User clicks a documents associated delete icon, the frontend then tells the API to delete the document by sending the documentID. The API then deletes the document in the backend. Response is sent, API tells frontend if it succeeded. If successful, the frontend retrieves the updated documents.

3.2.3 Starting and Finishing a Reading Session



The sequence diagram above shows what happens when a user initiates a reading session. It depicts the communication between the User, Frontend, FastAPI and backend over the course of a reading session from start to finish. When a user clicks on a document from the overview, the frontend routes to the ‘Reading-Page’ route. As the reading session begins, the frontend requests the document data using the 'readingIndex' variable to specify which pageThe document data is then displayed for the user on the frontend.

This process is repeated when the user navigates to either the previous or next page. The 'readingIndex' variable is used to retrieve the data for the specific page requested. Once the reading session is completed, the current value of the 'readingIndex' is sent in a PATCH request to the specific document on the backend. In addition, the user's wordBank is updated accordingly with words encountered and progress made.

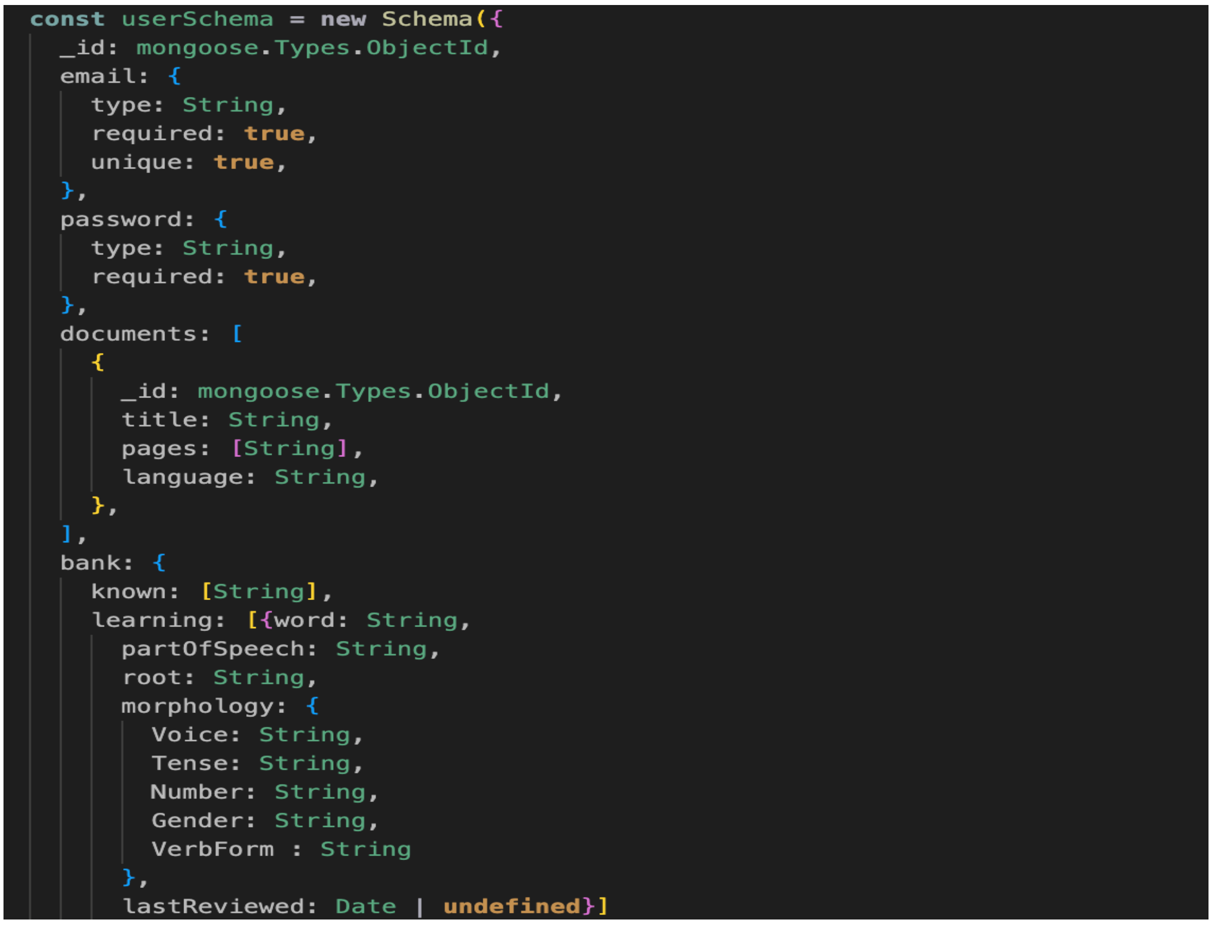
4. Implementation

4.1 Authentication and Authorization

Authentication and authorization is implemented through the ‘jsonwebtoken’ and ‘bcryptjs’ libraries. Upon sign-up, the user’s password is hashed using ‘bcrypt’ and compared for authentication on log-in. Successful authentication gives a signed ‘jsonwebtoken’ for user authorization which is verified against the secret on endpoints for HTTP operations like POST and DELETE.

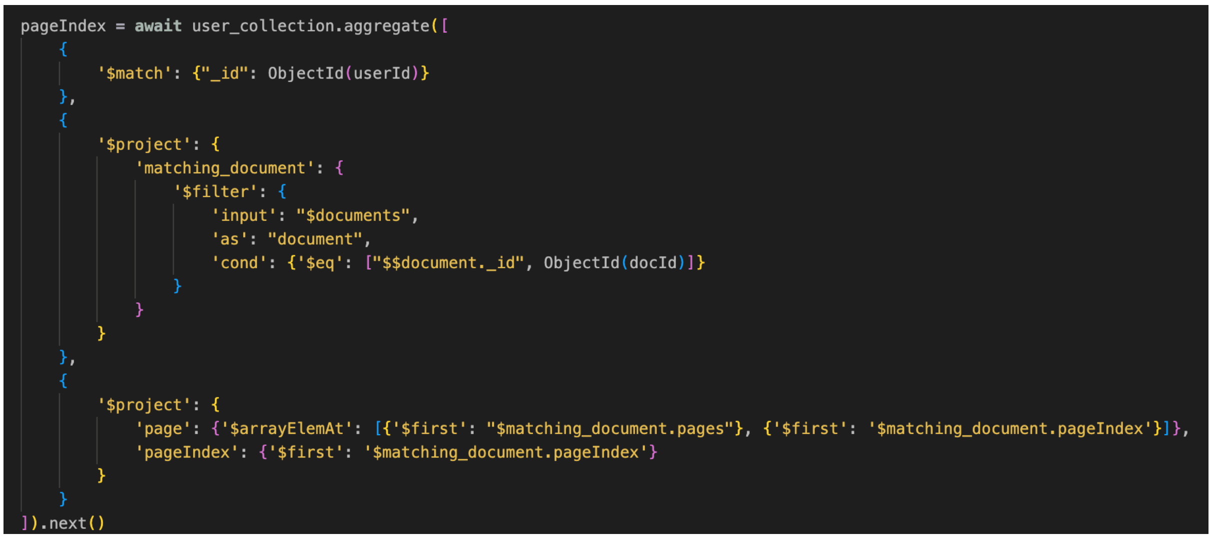
4.2 Database Implementation

Infer-Read uses a MongoDB Atlas cluster for storing data. Data models are defined in a schema using Mongoose on the Node.js server according to what was described in the database design.



```
const userSchema = new Schema({
  _id: mongoose.Types.ObjectId,
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  documents: [
    {
      _id: mongoose.Types.ObjectId,
      title: String,
      pages: [String],
      language: String,
    },
  ],
  bank: {
    known: [String],
    learning: [{word: String,
      partOfSpeech: String,
      root: String,
      morphology: {
        Voice: String,
        Tense: String,
        Number: String,
        Gender: String,
        VerbForm : String
      },
      lastReviewed: Date | undefined}]
```

CRUD operations are handled using Mongoose on the Node.js server and via the motor async-io wrapper for Pymongo on FastAPI. Due to the schema and embedding-based nature of MongoDB, manipulating data across the schema requires some relatively complex but quick aggregation pipeline queries to the database.



```
pageIndex = await user_collection.aggregate([
    {
        '$match': {"_id": ObjectId(userId)}
    },
    {
        '$project': {
            'matching_document': {
                '$filter': {
                    'input': "$documents",
                    'as': "document",
                    'cond': {'$eq': ["$$document._id", ObjectId(docId)]}
                }
            }
        }
    },
    {
        '$project': {
            'page': {'$arrayElemAt': [{'$first': "$matching_document.pages"}, {'$first': '$matching_document.pageIndex'}]},
            'pageIndex': {'$first': '$matching_document.pageIndex'}
        }
    }
]).next()
```

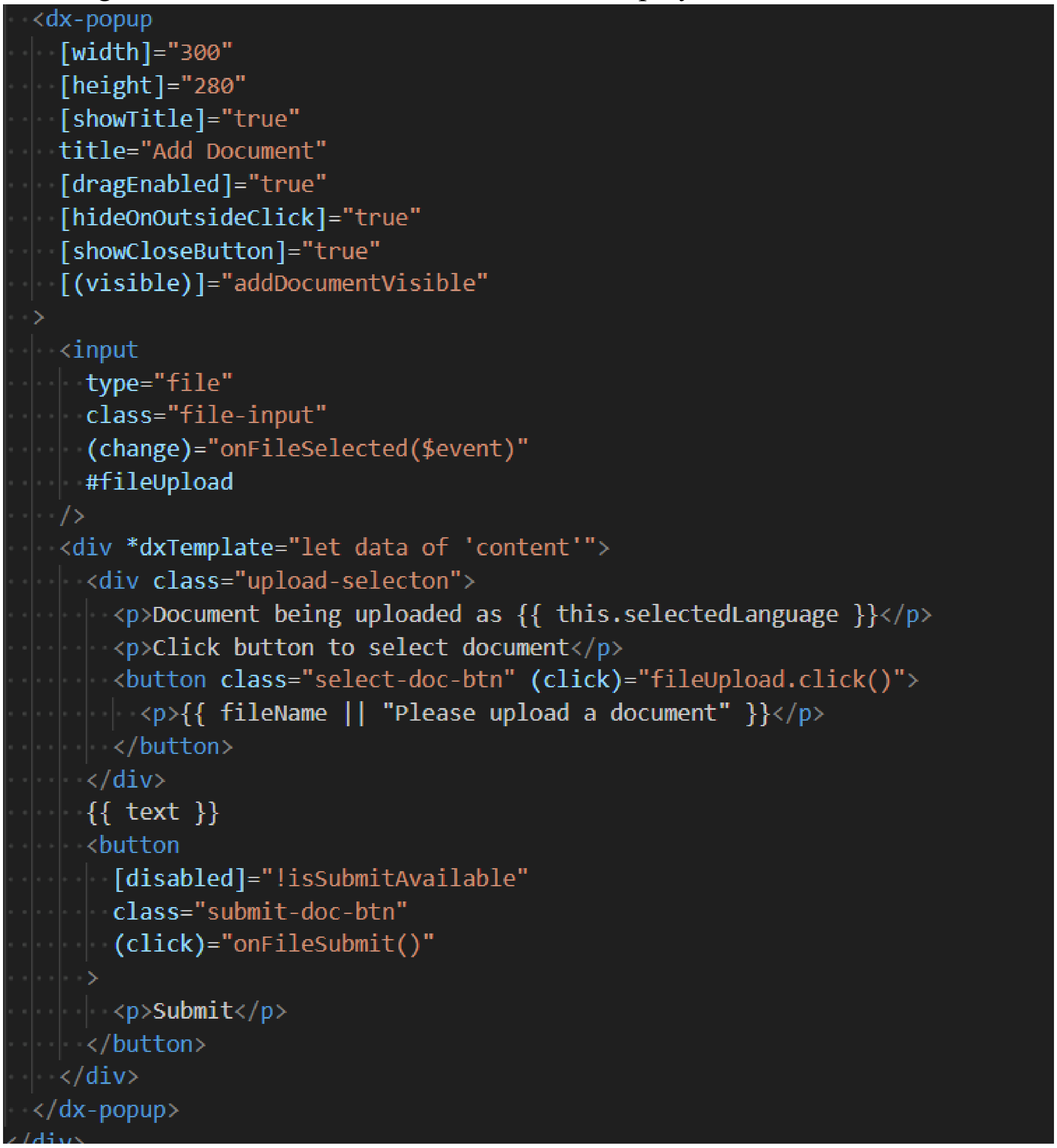
For example, retrieving the current page and pageIndex from the database for a given book.

4.3 Document Management

After a user has signed in, they will be able to access the overview screen. Within this screen is the functionality of document management, namely, deletion and uploading of documents. The user accesses both via the floating action button on the screen. The screen has two options, Add Document' and 'Delete Mode'.

When a user clicks 'Add Document', the frontend displays a popup window that shows input and reminds the user of the language they are in.

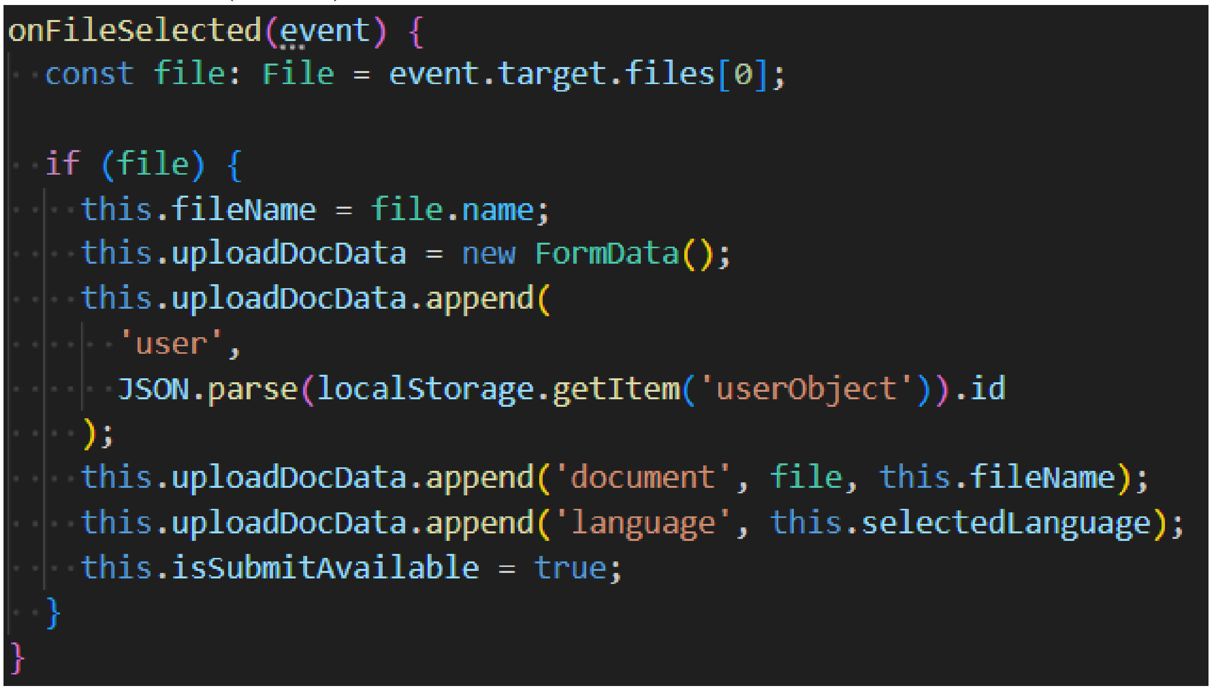
The Angular code for which in the frontend is displayed here:



```
  <dx-popup
    [width]="300"
    [height]="280"
    [showTitle]="true"
    title="Add Document"
    [dragEnabled]="true"
    [hideOnOutsideClick]="true"
    [showCloseButton]="true"
    [(visible)]="addDocumentVisible"
  >
    <input
      type="file"
      class="file-input"
      (change)="onFileSelected($event)"
      #fileUpload
    />
    <div *dxTemplate="let data of 'content'">
      <div class="upload-selecton">
        <p>Document being uploaded as {{ this.selectedLanguage }}</p>
        <p>Click button to select document</p>
        <button class="select-doc-btn" (click)="fileUpload.click()">
          <p>{{ fileName || "Please upload a document" }}</p>
        </button>
      </div>
      {{ text }}
      <button
        [disabled]="!isSubmitAvailable"
        class="submit-doc-btn"
        (click)="onFileSubmit()"
      >
        <p>Submit</p>
      </button>
    </div>
  </dx-popup>
```

As shown in the code, the visibility is managed by the ‘addDocumentVisible’ variable. When a user selects a file for the frontend to hold, the function “onFileSelected($event)” is run when a change is detected, e.g. empty to file1, file1 to file2.

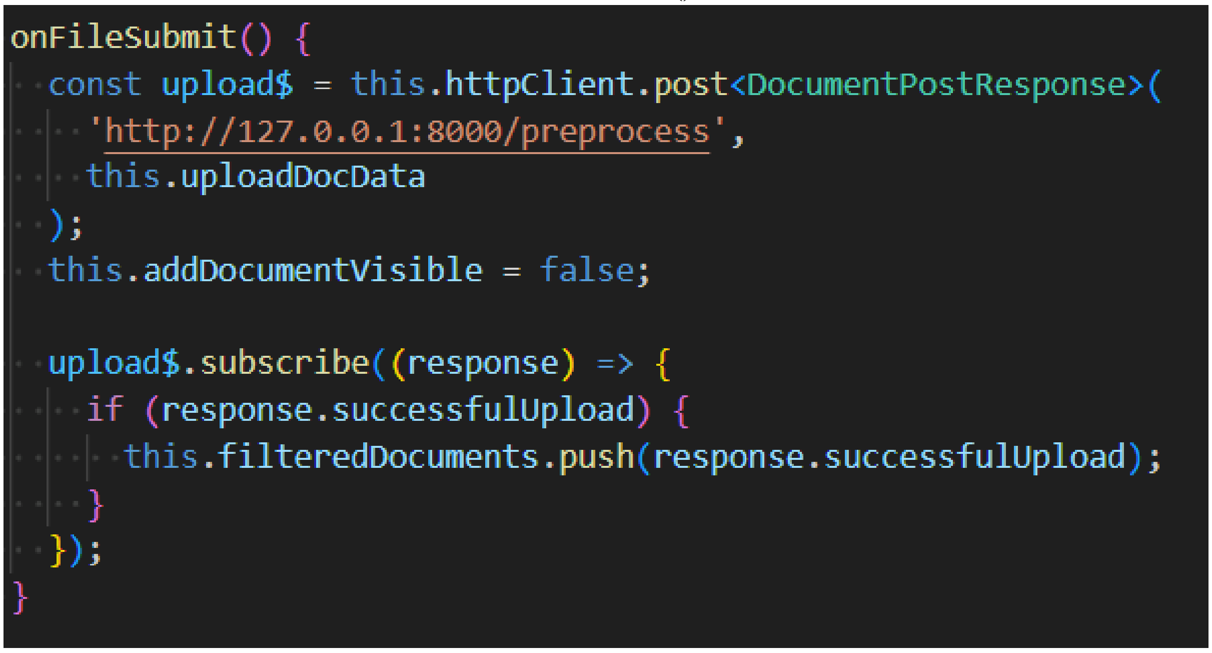
onFileSelected($event) is detailed below:



```
onFileSelected(event) {
  const file: File = event.target.files[0];

  if (file) {
    this.fileName = file.name;
    this.uploadDocData = new FormData();
    this.uploadDocData.append(
      'user',
      JSON.parse(localStorage.getItem('userObject')).id
    );
    this.uploadDocData.append('document', file, this.fileName);
    this.uploadDocData.append('language', this.selectedLanguage);
    this.isSubmitAvailable = true;
  }
}
```

This function specifies the steps involved in preparing the document file for being sent to the API. The name of the file, language and name of the document are obtained from the user's configuration and the document itself; it wraps these properties with the document's content. This data is temporarily stored in the ‘uploadDocData’ variable.

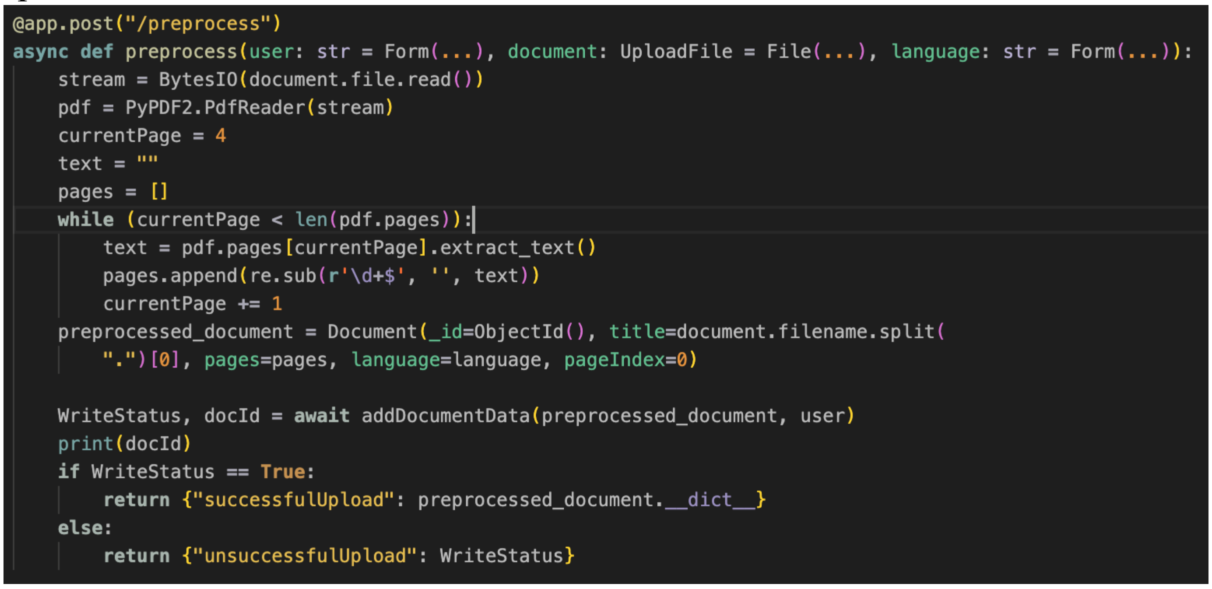
After this preparation is complete, the variable ‘isSubmitAvailable’ is set to true, this variable disables the button to submit when false. This is to avoid empty requests before file selection. This can be seen on the html, as well as the function this button activates, “onFileSubmit()”



```
onFileSubmit() {
  const upload$ = this.httpClient.post<DocumentPostResponse>(
    'http://127.0.0.1:8000/preprocess',
    this.uploadDocData
  );
  this.addDocumentVisible = false;

  upload$.subscribe((response) => {
    if (response.successfulUpload) {
      this.filteredDocuments.push(response.successfulUpload);
    }
  });
}
```

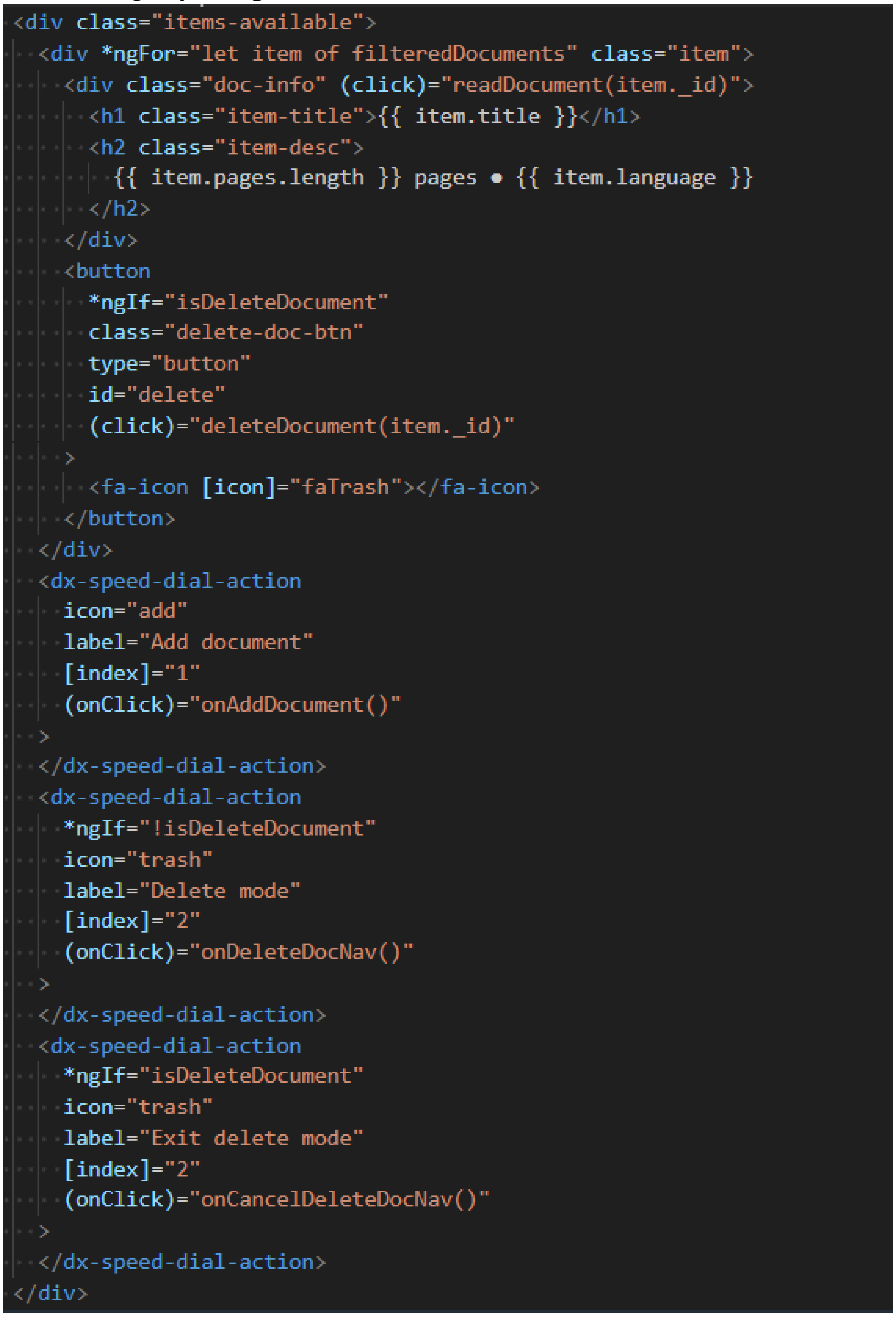
This function makes a POST request to the FastAPI NLP preprocess endpoint, sets the "addDocumentVisible" variable to false which stops displaying the popup. Then the response is subscribed to, which if successful, appends the new document locally.

The preprocess endpoint takes in the given document and processes it to fit the Document model and writing it to the MongoDB cluster via an update operation,



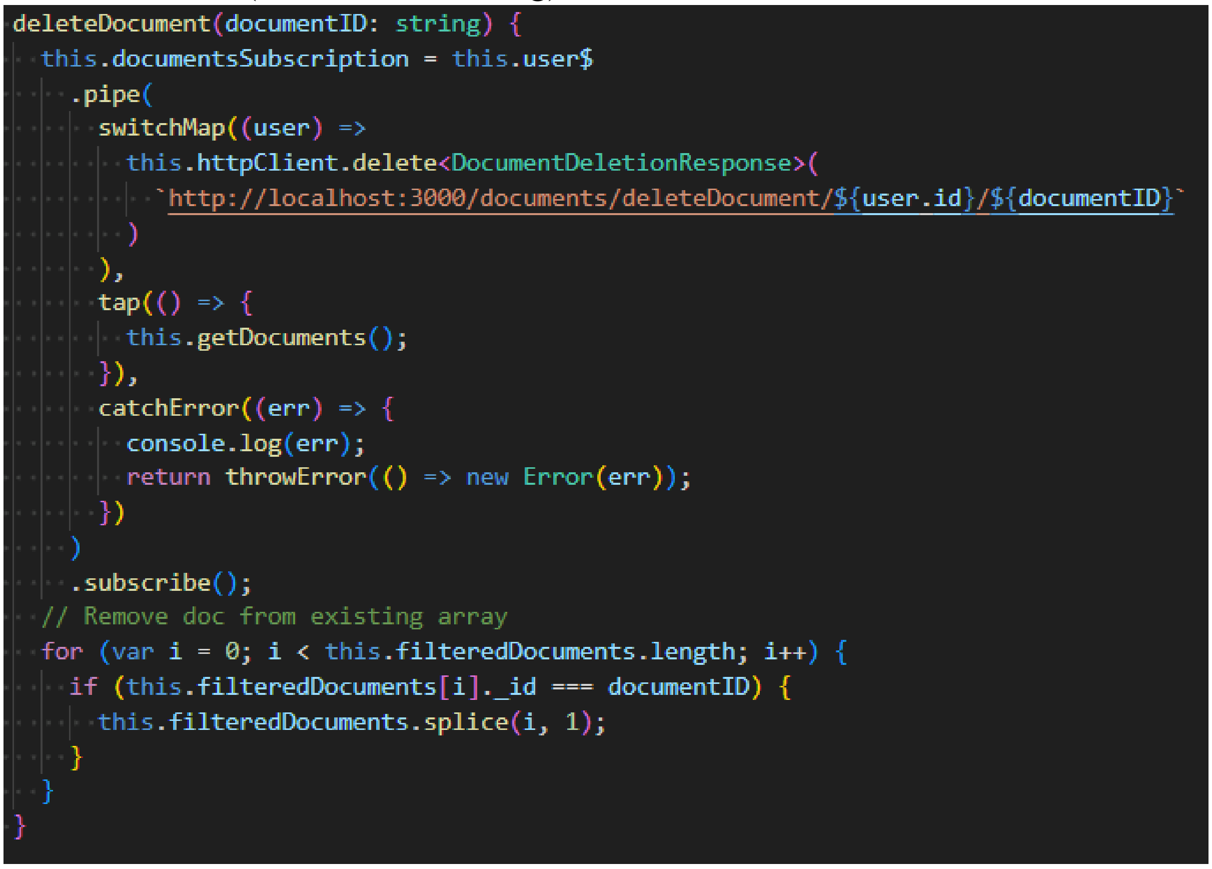
```python
@app.post("/preprocess")
async def preprocess(user: str = Form(...), document: UploadFile = File(...), language: str = Form(...)):
    stream = BytesIO(document.file.read())
    pdf = PyPDF2.PdfReader(stream)
    currentPage = 4
    text = ""
    pages = []
    while (currentPage < len(pdf.pages)):
        text = pdf.pages[currentPage].extract_text()
        pages.append(re.sub(r'\d+$', '', text))
        currentPage += 1
    preprocessed_document = Document(_id=ObjectId(), title=document.filename.split(
        ".")[0], pages=pages, language=language, pageIndex=0)

    WriteStatus, docId = await addDocumentData(preprocessed_document, user)
    print(docId)
    if WriteStatus == True:
        return {"successfulUpload": preprocessed_document.__dict__}
    else:
        return {"unsuccessfulUpload": WriteStatus}
```

For deletion, this is much the same except with the benefit of only needing the documents uniquely assigned ID to delete the document.



```html
<div class="items-available">
  <div *ngFor="let item of filteredDocuments" class="item">
    <div class="doc-info" (click)="readDocument(item._id)">
      <h1 class="item-title">{{ item.title }}</h1>
      <h2 class="item-desc">
        {{ item.pages.length }} pages • {{ item.language }}
      </h2>
    </div>
    <button
      *ngIf="isDeleteDocument"
      class="delete-doc-btn"
      type="button"
      id="delete"
      (click)="deleteDocument(item._id)"
    >
      <fa-icon [icon]="faTrash"></fa-icon>
    </button>
  </div>
  <dx-speed-dial-action
    icon="add"
    label="Add document"
    [index]="1"
    (onClick)="onAddDocument()"
  >
  </dx-speed-dial-action>
  <dx-speed-dial-action
    *ngIf="!isDeleteDocument"
    icon="trash"
    label="Delete mode"
    [index]="2"
    (onClick)="onDeleteDocNav()"
  >
  </dx-speed-dial-action>
  <dx-speed-dial-action
    *ngIf="isDeleteDocument"
    icon="trash"
    label="Exit delete mode"
    [index]="2"
    (onClick)="onCancelDeleteDocNav()"
  >
  </dx-speed-dial-action>
</div>
```

When ‘Delete Mode’ is clicked by the user, the function onDeleteDocNav() is run which just sets the ‘isDeleteDocument’ variable to true, which enables the ‘delete-doc-btn’ as seen in the html. For convenience, the frontend enters a deletion mode which enables multiple documents to be deleted without additional navigation for each document. An icon is displayed beside each document in Delete Mode, which when pressed executes the deleteDocument(documentID: string) function.

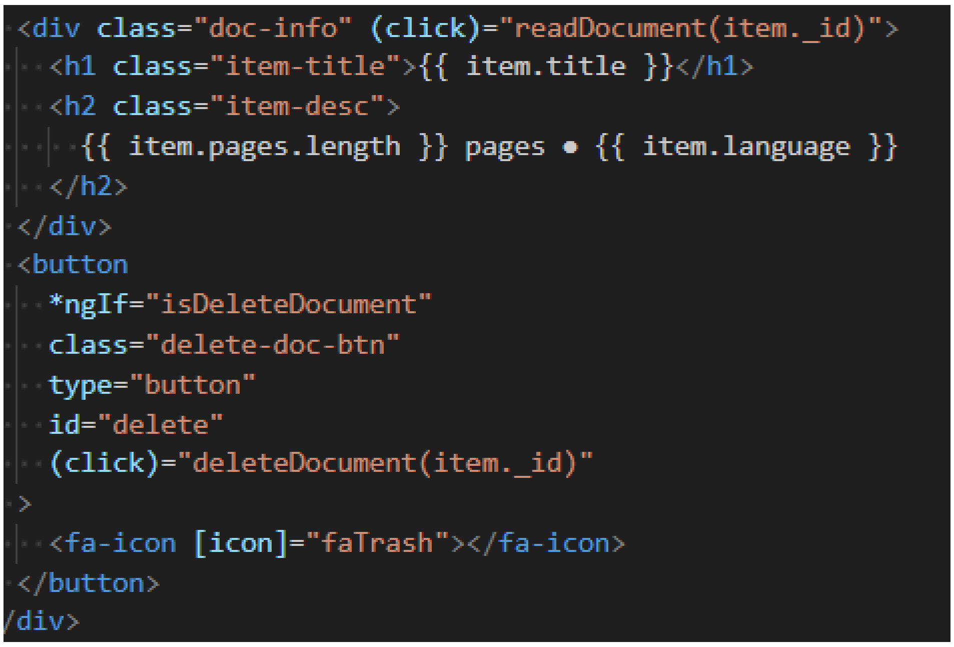


```
deleteDocument(documentID: string) {
  this.documentsSubscription = this.user$
    .pipe(
      switchMap((user) =>
        this.httpClient.delete<DocumentDeletionResponse>(
          `http://localhost:3000/documents/deleteDocument/${user.id}/${documentID}`
        )
      ),
      tap(() => {
        this.getDocuments();
      }),
      catchError((err) => {
        console.log(err);
        return throwError(() => new Error(err));
      })
    )
    .subscribe();
  // Remove doc from existing array
  for (var i = 0; i < this.filteredDocuments.length; i++) {
    if (this.filteredDocuments[i]._id === documentID) {
      this.filteredDocuments.splice(i, 1);
    }
  }
}
```

The function simply makes a delete request directly to the database, and catches any errors. It also manually removes the document from the local filteredDocuments array.

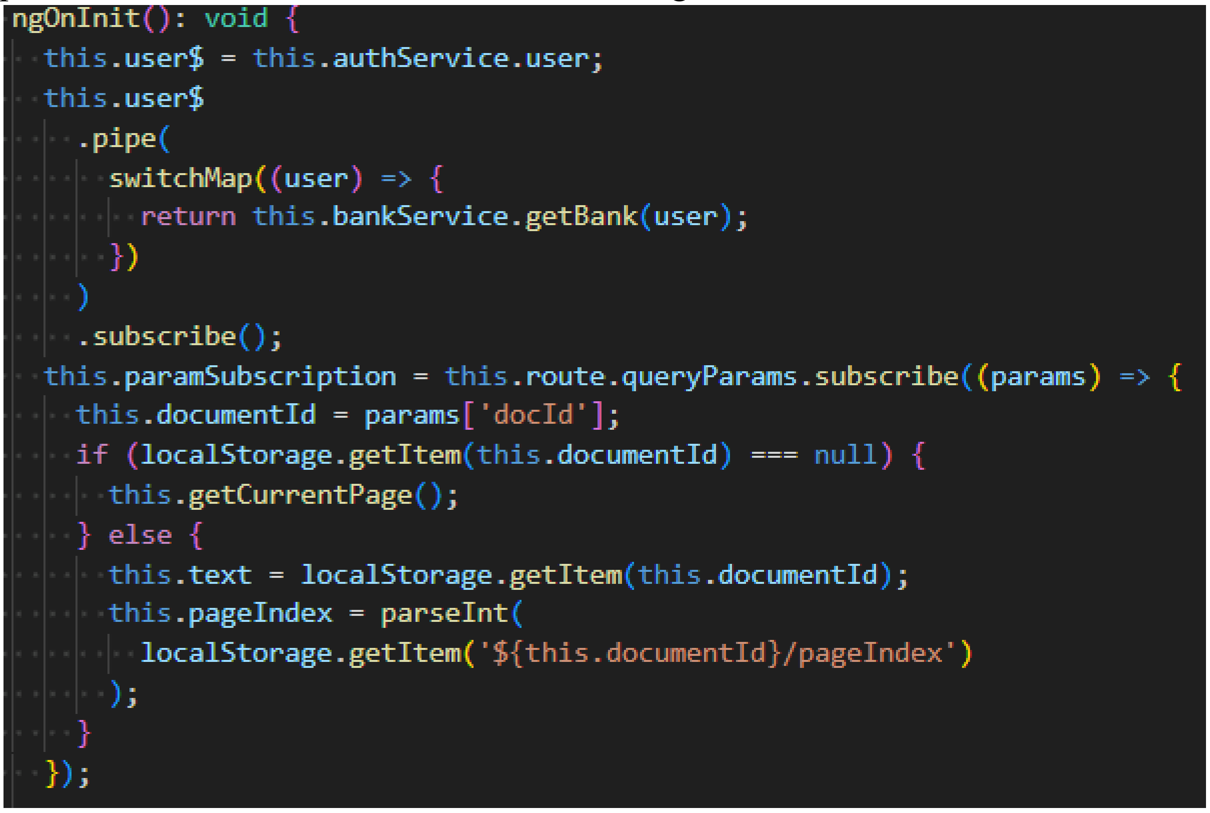
4.4 Reading View

The User accesses the reading view from overview, a book can be selected by clicking onto the document title. This can be seen here in overview:



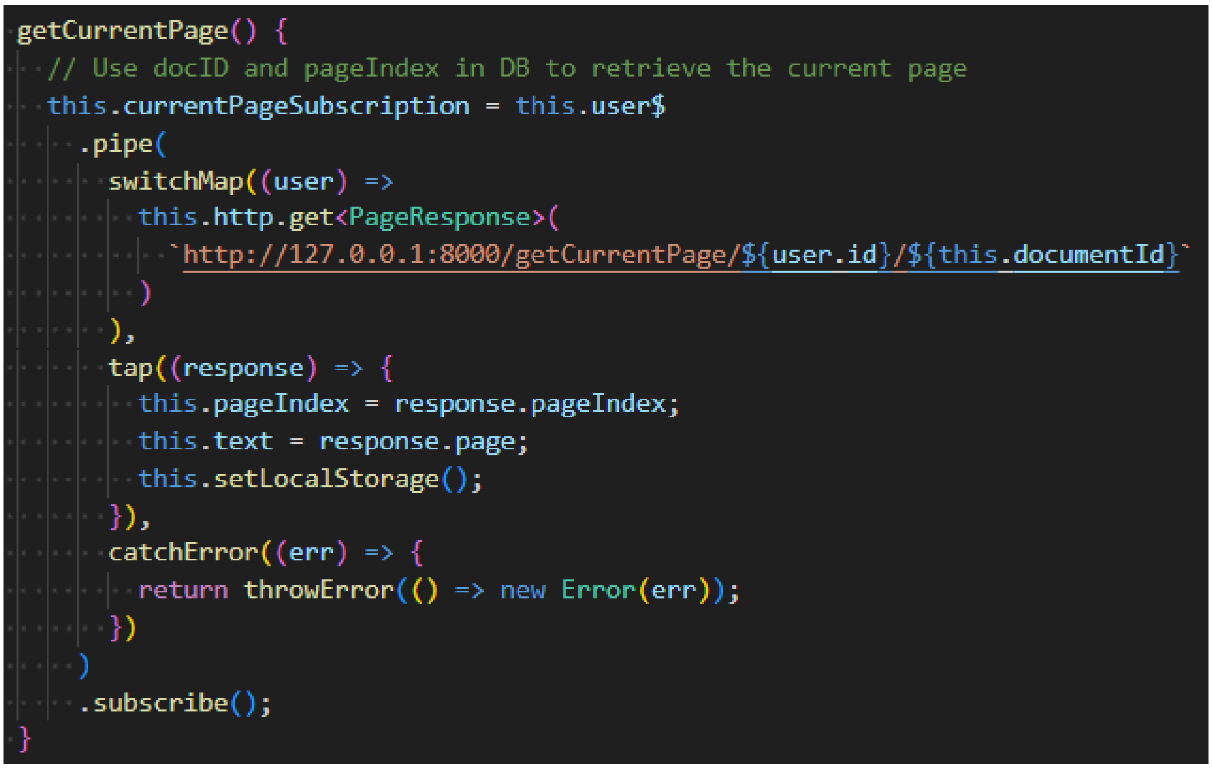
```
<div class="doc-info" (click)="readDocument(item._id)">
  <h1 class="item-title">{{ item.title }}</h1>
  <h2 class="item-desc">
    {{ item.pages.length }} pages • {{ item.language }}
  </h2>
</div>
<button
  *ngIf="isDeleteDocument"
  class="delete-doc-btn"
  type="button"
  id="delete"
  (click)="deleteDocument(item._id)"
>
  <fa-icon [icon]="faTrash"></fa-icon>
</button>
/div>
```

The readDocument function is run with the item’s id as a parameter. This function simply routes to the ‘read’ route with the documentID as a query parameter. When routed to read, the following code is ran:



```
ngOnInit(): void {
  this.user$ = this.authService.user;
  this.user$
    .pipe(
      switchMap((user) => {
        return this.bankService.getBank(user);
      })
    )
    .subscribe();
  this.paramSubscription = this.route.queryParams.subscribe((params) => {
    this.documentId = params['docId'];
    if (localStorage.getItem(this.documentId) === null) {
      this.getCurrentPage();
    } else {
      this.text = localStorage.getItem(this.documentId);
      this.pageIndex = parseInt(
        localStorage.getItem('${this.documentId}/pageIndex')
      );
    }
  });
```

This subscribes to the queryParam which it then uses to get the page in the this.getCurrentPage() function. Which is as follows:

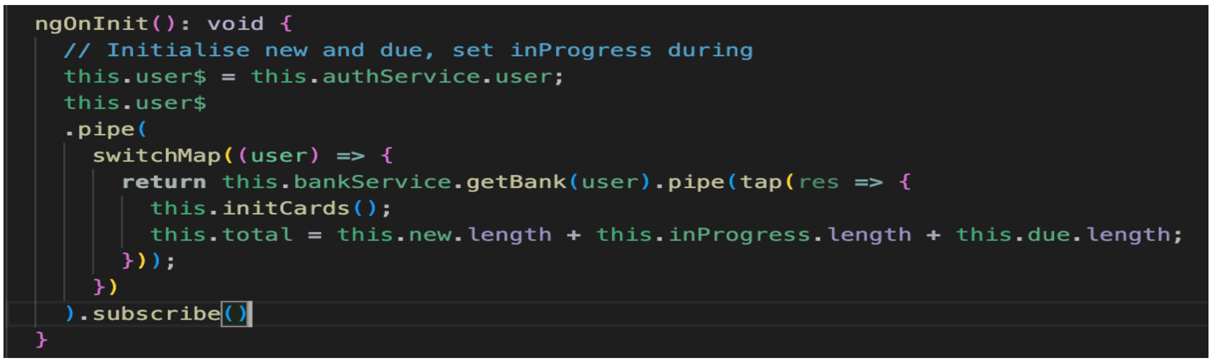


```
getCurrentPage() {
  // Use docID and pageIndex in DB to retrieve the current page
  this.currentPageSubscription = this.user$
    .pipe(
      switchMap((user) =>
        this.http.get<PageResponse>(
          `http://127.0.0.1:8000/getCurrentPage/${user.id}/${this.documentId}`
        )
      ),
      tap((response) => {
        this.pageIndex = response.pageIndex;
        this.text = response.page;
        this.setLocalStorage();
      }),
      catchError((err) => {
        return throwError(() => new Error(err));
      })
    )
    .subscribe();
}
```

The frontend then sends a request to the NLP backend to retrieve the document data for the page last accessed or first page if not previously viewed. This pulls the data into two variables in the frontend, pageIndex and text. The pageIndex is used as a pointer to the index of the current page in the pages array and to retrieve the next page and previous page. The text variable is simply the content of that page. The setLocalStorage() function caches the pageIndex and text variables for quick retrieval client-side if the user were to exit the reading session but return to the same book later in the same session.

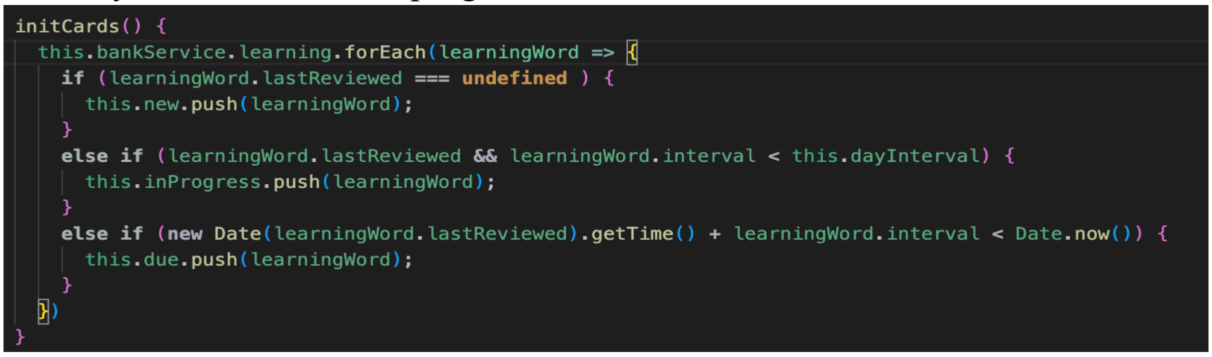
4.5 Review Page

The review page is where users revise words marked as learning. The principle behind this is similar to spaced repetition systems like Anki or SuperMemo; flashcards that appear and reappear over certain periods of time in an attempt to combat the "forgetting curve" (the boundary of memory where we begin to forget information we've encountered).
The review page functionality necessitated some additions to the modelling of the words in the learning word bank. These additions were the "lastReviewed" and "interval" fields. The former is the date that the word was last reviewed by the user and the interval is a millisecond value obtained from the user's assessment of their own knowledge of the word's features, which is used for scheduling the next review.



```
ngOnInit(): void {
  // Initialise new and due, set inProgress during
  this.user$ = this.authService.user;
  this.user$
  .pipe(
    switchMap((user) => {
      return this.bankService.getBank(user).pipe(tap(res => {
        this.initCards();
        this.total = this.new.length + this.inProgress.length + this.due.length;
      }));
    })
  ).subscribe()
}
```

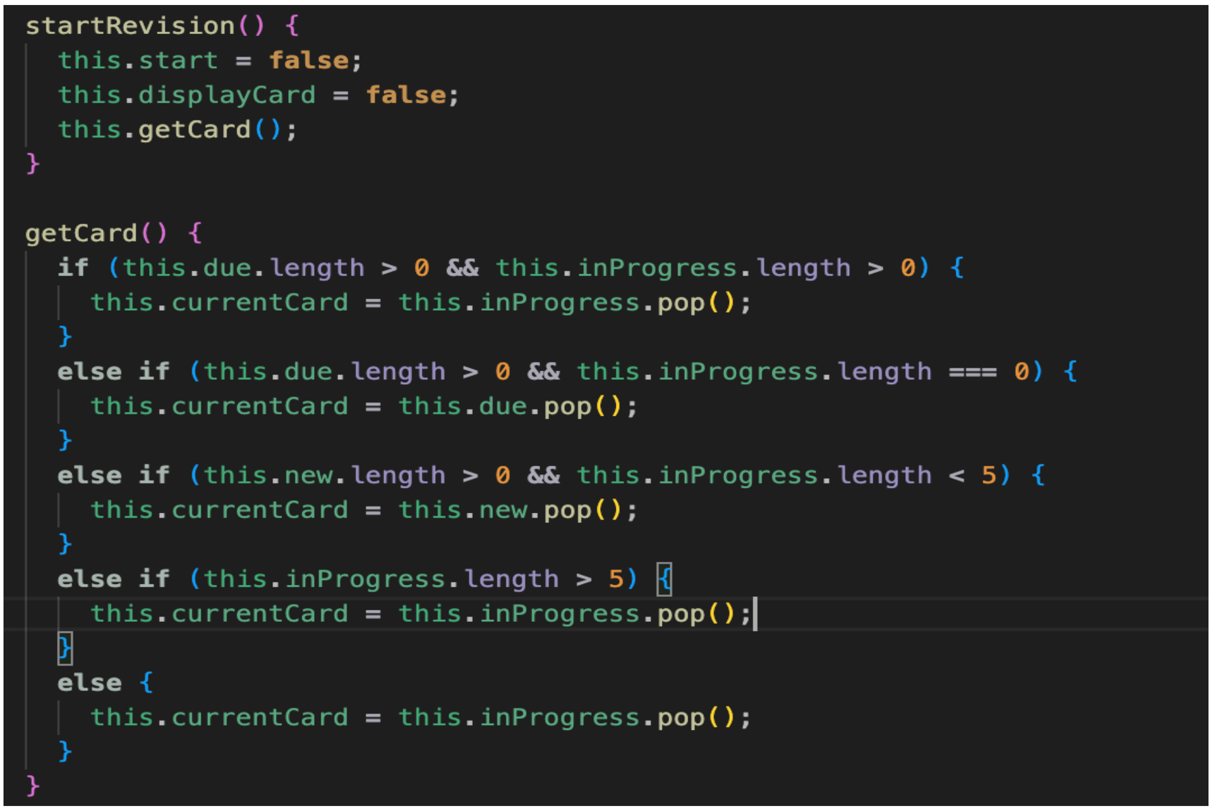
On component initialisation, we retrieve the word bank from the user via the bank service. The function initCards() divides words in the learning bank into the arrays new, due and in-progress.



```
initCards() {
  this.bankService.learning.forEach(learningWord => {
    if (learningWord.lastReviewed === undefined ) {
      this.new.push(learningWord);
    }
    else if (learningWord.lastReviewed && learningWord.interval < this.dayInterval) {
      this.inProgress.push(learningWord);
    }
    else if (new Date(learningWord.lastReviewed).getTime() + learningWord.interval < Date.now()) {
      this.due.push(learningWord);
    }
  })
}
```

A word is new in the review context if it has never been reviewed before, in-progress if it was left unreviewed in the previous session or incomplete in the current session and due if it is scheduled for another review given its interval.

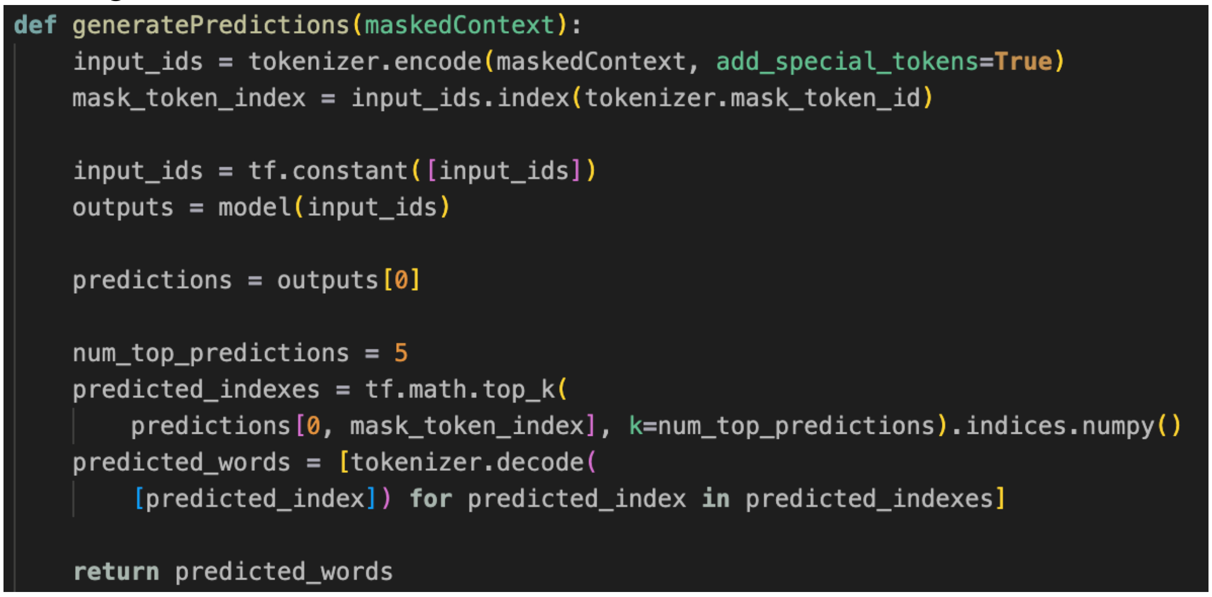
The user is greeted with a breakdown of their flashcards for the day in terms of new, in-progress and due. If they choose to start, the startRevision() function is called which fetches the first card via the getCard() function. The display logic of the UI for the review page is controlled by boolean variables that change with function calls accordingly.



```
startRevision() {
  this.start = false;
  this.displayCard = false;
  this.getCard();
}

getCard() {
  if (this.due.length > 0 && this.inProgress.length > 0) {
    this.currentCard = this.inProgress.pop();
  }
  else if (this.due.length > 0 && this.inProgress.length === 0) {
    this.currentCard = this.due.pop();
  }
  else if (this.new.length > 0 && this.inProgress.length < 5) {
    this.currentCard = this.new.pop();
  }
  else if (this.inProgress.length > 5) {
    this.currentCard = this.inProgress.pop();
  }
  else {
    this.currentCard = this.inProgress.pop();
  }
}
```

4.6 Predicting Words

Words are contextually predicted via masking tokens using BERT. The below example shows the code for prediction generation using TensorFlow. The context of the masked token is given as input and encoded. This gives a contextual word embedding, which is used to find the most suitable candidates for the given context.

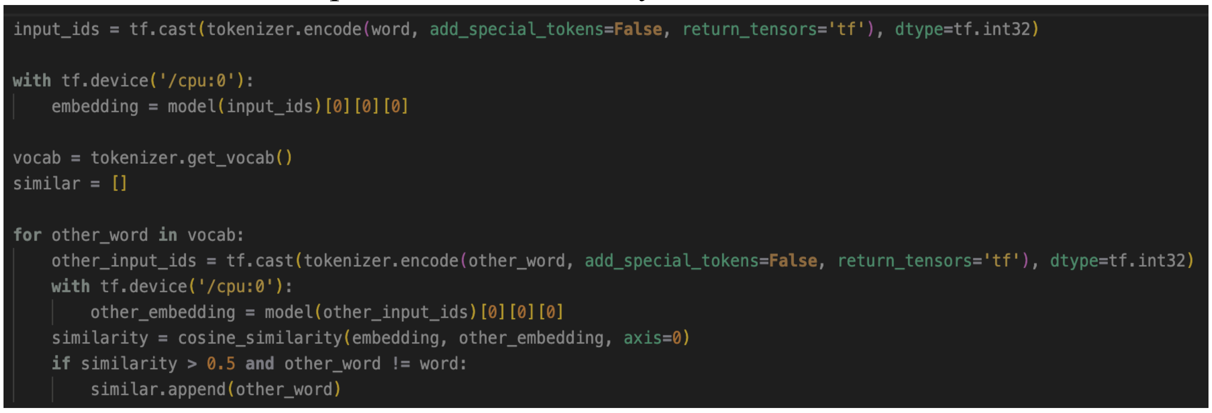


```
def generatePredictions(maskedContext):
    input_ids = tokenizer.encode(maskedContext, add_special_tokens=True)
    mask_token_index = input_ids.index(tokenizer.mask_token_id)

    input_ids = tf.constant([input_ids])
    outputs = model(input_ids)

    predictions = outputs[0]

    num_top_predictions = 5
    predicted_indexes = tf.math.top_k(
        predictions[0, mask_token_index], k=num_top_predictions).indices.numpy()
    predicted_words = [tokenizer.decode(
        [predicted_index]) for predicted_index in predicted_indexes]

    return predicted_words
```

5. Problems Solved

5.1 Predicting Words & Providing Synonyms

Given both developers had little background knowledge in the machine learning and natural language processing domains, one of the biggest challenges of the project was being able to process and return data in a timely manner. This was the barrier to being able to provide best candidate synonyms using the originally devised concept - measuring similarity of word vectors and embeddings in the vocabulary.
The first approach was using word vectors produced by SpaCy as a by-product of its training. However this rendered very poor results in speed and similarity; providing tokens that didn't qualify for fluency in the language (generally incomprehensible).
Using the French model, it was possible to measure the cosine similarity of BERT's contextual word embeddings to provide contextually suitable synonyms - done via this now deprecated functionality:



```
input_ids = tf.cast(tokenizer.encode(word, add_special_tokens=False, return_tensors='tf'), dtype=tf.int32)

with tf.device('/cpu:0'):
    embedding = model(input_ids)[0][0][0]

vocab = tokenizer.get_vocab()
similar = []

for other_word in vocab:
    other_input_ids = tf.cast(tokenizer.encode(other_word, add_special_tokens=False, return_tensors='tf'), dtype=tf.int32)
    with tf.device('/cpu:0'):
        other_embedding = model(other_input_ids)[0][0][0]
    similarity = cosine_similarity(embedding, other_embedding, axis=0)
    if similarity > 0.5 and other_word != word:
        similar.append(other_word)
```

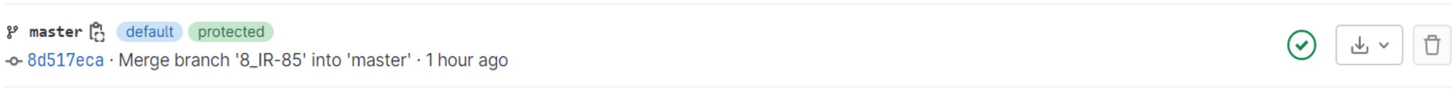
However, it was infeasible as even on a much smaller vocabulary, the process of encoding and comparing each embedding was computationally expensive and slow. The end implementation using BERT's masked language and instead providing morphological analysis using SpaCy was a tradeoff in functionality vs. usability for the user.

6. Testing

6.1 Gitlab CI/CD + Git Usage

Ensuring proper usage of Git, we established a set of standards to follow.
For all new feature requests, bugs, and miscellaneous items, we implemented a branch-based workflow, with a dedicated branch for each item. We maintained a

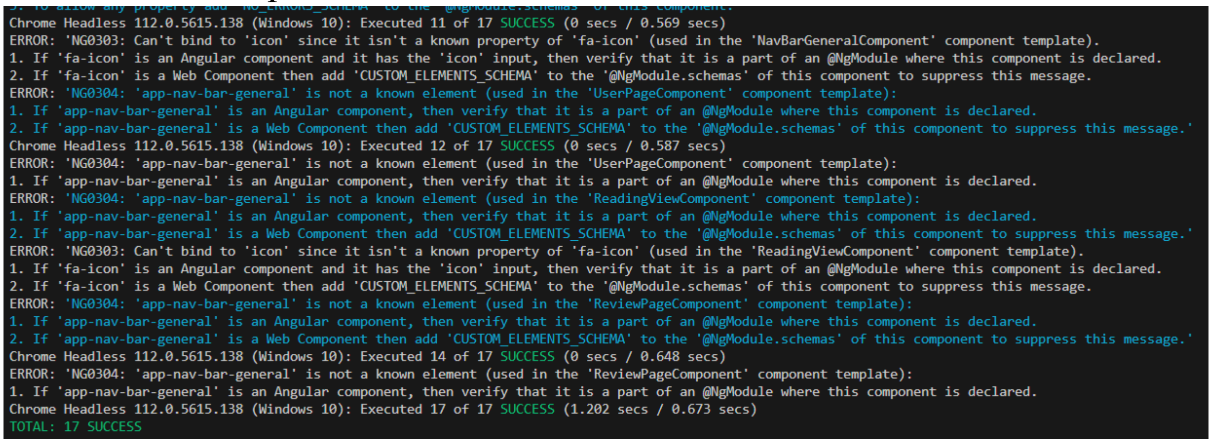
Jira backlog for each item, which included a detailed explanation for more complex items, if necessary. The branch names were based on the item number linked to the JIRA task, and also included the sprint number. Commit messages were required to detail the changes made to achieve the task, saving time during peer reviews.
Each merge request would be peer-reviewed by the other partner, if they were familiar with the application area, providing a second set of eyes and potentially catching any issues before merging. Additionally, the master branch was set as protected within GitLab to prevent the branch from being deleted or force pushed.
Each merge request required approval from either developer and must wait for the CI pipeline to succeed before changes could be merged.



master default protected
8d517eca · Merge branch '8_IR-85' into 'master' · 1 hour ago

6.2 Unit Tests

Infer-Read consists of an angular frontend, a mongoDB backend and a fastAPI backend API. As the frontend is written in Angular, Jasmine was provided for the unit tests in this area. 'ng test' is how the unit tests are run in the frontend, which returns a report like this:



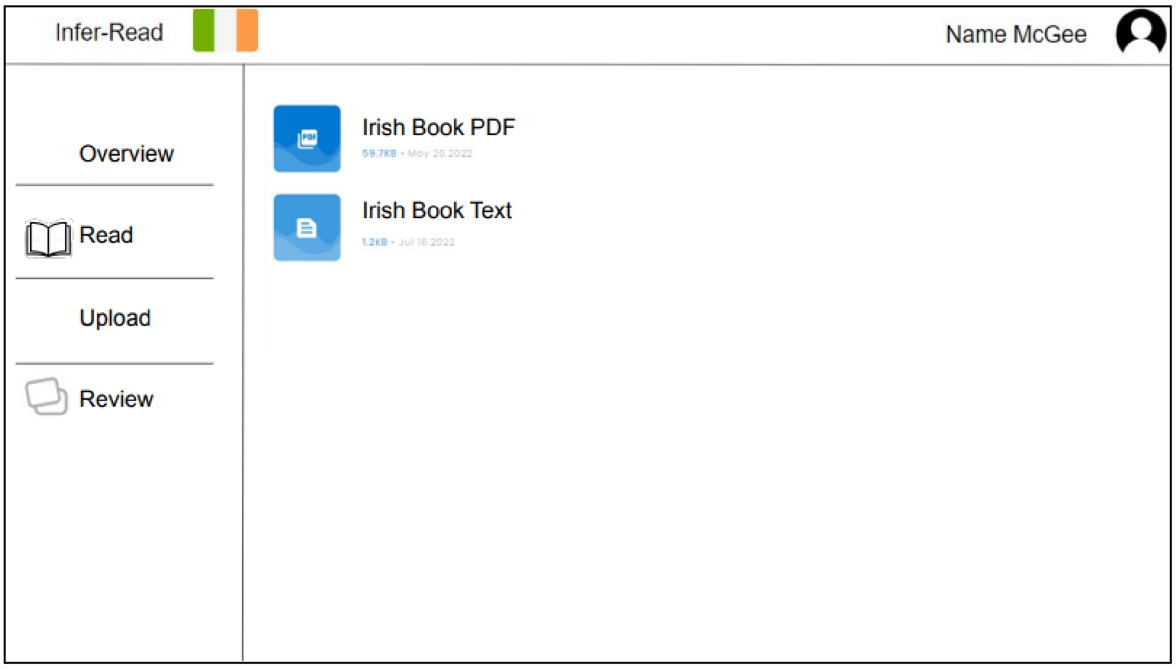
```
Chrome Headless 112.0.5615.138 (Windows 10): Executed 11 of 17 SUCCESS (0 secs / 0.569 secs)
ERROR: 'NG0303: Can't bind to 'icon' since it isn't a known property of 'fa-icon' (used in the 'NavBarGeneralComponent' component template).
1. If 'fa-icon' is an Angular component and it has the 'icon' input, then verify that it is a part of an @NgModule where this component is declared.
2. If 'fa-icon' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.
ERROR: 'NG0304: 'app-nav-bar-general' is not a known element (used in the 'UserPageComponent' component template):
1. If 'app-nav-bar-general' is an Angular component, then verify that it is a part of an @NgModule where this component is declared.
2. If 'app-nav-bar-general' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.'
Chrome Headless 112.0.5615.138 (Windows 10): Executed 12 of 17 SUCCESS (0 secs / 0.587 secs)
ERROR: 'NG0304: 'app-nav-bar-general' is not a known element (used in the 'UserPageComponent' component template):
1. If 'app-nav-bar-general' is an Angular component, then verify that it is a part of an @NgModule where this component is declared.
ERROR: 'NG0304: 'app-nav-bar-general' is not a known element (used in the 'ReadingViewComponent' component template):
1. If 'app-nav-bar-general' is an Angular component, then verify that it is a part of an @NgModule where this component is declared.
2. If 'app-nav-bar-general' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.'
ERROR: 'NG0303: Can't bind to 'icon' since it isn't a known property of 'fa-icon' (used in the 'ReadingViewComponent' component template).
1. If 'fa-icon' is an Angular component and it has the 'icon' input, then verify that it is a part of an @NgModule where this component is declared.
2. If 'fa-icon' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.
ERROR: 'NG0304: 'app-nav-bar-general' is not a known element (used in the 'ReviewPageComponent' component template):
1. If 'app-nav-bar-general' is an Angular component, then verify that it is a part of an @NgModule where this component is declared.
2. If 'app-nav-bar-general' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.'
Chrome Headless 112.0.5615.138 (Windows 10): Executed 14 of 17 SUCCESS (0 secs / 0.648 secs)
ERROR: 'NG0304: 'app-nav-bar-general' is not a known element (used in the 'ReviewPageComponent' component template):
1. If 'app-nav-bar-general' is an Angular component, then verify that it is a part of an @NgModule where this component is declared.
Chrome Headless 112.0.5615.138 (Windows 10): Executed 17 of 17 SUCCESS (1.202 secs / 0.673 secs)
TOTAL: 17 SUCCESS
```

It is configured to be 'chrome-headless', to enable its execution on a pipeline.

6.3 Design Heuristics & Mockups

In order to build our frontend application, we established a list of design heuristics that we were required to follow. These heuristics served as guidelines to ensure that our frontend design was accurate and met modern standards.

Early on, we developed mockups of what we wanted from the frontend on a functionality basis. As we went further, we reconsidered certain aspects both functionally and aesthetically. This allowed us to refine the design further. This initial mockup can be found in the docs directory.



To demonstrate, here is the mockup of our overview screen. The core functionality was sketched out before we started coding, then implemented into our application.

7. Future Work

7.1 Training Gensim Word Vector Models

We need to acquire more experience and knowledge with machine learning to train Gensim word vectors to see if we could create and finetune faster and more accurate vector similarity models. This would involve gathering more corpora in our target languages.

7.2 Explicit Machine Translation Model

It would be a big addition to the application to integrate a full transformer-based machine translation model. It would be trivial to implement calls to an API for MT like Google Translate or others but to grow our ML and NLP skills it would be rewarding to build out a full machine translation architecture.

Further, if we were to do this project again, focusing on one language but building it much further would be our wish.

8. Conclusion

Both students have learned a lot outside of their original knowledge going into the project. It involved combining components of altogether new domains for both of us and we enjoyed the development process despite the challenge of having to learn so much to even know where to begin solving the problems involved. Although some features were beyond our current abilities and we spent a lot of time on dead-ends and implementations that were later discarded, we have gained much more experience in the domain of machine learning and natural language processing. Both of us intend to keep working on the project to bring it closer to the original concept and execution we had in mind, and to use it ourselves in our own language learning endeavours.