TEDU Assistant

CMPE 492 Senior Project II

Low-Level Design Report

Advisor	Yücel Çimtay
Jury	Aslı Gençtav
	Emin Kuğu
Members	Deniz Akşimşek
	Enes Akmil
	Gizem Kurtcuoğlu
	Mehmet Batuhan Şenol

Introduction

TEDU Assistant is a voice-controlled, natural-language query system (i.e.: voice assistant, chatbot) which will answer common questions for university students, academic and administrative staff where answers are available but difficult to find or inconvenient. It will provide easy access to data published on university websites.

Previous reports about TEDU Assistant can be accessed on our project website [1].

The purpose of TEDU Assistant is to make information provided by TED University to its students and staff more accessible. We will achieve this by allowing users to query information using their voice and natural language. It should be possible to ask for and receive information through speech or text, for the user's convenience and ease of access.

Interface documentation guidelines

Because the server and client will be implemented in Python and JavaScript respectively, we will specify interfaces using Python type hint syntax for server modules and TypeScript syntax for client modules.

Object design trade-offs

In the construction of our system, we are aiming to maximize the following attributes:

Extensibility. It should take minimal effort to add features to the system. Clear extension points should be defined that allows extensions to a module to be decoupled. In particular for this system, it should be possible to add more language support and more kinds of questions that can be answered.

Maintainability: The system should not be difficult to maintain. When defects are found, fixing those defects should be possible with localized changes. Fixing defects and adding features should not introduce spurious defects.

Resilience: The system must deal with invalid input from users and changes to other systems it depends on. Because our system depends on many systems administered by

TEDU, the system should, to the extent possible, continue functioning when changes are made to these systems.
Usability: The system should be easy to use. Users should be able to know the capabilities of the system and how to use these capabilities to fulfill their needs without needing instruction. They should not feel confused or frustrated while using the system. They should be able to give feedback about the system.

Performance: Firstly, the system should respond to all user interaction on a timely basis. Secondly, the system must be able to function with a minimal computational expense in terms of memory use, CPU time and power consumption. To achieve this, we plan to make extensive use of caching and memoization. This is likely to increase the complexity of our system, which is a tradeoff with our goal of maintainability.

Availability: Ideally, the system should be ready to respond to users at all times to build user confidence. To achieve this, we need to ensure high uptime.

Security & privacy: The system must not provide information to users who are not authorized to see it. It should also make sure that users' questions are not intercepted by any third parties. To this end, our system will take a conservative approach and trade off versatility for security by only providing information that is readily publicly available on university websites.

Definitions, acronyms, and abbreviations

Follow-Up Question: questions asked by the system to the user when more information is required to answer the user's question
REST: Representational State Transfer
HATEOAS: Hypermedia as the Engine of Application State
HTTP: Hypertext Transfer Protocol
TLS: Transport Layer Security
TEDU: TED University

Overview

Server-side modules

The server side of the application uses a layered architecture, where every layer only depends on lower layers. This discourages brittleness and avoids mixing levels of abstraction. Dependency injection will be used to connect layers together.

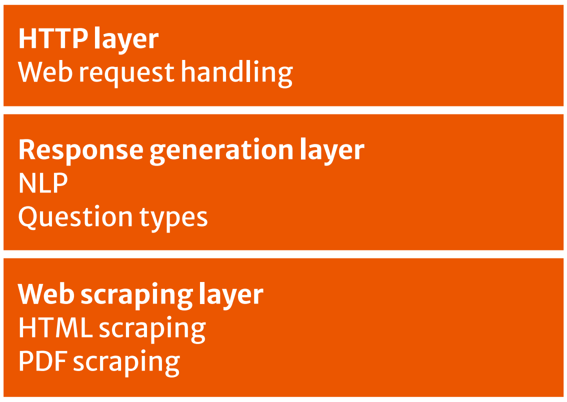

Table: Layered architecture of TEDU Assistant server, from our presentation [2]

Web module: This module is responsible for providing the web-based interface to the application. It consists of HTTP handlers and HTML templates. After deserializing and validating input, it will pass it along to the response generation module. The Flask framework [3] will be used to implement this module.

Response generation module: This module is the most significant in TEDU Assistant, as it's responsible for parsing and responding to questions. Its entry point is the `respond` function, which will receive a question as plain text and optionally a context object, and return an answer and the new context. The context object serves to hold data that needs to be retained between questions to form a longer conversation.

Web scraping module: The web scraping layer is responsible for fetching raw data from the university website, parsing it and returning the requested parts. We will need two scraping components at first: HTML scraper and PDF scraper. Due to the differences between these formats and the situations in which they are used, these components will not necessarily have a common superclass. This module will also contain the mechanism for fetching the data to scrape.

Client-side modules

TEDU Assistant will include a small amount of client-side code, as most of the functionality is implemented in the server. Our client-side components are implemented to be reusable, with most application-specific code in the server.

Speech recognizer element: Implements a speech recognition UI using Custom Elements [4]. The recognition will be handled by the Web Speech API [5]. It will start recognizing audio when actuated by the user and emit an event [6] when a string is recognized. Events will be the main architectural mechanism in the client. We currently have a prototype implementation of the speech recognizer, with a live demo on https://speech-recognizer-element.netlify.app/ (Only Google Chrome is supported).

Answer view: This component will take responses sent by the server in our format and display them to the user by converting them to HTML elements. It will also emit events when a follow-up answer or other interactive element is clicked.

Main program: The client's main program orchestrates the client components and communicates with the server. It will listen to events from the speech recognizer and when a question is recognized, send it to the server. After receiving a response to the server, it will deserialize it and pass it to the answer view. It will also listen to the answer view's events to send follow-up answers to the server.

Class and function interfaces

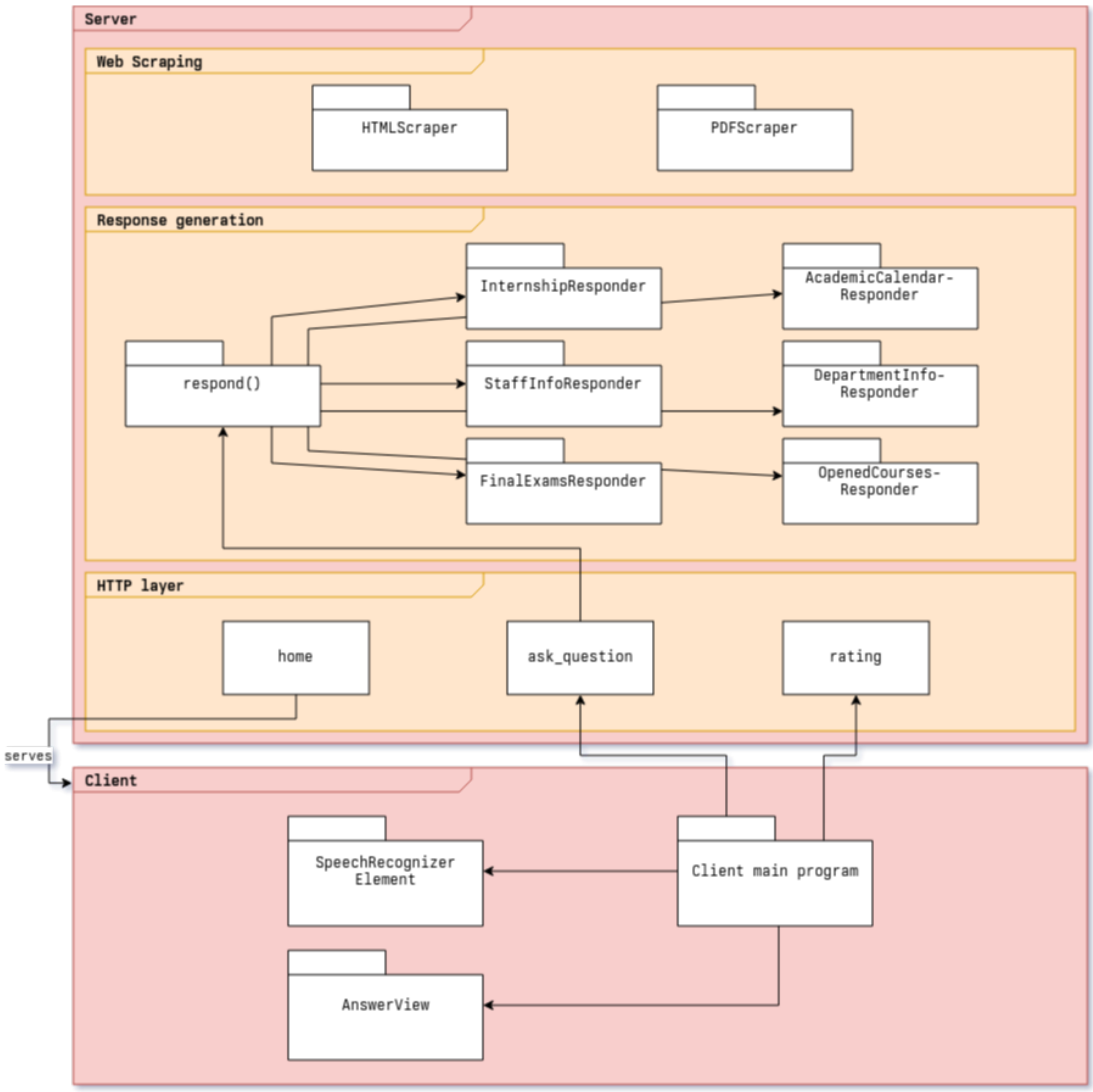

Figure: Diagram of system modules and components.

Web module

```
@app.get("/")
def home(): ...
```

Serves the home page of the application from a template.

```
@app.post("/ask")
def ask_question(): ...
```
Receives a question and context from the request body. Parses and validates questions and context, passes them to the response generation layer and returns the response in JSON format.

```
@app.post("/rate")
def rating():
```
Receives ratings for responses. The handling of these ratings is TBD.

Response generation module

```
def respond(
   question: str, context: Context
) -> tuple[Response, Context]: ...
```
Responds to a question. Delegates to the various responders in the application. The context object contains answers to any follow-up questions asked previously in the conversation. The return value contains a `Response` as well as a new context, since being asked a question can provide new information that alters the context.

```
class Responder:
  def can_answer(question: str, context: Context) -> bool: ...
  def answer(question: str, context: Context) -> Response: ...
```
The `Responder` interface represents a kind of question that our system can respond to. Each responder has the `can_answer` method that checks whether this responder can respond to a particular question, and the `answer` method that actually generates the response. The `can_answer` method is used by the `respond` function to determine which responder to use in answering a question.

```
class InternshipResponder(Responder):
```

The InternshipResponder directs users to their internship coordinator.

```
class StaffInfoResponder(Responder):
```

The StaffInfoResponder gives basic info about academic and administrative staff from public profiles.

```
class FinalExamsResponder(Responder):
```

The FinalExamsResponder provides dates of final exams.

```
class AcademicCalendarResponder(Responder):
```

The AcademicCalendarResponder provides dates from the academic calendar.

```
class DepartmentInfoResponder(Responder):
```

The DepartmentInfoResponder gives information about departments and faculties, such as staff, contact information and courses offered.

```
class OpenedCoursesResponder(Responder):
```

The OpenedCoursesResponder provides the list of opened courses.

```
class Response:
   content: str
```

The Response class represents a response to a user question, and it can also be a request for more information (a follow-up question).

```
class FollowUpQuestion(Response):
   options: list[str]
```

A follow-up question has a predetermined list of answers that the user can choose.

Web scraping module

```
class Scraper:
   def fetch(url: str) -> bytearray: ...
```

The scraper base class includes data fetching functionality.

```
class HtmlScraper(Scraper):
   def __init__(url: str): ...
   def find(selector: str) -> Element: ...
```

The `HtmlScraper` class allows users to fetch HTML data and extract information from it. The `find` method allows finding elements within a document using CSS selectors or XPath. It returns the element representation used by the underlying library (TBD).

```
class PdfScraper(Scraper):
```

The PdfScraper class is for extracting information from PDFs published on the website. Its exact design is to be determined.

Sample use cases

Internship use case:

- User presses the speech button and says: “Who can I ask about my internship?”.
- Speech recognizer element emits event.
- Client main program sends request to `/ask` endpoint of the server.
- `ask_question` function parses the question, and passes it to the `respond` function.
- respond function finds the appropriate responder (`InternshipResponder`).
- `InternshipResponder` looks at the context as well as the question content for the user's department. Not finding it, it returns a follow-up question as a response and a context containing the current question so that it can be answered later when all necessary information is available.
- ask_question serializes this response and context and returns it to the client.
- On the client, the answer view shows the follow up question to the user.
- Users pick their department.
- Answer view emits an event.
- Client main program receives the event and makes another request to `/ask`, with both the user’s choice and the context previously returned.
- The `ask_question` function calls the `respond` function again, which sees the original question in the context and passes the question to the InternshipResponder again.
- `InternshipResponder` can now add the user’s department to the context.
- `InternshipResponder` uses the `HtmlScraper` which was passed to it at construction time to fetch and parse the relevant page on the tedu.edu.tr website. It finds the internship coordinator for the user’s department and generates a response.
- `ask_question` serializes this response to JSON and sends it to the client.
- The answer view presents the response.
- The user knows who to consult for internship questions.

References

[1] Şenol, M.B. & Kurtcuoğlu, G. & Akşimşek, D. & Akmil, E., (2022), “TEDU Assistant — Final Project,” https://tedu-cmpe491-group10.netlify.app/.

[2] Şenol, M.B. & Kurtcuoğlu, G. & Akşimşek, D. & Akmil, E., (2023), "TEDU Assistant — CMPE 491 Project Presentation," https://docs.google.com/presentation/d/e/2PACX-1vSL3Q_06zQRSkKrzSguYNrpfn7UNzdJQhZJr0XiDNSaiUcLF-NSfJ9UETyEEcefoZ1jrSqbEWyiW8yK/pub?start=false&loop=false&delayms=3000.

[3] "Welcome to Flask — Flask Documentation (2.2.x)", https://flask.palletsprojects.com/en/2.2.x/.

[4] WHATWG, "4.13.4 The CustomElementRegistry interface," in "HTML Standard," https://html.spec.whatwg.org/multipage/custom-elements.html#custom-elements-api.

[5] WICG, (2020), "Web Speech API," https://wicg.github.io/speech-api/.

[6] WHATWG, "2.2. Interface `Event`," in "DOM Standard," https://dom.spec.whatwg.org/#interface-event.