TEDU Assistant

CMPE 491 Senior Project I
High-Level Design Report

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

Contents

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. [1]

Purpose of the system

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.

Design goals

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.

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. Our system will take a conservative approach to 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 [3]

HATEOAS: Hypermedia as the Engine of Application State

HTTP: Hypertext Transfer Protocol

TLS: Transport Layer Security

TEDU: TED University

Overview

TEDU Assistant is a Web application using NLP, the Web Speech API [2] and web scraping to provide a voice assistant experience. Users ask questions via speech, or text if they prefer. The voice input is recognized into text on the client, analyzed on the server and a response is generated by fetching the wanted information from TED University websites. This way, users can ask for the specific information they need in the way they understand, instead of needing to navigate the organizational structure of relevant web pages. Using the Web Speech API means that we will be able to use speech synthesis and voice recognition capabilities that are built into the user's device, instead of having to replicate these capabilities.

Proposed software architecture

Overview

As part of TEDU Assistant's architecture, we will decompose it into multiple subsystems, each one having a single responsibility, and restrict the interaction between subsystems to avoid tight coupling. Each subsystem can be built with the technologies most appropriate for its purpose (with some restrictions, for easy integration of subsystems). We take into consideration the constraints of our environment, such as the web using a client/server model. We also take inspiration from writings on software architecture, such as “Architectural Styles and the Design of Network-based Software Architectures” by Roy Fielding [3].

Subsystem decomposition

We use a client-server, request-response, layered architecture. The interface between the client and server is a REST [3] interface where the client sends questions as text (having recognized voice) and receives answers and follow-up questions. We have preferred to move responsibilities mostly to the server, for instance, the server will respond to questions with answers in a format that contains information about how the answer is presented.

Deriving from the idea of hypertext/hypermedia, the responses can also contain follow-up questions and other interactive elements. These elements can link the response to further responses. The user can complete linear flows of providing necessary details to get a specific piece of information, or explore information in a free-form manner.

The server subsystem is then further decomposed into the HTTP layer, question response generation and web scraping. The HTTP layer includes the webserver and views that construct domain objects from HTTP requests, and present responses. The HTTP layer only communicates directly with the response generation layer (through method calls) and the client layer (through HTTP). By ensuring each layer communicates only with adjacent layers, we create a linear order of layers with clear flow of data.

Hardware/software mapping

For the server, we will use the Python programming language and the Flask [4] web framework.

The client and server will communicate over HTTP via a REST API as mentioned previously. As opposed to many other “REST” APIs [5], we plan to use all constraints of the style, including HATEOAS [6].

The client will be a Web application that will run on the user’s own device. It will require a microphone on the device. We also rely on the Web browser on the device to support the Web Speech API [2].

Persistent data management

Subsystems may need to persist or cache information. For this purpose, a PostgreSQL (or similar SQL) database or a Redis key-value store may be used.

Access control and security

The system will not require signup or authorization. This will radically simplify the usage as well as reducing the implementation cost. This is possible as the system only provides information that is already publicly available.

For security, we will ensure that all client-server communication happens with TLS.

Global software control

In TEDU Assistant, there is a strict separation of responsibilities between server and client that avoids any components being shared or replicated between them. As a result of this, the server and client have separate architectures.

On the server, we have a layered architecture. We will use Python modules [10] to separate the layers. A rule we will follow to keep the layers in order is to make sure that each layer only communicates with the layer above and the layer below. For this communication, simple function or method calls will be used, with the HTTP layer that handles user requests being the “entry point”. The server will also make many HTTP requests to other servers, particularly the university website, and thus we may make use of Python’s asynchronous features [11] to make sure these requests do not block operations.

On the client, we use browser events [8] and custom events [9] as the main mechanism of connecting subsystems.

Boundary conditions

Initialization

All processing in TEDU Assistant is initiated by an HTTP request. The user can initialize the application by visiting the root URL on their browser. They can continue their session by interacting with the user interface (asking questions, answering follow-up questions, assigning ratings, &c.)

Termination

In REST-ful Web applications, each request to the server is self-contained. As such the user does not need to explicitly terminate TEDU Assistant — they can simply stop interacting and no more processing will happen for their session.

Failure

Lack of internet connectivity can result in failures because most of TEDU Assistant's features depend on this connection to function.

Subsystem services

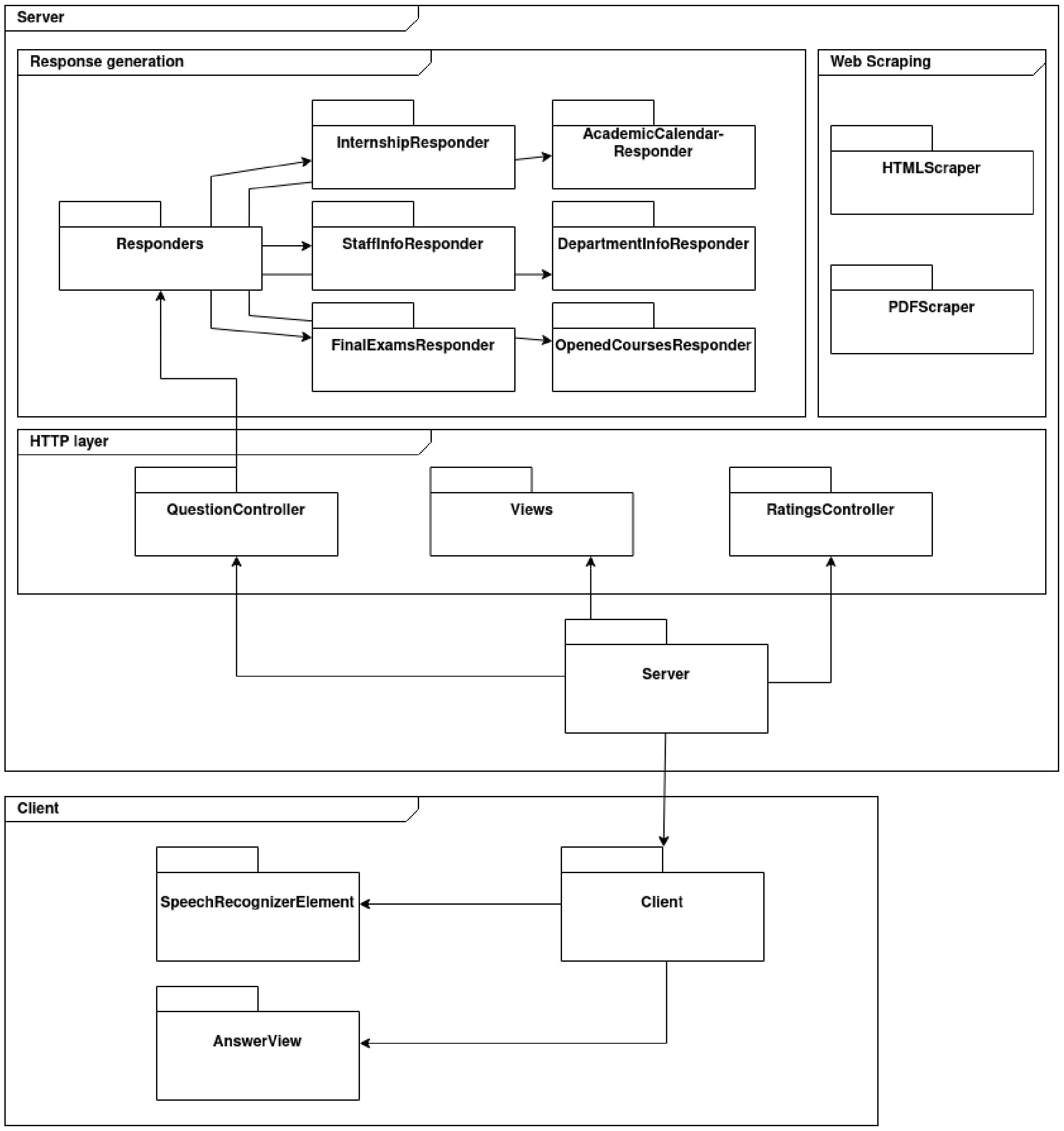

Figure 1: Subsystems and components of TEDU Assistant

Client

The client is an HTML & JavaScript application. It can be used on any device with a browser. It can also be converted into a mobile app via hybrid app technologies such as Apache Cordova or similar.

By moving as many responsibilities to the server as we reasonably can, we are able to keep the client very simple. As a result, we will not need any JavaScript framework and will construct the client from only a few components.

The **speech recognizer element** will encapsulate the speech recognition UI using Custom Elements [7]. It will start recognizing audio when actuated and emit events [8] when a string is recognized. Events will be the main architectural mechanism in the client.

The **answer view** will accept responses to questions 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 similar interactive element is clicked.

The **client** entry point is responsible for communication between these two client components and the server. It will listen to events from the speech recognizer and when a question is complete, send it to the server. After receiving a response to the server, it will deserialize it and pass it to the answer view to present to the user. It will also listen to the answer view's events to send follow-up answers to the server.

Server

The server holds the majority of the system's functionality and thus needs to be split into multiple modules. We have arranged the modules in a layered manner where each layer only communicates to the layer above and the layer below. This creates loose coupling and clear data flow. The topmost layer is the client, followed by the server's HTTP layer, response generation and web scraping.

There will be a few services that will be needed in multiple layers, such as persistent data storage. We plan on using dependency injection methods for such services.

HTTP layer

The HTTP layer deals with the web-related aspects of the application. It acts as a translation layer between HTML or JSON payloads and domain objects. It will also be the only part of the system that depends on Flask, meaning the rest of the server could be remodeled as a non-web application. It has three sub-components:

Views. This component consists of HTML templates and a static file server for delivering the client application to users' devices.

Question Controller. This section deals with responding to questions. It will accept requests containing questions in JSON format, and after deserializing, pass them on to the response generation layer. After a response is generated, it will respond to the HTTP request with the JSON representation of this response.

A contract needs to exist between the question controller and the client about the format of questions and answers. This format will be able to represent rich content as well as follow-up questions and answers.

Response generation

We will create a *Responder* object for each kind of question TEDU Assistant is able to answer. The Responder interface is as follows:

```
canAnswer(question: Question) -> bool
answer(question: Question) -> Response
```

We plan to also have a Responder abstract base class that contains common behaviors for all questions, such as how to respond when an adequate answer cannot be provided.

The *Responders* object is the registry of these Responder objects. When given a question, it can query the responders and forward the question to the responder that can answer it.

Responders may invoke the web scraping layer to acquire information. For instance, AcademicCalendarResponder will ask the web scraping layer to scrape the academic calendar page (https://www.tedu.edu.tr/en/academic-calendar) and extract data from the tables (CSS selector `.academic-calendar-view > table`), then search through this data to find the date that the user asked for. Sophisticated responders such as this one may have a separate data access component, keeping only response generation code in the main Responder, whereas for others this might not be necessary.

Web scraping

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.

While building these components, we expect to use a combination of off-the-shelf libraries and custom code for TEDU Assistant's scraping needs.

It's important that this subsystem is easy to modify and resilient to unexpected situations. This is because it interacts directly with the university website, an external system with which we have no contract.

Glossary

Chatbot: a software system accepting questions from users and automatically writing responses in a way that mimics human conversation

Hybrid App: a software application, usually for mobile devices, that combines elements of both native apps (apps made with the platform’s toolkit) and web applications

NLP: Natural Language Processing

Voice Assistant: a software system assisting users in their day-to-day tasks that can be controlled by spoken commands

Web Scraping: extracting data from websites, usually by parsing HTML

References

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

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

[3] Fielding, R., (2000), “Architectural Styles and the Design of Network-based Software Architectures,” CA: University of California, Irvine, https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

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

[5] htmx, “How Did REST Come To Mean The Opposite of REST?” https://htmx.org/essays/how-did-rest-come-to-mean-the-opposite-of-rest/.

[6] htmx, “HATEOAS,” https://htmx.org/essays/hateoas/.

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

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

[9] WHATWG, “2.4. Interface `CustomEvent`,” in “DOM Standard,” https://dom.spec.whatwg.org/#interface-customevent.

[10] Python Software Foundation, “6. Modules,” in “Python 3.11.1 Documentation,” https://docs.python.org/3/tutorial/modules.html.

[11] Python Software Foundation, “8.9.1. Coroutine function definition,” in “Python 3.11.1 Documentation,” https://docs.python.org/3/reference/compound_stmts.html#async-def.