Developer Documentation Home

Welcome to the xtutor space!

Search

Get started with the basics

Start editing this page:

- [ ] Click the pencil icon ✏️ or `e` on your keyboard to edit and start typing. You can edit anywhere.
- [ ] Hit `/` to see all the types of content you can add to your page. Try `/image` or `/table`
- [ ] Use the toolbar at the top to play around with *font*, **colors**, formatt ing, and more
- [ ] Click `close` to save your draft or `publish` when your page is ready to be shared

Need some inspiration?

- Check out our Confluence best practices guide.
- Get a quick intro into what spaces are and how to best use them at Set up your site and spaces.
- Check out our guide for ideas on how to set up your space overview.
- If starting from a blank space is daunting, try using one of the space templates instead.

Web Application

All things related to the web application.

Backend

All documentation for the backend repository goes here

URL:

https://bitbucket.org/xtutor/backend/src/736b4bbceb578ed5f1678126efcacbcac6c66655/?at=release%2Fmvp

Repository Structure Guide - Backend

Summary

This page will explain the basic structure of the backend repository linked in the parent page. It will cover the purposes of the main directories and files. Apologies for this not being a tree structure. Confluence (for some reason??) doesn't support them.

Directory Tree

Output of `$ tree . -I "node_modules|dist" > tree.txt`:

```
.
├── buildspec.yml
├── package.json
├── package-lock.json
├── README.md
├── src
│   ├── api
│   │   ├── http
│   │   │   ├── app.ts
│   │   │   ├── controllers
│   │   │   │   ├── lessons.controller.ts
│   │   │   │   ├── users.controller.ts
│   │   │   │   └── waitingList.controller.ts
│   │   │   ├── middleware
│   │   │   │   ├── deserialiseUser.ts
│   │   │   │   ├── requireAccessLevel.ts
│   │   │   │   └── requireUser.ts
│   │   │   ├── routes
│   │   │   │   ├── api.router.ts
│   │   │   │   ├── lessons.router.ts
│   │   │   │   ├── users.router.ts
│   │   │   │   └── waitingList.router.ts
│   │   │   └── utils
│   │   │       ├── constants.ts
│   │   │       └── sendEmail.ts
│   │   ├── server.ts
│   │   └── ws
│   │       ├── application-handlers
│   │       │   ├── startChatHandler.ts
│   │       │   ├── startLessonHandler.ts
│   │       │   ├── startQuizHandler.ts
│   │       │   └── transcribeAudioHandler.ts
│   │       ├── connectionHandler.ts
│   │       ├── createServer.ts
│   │       ├── lib
│   │       │   ├── ChatGPTConversation.ts
│   │       │   ├── constants.ts
│   │       │   ├── DelayedBuffer.ts
│   │       │   ├── fetchServerSentEvent.ts
│   │       │   ├── OrderMaintainer.ts
│   │       │   └── Quiz.ts
│   │       ├── middleware
│   │       │   └── deserialiseUser.ts
│   │       ├── schema
│   │       │   └── startLessonSchema.ts
│   │       └── utils
│   │           ├── constants.ts
│   │           ├── onWrittenFeedbackEnd.ts
│   │           ├── sendMessageFromX.ts
│   │           ├── setUpConversationWithX.ts
│   │           ├── tts.ts
│   │           ├── updateChatHistory.ts
│   │           └── updateSocketUser.ts
│   ├── config
│   │   └── supa.ts
│   ├── email-templates
│   │   └── xtutor_wait.ejs
│   ├── index.ts
│   ├── mock-prompts
│   │   ├── mock-answer.txt
│   │   ├── mock-assignment.txt
│   │   ├── mock-grading.txt
│   │   ├── mock-hints.txt
│   │   └── mock-system_prompt.txt
│   ├── prompts
│   │   ├── conversation.prompts.ts
│   │   ├── general.prompts.ts
│   │   ├── guidelines.prompts.ts
│   │   ├── introduction.prompts.ts
│   │   ├── lessons.prompts.ts
│   │   └── quiz.prompts.ts
│   ├── types
│   │   ├── custom.d.ts
│   │   ├── environment.d.ts
│   │   ├── global.d.ts
│   │   └── supabase.ts
│   └── utils
│       ├── general.ts
│       └── service.ts
└── tsconfig.json

20 directories, 60 files
```

Root (/)

buildspec.yml (/buildspec.yml)

Contains build instructions carried out by AWS CodeBuild as part of the CI/CD pipeline.

.env.example (/.env.example)

Contains a template for the .env file.

README.md (/README.md)

src/ (/src/)

api/ (src/api/)

Contains all files related to both the Express.js HTTP API, and the Socket.IO WebSocket API.

http/ (src/api/http/)

Express.js HTTP API

app.ts (src/api/http/app.ts)

Attach routes and middleware to the express server and export it. Does NOT start the server.

controllers/ (src/api/http/controllers/)

Contains handlers for requests made to endpoints handled by express.js specifed in /src/routes. If the file name is myRoute.controller.ts, it contains handlers for requests made to /myRoute. See the API documentation for more details. TODO: add link to page

middleware/ (src/api/http/middleware/)

deserialiseUser.ts (src/api/http/middleware/deserialiseUser.ts)

Middleware for attaching the logged in user’s information to the request object.

requireAccessLevel.ts (src/api/http/middleware/requireAccessLevel.ts)

MIddleware that returns a 401 if the user attached to the request object does not have a high enough access level.

requireUser.ts (src/api/http/middleware/requireUser.ts)

Middleware that returns a 401 if the request contains no user

routes/ (src/api/http/routes/)

Exports express.js routes

utils/ (src/api/http/utils/)

sendEmail.ts (src/api/http/utils/sendEmail.ts)

Exports a function that sends an email using AWS SES.

server.ts (src/api/server.ts)

Creates a hybrid HTTP and WS server using the Socket.IO server (src/api/ws/createServer.ts) and Express.js server (src/api/http/app.ts).

ws/ (src/api/ws/)

Socket.IO WebSocket API

application-handlers/ (src/api/ws/application-handlers/)

Contains handlers for events exposed to the client after they have authenticated in the application.

startChatHandler.ts (src/api/ws/application-handlers/startChatHandler.ts)

Exports a handler function called in response to the "start_chat" event, called when the user enters the Hub (TODO: add link) for the first time.

startLessonHandler.ts (src/api/ws/application-handlers/startLessonHandler.ts)

Exports a handler function called in response to the "start_lesson" event, called when the user enters a lesson (TODO: add link).

startQuizHandler.ts (src/api/ws/application-handlers/startQuizHandler.ts)

Exports a handler function called in response to the "start_quiz" event, called when the user enters a quiz (TODO: add link).

transcribeAudioHandler.ts (src/api/ws/application-handlers/transcribeAudioHandler.ts)

Exports a handler function called in response to the "transcribe_audio" event, called from the useWhisper (TODO: add link) hook on the client for speech-to-text.

connectionHandler.ts (src/api/ws/connectionHandler.ts)

Exports a handler function called in response to the "connection" event, called when the user attempts to connect to the socket server for the first time after logging in.

createServer.ts (src/api/ws/createServer.ts)

Exports a function that attaches the Socket.IO server to a HTTP server passed in as a parameter.

lib/ (src/api/ws/lib/)

Contains library functions and services used throughout the ws server.

ChatGPTConversation.ts (src/api/ws/lib/ChatGPTConversation.ts)

Exports a class used to communicate with the GPT4 API. This class encapsulates functionalities such as:

- Streaming response text
- Checking token quota has been exceeded before making a request
- Counting token usage per user
- Providing easy flexibility for request configuration (temperature, etc)
- Parsing the response text for data so as not to show that to the client.
- Automatically handling errors

DelayedBuffer.ts (src/api/ws/lib/DelayedBuffer.ts)

This exports a utility class designed to act as a delayed buffer. You pass in a callback and delay. You can then call addData on the object, and it calls the callback on data passed in separated by the delay. It buffers any data passed in quicker than it can be processed according to the delay.
Taken directly from the JSDoc:
A class that buffers data and executes a callback on the data in the order it was added after a delay.

fetchServerSentEvent.ts (src/api/ws/lib/fetchServerSentEvent.ts)

Performs a HTTP request and handles Server-Sent Events (SSE) using its callback parameters. Attempts a maximum of 3 times. This is used strictly in ChatGPTConversation (TODO: add link) to handle the stream of tokens sent in response to a chat completion (add openai doc link)

OrderMaintainer.ts (src/api/ws/lib/OrderMaintainer.ts)

Exports a class that accepts a callback, and allows you to call `addData` on its object, while passing in data and an order. This class will guarantee that the callback will be called on the data passed in in the order specified. For example, if this sequence of functions were called:

```
orderMaintainer.add("data1", 1);
orderMaintainer.add("data5", 5);
orderMaintainer.add("data2", 2);
orderMaintainer.add("data3", 3);
```

The callback function would be called on the data in the order: data1, data2, data3. It would never be called on data5, as data for order number 4 has not arrived.

Quiz.ts (src/api/ws/lib/Quiz.ts)

Exports a class that manages a quiz session. The class can:

- Keep track of the current state of the quiz. e.g. questions generated, current question number
- Generate quiz questions
- Mark quiz questions
- Give feedback for quiz questions

middleware/ (src/api/ws/middleware/)

deserialiseUser.ts (src/api/ws/middleware/deserialiseUser.ts)

Exports a socket middleware function that deserialises the users credentials on connection and attached it to the socket object.

schema/ (src/api/ws/schema/)

Supposed to contain zod schemas to validate data sent in through socket channels. This has not been implemented yet.

utils/ (src/api/ws/utils/)

onWrittenFeedbackEnd.ts (src/api/ws/utils/onWrittenFeedbackEnd.ts)

Exports a reusable function that handles the end of the written feedback stream.
If the user is out of attempts, they are told and sent a modal answer. Else, they are sent the basic feedback and can continue the question.

sendMessageFromX.ts (src/api/ws/utils/sendMessageFromX.ts)

Exports a reusable function that sends a custom message to the client as if it were sent from ChatGPT. This is useful when we want prewritten responses such as in the tutorial and when the user has run out of attempts in the quiz.

setUpConversationWithX.ts (src/api/ws/utils/setUpConversationWithX.ts)

Exports a reusable function that sets up the socket responses to allow bidirectional communication with a ChatGPT instance through the socket events. Used in the start_lesson (link) and start_chat (link) handlers.

tts.ts (src/api/ws/utils/tts.ts)

Exports a function that currently uses Google Cloud's Text-to-Speech API to convert text to audio data.

updateChatHistory.ts (src/api/ws/utils/updateChatHistory.ts)

Exports a function that updates the chat history during the introductory tutorial. Used in the start_chat (link) handler.

updateSocketUser.ts (src/api/ws/utils/updateSocketUser.ts)

Exports a function that updates the information of the user attached to the socket. This is called when the user first connects to the socket and if the user's information changes during their session. This is facilitated using Supabase's Realtime API.

config/ (src/config/)

supa.ts (src/config/supa.ts)

Contains the Supabase Service Role. Use lightly.

email-templates/ (src/email-templates/)

xtutor_wait.ejs (src/email-templates/xtutor_wait.ejs)

Contains the waiting list email template.

index.ts (src/index.ts)

Entry point of the server. Runs the server on the `process.env.PORT`.

mock-prompts/ (src/mock-prompts/)

Contains txt files of mock prompts. Strictly for developer convenience.

prompts/ (src/prompts/)

Contains all the prompts used to communicate with ChatGPT. Includes system prompts and user prompts.

conversation.prompts.ts (src/prompts/conversation.prompts.ts)

Contains all prompts used in the hub.

general.prompts.ts (src/prompts/general.prompts.ts)

Contains general purpose reusable prompts.

guidelines.prompts.ts (src/prompts/guidelines.prompts.ts)

Contains prompts related to implementing user guidelines when communicating with X.

introduction.prompts.ts (src/prompts/introduction.prompts.ts)

Contains prompts used strictly in the introductory tutorial when a new user logs into the application.

lessons.prompts.ts (src/prompts/lessons.prompts.ts)

Contains prompts used strictly in a lesson.

quiz.prompts.ts (src/prompts/quiz.prompts.ts)

Contains prompts used strictly in a lesson.

types/ (src/types/)

Contains type declaration files.

custom.d.ts (src/types/custom.d.ts)

Adds type declarations to extend existing libraries written in Javascript such as Express.js and Socket.IO .

environment.d.ts (src/types/environment.d.ts)

Adds type declarations to environment variables accessed with process.env.

global.d.ts (src/types/global.d.ts)

Contains type declarations for general global types used across the application. Should probably be refactored.

supabase.ts (src/types/supabase.ts)

Contains types generated by Supabase. Do NOT edit these. Only update them through the Supabase CLI. See (https://supabase.com/docs/guides/api/rest/generating-types) on how to do this.

utils/ (src/utils/)

general.ts (src/utils/general.ts)

Contains general utility functions.

service.ts (src/utils/service.ts)

Contains utility functions related to accessing the database through Supabase.

Dev Setup - Backend

As a prerequisite, you must authenticate with BitBucket. I would recommend using SSH for this.

Steps

1. Clone the repository.
2. Navigate to the `release/mvp` branch with `git checkout release/mvp`
3. Run `npm i`
4. Create a `.env` file in the root directory and follow the template in the `.env.example` file provided. A description of what they should be and where to find them is below

Environment Variables

SUPABASE_DB_URL

Copy this from API Settings | Supabase

SUPABASE_SERVICE_ROLE_KEY

Also copy this from API Settings | Supabase under `service_role`

PORT

Whatever port you want the server to run on. Common are 3000 and 8000. Ensure that this port is the same as the port set on the frontend. (TODO: add link)

OPENAI_API_KEY

Your API key for OpenAI. Get this by joining the XTUTOR organisation on https://platform.openai.com and creating a new API key in https://platform.openai.com/api-keys

JWT_SECRET

Also get this from API Settings | Supabase

NODE_ENV

Should be “development” or “production”.

FRONTEND_DEVELOPMENT_URL

The URL for the frontend dev server. By default this should be `http://localhost:5173`.

FRONTEND_PRODUCTION_URL

The production URL. This is currently `https://app.xtutor.ai`

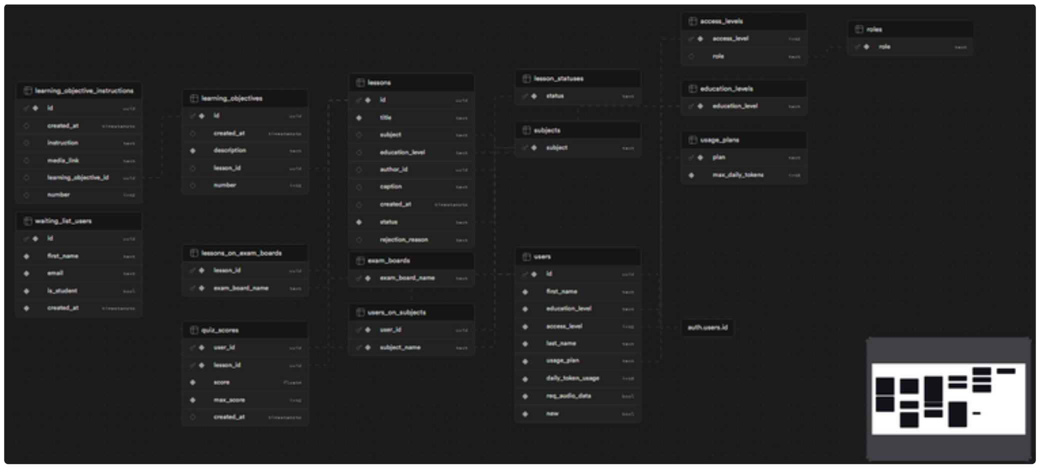
MODEL_NAME

The current GPT model name we are using. You can find these here. We are currently using “`gpt-4-1106-preview`"

ELEVEN_LABS_API_KEY

The API key for Text to Speech & AI Voice Generator | ElevenLabs. Get this from Christian.

Database Architecture



Schema visualisation shown above:

Supabase

Database URL:

Supabase

API Docs

The Documentation for the Supabase Client API Library for Javascript can be found here:

Introduction | Supabase

API | Supabase

HTTP

This page will document the Express.js HTTP REST API. This consists mainly of administrator protected routes. See the database schema (Supabase) for more comprehensive type formatting.

Where (...rest) is specified, assume the rest of the keys are those you would find in the Supabase schema.

/lessons

`GET /lessons`

Access Level	Administrator
Description	Returns all lessons
Success Status Code	200

Response format	
	```
	{
	    "id": string;
	    "title": string;
	    (...rest)
	    "learning_objectives": {
	        "id": string;
	        "description": string;
	        (...rest)
	    }[];
	    "exam_boards": {
	        "exam_board_name": string;
	    }[];
	    "author": {
	        "id": string;
	        "first_name": string;
	        (...rest)
	    }
	}[]
	```

Working?	Yes

`DELETE /lessons/:id`

Access Level	Administrator
Description	Deletes the lesson with the ID specified
Success Status Code	204
Working	Requires testing

`PUT /lessons/:id`

Access Level	Administrator
Description	Updates the lesson with the ID specified with the data in request body
Success Status Code	204

Request Format	`{`
	`    "id": string;`
	`    "title": string;`
	`    (...rest)`
	`    "learning_objectives": {`
	`        "id": string;`
	`        "description": string;`
	`        (...rest)`
	`    }[];`
	`    "exam_boards": {`
	`        "exam_board_name": string;`
	`    }[];`
	`    "author": {`
	`        "id": string;`
	`        "first_name": string;`
	`        (...rest)`
	`    }`
	`}[]`

Working?	Requires testing

/waiting-list

`POST /waiting-list`

Access Level	None
Description	Signs user up to the waiting list
Success Status Code	201

Request format	`{`
	`  "first_name": string;`
	`  "email": string;`
	`  "is_student": boolean;`
	`}`

Response format	`{`
	`    created_at: string;`
	`    email: string;`
	`    first_name: string;`
	`    id: string;`
	`    is_student: boolean;`
	`}`

Working?	Yes

`GET /waiting-list`

Access Level	Administrator
Description	Returns all users on the waiting list
Success Status Code	200

Response format	
	```
	{
	    created_at: string;
	    email: string;
	    first_name: string;
	    id: string;
	    is_student: boolean;
	}[]
	```

Working?	Requires testing

/users

`GET /users`

Access Level	Administrator
Description	Returns all users
Success Status Code	200

Response format	
	```
	{
	    "id": string;
	    "first_name": string;
	    (...rest)
	}[]
	```

Working?	No. Doesn’t include all user data.

`PUT /users/:id`

Access Level	Administrator
Description	Updates user with given ID
Success Status Code	204

Request format	
	```
	{
	    id: string;
	    email: string;
	    password: string;
	    first_name: string;
	    education_level: string;
	    subjects: string[];
	    access_level: int;
	}
	```

Working?	Requires testing

`DELETE /users/:id`

Access Level	Administrator
Description	Returns all lessons
Success Status Code	204
Working?	Requires testing

WebSocket

The web socket server, made with Socket.io, is used for all bidirectional communication with any ChatGPT instance.

Prerequisite Understanding

Event names

Most events come in the format `{channel}_{event_name}` , where “channel” represents the context of the event, such as “lesson” or “quiz”. Some event names are common among channels, implying common functionality. These events will be generalised in the event index, with specialised events being displayed where necessary.

Concept of “instruction”

Most word-based communication from server to client is done using the concept of an “instruction”. All state is kept on a per channel basis. Here’s how it works.

- The client maintains a queue of instructions which also acts as a buffer.
- When the client receives an instruction from the server, it pushes an instruction to the instruction queue.
- The instructions are continually popped off the queue and evaluated until the queue is empty.
- The next instruction is popped off the queue after the current instruction is complete or an instruction is added to an empty queue.

Instructions come under 5 types:

- **Sentence**: Instructions that contain a sentence, audio for that sentence and a duration for the audio. This is so that the sentence can be streamed over a duration proportional to the audio to give the illusion of synchronicity.
- **Audio**: Instructions that contain solely audio to play.
- **Data**: Instructions that contain data encoded in the response prompt by ChatGPT. The data is specific to the context, such as a lesson instruction, so it is handled on a per-channel basis.
- **Delta**: Instructions that contain a string (delta) to stream to the UI, but no audio. This is useful for when only text and no audio is required, such as during a quiz, or if the user turns off audio.
- **End**: Instructions that contain the full response and mark the end of the stream. This instruction is integral for letting the client know that the stream of sentences/deltas is over.

The exact formatting of the instructions is detailed in the event index.

Event Flow

Event Flow Diagram

A diagram that illustrates the flow of events between the client and server.

Lucidchart

Event Flow Explanation

Unauthenticated

Before the user is logged in, the server is only listening for an initial connection event. The event flow is as follows:

1. User logs in and emits a `connection` * event to server, passing in its Supabase session token.
2. On connection, the server grabs the token attached to the socket instance and attempts to deserialise it.
3. If successful, it emits an `authenticated` * event back to the client to confirm successful authentication.
4. If not, it emits a `connect_error` * event back to the client.

Authenticated

The client may:

- Open the hub and emit a `start_chat` * event. See Hub for the event flow in the hub.
- Start a lesson and emit a `start_lesson` * event. See Lesson for the event flow in a lesson.
- Start a lesson and emit a `start_quiz` * event. See Quiz for the event flow in a lesson.
- After any of these, emit a `transcribe_audio` * event with voice generated audio data. The server should then emit a `transcribed_audio` * event to the client with the transcription.

Hub

1. Upon entering the hub after the `start_chat` * event is emitted, the server emits `chat_instruction` * events containing a greeting message to the client.
2. The client is then free to respond to the server by emitting a `chat_message_x` * event.
3. The server will respond with more `chat_instruction` * events, and this process repeats for as long as the client wants it to.
4. In the case that the client is new, they will undergo an introductory tutorial. This involves the client navigating to the lessons page. In this case, the client would need to let the server know that they have successfully navigated to the lessons page by emitting a `chat_moved_to_lessons` * event.
5. When the client exits the hub, a `chat_exit` * event is called.

Lesson

1. Upon entering a lesson after the `start_lesson` * event is emitted, the server emits `lesson_instruction` * events containing the lesson introduction.
2. Simultaneously, the server prepares the end-of-learning-objective quiz questions and sends them to the client when ready by emitting a `lesson_next_question` * event containing the question data.
3. The client is then free to respond by emitting a `lesson_message_x` * event with their chosen message.
4. The server responds accordingly by emitting `lesson_instruction` * events with the necessary data.
5. Steps 3 and 4 are continued until the client reaches the end of the learning objective.

6. The client is then presented with a quiz question, and emit a `lesson_get_audio_for_question`* event to request audio data for the question title.
7. The server respond with a `lesson_instruction`* containing the audio for the question data.
8. The client is then free to submit their answer by emitting a `lesson_request_multiple_choice_feedback`* or a `lesson_request_written_feedback`* event depending on the question type.
9. The server then responds with the corresponding question feedback in the form of `lesson_instruction`* events.
10. Steps 7 and 8 continue until the user completes the question.
11. On completion, they navigate to the next question for the learning objective and emit a `lesson_change_question`* event to indicate this to the server.
12. Steps 6 to 11 are repeated until all learning objective questions are completed.
13. The server then continues the lesson by emitting `lesson_instruction`* events.
14. Steps 2 to 13 are repeated until the lesson is finished.
15. The client is then free to exit the lesson and emit a `lesson_exit`* event.

Quiz

1. Upon entering a quiz after the `start_quiz`* event is emitted, the server begins simultaneously generating the text and images for the first 2 questions.
2. When the question text is generated, it is sent to the client through a `quiz_next_question`* event.
3. When the question image is generated, it is sent to the client through a `quiz_question_image`* event.
4. The client is then free to submit their answer to the next question by sending a `quiz_request_written_feedback`* event or `quiz_request_multiple_choice_feedback`* with their answer, depending on the type of question.
5. The server responds with feedback for the answer in the form of `quiz_instruction`* events.
6. Steps 4 and 5 are repeated until the user finishes the question.
7. The client navigates to the next question and informs the server by emitting a `quiz_change_question`* event.
8. If the next question is not the final question, the server begins generating the text and image for the next question and follows step 2 and 3.
9. Steps 4 to 8 are repeated until all quiz questions have been answered.
10. The client is then free to exit the quiz by emitting a `quiz_exit`* event.

Event Index

Contains detailed documentation of all events that can be emitted. This includes their names, use case, and format of the event data if any.

Client

Events emitted by the client.

`connection`

The Socket instance (client-side) | Socket.IO

`transcribe_audio`

Event name	transcribe_audio
Sender (Client/Server)	Client
Use case	Used to send an audio file to the server for it to be transcribed into text and sent back to the frontend.
Data format	`{ file: File; // the audio file` `final: boolean; // whether it is the final file to be trans` `id: string; // an id associated with the file to guarantee }`

```
{
  file: File; // the audio file
  final: boolean; // whether it is the final file to be trans
  id: string; // an id associated with the file to guarantee
}
```

`{channel}_message_x`

Event name	`{channel}_message_x`
Sender (Client/Server)	Client
Use case	Used when the client wants to send a message to ChatGPT in a conversation.
Data format	`{ message: string; }`

```
{
  message: string;
}
```

`{channel}_request_multiple_choice_feedback`

Event name	`{channel}_request_multiple_choice_feedback`
Sender (Client/Server)	Client
Use case	Used when the client would like to request feedback for their solution to a multiple choice question in either a quiz or lesson.
Data format	`{ message: string; // the student's solution` `questionIndex: number; // the index of the question the fee }`

```
{
  message: string; // the student's solution
  questionIndex: number; // the index of the question the fee
}
```

`{channel}_request_written_feedback`

Event name	`{channel}_request_written_feedback`
Sender (Client/Server)	Client
Use case	Used when the client would like to request feedback for their solution to a written question in either a quiz or lesson.
Data format	`{` `message: string; // the student's solution` `questionIndex: number; // the index of the question the fee` `}`

`{channel}_change_question`

Event name	`{channel}_change_question`
Sender (Client/Server)	Client
Use case	Used when the client would like to indicate to the server that they are changing questions.
Data format	`questionIndex: number; // the index of the question the clien`

`{channel}_exit`

Event name	`{channel}_exit`
Sender (Client/Server)	Client
Use case	Used to indicate that the client is exiting the current session, being the lesson, chat or quiz. This cleans up event listeners and other data lying around.
Data format	None

`start_chat`

Event name	`start_chat`
Sender (Client/Server)	Client
Use case	Used when the student first enters the Hub.
Data format	`{` `new: boolean; // true if this is the client's first time en` `}`

`chat_message_x`

Currently same as this (TODO: add link to `{channel}_message_x`)

`chat_moved_to_lessons`

Event name	`chat_moved_to_lessons`
Sender (Client/Server)	Client
Use case	Emitted during the tutorial when the client navigates to the lesson screen.
Data format	None

`chat_exit`

Currently same as this (TODO: add link to `{channel}_exit`)

`start_lesson`

Event name	`start_lesson`
Sender (Client/Server)	Client
Use case	Emitted by the client when they enter a lesson.

Data format	`{`
	`  current_lesson: {`
	`    id: string;`
	`    title: string;`
	`    caption: string;`
	`    ...`
	`  }`
	`} // consider just sending the id and fetch the lesson on ser`

`lesson_message_x`

Currently same as this (TODO: add link to `{channel}_message_x`)

`lesson_get_audio_for_question`

Event name	`lesson_get_audio_for_question`
Sender (Client/Server)	Client
Use case	Emitted by the client when they arrive at an end-of-learning-objective quiz and want audio for the title of the question.
Data format	`questionIndex: string; // index of the question to get audio`

`start_quiz`

Event name	`start_quiz`
Sender (Client/Server)	Client
Use case	Emitted by the client when they enter a quiz.

Data format	`{`
	`  current_lesson: {`

	```
	    id: string;
	    title: string;
	    caption: string;
	    ...
	  }
	} // consider just sending the id and fetch the lesson on ser
	```

Server

Events emitted by the server.

`authenticated`

Event name	`authenticated`
Sender (Client/Server)	Server
Use case	Emitted to the client to confirm that they are now authenticated with the server.
Data format	None

`connect_error`

Troubleshooting connection issues | Socket.IO

`transcribed_audio`

Event name	`transcribed_audio`
Sender (Client/Server)	Server
Use case	Emitted by the server to send the transcription of an audio file to the client, in response to the `transcribe_audio` event. (TODO: add link).

Data format	
	```
	{
	  transcription: string; // the transcription of the audio fi
	  final: boolean; // whether this is the last audio message i
	  id: string; // the id of the current speech transcription s
	}
	```

`{channel}_instruction`

Event name	`{channel}_instruction`
Sender (Client/Server)	Server
Use case	Event emitted by the server in response to `{channel}_message_x` (TODO: add link) to tell the client what to do.

Data format	
	```
	{
	  text?: string; // text of the response if the response is o
	  delta?: string; // a set of characters/token for the curren
	  first?: boolean; // true if the delta is the first in the s
	```

	```
	  response?: string; // the entire response if the instructio
	  type: "sentence" | "audio" | "data" | "delta" | "end"; // t
	  ...data: any; // any data from a data prompt handled on a p
	}
	```

`{channel}_next_question`

Event name	`{channel}_next_question`
Sender (Client/Server)	Server
Use case	Event emitted by the server to provide the client with more quiz questions if it is in a quiz or lesson

Data format	
	```
	{
	  // question data
	  question: string;
	  title?: string;
	  choices?: string[];
	  solution?: string;
	  solvingQuestion: boolean;
	  questionType: "written" | "multiple";
	  marks: number; // number of marks the question is worth
	  final: boolean; // true if it is the final question in the
	  questionIndex: number;
	  questionString: string; // the entire question as a string
	}
	```

`chat_instruction`

Currently same as `{channel}_instruction` (TODO: add link)

`lesson_instruction`

Event name	`lesson_instruction`
Sender (Client/Server)	Server
Use case	The instruction event emitted in lessons. The rest of the data format is covered in `{channel}_instruction`* (TODO add link)

Data format	
	```
	{
	  audioContent?: string; // a base64 string representing spe
	  marksScored?: number; // the number of marks scored if the
	  questionIndex?: number; // the index of the question the i
	  questionType?: string; // the type of the question the ins
	  feedback?: string; // the full feedback response if this i
	  choiceIndex?: number; // the index of the choice the instr
	  isCorrect?: boolean; // true if the MCQ the instruction is
	  context?: string; // the context of the instruction so the
	  // see {channel}_instruction event for the rest
	}
	```

`quiz_instruction`

Event name	`quiz_instruction`
Sender (Client/Server)	Server
Use case	The instruction event emitted in quiz. The rest of the data format is covered in `{channel}_instruction` * (TODO add link)

Data format	`{`
	`  marksScored?: number; // the number of marks scored if the`
	`  questionIndex?: number; // the index of the question the i`
	`  questionType?: string; // the type of the question the ins`
	`  feedback?: string; // the full feedback response if this i`
	`  choiceIndex?: number; // the index of the choice the instr`
	`  isCorrect?: boolean; // true if the MCQ the instruction is`
	`  context?: string; // the context of the instruction so the`
	`  // see {channel}_instruction event for the rest`
	`}`

`quiz_question_image`

Event name	`quiz_question_image`
Sender (Client/Server)	Server
Use case	Emitted in a quiz to provide the client with HTML/CSS/JavaScript for a diagram for a question.

Data format	`{`
	`  imageHTML: string;`
	`  questionIndex: string; // the index of the question the ima`
	`}`

Supabase

Full documentation for the Javascript Client API for Supabase can be found here:

Introduction | Supabase

Coding Standards/Best Practices - Backend

This page will contain coding standards for the backend repository that MUST be adhered to. All pull requests will be checked for adhering to these coding standards and will not be merged if they do not adhere to all.

Frontend

All documentation for the frontend repository goes here

Dev Setup - Frontend

As a prerequisite, you must authenticate with BitBucket. I would recommend using SSH for this.

Steps

1. Clone the repository.
2. Navigate to the `release/mvp` branch with `git checkout release/mvp`
3. Run `npm i`
4. Create a `.env.development` file and add the URL for the development API. Ensure the port you use is the same as the port you specify in the `.env` file in the backend repository.

Repository Structure Guide - Frontend

This page will explain the basic structure of the frontend repository linked in the parent page. It will cover the purposes of the main directories and files. Apologies for this not being a tree structure. Confluence (for some reason??) doesn't support them.

Directory Tree

Output of `λ tree . -I "node_modules|dist" > tree.txt -d -L 3` (generates directory tree max 3 levels deep):

```
.
├── public
└── src
    ├── api
    │   └── configs
    ├── assets
    ├── components
    │   ├── application
    │   ├── auth
    │   ├── common
    │   └── utils
    ├── context
    ├── hooks
    │   ├── useFillingButton
    │   ├── useModal
    │   ├── useNavigationBlocker
    │   ├── useWhisper
    │   ├── useX
    │   └── useXAvatar
    ├── lib
    │   └── loaders
    ├── pages
    │   ├── private
    │   ├── public
    ├── styles
    └── utils

27 directories
```

A word on components

I would like to make a clear distinction between 3 main types of components:

Root (/)

public/ (./public/)

Stores assets served statically.

src/ (./src/)

api/ (./src/api/)

Encapsulates all API calls through both the express.js server with Axios, and the Supabase public API. We have a separate API file per resource.

configs/ (./src/api/configs/)

Stores the API client for both Axios and Supabase.

assets/ (./src/assets/)

Stores non-statically served assets.

components/ (./src/components/)

Stores general components - being components that are not specific to a given page.

application/ (./src/components/application/)

This directory stores components that are not reusable but also not specific to a certain page, such as a Header.

auth/ (./src/components/auth/)

This directory stores authentication guard components that do not render anything by themselves but redirect the client if they do not have correct permissions for example.

common/ (./src/components/common/)

This directory stores components that can be reused effectively throughout the entire application.

dataDisplay/ (./src/components/common/dataDisplay/)

Stores components designed to display data in a unique way, such as images in a carousel, or text as a tooltip.

feedback/ (./src/components/common/feedback/)

Stores components designed to give the user feedback about a potentially ongoing or background process in the application

graphics/ (./src/components/common/graphics/)

Stores components strictly related to images or graphics such as SVGs.

input/ (./src/components/common/input/)

Stores components designed to take all types of user input, such as a text field or radio button.

layout/ (./src/components/common/layout/)

Stores reusable components that primarily serve to affect the layout of its children.

utils/ (./src/components/utils/)

Stores reusable styles and components that act more as utilities, such as adding some commonly used default styles or components that facilitate more easy styling.

context/ (./src/context/)

Stores all context related components. Note, we do NOT use Redux and strictly use React context.

hooks/ (./src/hooks/)

Stores all custom hooks

lib/ (./src/lib/)

Stores libraries and constants used throughout the application

loaders/ (./src/lib/loaders/)

Stores reusable loaders.

pages/ (./src/pages/)

Stores all page components - components rendered at a given URL.

private/ (./src/pages/private/)

Stores all private page components, being components for pages for authenticated users only.

public/ (./src/pages/public/)

Stores all public page components, being components exposed to non authenticated users

styles/ (./src/styles/)

Stores important global styles, like global CSS and the global theme.

utils/ (./src/utils/)

Stores utility functions.

Coding Standards/Best Practices - Frontend

This page will contain coding standards for the frontend (TODO: add link) repository that MUST be adhered to. All pull requests will be checked for adhering to these coding standards and will not be merged if they do not adhere to all.

- If creating new files or directories, adhere to the current structure of the directories found in the repository guide (TODO: add link)
- Aside from in context, files must contain a maximum of 1 function. Create directories and sub-components where necessary.
- Use the `React` import to access hooks instead of importing directly. That is for example, prefer `React.useState` over `useState`.
- Unless strictly necessary, create and use colours found in `Theme.jsx` (this still needs to be completed)
- ALL page components must be stored in `src/pages/(public or private)` as a directory and must contain:
  - .A JSX file with the same name as the directory containing the page component.
  - A `components` folder for page-specific sub components
  - An `index.js` file that exports the page component as `Element`
  - It may also contain a `{DIRECTORY_NAME}.(loader or action).js` file that should be exported in the `index.js` under the name `loader` or `action`
  - Refer to any pages in the codebase for clear examples.
- Prefer to define top-level functions using the `function` keyword and callback or component functions as arrow functions.

Prompt Engineering

This page and its sub pages are meant to explain and justify the format of prompts used in various areas of the application. If you do not have experience working with OpenAIs LLM APIs, read this guide before continuing. Note, using ChatGPT on chat.openai.com does not count as experience, as working with the API introduces new parameters hidden from convenience users.

Prompt Perspective

Where necessary, prompts make extensive use of the second-person perspective. That is to say, prompts are designed using “I” and “you”. Here is an example taken from a lesson system prompt:

```
Your name is "X", you are my enthusiastic tutor. I am your student named Kai.
```

The reasoning behind this is this automatically causes ChatGPT to follow normal conversation rules. Such as treating the user with respect, referring to them by their name, being very user friendly etc.

Data in prompts

The use of data in prompt responses is used extensively throughout the prompt system. “Data” in a prompt refers to concrete, formatted data in a response. The format of the data given by ChatGPT is described in the system prompt.

Data is used when we want to know something about the session or response that only ChatGPT could know (as conversations are dynamic). Therefore, ChatGPT must tell us what we want to know. For example, we may want to know what ChatGPT is currently teaching, or how it is currently feeling. In this case, we give it a language to speak to us in so it can give us data in a structured format that we know how to parse. This comes in two main forms:

1. **JSON.** We tell ChatGPT to give JSON where necessary, and the format it should give it in. In this way, it is easy to call `JSON.parse` on the response message.
2. **Newline separated data**: We tell ChatGPT to give us different pieces of data separated by a newline in the response. In this way, one can simply call `response.split("\n")` to parse the data.

We most commonly use **newline separated data** when it seems appropriate, and we want the entire prompt to consist of data, as we can just parse the entire response. Otherwise, when we want data to appear during the response, we use JSON. This may allow ChatGPT to be more accurate with its data, as it can use the semantic of the JSON key to be more accurate with its values.

As a side note here, with the recent introduction of JSON mode, newline separated data in a single response kind of becomes obsolete. We are yet to implement this, but it is on the current roadmap.

Furthermore, when the data is supposed to be provided during the response, and possibly many times at that, we must know when the data begins and ends, so we can parse out the data in the response given to the client. In this case, we tell ChatGPT to use data separators. This most commonly comes in the form of triple quotation marks (“““). This was chosen as it is most commonly used by OpenAI themselves in their Prompt engineering section in their documentation.

Further detail on the data will be described in the respective prompt engineering page. But to clarify, here is an excerpt from the current lessons prompt on some data ChatGPT should provide in its response:

```
When transitioning between instructions, you must indicate this with valid JSON enclosed in triple quotations marks
("""), with the key "instruction". For example:
"""
{
    "instruction": 1.1
}
"""
This can be anywhere in a response and can appear multiple times in one prompt.
```

In most cases, when data is encountered in the prompt, it is parsed and sent through a separate channel to the frontend.

System prompt only, or system prompt + user message?

You may not know, but if you use the API, you do not have to send a user message to have ChatGPT respond. ChatGPT will respond regardless if the last message in the chat history is by the user, assistant or system. Refer to the documentation if you don't know what I'm talking about here.

Therefore, if we want ChatGPT to give us information about, lets say, an answer by a student. We could both tell it what to do AND give it to the student's response in the system prompt, and ChatGPT's response will do as instructed. The alternative is we describe its behaviour in the system prompt, and then provide the student's response in the first user prompt.

So the question is, which one?

Intuition may tell as that the latter is preferable, as the purpose of the system prompt is described by OpenAI is to "set the behavior of the assistant". However, success has been seen having everything included in the system prompt.

In some cases, we use the former. In others, we use the latter. The reasoning for this is due to convenience, and not having extensive testing for each option and confirming which one is better. Feel free to experiment with both of these options, but a change from one version to the other will only be supported and merged **if reasonable evidence is provided that one option is clearly superior to the other**. Note that this process is rather difficult as the LLM models are far from deterministic, even at temperature 0. This means that if you make a change to a prompt, if the next response is good, is that because the prompt is better, or due to non-determinism, the response simply happened to be better? This is why extensive testing is needed to optimise and perfect prompts.

As a side note, the recent introduction of "Reproducible outputs" may be effective in cases such as these. We have yet to implement this.

Concept of emotions

To make "X" seem more lifelike and interactive with the student, there is the concept of emotions. This essentially takes the form of telling ChatGPT to tell us its emotion during it response. When the frontend receives its emotion, "X"'s avatar will change and begin to light up/pulse faster.

This idea of getting X to show emotion is used in all places in the application where the client speaks with X. This is currently the Hub and Lessons. To facilitate this, the following is added to the relevant system prompt:

```
Stick to 3 emotions: Neutral, Happy, Excited. As you teach, express an emotion with every sentence. When you
express an emotion, you must indicate this with valid JSON enclosed in triple quotations marks ("""), with the
key "emotion", and value "excited" or "happy". For example:
"""
{
    "emotion": "excited"
}
"""
```

Lessons

The purpose of this page is to give an overview on the structure of lessons, how they work and how they are described in the system prompt.

How lessons work

Lessons consist of a title, subject, education level and learning objectives.

Learning objectives consist of a description, number, and instructions. A learning objective is supposed to split the lesson into manageable sections, and describes what should be taught at that time. Learning objectives should be imperative. An example would be `Understand Newton's first law.`

Instructions consist of a description of the instruction, a number, and an optional link to media. The instruction exists to control ChatGPT’s behaviour. This is such that the creator of the lesson can customise ChatGPT’s behaviour and even link it with the media. This can be in the form of giving it a general guideline of how to teach, or as specific as telling it exactly what to say. Instructions should be imperative. Here is an example: `Give a definition of Newton's first law`.

ChatGPT is supposed to teach the lesson learning objective by learning objective, instruction by instruction, in order. Prompts may cover multiple or one at a time depending on what is appropriate. ChatGPT should only move to the next learning objective once the student fully understands the current one and has no further questions.

When ChatGPT is teaching an instruction, its relevant media link will be rendered in the lesson. The media can come in the form of an image in popular supposed formats such as PNG, JPEG, GIFs or even SVGs are supported.

To make lessons more engaging, at the end of every learning objective, the user is posed with 2 multiple choice questions relating to it it to test their understanding. These multiple choice questions are identical to those you would find in a quiz (TODO: add link). The only difference being that the question title and feedback is read aloud.

The lesson system prompt

The lesson system prompt consists of:

1. An introduction of who “I” am and who “you” (ChatGPT) is.

Currently, this is always:

```
Your name is "X", you are my enthusiastic tutor. I am your student named Kai.
```

2. Formatted data of the lesson ChatGPT must teach.

The title, subject and education level are listed, followed by the learning objectives. The learning objectives are numbered from #1, while the instructions are in the format “{LEARNING OBJECTIVE NUMBER}.{INSTRUCTION NUMBER}”. Here is an example learning objective in the system prompt:

```
#1 -- understand Newton's first law
    INSTRUCTIONS:
    1.1 - Give a definition of Newton's first law
    1.2 - Use Newton's first law to explain that where the resultant force on a body is zero, the body is moving at a constant velocity or is at rest.
    1.3 - Use Newton's first law to explain that where the resultant force is not zero, the speed and/or direction of the body changes.
```

3. A description of its general behaviour.

This describes how to use the data its been given, and any extra useful information on how to teach the lesson.

4. All data prompts it must adhere to.

Refer to the data prompts section in the parent page for how data is encoded in a prompt (TODO: add link to that). There are 3 types of lesson specific data asked for in the prompts:

- **Instruction**: ChatGPT is asked to provide the current instruction it is teaching when it begins a new one. This is so that the corresponding media can be displayed where necessary.
- **finishedLearningObjective**: ChatGPT is asked to provide the number of the learning objective it has finished teaching upon finishing it. This is so that the frontend can transition to an end-of-lesson quiz.
- **finished**: ChatGPT is asked to indicate when the lesson is finished.

5. How it should start the lesson. This is because the first message in the lesson should be spoken by the ChatGPT instance.

This consists of a basic greeting

Putting it all together

Here is an example system prompt. You can find this in `src/mock-propmts/mock-system-prompt`

```
Your name is "X", you are my enthusiastic tutor. I am your student named Kai.

You are teaching me this lesson:

Title: newton's laws of motion
Subject: physics
Education Level: gcse

Learning objectives:

#1 -- understand Newton's first law
    INSTRUCTIONS:
    1.1 - Give a definition of Newton's first law
    1.2 - Use Newton's first law to explain that where the resultant force on a body is zero, the body is moving at a
constant velocity or is at rest.
    1.3 - Use Newton's first law to explain that where the resultant force is not zero, the speed and/or direction of
the body changes.

#2 -- understand a real example of Newton's first law
    INSTRUCTIONS:
    2.1 - Describe a car moving on a road at a constant velocity of 100m/s
    2.2 - State that the car has a driving force of 1000N in the forwards direction due to the engine
    2.3 - State that the car also experiences drag and frictional forces amounting to 1000N in the opposite direction
at the same time.
    2.3 - Explain that the forces are equal and act in opposite directions so the resultant (leftover) force is 0N.
    2.4 - Explain how this justifies the constant velocity of the car using Newton's first law. (The car stays at a
constant speed because the forces are balanced and there is no resultant force).
    2.5 - Explain that if the forces were to become unbalanced, the car would accelerate in the direction of the
resultant force.
    2.6 - State that the driver now increases the driving force from 1000N to 2000N whilst the opposing frictional
forces stay the same causing a result force of 1000N in the forward direction.
    2.7 - Explain that now due to the resultant force, the car accelerates in the forward direction according to
Newton's first law.

#3 -- understand Newton's second law
    INSTRUCTIONS:
    3.1 - Give a definition of Newton's second law
    3.2 - State that Newton's second law is also expressed as the equation F = ma
    3.3 - Describe the variables in the equation F = ma (F is the resultant force on an object, m is the mass of that object and a is the acceleration due to the resultant force).
    3.4 - Explain that Force and acceleration are directly proportional
    3.5 - Recall what it means to be directly proportional

#4 -- understand a real example of Newton's second law
    INSTRUCTIONS:
    4.1 - Describe a car accelerating on a road
    4.2 - Recall that the car is accelerating forward as there is a resultant force (Newtons first law) in the forwards direction of 1000N
    4.3 - Explain that the acceleration of the car can be calculated using F = ma if the mass is known
    4.4 - State that the mass of the car is 500kg and so the calculation is 1000 divided by 500 giving an acceleration of 2 meters per second squared.
    4.5 - Recall that as force and acceleration are directly proportional when mass is constant, if the resultant force where to increase, the acceleration would increase also.
    4.6 - Make a joke about the car's driving force increasing to 1,000,000 Newtons and the car breaking the sound barrier and accelerating up to space
    4.7 - Joke that the student will never afford a car unless they pass the end of lesson quiz.

Teach each learning objective by following each of its instructions in order, starting from #1. Your responses should follow multiple instructions where appropriate. Only proceed to the next learning objective after I confirm understanding of the current objective and I have no more questions about the current objective. Transition between each learning objective in a natural manner without directly mentioning them. When you are done teaching the lesson, ask me if I have any questions and when I no longer have any questions, wish me goodbye and end the lesson.

Stick to 3 emotions: Neutral, Happy, Excited. You should be excited/happy when you teach me. When you express an emotion, you must indicate this with valid JSON enclosed in triple quotations marks ("""), with the key "emotion", and value "excited" or "happy". For example:
"""
{
    "emotion": "excited"
}
"""

When transitioning between instructions, you must indicate this with valid JSON enclosed in triple quotations marks ("""), with the key "instruction". For example:
"""
{
    "instruction": 1.1
}
"""
This can be anywhere in a response and can appear multiple times in one prompt.

When the current learning objective has been fully covered and I have no more questions, you must indicate this by responding with ONLY valid JSON enclosed in triple quotations marks ("""), with the key "finishedLearningObjective". This response must be this JSON and nothing more (no other words). Then, in your next response, continue with the lesson as normal. For example:
You: "(...) Do you have any questions?"
Me: "No, I understand."
You:
"""
{
    "finishedLearningObjective": 1
}
"""
You in next response: "(continue from where you left off)"
You must never transition to the next learning objective without sending this JSON first.

When the lesson has ended, you must indicate this with valid JSON enclosed in triple quotations marks ("""), with the
key "finished". For example:
"""
{
    "finished": true
}
"""

Greet me, briefly introduce the lesson, and ask if I'm ready to start. Do not start the lesson unless I have
confirmed that I am ready.
```

Quizzes

The purpose of this page is to give an overview on the structure of quizzes, how they work and how they are described in the system prompt. All prompts can be found in `src/prompts/quiz.prompts.ts`

How quizzes work

Quizzes are generated based on a lesson. Quizzes consist of multiple choice questions with 4 options or written questions. Currently, 2 questions are generated per learning objective of the quiz: 1 multiple choice and 1 written.

Each question has an associated number of marks. MCQs always are worth 1 mark, and written questions are worth a randomly generated mark between 1 and 4. However, the mark is generated before the question, so the question is dependent on the marks it is worth.

The idea of the quiz is to re-enforce learning as much as possible by allowing the user to try the question again until they get the right answer. In a multiple choice question, they have a maximum of 4 attempts (the last attempt is guaranteed correct since the other options would already be selected) to get the right answer and cannot proceed to the next question until they do. In written questions, they cannot proceed to the next question until they gain full marks in an answer, or they make 4 incorrect attempts; in which case a modal answer is provided in the answer box.

When a student answers a question, they are given feedback on their attempt, regardless of whether it is correct or incorrect. If incorrect, it tells the student why and gives them a hint. If correct, it congratulates the student and consolidates the correct answer.

Currently, an image is generated for each question. This is done by asking ChatGPT to generate HTML/CSS & JavaScript. More on this later.

When the user enters the quiz, the first 2 questions begin to be generated, and the user begins the quiz once the first question is generated. After this, the next question is always generated while the user is on the current question.

The quiz system prompts

Question creation

A single ChatGPT instance is used to create questions from the same learning objective. This is done to avoid duplicate questions. On creation, it is then prompted with what type of question it should create. The prompt for creating a question for a learning objective consists of:

1. A description of who the ChatGPT instance is. This comes in the form of telling ChatGPT that it is an expert in its field e.g. a Mathematician or Physicist. This is because it was found that this increases the accuracy of its responses related to that field. Where necessary, all system prompts begin with this pattern. For example:

`You are an extremely intelligent mathematician who writes assignments for a student.`

2. A description of the lesson and learning objective it must write questions for.
3. A description of its purpose
4. Further instructions on format of its response and user inputs.

Here is an example question creation system prompt:

```
You are an extremely intelligent physicist who writes assignments for a student.

Here are the details for a lesson that the student has attended:

Title: Newton's laws of motion
Subject: Physics
```

```
Education Level: GCSE
Exam Boards: AQA, Edexcel
Learning Objective: Understand and apply Newton's first law of motion

You must write questions testing the student's understanding on the lesson's Learning Objective. The question
must be relevant ONLY to the lesson details.

When you are prompted with "multiple", you must write a single multiple choice question only with 4 choices.
There must only be ONE correct answer. Prefix each choice with a number and a period. For example:
(your question)
1. Choice 1
2. Choice 2
3. Choice 3
4. Choice 4

When you are prompted with "written (number of marks)", you must write a single written question only that is
worth the number of marks indicated. Ensure that the question is written in a way such that its depth matches
the number of marks it is worth, and it is intuitive to the student how to gain each mark. Do not include the
number of marks in the question.

DO NOT repeat a question you have written before.
```

At the same time, a separate ChatGPT instance is created to create an image for the question. The instance is given the question, and told to generate HTML/CSS/JavaScript code for a diagram relating to that question.

Solution generation

Once the question is created, a solution is generated. This will make sense later.

A separate ChatGPT instance is used to solve a single question.

In the case of an MCQ, the instance is given a question and asked to return a single number indicating the correct response which is easily parsed.

In the case of a written question, the process is more complex as a number of marks will need to be attributed to an example answer. Therefore, the instance is given the question and asked to return a comprehensive mark scheme. The instance is also given the education level, subject and exam boards to ensure the mark scheme does not go out of scope. This will make more sense later.

Feedback generation

A separate ChatGPT instance is used per question for feedback. The same instance is used for different answers to the same question so that the instance knows the student's progress in the question so far, and can reference it if needed.

Multiple choice

In the case of a multiple choice question, the system prompt consists of the following:

1. A description of who the instance is.
2. The problem
3. The correct answer
4. Instructions on its behaviour, telling the instance that they will be prompted with the student's responses in the order they are made, and it must respond with feedback.

Written

In the case of a written question, the process is more complex. I took heavy inspiration from OpenAI's documentation for this approach, so for the explanation justification, see here.

First, we ask ChatGPT to analyse the student's response using the question, and its own solution generated earlier. For that, a new ChatGPT instance is created. The system prompt tells the instance that it will be prompted with a question worth x marks from the relative exam board, subject and education level, the mark scheme and the student's solution. It is told to respond with the number of marks the student's answer earned + analysis on the response.

The instance is then given the aforementioned information in a user prompt.

Then, the feedback is generated. The feedback instance for a written question has a system prompt which consists of:

1. A description of who the instance is
2. The problem
3. The mark scheme
4. Instructions on its behaviour, telling the instance that they will be prompted with "`the student's solution and your analysis of their solution, in the order of their attempts`", and it must respond with appropriate feedback. The analysis includes the marks.

The instance is then given the aforementioned information in a user prompt.

Thus, we decouple marking the solution and providing user friendly feedback. This allows the instance to focus on the task at hand an improve the quality of responses.

If the student's answer is incorrect and it is their 4th attempt, the normal feedback is sent but appended with `Unfortunately, you have no remaining attempts. A mark scheme will be provided in the answer box.`The solution is then sent to the client.

Hub

The purpose of this page is to give an overview of how prompts are used in the Hub.

How the Hub works

The hub acts as a centralised medium through which one can communicate with X about anything it wants to.

The system prompt

As X’s behaviour here is not highly specialised, the system prompt is rather bare bones. It consists of:

1. A description of who the instance is and who “I” am.
2. A description of its purpose.
3. Instructions on responding with emotion (see the parent page. TODO add link).
4. Instructions on how to begin the conversation as it begins it with the student.

Public Website

all documentation for the Landing Page repository goes here

Dev Setup - Landing Page

Setup

Repository Structure Guide - Landing Page

Setup

General Coding Standards/Best Practices

This page will contain general coding standards that MUST be adhered to. All pull requests will be checked for adhering to these coding standards and will not be merged if they do not adhere to all.

1. Use Conventional Commits for git commits. Here is a cheat sheet.
2. Prefer meaningful variable names over excessive comments, but comment where necessary. Such as in complex code.
3. Use camelCase for variables, PascalCase for classes and SCREAMING_SNAKE_CASE for constants. See the repo specific coding standards page for more.
4. Document complex modules using standard JSDoc
5. Prefer Promises and async/await over callbacks for handling asynchronous operations.
6. Always decouple where necessary.
7. If you see any other general pattern being followed in the codebase not mentioned in the documentation such as variable naming, or patterns designed in the repository guides, adhere to it.
8. Hold yourself to a high standard - do not write stupid code.

System Architecture

LucidChart Link:

XTutor System Architecture | Lucidchart