NEW PARTS OF THE CODE RELATED TO:  
USER ACCOUNTS AND SESSION STATE MANAGEMENT  
  
—  
  
[BACKEND]:  
  
## ‘Main.py’  
  
- Now includes configuration statements for secret key for token generation, and database uri for SQLite database setup

app.config['SECRET\_KEY'] = os.environ.get('SECRET\_KEY')

app.config['SQLALCHEMY\_DATABASE\_URI'] = os.environ.get(

'SQLALCHEMY\_DATABASE\_URI')

Note: secret key is a random string saved in environment variables (secrets). And the sqlalchemy database uri is just sqlite:///app.db which creates a local database file when used. Database get’s initially created at the bottom of main.py here:  
  
 with app.app\_context():

db.create\_all()

## ‘routes.py’

- new routes added for the following:

– /register calls chat\_services to render the register.html template

– /register (POST method) should receive request from frontend to register a new user

– /settings calls chat\_services to render the settings.html template

– /login (POST method) receives request from frontend to verify an existing user when they login (and should return a token to frontend which is used in all subsequent requests to access user data and state)

– /update\_api\_key (POST method) receives frontend request, specifically from a script on the settings.html page to set the user’s api key when submitted. Storing it on the backend within the user model on the database

– /get\_user\_state (GET method) receives frontend request to get the current state of the app for the user. This includes an object full of variables that are used throughout the frontend javascript, and is replacing our previous use of the browser’s local storage.

– /set\_user\_state (POST method) this is the counterpart to the last endpoint, and sends the global state object from the frontend, to the backend, to update the state (thus allowing updated information to be retrieved by other frontend functions later).

- existing routes /run, /start, and /list-assistants are nearly the same as live version, but have been updated to pass an additional parameter ‘app.config’ to their respective functions in chat\_services. (used to get user data for subsequent logic).

## ‘chat\_types.py’  
  
- remains the same as live version. Includes duo (defined as duo\_i) and x\_poll (requires some minor updates to accept entire agent objects instead of just assistant\_ids)

## ## ‘tools.py’

- remains the same as live version. Includes ‘giint’ (which runs the main duo), ‘awaken\_assistant’ (which can be called by user proxy to make make a domain specific version of the main duo agent using procreate and the agent factory), ‘giint\_flow’ (can be called by the main duo agent to make domain specific progeny agents via procreate and agent factory), ‘SDNA\_Progenitor’, ‘SDNA\_Chaining’, and ‘SDNA\_Morph’ (all create agents with assistant based on their respective instructions, and does a run on a new thread with them given any input), ‘procreate’ (chains the 3 SDNA guys together with some specific handling logic, and returns the new instructions which can be used by agent factory to create new giint agents with assistants that include those instructions).  
  
  
  
## ‘utils.py’  
  
- same as live version. Just as utility functions used throughput rest of code.  
  
  
  
## ‘chat\_services.py’  
  
- renders the 4 html pages and…

- ‘handle\_start’ function

– should accept token from frontend, and use that and the app.config to get the user object, and then uses the api\_key stored for that user to initialize the open ai client like this:‘client=OpenAIClientWrapper(user.api\_key)’. Then it creates a GiintService which allows us to call the method ‘run\_giint’ which if called WITHOUT providing a giint\_id (as is the case here) immediately creates a new giint\_thread (with open ai threads included) and returns the giint\_id for it. This giint\_id would then go back to the frontend, get stored in global state as a sessionId (with a particular window number attached) and then that global state is also then supposed to be sent back to the backend to update the persistent backend state as well via those other routes)

- ‘handle\_run’ function

– accepts token and config, as well as a giint\_id, message, and assistant\_id (optional) from the data payload of request body. Just like in the start, the users api key is retrieved to the create the client (the client is passed along through all of the logic from this point on). Then the GiintService, and we call run\_giint, but now with a message and giint\_id, which allows the previously saved giint\_thread to be retrieved, reinstanced (have a main agent added to it if one is not already there), and then executes a run with the message, and returns the entire thread message list back to the handle\_run, which is then processed to extract the most recent assistant messages (since there may be more than one), and then formats it to be returned to frontend via route.  
  
- ‘handle\_list\_assistants’ function

– just like before, but now uses the token to get the user’s api. Gets all the assistants, and returns formatted list. The actual ‘get\_assistants’ method within the OpenAIClientWrapper contains the logic to iterate through multiple pages of assistants to ensure that you get all of them if there are more than 100.  
  
  
  
## ‘openai\_client.py’  
  
- no change. Same as live version.  
  
  
  
## ‘models.py’  
  
- this is a NEW module which initializes the database using SQLAlchemy  
- then a ‘User’ class is defined as a subtype of ‘db.Model’  
– The user class has attributes for ‘id’, ‘emai’, ‘password’, ‘api\_key’, and ‘state’  
– it also has methods for setting the password, checking the password, getting the state, and setting the state.  
— apparently it does not need an \_\_init\_\_ method because of the inheritance from db.Model.  
  
  
  
## ‘auth\_services.py’  
  
- this is a NEW module which has several functions used for authentication of user requests.  
- generate token, verify token, register user (to make a new user model), login user (to generate a token for the frontend session whenever a user logs in), and update api key

## ‘giint\_agent.py’  
  
- remains unchanged from live version

## ‘giint\_thread.py’

- unchanged

## ‘giint\_factories.py’  
  
- unchanged. Used to make giint agents and threads. Abstraction from the actual agent and thread classes allows for much cleaner code and less problems. Factories are used within the giint service.  
  
  
  
## ‘giint\_repositories.py’  
  
- unchanged. Used to retrieve giint\_thread and giint\_agent objects that were made and saved previously.

## ‘giint\_service.py”  
  
-unchanged. Handling for all the other giint classes, and defines the run\_giint method which is the main agent chat logic.

[/BACKEND]

[FRONTEND]:  
  
  
## ‘index.html’  
  
- has been updated to accept an email (username) and password (instead of api key).

- And a new button has been added to redirect to the ‘register’ page for making new user accounts

## ‘chat.html’  
  
- almost exactly the same BUT…

- new button has been added to redirect you to the ‘settings’ page where the user can submit and set their open ai api\_key

## ‘register.html’

- NEW page includes fields for username (email) and password, and a submit button

## ‘settings.html’  
  
- NEW page includes field to input api key, and button to submit it  
  
  
  
## ‘api.js’

- all of the fetch request to the backend have been updated to include an authorization header containing the token that is used on backend to actually access the user model. This is done as follows:  
– 1st the token value is set like this: const token = sessionStorage.getItem('token');

— (the token will already be in sessionStorage once the user is on the chat p[age because that is what is returned from the backend login request that is run when they initially log in)

– 2nd the authorization statement is added to the request header like this: 'Authorization': `Bearer ${token}`

- Note: ALL functions which previously access ‘localStorage’ now access the ‘globalState’ object, either directly or by using the ‘setStateValue’ and ‘getStateVale’ functions.

- Also new functions have been added:  
– ‘getUserState’, ‘getUserState’, ‘register’, ‘login’, ‘updateApiKey’

— they all acces their respective endpoints to perform those actions.

- Final Note: the functions ‘fetchConsoleMessages’, ‘fetchTranscripts’, and ‘sendTitleToBackend’ are still outdated, and don’t access anything on the backend.

## ‘chat.js’

- The ‘startNewChat’ function has also been updated to provide the token within the authorization header of the fetch request.

- Note: ALL functions which previously access ‘localStorage’ now access the ‘globalState’ object, either directly or by using the ‘setStateValue’ and ‘getStateVale’ functions.

## ‘chatEntryPoint.js’  
  
- This is the main script that runs the ‘chat.html’ page. It has the event listener that sets up the entire page once DOMContentLoaded, and calls all of the rest of the functions it needs to retrieve the state, and then rebuild all the windows and load in their states, and add event listeners to elements, etc…

- it also has the window event listener ‘beforeunload’ which captures various state values and saves them before the page is unloaded (refreshed or closed). Basically all the global state values should be set up to have their state saved and persisted to backend during the course of the functions that intercat with them, so there is probably some unnecessary stuff in this block, BUT…

– The before unload event listener most importantly saves the openWindowsCount which counts the window number, which can then be used to re-instance all of the windows that were open before (and their states, already saved and numerically associated) when the page is loaded again, even if the user didn’t explicitly “save” the window (which uses a different vale “savedWindowState” to keep track of those explicitly saved windows)

- Note: ALL functions which previously access ‘localStorage’ now access the ‘globalState’ object, either directly or by using the ‘setStateValue’ and ‘getStateVale’ functions.

## ‘localStorage.js’  
  
- just includes one function, which should probably be refactored into a different module.

– also this one function isn’t even really used, since the transcripts are not being fetch properly anyway. But it has been updated to use global state getter and setter functions nonetheless.  
  
  
  
## ‘loginEntryPoint.js’

- This is the main script that runs on the login page, ‘index.html’.  
- it sets up the event listeners for the elements, and then calls the login function upon submit.  
– (or redirects to register.html on button click)

## ‘state.js’  
  
- pretty important module!  
- include the ‘globalState’ object which contains all of the variables that need to be used and updated throughout the app. That were previously just saved into localStoage naked  
- the persistentStateProperties is a set of keys that are referenced by the setStateValue function and specifically filter the state values down to the list of them which need to be persisted to the backend model (those not on the list but still part of the globalState object are ephemeral and don’t logically need to be persisted between sessions)  
  
- ‘getStateValue’ function

– returns the value of the globalState object at the provided key

- ‘setStateValue’ function

– sets the value of a particular key within the globalState, and can be called with the “syncWithBackend” flag set to TRUE, which will then also automatically call the ‘setUserState’ function with the session token to persist the state information to the backend  
  
- ‘fetchAndInitializeUserState’ function

– supposed to use the token to call ‘getUserState’ and get the sate object from backend  
– Then it is supposed to take that state and call ‘initializeGlobalStateFromBackend’ with it to set the globalState object on the frontend for use by the other functions (via getter and setter explained above)

- ‘getUserState’ function  
– this gets the user state from the backend.   
– Note: I think this is the same as the function of the same name within the api.js module, and so one should be deleted probably

- ‘initializeGlobalStateFromBackend’ function

– takes userState argument and sets the globalState on the frontend by calling the ‘setStateValue’ function

- ‘updateBackendState’ function

– supposed to update the backend state using the token and global state as arguments for the ‘setUserState’

- ‘loadSelectedHistoryIndexes’ function

– this function should not even be in this module, and should be moved to utils probably  
– purpose is to get the saved indexes of which history from the dropdown of is selected for each of the open chat windows.

## ‘style.css’

- mostly unchanged.  
- added a style for the .nav-button  
- needs more work to both organize, modularize, and add styles for the elements on register and settings page  
  
  
  
## ‘ui.js’

- mostly unchanged  
- Note: ALL functions which previously access ‘localStorage’ now access the ‘globalState’ object, either directly or by using the ‘setStateValue’ and ‘getStateVale’ functions.

- includes the very important function ‘initializeApplicationState’ which is called by the DOM event listener on the chatEntryPoint.js script.  
– it needs to get backend state for setting up everything else.   
– Should probably refactor between these 2 scripts

## ‘utils.js’

- mostly unchanged  
- Note: ALL functions which previously access ‘localStorage’ now access the ‘globalState’ object, either directly or by using the ‘setStateValue’ and ‘getStateVale’ functions.

## ‘registerEntryPoint.js’

- NEW module that runs the main script for the new register.html page

- sets up event listeners.  
- calls the ‘register’ function with the provided email and password

## ‘settingsEntryPoint.js’  
  
- NEW module that runs the main script for the new settings.html page  
- sets up the event listener, and updates api key on backend model when submitted.  
- Note: this should be refactored to actually USE the ‘update\_api\_key’ function, instead of defining the fetch request directly in the script as it is now.