Design:

dPhilosophical:

Our design has three important terms when it comes to defining our approach. These are **content**, **categories**, and **rooms**. More detail can be found in the glossary, but a brief description is as follows. **Content** is the data which is published by the users of the application. Every piece of content has a **category**, which is essentially a way to describe what the content is. Finally, each **room** has one category associated with it, and only content that belongs to that category can be sent in said room.

One of the priorities when it came to this project was extensibility, and the notion of categories is important to achieving this goal. By forcing all content to have an associated category, the frontend can delegate to appropriate strategies when it comes to both publishing and rendering content. Delegation for publishing is required to account for all the possible future categories.

For example, instead of attempting to design a single communication method capable of sending any data type, our solution was to instead design around only one data type. Then, all new categories must come with a new strategy to encode its content into the single data type. This means the frontend only needs to send and receive that single data type, as the render strategy can then turn the encoded content back into a viewable state.

Delegation to strategies also extends to how new content is entered into the client to be published, with the GUI delegating to strategies based on what category the content belongs to. For example, a Text category would most likely use a text box, while an Image category might use a file picker. If a new type of category is desired, all that is required is a new strategy.

Implementation:

Our delegation strategy is implemented through the usage of mappings from category types, stored as strings, to the various strategies. This leads to even more extensibility, as a new category only requires the addition of several new strategies to the frontend. In the extreme case, it is theoretically possible for categories to be dynamically added to the mapping rather than being statically defined in the frontend. This would necessitate some further work, as new API functions in the frontend and back end as well as collections in the database would be required, but no fundamental design reworks need occur.

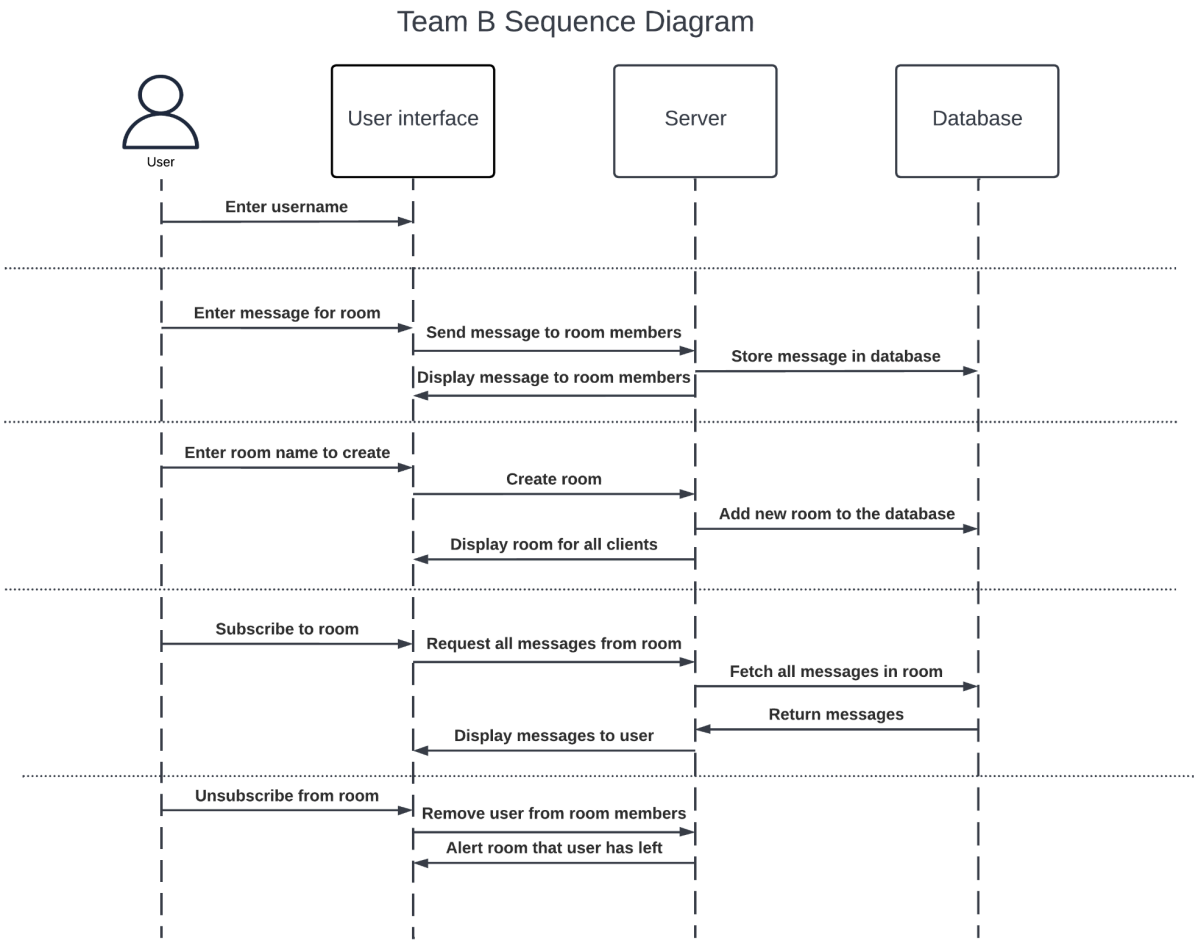
In more broad terms, our frontend is built using Reactjs. This allows our design to easily react to changes through the usage of states. This is not perfect, since there are some difficulties with asynchronous content and React, as described here. However, they are not insurmountable as that same link shows.

For the actual communication between the frontend and backend we used SignalR. SignalR was a good choice in our opinion because it simplified the creation of the API endpoints compared to a REST API, as the various functions are the endpoints. Additionally, SignalR automatically provides the notion of a connection, which each tab connected to the backend has. This obviates the necessity of creating any form of user ID, as the connections already represent this information. SignalR also had Groups, which are essentially a pool of connections. This maps nearly perfectly to our definition of rooms, especially when considering that messages can easily be sent to every member in a group at once. Finally, all these tasks can generally be done asynchronously, which is especially beneficial when it comes to managing multiple users.

Use Case Diagram:

1-20 Use Case Diagram

Sequence Diagrams:



https://lucid.app/lucidchart/cf52ffd3-1d93-44a4-a557-63108c55035c/edit?viewport_loc=-388%2C-85%2C3072%2C1577%2C0_0&invitationId=inv_d4d84051-6a4e-46df-ac03-29e11ce06769

Pros and Cons of Selected Design:

React

Pros - Component-based design (reusable components), uses virtual DOM (faster, better performance, only updates necessary states), easy to debug, easy to integrate with ASP.NET application,

Cons -

- Asynchronous pushing of data requires workarounds
- State based architecture requires new design considerations

Performance Metrics:

Each client when initially loading the page requires about 1.1 MB from the frontend. After the initial load, all further requests are to the other services, like the backend. This gives us our base measure for calculating performance.

An Azure Static Web App is given 100 GB of free bandwidth to start out. This is enough to perform nearly 88,000 loads for clients. Afterwards, there is a cost of $0.20 for each additional gigabyte, which works out to $1 for nearly 4,400 additional loads. Additionally, the paid hosting plan gives access to enterprise-grade edge, which will provide several benefits, such as DDoS protection and caching closer to cities which will generally increase access speeds.

This means that the limiting factor when it comes to the frontend scaling is how much money can be spent to pay for Azure, not any intrinsic limit to the frontend.

API Endpoints:

Callers

- joinRoom(String room) -> JoinRoom(string room, string user)
  - String room - The name of the room to join
  - String user - The name of the user who is joining
  - Calls the backend function JoinRoom, indicating that the user wishes to be subscribed to the indicated room.
- leaveRoom(String room) -> LeaveRoom(string room, string user)
  - String room - The name of the room to leave
  - String user - the name of the user who is leaving
- addRoom(String room, String category) -> AddRoom(string room, string category)
  - String room - The name of the room to create
  - String category - The name of the category for the new room

    - Calls the backend function AddRoom, requesting that a new room be created with the specified name and category.
- sendMessage(string room, string message) -> SendMessage(string room, string user, string value)
    - String room - The name of the room to send the message to
    - String message - The encoded form of the message
    - Sends a message, i.e. publishing content, to the specified room with specified message.
- getRooms() -> GetRooms()
    - Calls the backend function GetRooms, requesting all rooms

Callees

- ReceiveRooms(List<Room> rooms)
    - List<Room> rooms - rooms received by client from the backend
    - Called by backend on initial load, and when new rooms are created
- ReceiveSendMessages(List<Message> messages, string room)
    - List<Message> messages - messages received by client from the backend sent by other users
    - String room - room of the messages
    - Called by backend when another user sends a message in the specified room the user is in
- ReceiveUpdateMessages(List<Message> messages, string room)
    - List<Message> messages - messages received by client from the backend caused by users joining or leaving room
    - String room - room that was left or joined
    - Called by backend when another user joins or leaves the specified room the user is in
- ReceiveGetMessages(List<Message> messages, string room)
    - List<Message> messages - messages received by client from the backend caused by the user subscribing to room
    - String room - room that was subscribed to by the user
    - Called by backend when the user subscribes to the specified room, receiving all messages from the room