Voice Dev 2

CS571: Building User Interfaces

Cole Nelson

Today's Warmup

- Clone today's code to your machine.
  - Run the command `npm install` inside of the `starter` and `solution` folders.
- Create and train the following intents in wit.ai...
  - `get_comments` e.g. "get 4 comments"
  - `login` e.g. "log me in"
  - `logout` e.g. "log me out"
  - `create_comment` e.g. "create a comment"

Announcements

- HW11 will be due on Friday, May 3rd at 11:59 pm, regardless of late days.
  - It *cannot* be submitted after this time.
  - It will be worth **5 points** instead of 4 points.
  - HW12 (2 pts) has been removed!
    - The HW will be done as an in-class exercise.
    - Enjoy another free point (:
  - **Start early!**

Announcements

1 total bonus point will be available...

- +0.5 pts for completing the HCI Workshop Quiz
  - Released 4/24, submit by 5/3.
- +0.5 pts for completing the security CTF
  - Released 5/2 & submit by 5/3.
  - We will complete it in-class, more reason to watch/attend the lecture! :)

There is no rounding of grades.

Learning Objectives

1. Solidify understanding of chat agent implementation.
2. Be able to keep the context of the conversation.
3. Be able to delegate to subagents.
4. Be able to appreciate other neat features, such as traits, speech-to-text translation, and text-to-speech synthesis.

Key Concepts in Wit.AI

- **Agent:** The overarching project consisting of *intents*, *utterances*, and *entities*.
- **Intents:** A higher level meaning of many *utterances*.
- **Utterances:** A string of words.
- **Entities:** Special attributes of an *intent*.

The goal of our **agent** is to extract the **intent** and any **entities** out of a new **utterance** and map it to a function.

Intents

Consider the following **utterances**...

- What is the weather like *tomorrow*?
- How's it looking out there *right now*?

What is the **intent** of these requests? They're both some sort of `weather_inquiry`!

These also have an **entity** of a time/date.

```
const createChatAgent = () => {
    const CS571_WITAI_ACCESS_TOKEN = "...";

    const handleInitialize = async () => {
        return "Welcome to BadgerChat Mini";
    }

    const handleReceive = async (prompt) => {
        return "Your message has been received...";
    }

    return {
        handleInitialize,
        handleReceive
    }
}
export default createChatAgent;
```

Implementation

This is implemented with a **closure**, meaning that the inner functions `handleInitialize` and `handleReceive` have access to the outer variables.

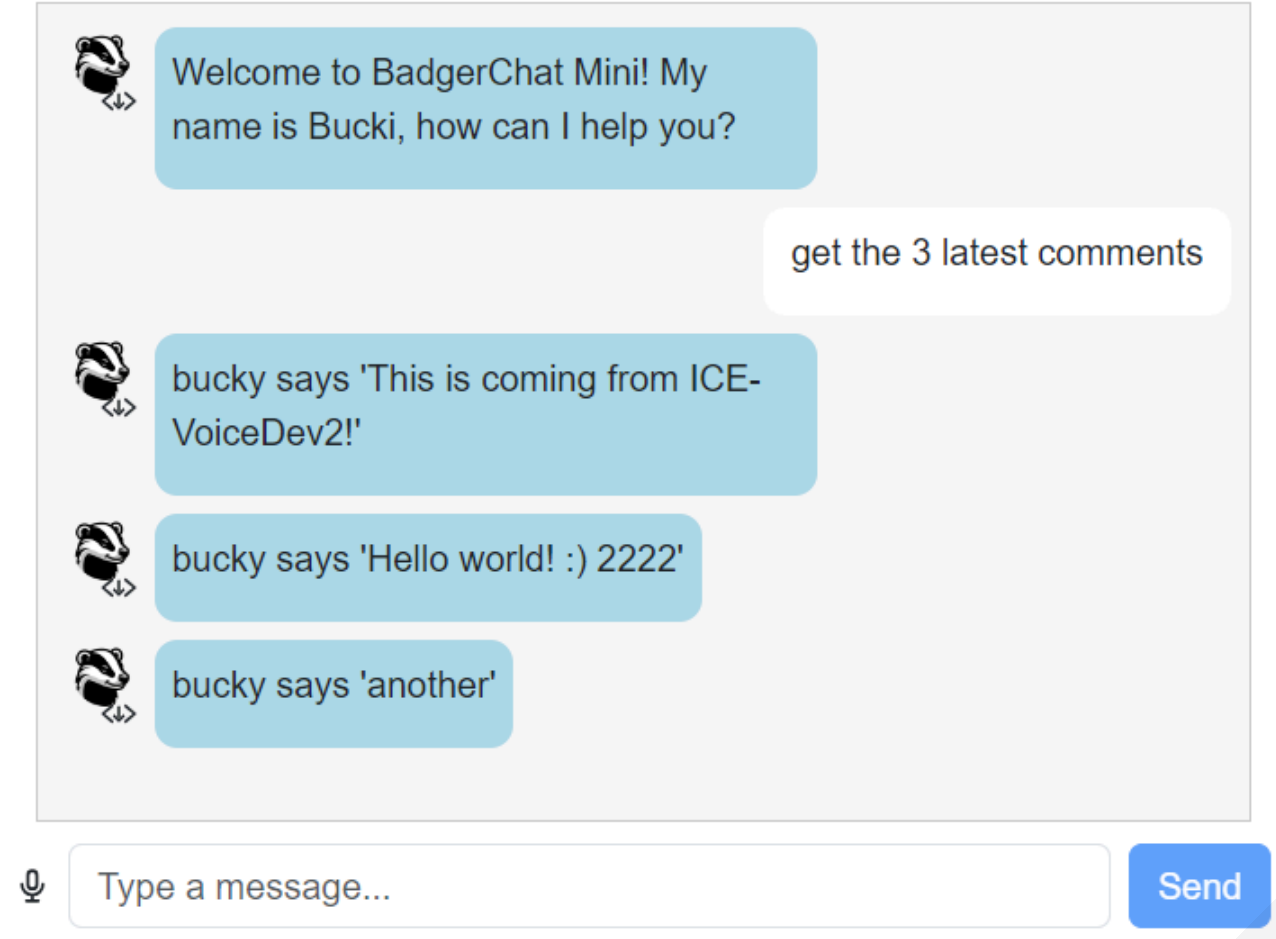
What is exposed to the consumer is controlled via `return`, everything else is 'private'.

Alternatively, you may consider ES6 classes.

Your Turn!

Implement `get_messages` for BadgerChat Mini!

Here's the web version you built...



```
const createChatAgent = () => {
    const CS571_WITAI_ACCESS_TOKEN = "...";

    // Define conversation context here!
    let stage;

    const handleInitialize = async () => {
        return "Welcome to BadgerChat Mini";
    }

    // ...
}
export default createChatAgent;
```

Closures allow us to share some **state**. This state defines the *context* of our conversation.

Looking Ahead...

We still need to implement `login` and `create_comment`, but how would these be expressed?

"Log me in with username **bucky124** and password **myp@ssw0rd**." ← seems silly!

There is no Wit.AI entity to handle this, usernames and passwords are unpredictable!

Possible Solution

Step out of Wit.AI for a moment!

When the `login` intent has been triggered, ask the user for their username and set the `stage` to `"FOLLOWUP_USERNAME"`

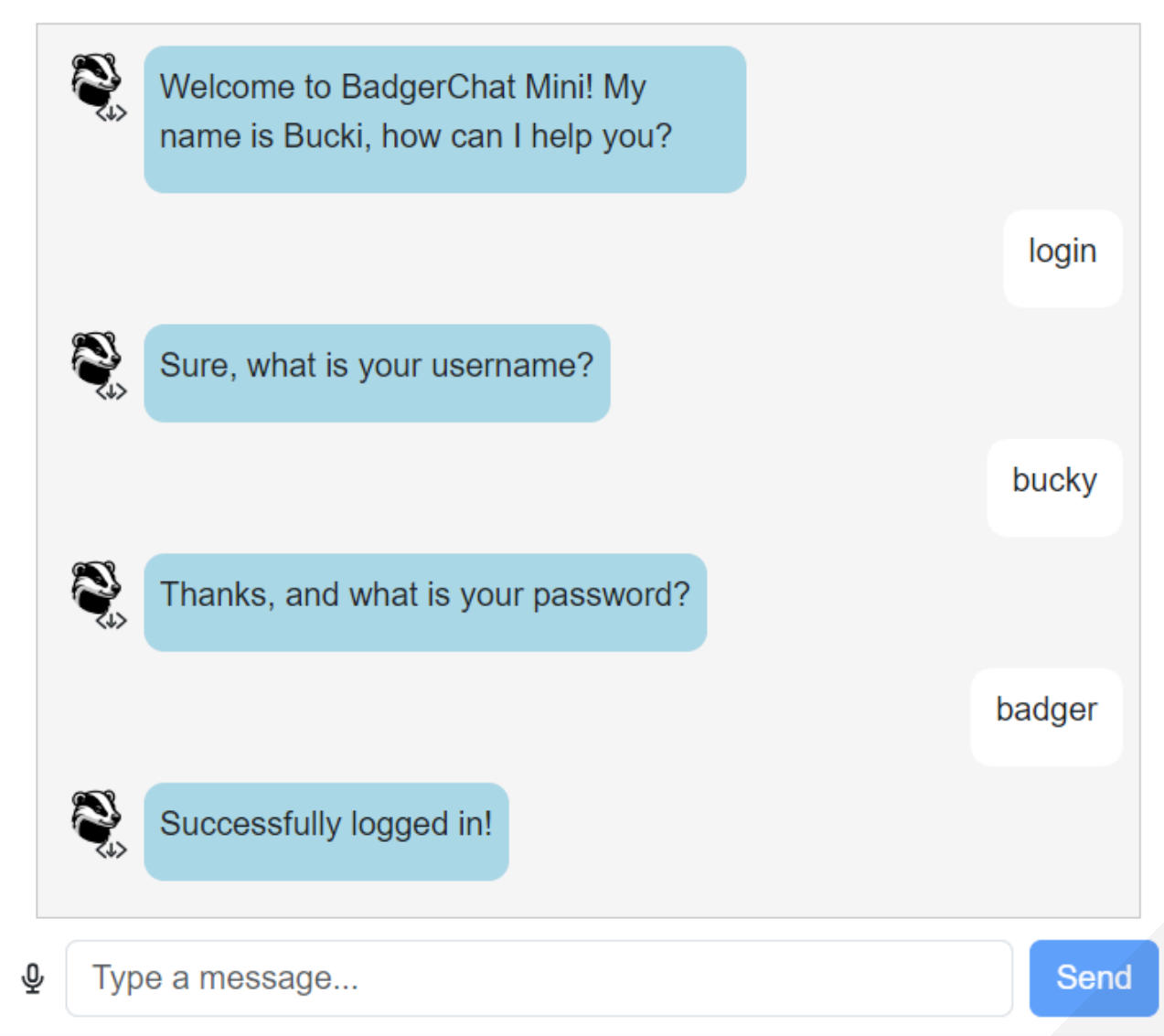
On the next `handleReceive`, ask for their password and set the stage to `"FOLLOWUP_PASSWORD"`

Finally, on the next `handleReceive`, attempt to login.

Your Turn!

Implement `login` for BadgerChat Mini!

- bucky,badger
- pete608,gopioneers!
- gophy77,boooo



```
const createChatAgent = () => {
    const CS571_WITAI_ACCESS_TOKEN = "...";

    // Define conversation context here!
    let stage;

    const handleInitialize = async () => {
        return "Welcome to BadgerChat Mini";
    }

    // ...
}
export default createChatAgent;
```

Do you have any concerns about this code?

Asynchronous Code

❗ Two `async` functions sharing state? JavaScript is actually single-threaded!

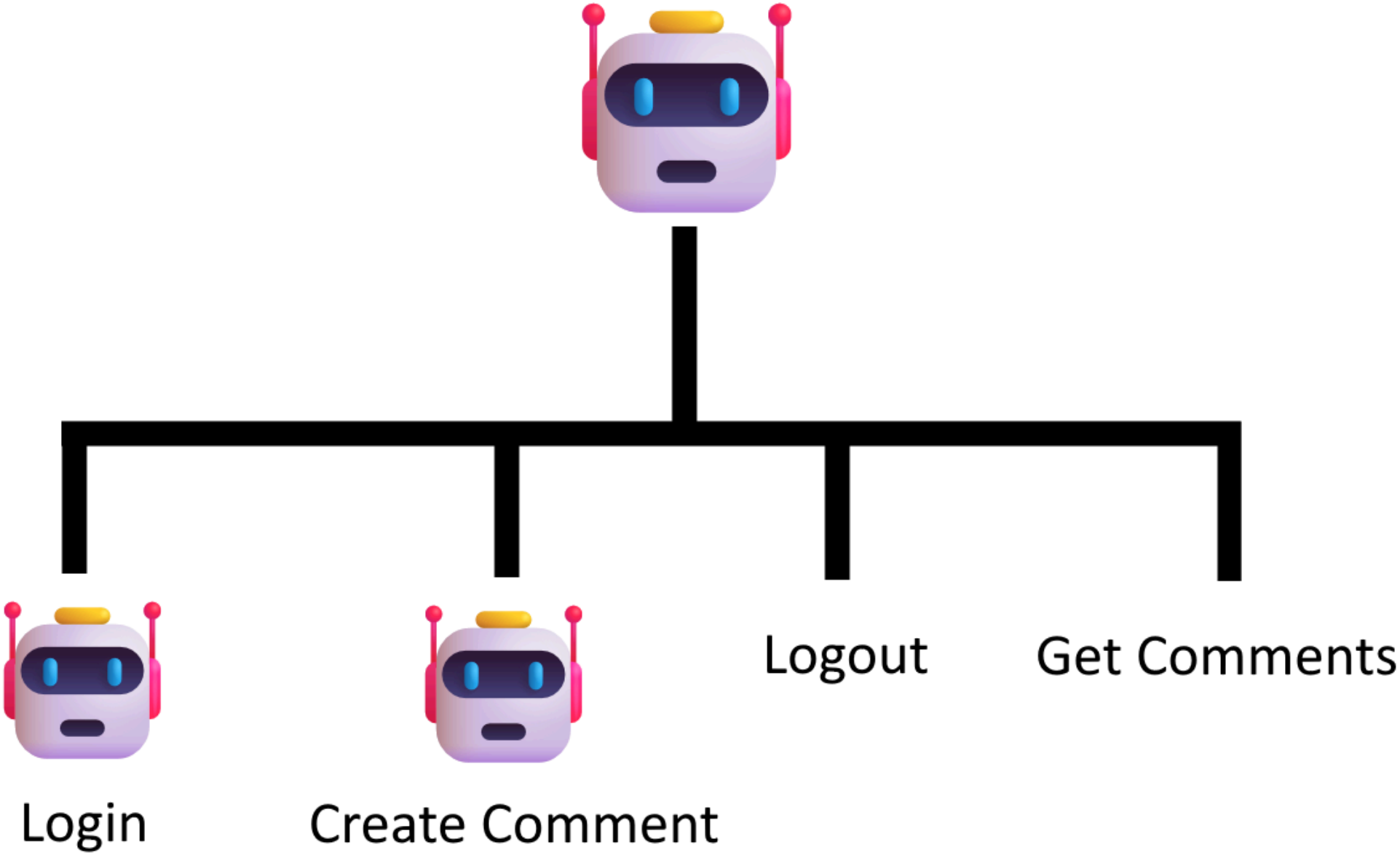
❗ ❗ JavaScript is single-threaded? Things like `fetch` and `setTimeout` are run by the browser *outside of JavaScript* on a seperate thread.

Takeaway: You are *guaranteed* that the JS doesn't execute these 2 functions the same time.

Learn more about the event loop... (1) (2)

```
const createChatAgent = () => {
    const CS571_WITAI_ACCESS_TOKEN = "...";

    // We may have a lot to keep track of!
    let stage;
    let comment, commentConfirm;
    let loginUsername, loginPassword;

    const handleInitialize = async () => {
        return "Welcome to BadgerChat Mini";
    }

    // ...
}
export default createChatAgent;
```

Having many intents → Having many context variables


Login
Create Comment
Logout
Get Comments

In `ChatAgent.js` ....

```
const createChatAgent = () => {

    const delegator = createChatDelegator();

    const handleReceive = async (prompt) => {
        if (delegator.hasDelegate()) { return delegator.handleDelegation(prompt); }
        // ...
    }

    // ...

    const handleLogin = async (promptData) => {
        return await delegator.beginDelegation("LOGIN", promptData);
    }

    const handleCreateComment = async (promptData) => {
        return await delegator.beginDelegation("CREATE", promptData);
    }

    // ...
```

In `ChatDelegator.js` ....

```
const createChatDelegator = () => {

    let delegate;

    const DELEGATE_MAP = {
        "LOGIN": createLoginSubAgent,
        "CREATE": createCommentSubAgent
    }

    // ...
```

... with functions to begin and end delegation.

In `LoginSubAgent.js` …

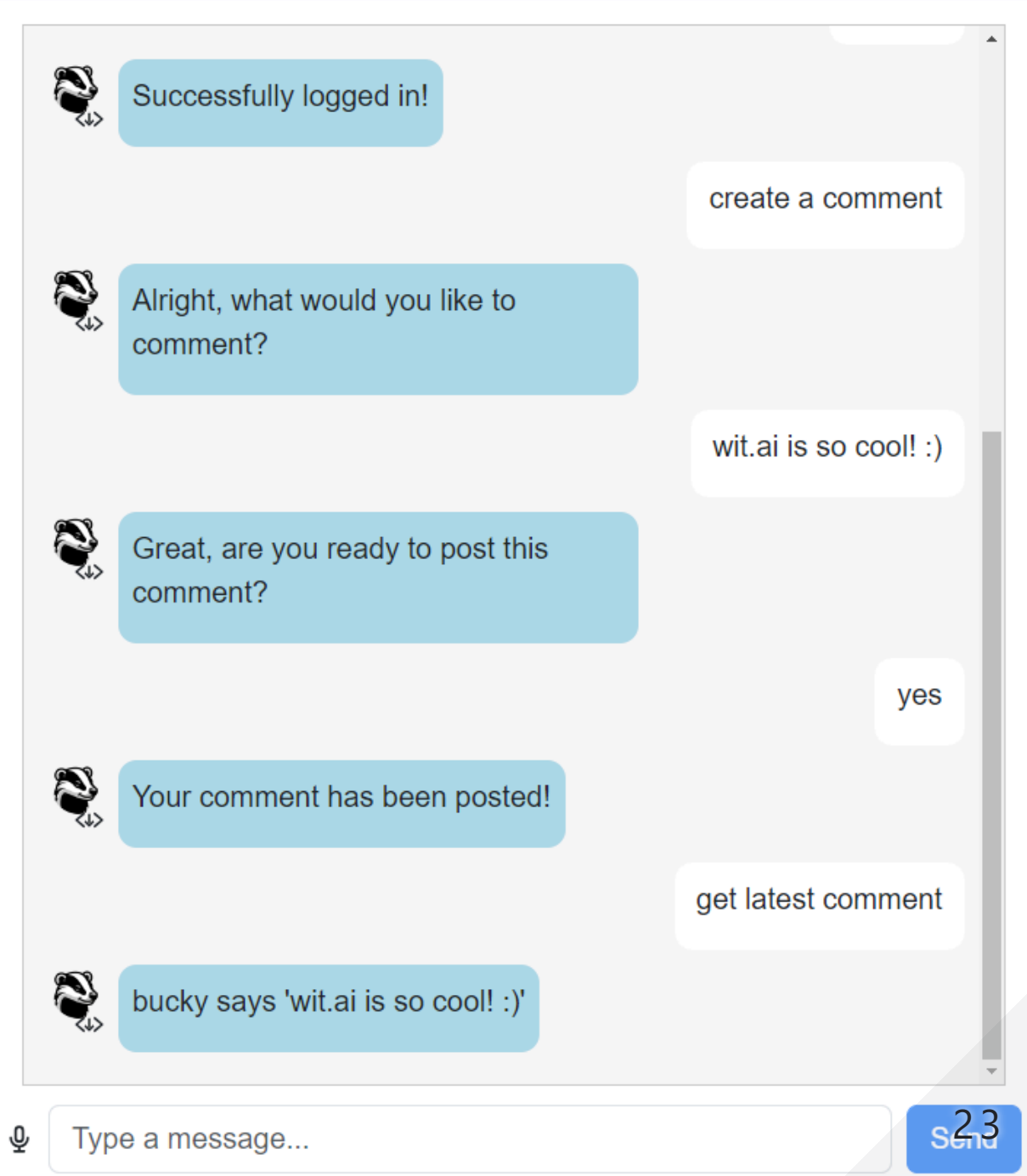
```
const createLoginSubAgent = (end) => {

    let stage;

    const handleInitialize = async (promptData) => {
        return "I should handle logging in!";
    }

    const handleReceive = async (prompt) => {
        switch(stage) {
            case "FOLLOWUP_USERNAME": return await handleFollowupUsername(prompt);
            case "FOLLOWUP_PASSWORD": return await handleFollowupPassword(prompt);
        }
    }

    const handleFollowupUsername = async (prompt) => { }

    const handleFollowupPassword = async (prompt) => {
      // ...
      end();
    }

    // ...
```

In `CreateCommentSubAgent.js` …

```
const createCommentSubAgent = (end) => {

    let stage;

    const handleInitialize = async (promptData) => {
        return "I should handle creating a comment!";
    }

    const handleReceive = async (prompt) => {
        switch(stage) {
            case "FOLLOWUP_COMMENT": return await handleFollowupComment(prompt);
            case "FOLLOWUP_CONFIRM": return await handleFollowupConfirm(prompt);
        }
    }

    const handleFollowupComment = async (prompt) => { }

    const handleFollowupConfirm = async (prompt) => {
        // ...
        end();
    }

    // ...
```

Your Turn!

Delegate and implement `create_comment` for BadgerChat Mini!



Other Features

Text-to-Speech `handleSynthesis`

```
const resp = await fetch(`https://api.wit.ai/synthesize`, {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
        "Accept": "audio/wav",
        "Authorization": `Bearer ...`
    },
    body: JSON.stringify({
        q: "This is the text I would like to transcribe!",
        voice: "Rebecca",
        style: "soft"
    })
})
```

Limited to 60 requests/minute, 280 characters max.

Speech-to-Text `handleTranscription`

```
const resp = await fetch(`https://api.wit.ai/dictation`, {
    method: "POST",
    headers: {
        "Content-Type": "...",
        "Authorization": `Bearer ...`
    },
    body: rawSound
})
```

`Content-Type` is limited to a range of audio types, *not* what is recorded by default.

Traits

Used for detecting certain traits like user sentiment.

Questions?