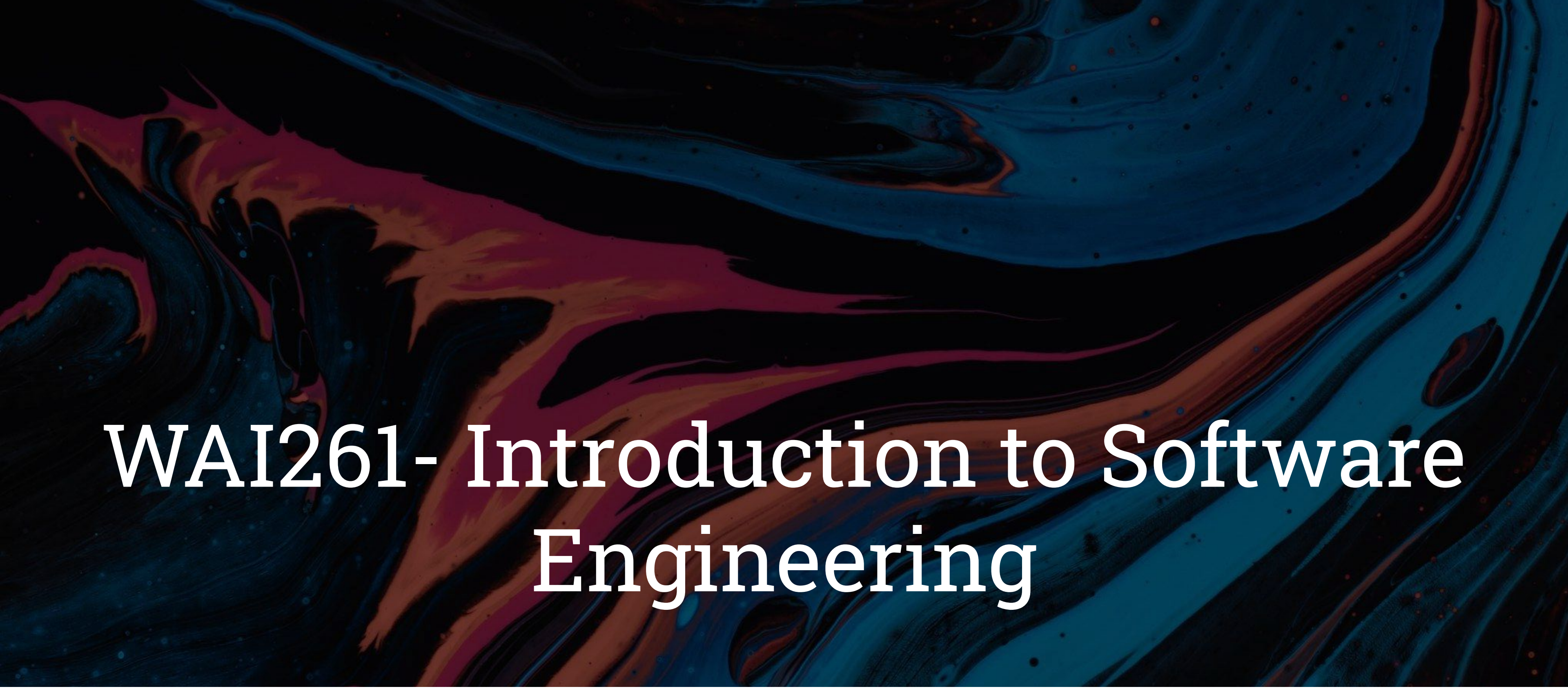
Welcome back...

If you have not completed Tutorial 1, go to `github.com/WarwickAI/wai161` to get the latest code.

Once downloaded, open the folder in VSCode and in the terminal run `npm install` to install the packages



WAI261- Introduction to Software Engineering

Tutorial 2 - React State Management, Lifecycle & Querying

Course Breakdown

Tutorial 0: Setup VSCode, Git and NodeJS (Medium article)

Tutorial 1: Introduction to web development, setting up NextJS project, some UI stuff

Tutorial 2: More React; state management, component lifecycle (+hooks), basic querying

Tutorial 3: Creating and developing server in NodeJS

Tutorial 4: Finishing touches and deployment to Vercel

Following the Tutorial

- Head to *github.com/WarwickAI/wai161* for resources:

Where are we?

- Set up a React project
- Added DaisyUI package
- Used DaisyUI and TailwindCSS to create a basic UI

Today we are going to...

- Customise our Message component with properties

- Use React states to manage variables belonging to a component
    - To store the current message in the text field
    - And to store the messages

- Query Hugging Face’s API to get NLP analysis of our messages

Customising Components with Properties

HTML Button

```
<button
  className="btn btn-sm"
  onClick={
    () => console.log("Button Clicked")
  }
>
  Send
</button>
```

Properties or props of the component

Format:

    <key>=<value>

Example:

    className="..."

will set the CSS styles

Actually also a property, equivalent of setting children property of component:

    children="Send"

Customising our Message Component

What properties do we want to be able to specify:

- Message contents: this will be the string of the message

- Message author: ID or name of the message sender (user/AI)

- Message time: when the message was sent

Properties are Read-Only

Code... (in Message.tsx)

```
type MessageProps = {
  contents: string;
}


// This function is our custom Message component
const Message = ({ content }: MessageProps) => {
  return (
    ...
  );
}
```

TypeScript type with a property called contents of string type

Here we are saying that the Message component accepts properties of type `MessageProps`

Here we are separating our `props` into the separate variables (aka destructuring)

Accessing Properties (in Message.tsx)

```
<p className="text-white">
     {contents}
</p>
```

Access the message `contents` from the props argument

Passing Properties into Message (in `App.tsx`)

```
<Message contents="Message 1" />
<Message contents="Message 2" />
<Message contents="Message 3" />
<Message contents="Message 4" />
```

Setting message contents to be displayed

Adding More Properties

1. Add a `user` property (can just be of `String` type)

2. Change the background colour and position dependent on the `user` (have a look at JavaScript ternary operator).

3. Add a `date` property (check out the JavaScript `Date` type).

But what about Component State...

We have looked at passing properties which are read-only

What if we want to track a value within a component

To do this we use **States**:

From React library

```
const [state, setState] = useState<number>(0);
```

State variable type

State variable (use this to access the value) *Modifying the value in this variable will not work correctly*

Initial value for the state variable

State update function (use this to update the value of the state variable)

Tracking Entered Message (in `index.tsx`)

```
const Home: NextPage = () => {
  const [msg, setMsg] = useState<string>("");


  return (
    ...
  )
}
```

Add this line within our function component (before the `return` statement)

We can then access the variable `msg`, and update the message state using `setMsg(<newValue>)`

Accessing and Updating Message (in `index.tsx`)

```
<input
    placeholder="Enter Message..."
    value={msg}
    onChange={(e) => setMsg(e.target.value)}
/>
```

Set value (string in the text field) to the message state value

When the value of the state variable changes, it will automatically update the component with the new property

Update the message state value with the text in the text field

This will trigger any component using the value to update

Our variable `msg` will now contain whatever has been written in the text field

Let’s output the message when the send button is clicked

In `index.tsx`

$$\texttt{<button ... onClick=\{() => }\underbrace{\texttt{console.log(msg)}}_{\text{Here we log the message state variable to console}}\texttt{\} > Send </button>}$$

```
<button
  ...
  onClick={() => console.log(msg)}
>
  Send
</button>
```

Here we log the message state variable to console

Keeping Track of Messages

Instead of hardcoding the messages, we should store them in some kind of data structure.

We will use an `Array` of message objects

Add the following to the `index.tsx` file above the Home function component:

```
interface MessageObject {
  contents: string;
  user: string;
  time: Date
}
```

Create Message Array State

To keep track of all the messages, we will store them in a React State so that we can use and update the value within our `Home` page component.

Add the state declaration to `index.tsx` at the start of the root function component:

```
const [messages, setMessages] = useState<MessageObject[]>([]);
```

Type is an array of message objects

Initialise with empty array i.e. no messages

Adding Messages

When the user presses the send button, we want to add a message to the message state variable.

```
<button
  ...
  onClick={() => {
    let newMessage: MessageObject = {
      contents: msg,
      user: "me",
      time: new Date(),
    };
    setMessages([...messages, newMessage]);
  }}
>
  Send
</Button>
```

When the send button is pressed, we:

1. Create a new message object with the details we want.
2. Add the new message (using the spread operator - Google for more info)

We must use the spread operator due to how set state manages updating

Showing our Messages

Instead of hardcoding our message rendering, we can iterate over our array of messages creating a `Message` component for each.

Replace the hard-coded message declarations with the following:

```
{messages.map((message) => (
  <Message
    key={message.time.toISOString()}
    contents={message.contents}
    user={message.user}
    time={message.time}
  />
))}
```

This will iterate over each element in our array, and return a new array with the result of the function for each element.

In this case we are returning an array of `Message` components.

React requires us to specify a **unique** `key` when we are creating arrays of components.

Try typing and sending a message,
it should show up in your chat

Now for the NLP Analysis

We want to analyse our messages and return some interesting analysis.

For this we are going to do NLP analysis.

We are going to use **HuggingFace.co** to do the NLP analysis for us.

Go ahead and create an account on Hugging Face and pick a model that will return some info about a string that we send it.

We are going to send an API call with the message to the model we want to use for NLP analysis, then show the results as a message.

Querying Hugging Face through an API Call

Add the following near the top of your `index.tsx`:

```
const API_URL = "<yourModelURL>";

var HEADERS = new Headers();
HEADERS.append("Authorization", "Bearer <yourAPIKey>");
```

Set these up with the values from Hugging Face

And add the following to the function for the send message button
You will need to make the function asynchronous using the `async` keyword e.g. `async () => {`

```
const requestInit: RequestInit = {
  method: "POST",
  headers: HEADERS,
  body: JSON.stringify({ inputs: msg }),
};
const response = await fetch(API_URL, requestInit);
const json = await response.json();
console.log(json);
```

Setup the config for the API call

Call the API, and convert the results to JSON to be able to read them

Adding the Response as a Message

With the code from the previous slide, the results from the API call will be shown in the console (in developer tools).

We want to show a meaningful message in our chat representing this response.

Analyse the response and add a new message to the chat

The next slide shows the complete send button function for my model

```
<div
  ...
  onClick={async () => {
    const newMessage: MessageObject = {
      contents: msg,
      user: "me",
      time: new Date(),
    };

    const requestInit: RequestInit = {
      method: "POST",
      headers: HEADERS,
      body: JSON.stringify({ inputs: msg }),
    };
    const response = await fetch(API_URL, requestInit);
    const json = await response.json();

    const sortedLabels = json[0].sort(
      (val1: any, val2: any) => val2.score - val1.score
    );
    const aiMessage: MessageObject = {
      contents: "Message was " + sortedLabels[0].label,
      user: "ai",
      time: new Date(),
    };
    setMessages([...messages, newMessage, aiMessage]);
  }}
>
  Send
</Button>
```

Function defined as asynchronous

Create message from us

Setup config and call NLP model API with the message. Convert the response to JSON.

Sort labels in descending order of confidence

Create “AI” message, including NLP analysis. In this case we use the most confident label.

Update the message state with our message and the “AI”’s message

Lastly, Making our code more Maintainable

Having the function for button presses within our UI declaration isn't very tidy, especially for large functions like this one.

Let's move it out.

Create a new function within the `App` component's functional component like so:

```
const handleMessages = async () => {
    // Code for handling creating messages
    // and sending API request
}
```

Modify the Button component to call this function:

```
<button
  ...otherProperties
  onClick={handleMessages}
>
```

Note:
we don't call the function here, just pass it as a variable

Your Turn

Work through what has been covered here

Maybe try a different model that returns some other NLP analysis data.

Before next week complete the following:

- Tutorial 2 (slides will be online)
- Implement the ChatBot functionality:
  - Display current messages
  - Allow sending messages
  - Show results of NLP analysis as a message

Next week we will be:

- React `useEffect` hook
- Creating a server to handle storing our messages
- Allowing multiple users to connect to the same chatroom