Topic	**jQuery AJAX METHOD**	
Class Description	**Students will understand the use of AJAX techniques used in web applications. Students will learn about the jQuery ajax() method. Students will also learn to use jQuery ajax() method call to update the score result of the Mad Libs story.**	
Class	**C177**	
Class time	**45 mins**	
Goal	● Understand the use of AJAX techniques in web applications. ● Learn about the jQuery ajax() method. ● Show the score result of the Mad Libs story using jQuery ajax() method call.	
Resources Required	● Teacher Resources: ○ Visual Studio Code Editor ○ laptop with internet connectivity ○ smartphone ○ earphones with mic ○ notebook and pen ● Student Resources: ○ Visual Studio Code Editor ○ laptop with internet connectivity ○ smartphone ○ earphones with mic ○ notebook and pen	
Class structure	**Warm-Up** **Teacher-led Activity** **Student-led Activity** **Wrap-Up**	**5 mins** **15 mins** **20 mins** **5 mins**
WARM-UP SESSION - 5 mins		

CONTEXT ● **Understanding the jQuery ajax() method call.**		
Teacher Starts Slideshow **Slide 1 to 4** Refer to speaker notes and follow the instructions on each slide.		
Hey <student's name>. How are you? It's great to see you! Are you excited to learn something new today? **Following are the WARM-UP session deliverables:** ● Greet the student. ● Revision of previous class activities.		**ESR**: Hi, thanks! Yes I am excited about it! Click on the slide show tab and present the slides
WARM-UP QUIZ Click on In-Class Quiz		
Continue WARM-UP Session **Slide 5 to 13**		
Following are the session deliverables: ● Appreciate the student. ● Narrate the story by using hand gestures and voice modulation methods to bring in more interest in students.		
Class Steps	**Teacher Action**	**Student Action**
Step 1: Warm-Up (5 mins)	Hi, how are you? Great!	**ESR:** I am good!
	Do you remember we discussed why we should use jQuery at the beginning of the previous class?	**ESR**: Yes.

	Can you tell me why? Perfect! jQuery is “write less, do more” JavaScript library. jQuery also makes web applications load a lot faster than JavaScript. Before we can see how jQuery is faster, let’s discuss how the web applications are loaded! Can you tell me how a web page loads? A lot of things happen between the time when we search for something on a web browser and the time when a web page is completely loaded. There are lot of steps to describe this but it can be broken down into a few general steps: ● Browser sends an HTTP request.	**ESR**: jQuery helps to reduce the JavaScript code. That means we have to write very less code compared to the JavaScript code. **ESR**: Varied.

	<ul><li>Server responds and sends back the requested HTML file.</li><li>Browser begins to render HTML.</li><li>Browser then sends additional requests for objects present in the HTML file (CSS files, images, JavaScript, etc.).</li></ul>The faster all steps are completed, the faster the web page will be loaded! Did you know around 57% of people leave (or close) a web page that takes longer than 3 seconds! We all have used Google's Chrome browser for searching anything we want to know about and Google gives the result in the blink of an eye!! Can you imagine how this is even possible? With millions of people using the browser at the same time, how is this possible for Google to give such fast results? ***Note**: Encourage the student to discuss what they know about the technology and help them to be more involved.* Well the Google Chrome browser is built on one of the very powerful	**ESR**: Yes/No. *The student discusses his/her views with the teacher.*

	technologies used in the web application, that is, AJAX. AJAX is **A**synchronous **Ja**vaScript and **X**ML. In simple terms, this technique helps to load/update the specific portion of the page, instead of loading the whole page again and again. For example, in our Mad Libs game, when we click on the Submit button the result should be updated without reloading the page again! That means only the result section should be updated! In today's class, we are going to learn how to use the jQuery ajax() method to show the score when the user clicks on the submit button for a particular Mad Libs story. Are you excited? Let's get started then.	**ESR**: Yes.

Teacher Ends Slideshow

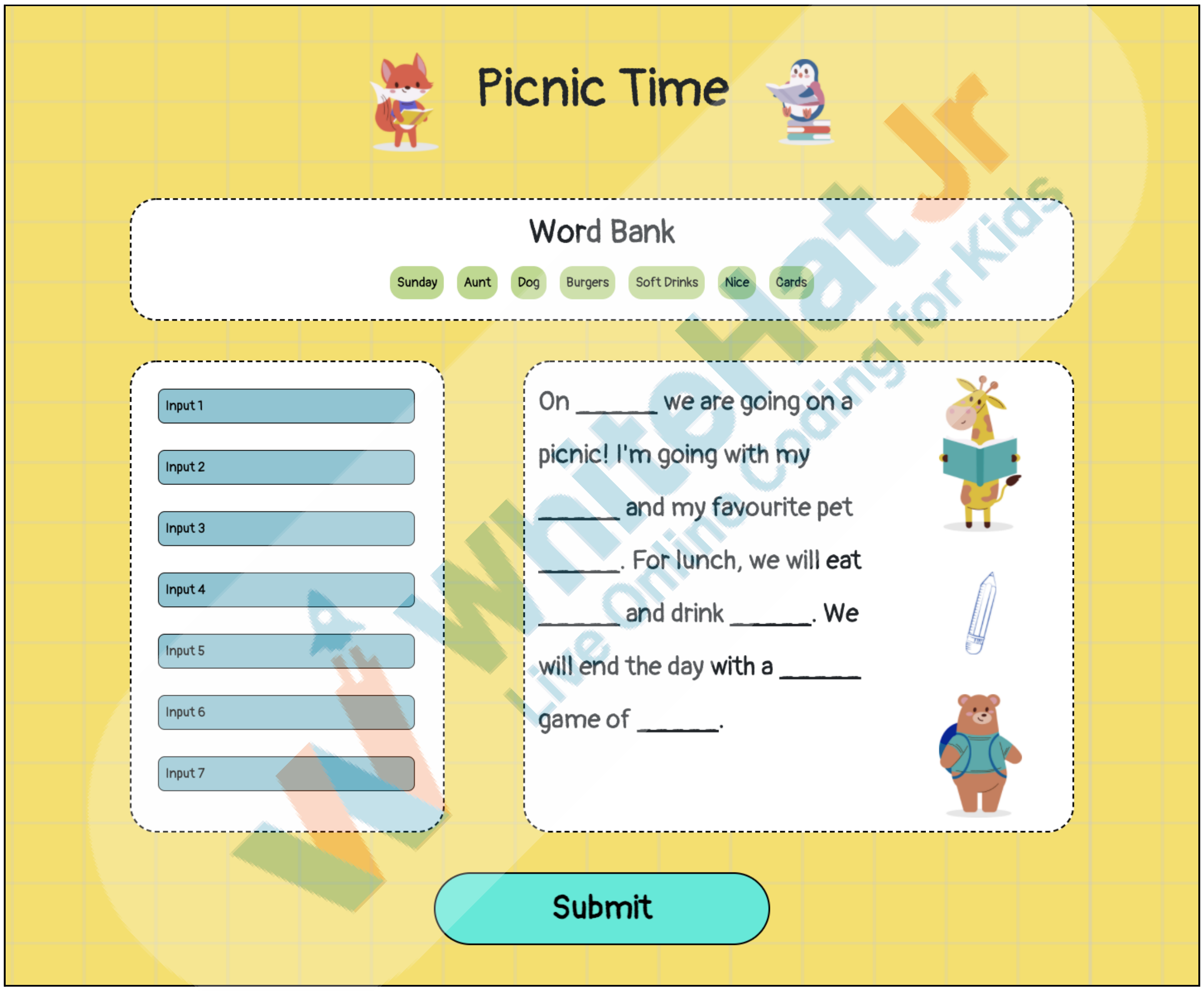
TEACHER-LED ACTIVITY - 15 mins

Teacher Initiates Screen Share

CHALLENGE ● **Show the Mad Libs story score result on the page using jQuery ajax() method.**		
Step 2: Teacher-led Activity (15 mins)	*<The teacher clones the activity from the Teacher Activity 1 and shows the output.>* ***[Teacher Activity 1]*** Let's first go through this boilerplate code. If we take a closer look, we converted our HTML's page into a Flask App. Do you remember what a Flask is? Awesome! Just as we have used Flask to create APIs until now, we can also create websites with it. Do you remember how we used to run Flask Apps?	**ESR:** Flask is a Python's Framework to create APIs. **ESR:** 1. Create a Virtual Environment. 2. Activate the Virtual Environment. 3. Install Modules/Dependencies. 4. Run the server.

	Okay!
	
	First we will go to the project directory through terminal or command prompt.
	
	Then, we'll create a virtual environment first -
	
	`python3.8 -m venv venv`
	
	Next, we will activate the virtual environment -
	
	MacOS/Linux -
	
	`source venv/bin/activate`
	
	Windows -
	
	`venv\Scripts\activate.bat`
	
	Next, we will install flask -
	
	`pip install flask`
	
	Finally, we can run the server -	

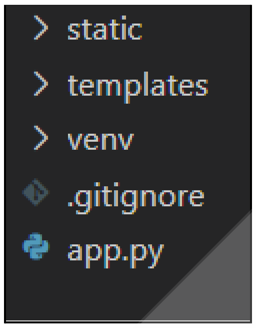
	```python app.py``` If we go to **localhost:5000** on the browser, we can see our app:	

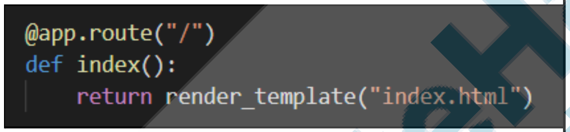


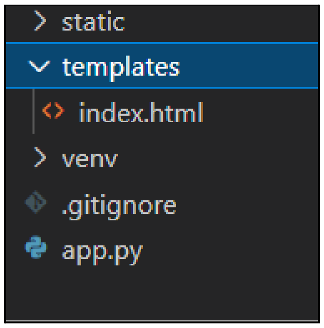
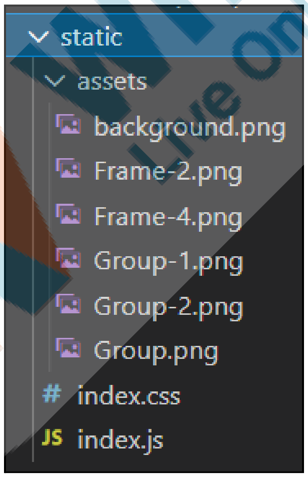
Picnic Time

Word Bank

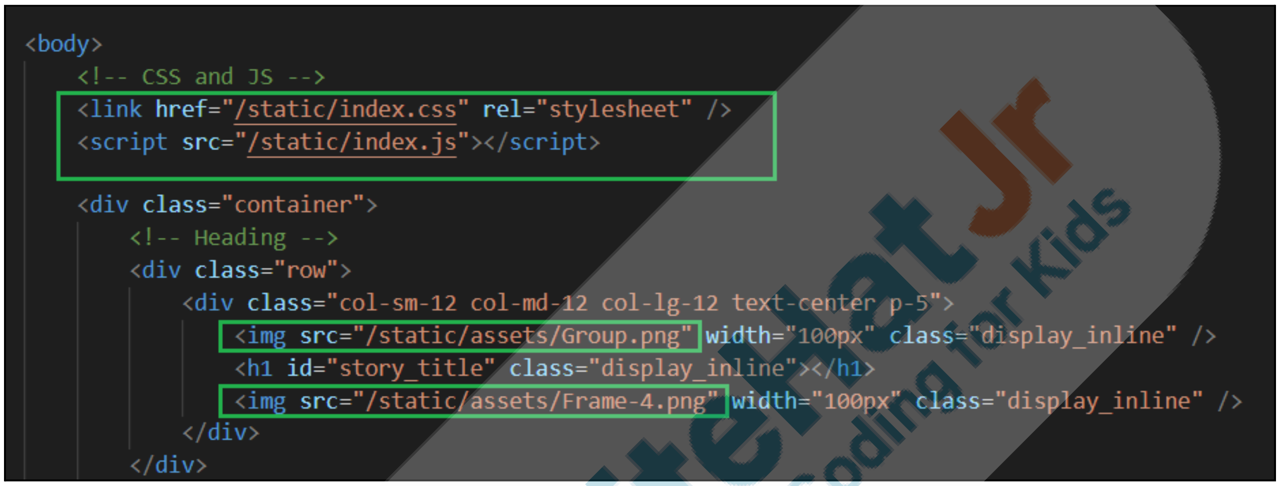
On ______ we are going on a picnic! I'm going with my ______ and my favourite pet ______. For lunch, we will eat ______ and drink ______. We will end the day with a ______ game of ______.

	Let’s go through the boilerplate code now. If we take a look at the code structure, we can notice the following structure -  static templates venv .gitignore app.py Here, we know that our main APIs and Flask App is in **app.py.** Our **.gitignore** file contains the files/folders we want GitHub to ignore. What are the **templates** and **static** folders? ***Note**: The name **templates** and **static** are predefined keywords.*	**ESR:** Varied.

	Just like programming languages have special keywords that have a specific meaning, such as **“let/var/const”** to define variables in JavaScript**. Flask** has some special words reserved for folders that have special meaning!
	
	“templates” folder is where **Flask** looks for HTML files when a function **render_template** is called.
	
	If we take a look at our **app.py** file -
	
	
	`@app.route("/")`
	`def index():`
	`    return render_template("index.html")`
	
	We can see here that we have created a route **“/”** where we are calling the **render_template()** function and passing **“index.html”** into it.
	
	What this means is that on the route
	
	localhost:5000/
	
	We are rendering a template **“index.html”** with a **render_template()** function.
	
	When the **render_template()** function is called, it looks for the HTML file only in a folder named **templates** like we have.	

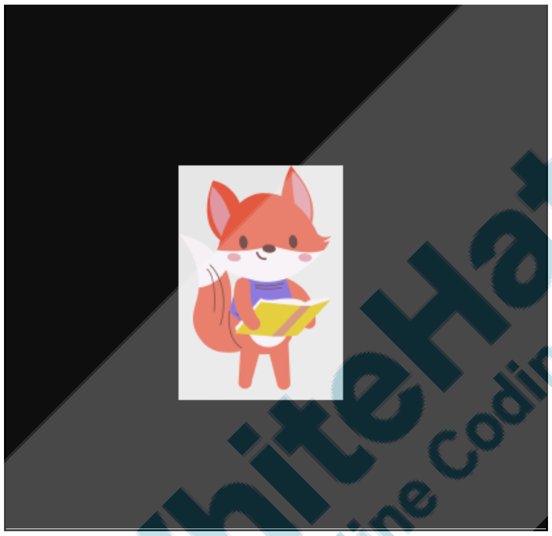
	 static templates index.html venv .gitignore app.py If, however, the HTML file was present at a different location, then the server would have given an error because it wouldn’t have been able to find the file.	
	Next, we have a **“static”** folder. Inside this folder, we have our **CSS, JavaScript** files and **Images** that we are using in an **assets** folder.  static assets background.png Frame-2.png Frame-4.png Group-1.png Group-2.png Group.png index.css index.js Now Flask creates a URL endpoint of **“/static”** by default, which contains all the static data. Everything we have in	

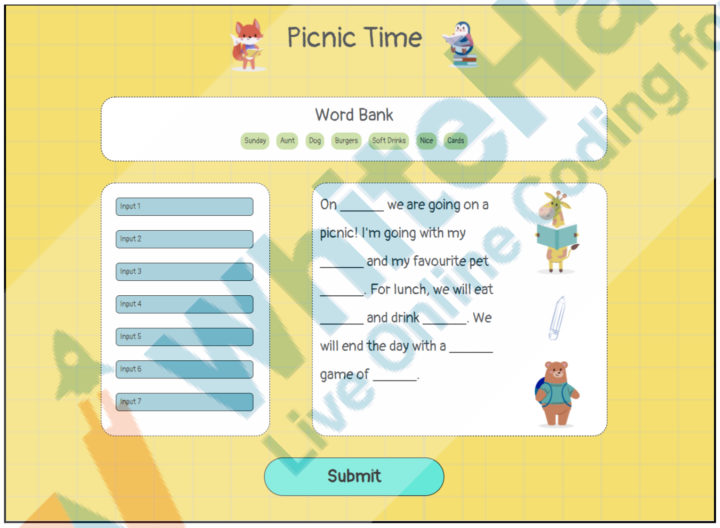
	the “static” folder can be accessed with this. To understand this, let’s take a look at our **index.html** file:	



```
<body>
    <!-- CSS and JS -->
    <link href="/static/index.css" rel="stylesheet" />
    <script src="/static/index.js"></script>

    <div class="container">
        <!-- Heading -->
        <div class="row">
            <div class="col-sm-12 col-md-12 col-lg-12 text-center p-5">
                <img src="/static/assets/Group.png" width="100px" class="display_inline" />
                <h1 id="story_title" class="display_inline"></h1>
                <img src="/static/assets/Frame-4.png" width="100px" class="display_inline" />
            </div>
        </div>
```

	Here, we can see that we are adding our CSS file with the path **“/static/index.css”**. Similarly, we have our JS file in path **“/static/index.js”**. We also have our images in the **“/static/assets/<image_name>”** path. This is because Flask has **“static”** as a special folder we can create to hold our static data and we can access this data with the help of a URL. Try opening the following link in the browser -	

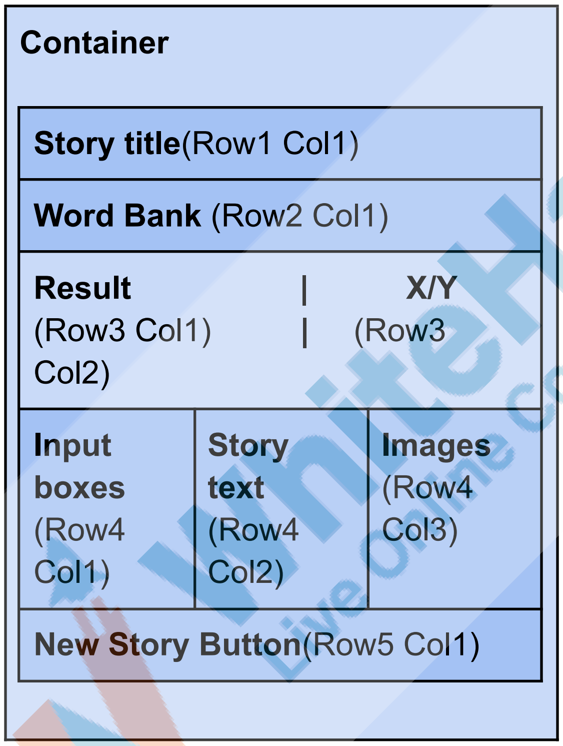
	localhost:5000/static/assets/Group.png You will see the image from the **Group.png** image in the browser -	
		
	Amazing, isn't it? Now to summarize our **folder structure**: ● **"static"** folder: We have our static assets and files ● **"templates"** folder: Our HTML templates. ● **"app.py"** file: We have our APIs.	**ESR:** Yes.

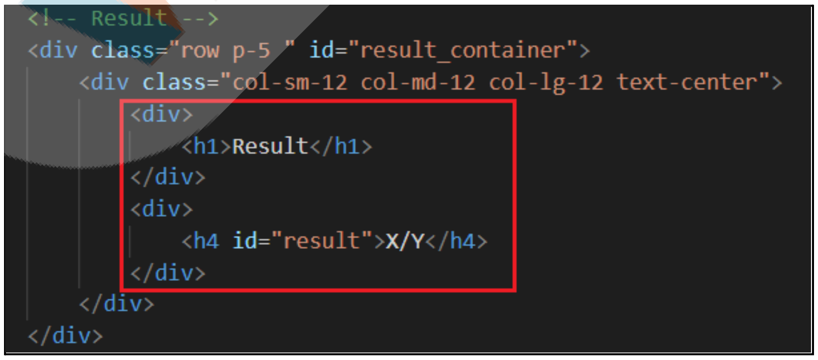
	Let’s now take a look at the page. The “**New Story**” **button** has now been updated to the “**Submit**” button. Earlier when the user was clicking on the “**New Story**” button, the next story was picked randomly. But now, since we want to show the scores based on how many inputs are correct, we will use the “**Submit**” button to show the result on click.	
		

On ______ we are going on a picnic! I'm going with my ______ and my favourite pet ______. For lunch, we will eat ______ and drink ______. We will end the day with a ______ game of ______.

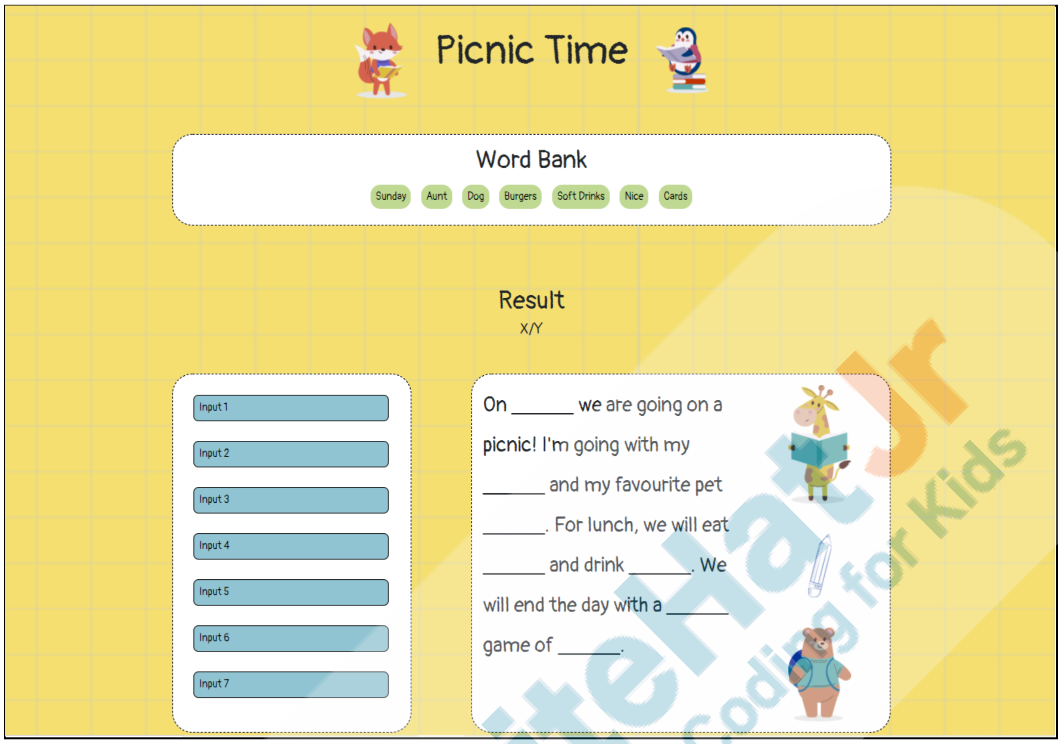
To begin with, we are going to create a <div> tag showing the Mad Libs result on the page below the Word Bank.

<The teacher adds the <div> tags in the following structure after the Word Bank in index.html>.





```
<!-- Result -->
<div class="row p-5 " id="result_container">
    <div class="col-sm-12 col-md-12 col-lg-12 text-center">
        <div>
            <h1>Result</h1>
        </div>
        <div>
            <h4 id="result">X/Y</h4>
        </div>
    </div>
</div>
```



Picnic Time

Word Bank

Sunday Aunt Dog Burgers Soft Drinks Nice Cards

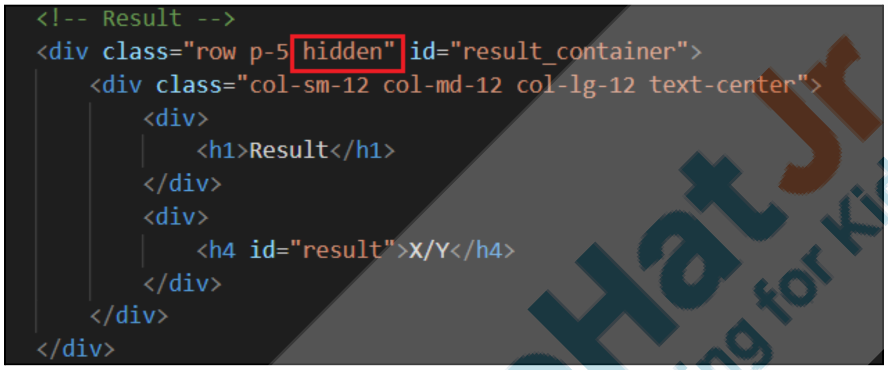
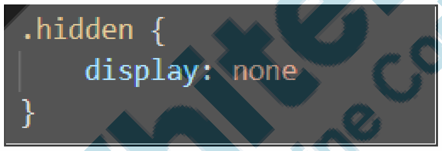
Result

X/Y

Input 1
Input 2
Input 3
Input 4
Input 5
Input 6
Input 7

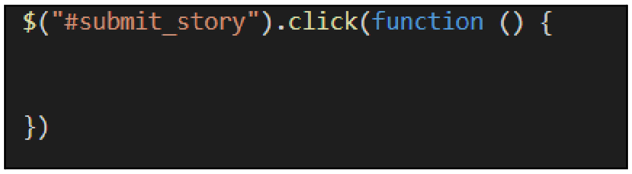
On ______ we are going on a picnic! I'm going with my ______ and my favourite pet ______. For lunch, we will eat ______ and drink ______. We will end the day with a ______ game of ______.

	But the "**Result**" is visible on the page all the time, even though we have not clicked on the "**Submit**" button. The result must be visible only after Submit is clicked. Any idea what should we do for this? Amazing! Let's take "**hidden**" as our own class name and assign it to the row <div> and set its initial display property as none in the style sheet.	**ESR:** We can initially hide the element and then show it only when the button is clicked.

	<The teacher adds the class name in the index.html file>. *<The teacher adds the style to class in the index.css file>.*	
	 	
	Now we are going to add the click event on the Submit story button. Remember how we add events in jQuery? Superb! *<The teacher selects the submit button element using #id and adds a click event in the index.js file>.*	**ESR**: We use $ to select the element and then .click() method on the element.

```
<!-- Result -->
<div class="row p-5 hidden" id="result_container">
    <div class="col-sm-12 col-md-12 col-lg-12 text-center">
        <div>
            <h1>Result</h1>
        </div>
        <div>
            <h4 id="result">X/Y</h4>
        </div>
    </div>
</div>
```

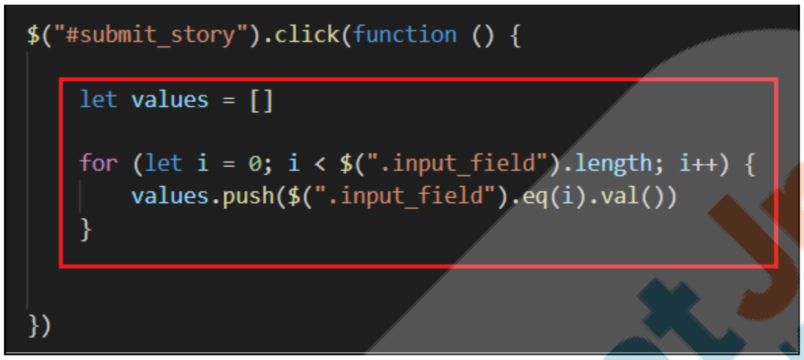
```
.hidden {
    display: none
}
```



```
$("#submit_story").click(function () {


})
```

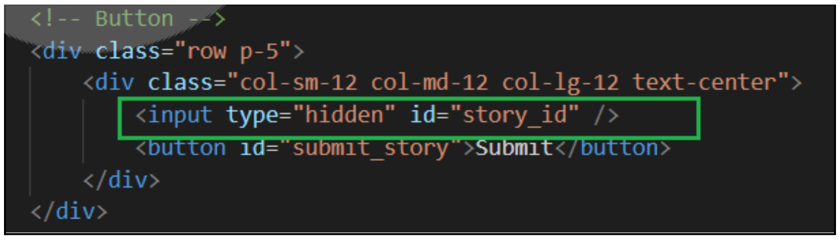
	Now let's frame logic for what all should happen when the submit button click event is triggered! Where shall we begin? Can you help me with that? Prefect! Let's take an array variable "**values**". We will **loop** through all the **#input_field** filled by the user: ● We will select the #input_field ○ $(".input_field") ● We can get the index i using eq() method ○ $(".input_field").eq(i) ● We can get the value of the selected input field using val() method ○ $(".input_field").eq(i).val() ● We push the #input_field value in the array	**ESR**: We need to know how many inputs are entered by the user out of the total input fields.

	○ values.push($(“.input_field”).eg(i).val()	



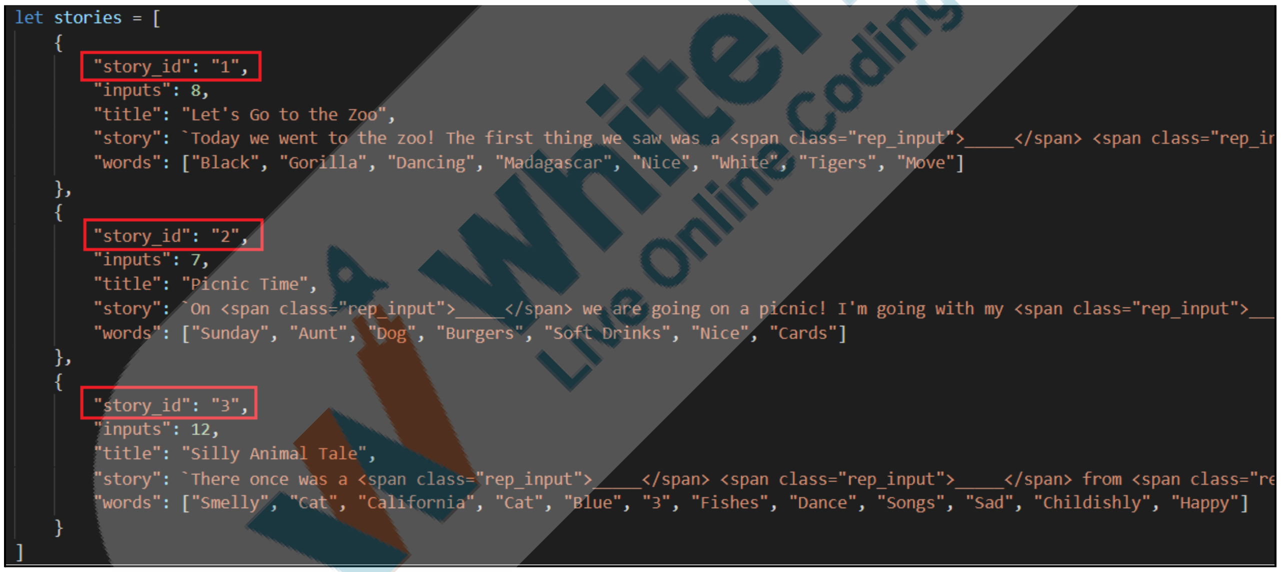
```
$("#submit_story").click(function () {

    let values = []

    for (let i = 0; i < $(".input_field").length; i++) {
        values.push($(".input_field").eq(i).val())
    }

})
```

	Now once we have all values entered by the user, how would you make sure that the input words entered by the user are correct? For this, first we should know which story the input values have been submitted. To identify the story we are going to have the **#story_id** <div> in the index.html file, which will be **hidden** initially. This will help us identify the story for which answers have been submitted!	**ESR**: Varied.

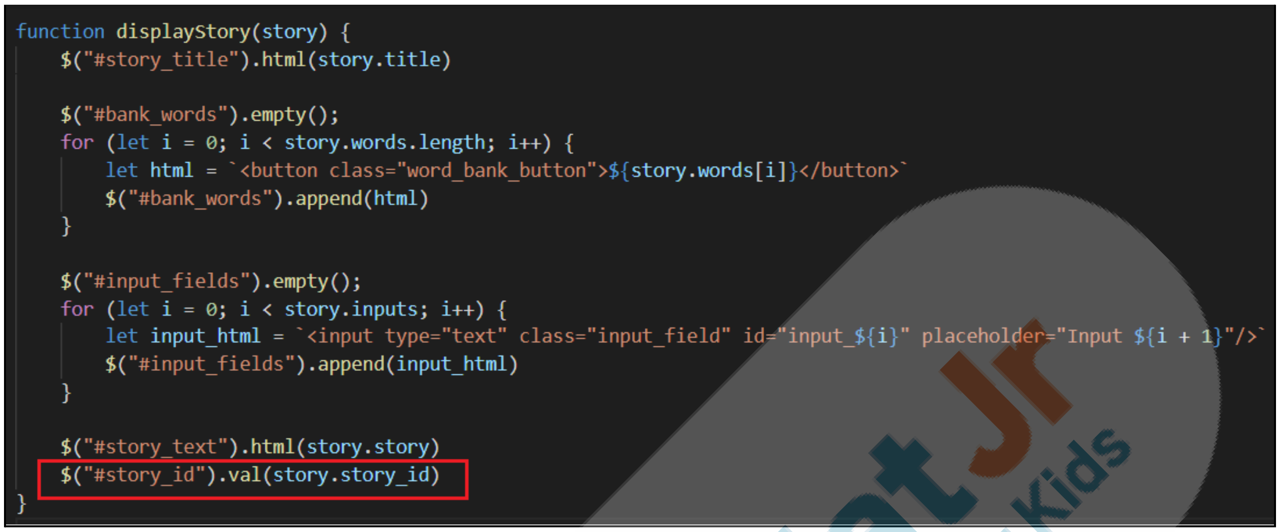


```
<!-- Button -->
<div class="row p-5">
    <div class="col-sm-12 col-md-12 col-lg-12 text-center">
        <input type="hidden" id="story_id" />
        <button id="submit_story">Submit</button>
    </div>
</div>
```

	Remember we were using the **displayStory()** function to show all HTML using jQuery? We need to update displayStory() function with **#story_id** in the HTML. For this let's first add the "**story_id**" in the **stories** array (in index.js) that we had in the previous class. Once the user submits the story we will use this to identify which story's answers have been submitted!	**ESR**: Yes.

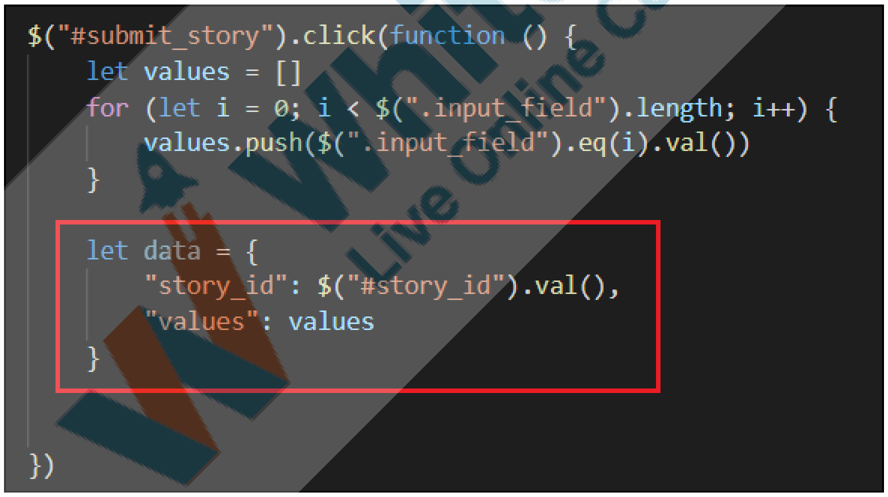


```
let stories = [
    {
        "story_id": "1",
        "inputs": 8,
        "title": "Let's Go to the Zoo",
        "story": `Today we went to the zoo! The first thing we saw was a <span class="rep_input">_____</span> <span class="rep_ir
        "words": ["Black", "Gorilla", "Dancing", "Madagascar", "Nice", "White", "Tigers", "Move"]
    },
    {
        "story_id": "2",
        "inputs": 7,
        "title": "Picnic Time",
        "story": `On <span class="rep_input">_____</span> we are going on a picnic! I'm going with my <span class="rep_input">__
        "words": ["Sunday", "Aunt", "Dog", "Burgers", "Soft Drinks", "Nice", "Cards"]
    },
    {
        "story_id": "3",
        "inputs": 12,
        "title": "Silly Animal Tale",
        "story": `There once was a <span class="rep_input">_____</span> <span class="rep_input">_____</span> from <span class="re
        "words": ["Smelly", "Cat", "California", "Cat", "Blue", "3", "Fishes", "Dance", "Songs", "Sad", "Childishly", "Happy"]
    }
]
```



```
function displayStory(story) {
    $("#story_title").html(story.title)

    $("#bank_words").empty();
    for (let i = 0; i < story.words.length; i++) {
        let html = `<button class="word_bank_button">${story.words[i]}</button>`
        $("#bank_words").append(html)
    }

    $("#input_fields").empty();
    for (let i = 0; i < story.inputs; i++) {
        let input_html = `<input type="text" class="input_field" id="input_${i}" placeholder="Input ${i + 1}"/>`
        $("#input_fields").append(input_html)
    }

    $("#story_text").html(story.story)
    $("#story_id").val(story.story_id)
}
```

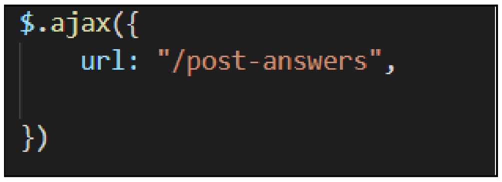
	We can keep the **#story_id** and **values** of all the input_field in a JSON object variable, **data**.	



```
$("#submit_story").click(function () {
    let values = []
    for (let i = 0; i < $(".input_field").length; i++) {
        values.push($(".input_field").eq(i).val())
    }

    let data = {
        "story_id": $("#story_id").val(),
        "values": values
    }

})
```

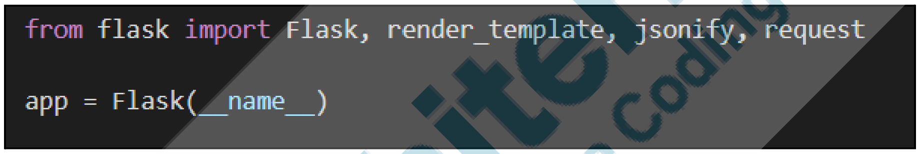
	Now we are going to use the **jQuery ajax()** method to send a POST request to the server. <u>The jQuery ajax() method handles AJAX (asynchronous HTTP) requests, that means, to get and post data from</u>	

	the server without reloading the whole page again!
	
	Remember how we used **fetch()** or **axios()** in React Native?
	The Fetch API and Axios help to call the API at a given URL and get some results corresponding to that URL.
	
	Ajax requests are similar to that!
	
	When we make a request to a server to get or post some data, conventional JavaScript methods redirect(sends requests) to the next route (or we can say next page) after the request is completed or the same page will be reloaded again.
	
	But using jQuery **ajax()** method for making server requests the result is updated on the same page without reloading the whole page again.
	
	Syntax:
	
	`$.ajax({`
	`    name:value,`
	`   name:value,`
	`   ….`
	`   ….`
	`})`
		ESR: Varied.

	There are many possible values for the **name:value parameters** in the ajax() method. We are going to use a few in our AJAX request: ● **url**: URL of the page where we want to send the request; ● **type**: type of the request, GET or POST; ● **data**: actual data that needs to be sent to the server; ● **dataType**: datatype of the response; ● **contentType**: type of the content used while sending a request to the server; ● **success**: a function after the request is successful; and ● **error**: a function after the request fails. Let's write the ajax() method now. We will begin with setting the url parameter.	
		

```
$.ajax({
    url: "/post-answers",

})
```

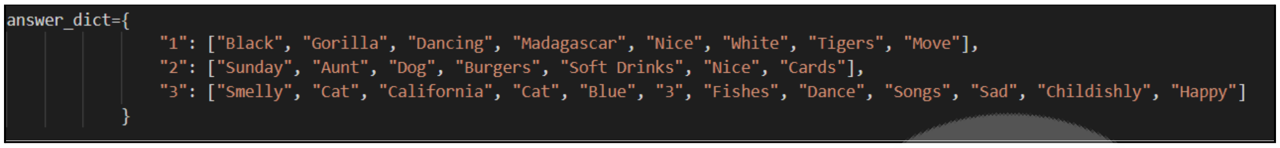
	Why is this url used? Yes! That’s because we have an API already created in the boilerplate code. Let’s take a look at the **“app.py”** file. Here, we are first importing all the important methods and functions from Flask. Next, we are initializing our Flask App	**ESR:** This will help us to call the API.



```
from flask import Flask, render_template, jsonify, request

app = Flask(__name__)
```

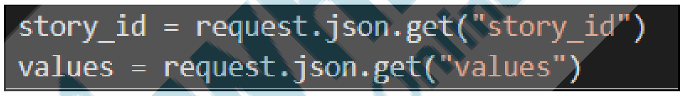
	Next, we have a dictionary in variable **answer_dict** in the following format:
	answer_dict = {
	
	story_id1: [answer1, answer2,...],
	story_id2: [answer1, answer2,...],
	...
	}
	
	Here, we have already mapped the expected answers for all the story IDs, where the **story_id** is the key and its value is the expected answers in order.
	
	We then created an API to render our HTML at the **“/”** url, just as how we	

	discussed above.	



```
answer_dict={
            "1": ["Black", "Gorilla", "Dancing", "Madagascar", "Nice", "White", "Tigers", "Move"],
            "2": ["Sunday", "Aunt", "Dog", "Burgers", "Soft Drinks", "Nice", "Cards"],
            "3": ["Smelly", "Cat", "California", "Cat", "Blue", "3", "Fishes", "Dance", "Songs", "Sad", "Childishly", "Happy"]
        }
```

	Then we have our API on the **“/post-answers”** url. This API responds to a POST request. Since the user will be sending us the **story_id** and **values,** we are fetching that data. Since AJAX is going to send the data in **JSON** format, we are going to use **request.json** to get the values of **story_id** and **values.**	



```
story_id = request.json.get("story_id")
values = request.json.get("values")
```

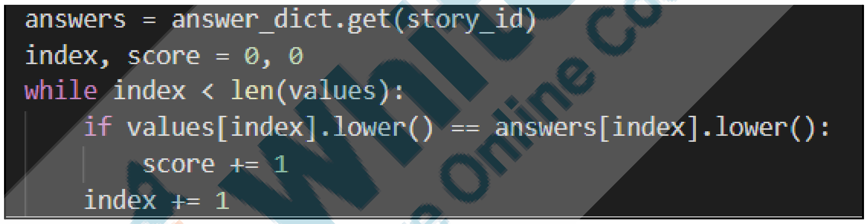
	Next, based on the **story_id** that we receive, we get the answer there in the **answer_dict** and compare the values that user entered v/s what the expected answer should be. Here, one thing to note is that we are using the **.lower()** function while comparing the answers. **.lower()** converts them to lowercase before comparing, i.e., all letters of the word will be lowercase after that as the result. For example:	

- The answer “**Black**” in **answer_dict** after **.lower()**, it will be “**black**”.

- Or if the user types “**blacK**” then after **.lower()**, it will be “**black**”.

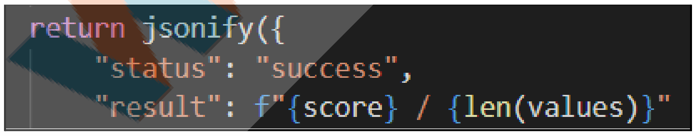
This is done so that users don’t have to worry about matching the case to get the score for that answer.

And then if the user’s answer matches with the expected answer, we are increasing the score by 1.



```python
answers = answer_dict.get(story_id)
index, score = 0, 0
while index < len(values):
    if values[index].lower() == answers[index].lower():
        score += 1
    index += 1
```

We are finally returning the result to the user.



```python
return jsonify({
    "status": "success",
    "result": f"{score} / {len(values)}"
```

Coming back to the AJAX request, we can set the other parameters as:

- **type**: post

- **data**: JSON.stringify(**data**)

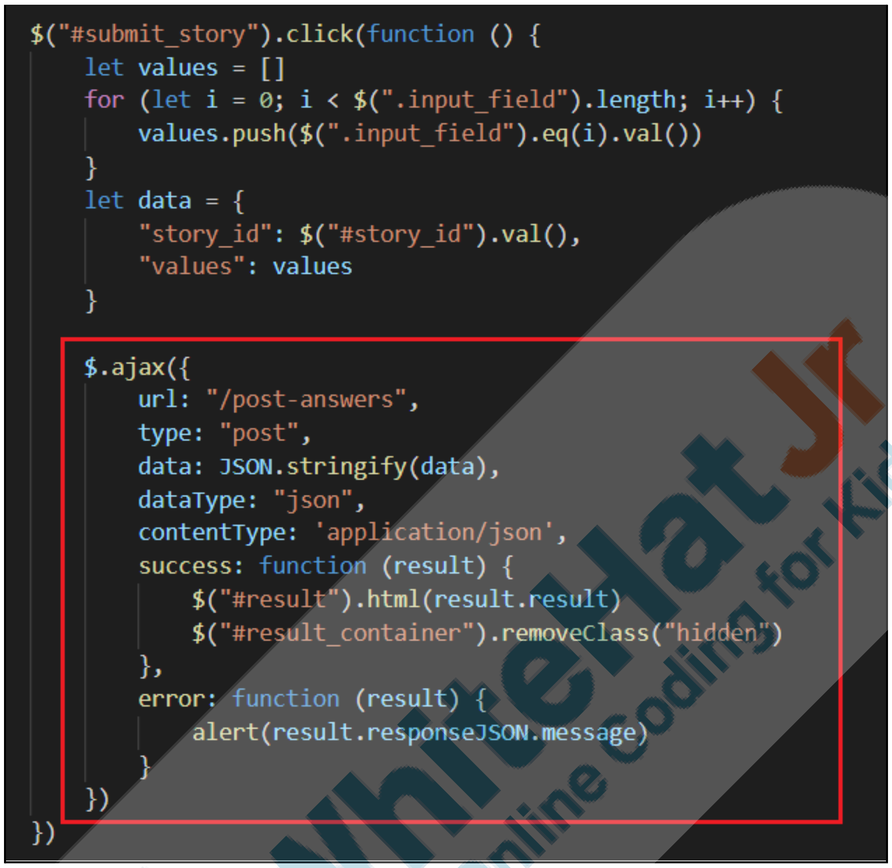
- **dataType**: json
- **contentType**: application/json
- **success**: a **function** to set result using jQuery html()
- **error**: a **function** to send alert

For the **success function**, we first need to update the result in place of **X/Y** that we had in the boilerplate code, and we have to remove the **"hidden"** class to ensure that our result is visible.

In the **error()** function, we can have:

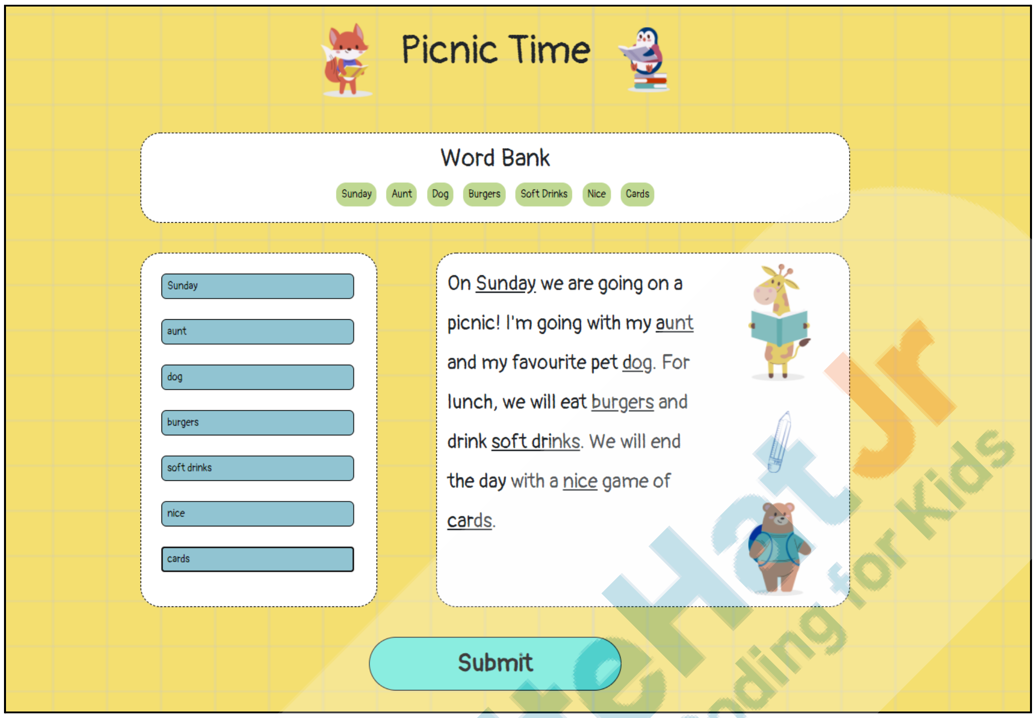
```
alert(result.responseJSON.message
)
```

This will give an alert with the error message, in case of any error.



```
$("#submit_story").click(function () {
    let values = []
    for (let i = 0; i < $(".input_field").length; i++) {
        values.push($(".input_field").eq(i).val())
    }
    let data = {
        "story_id": $("#story_id").val(),
        "values": values
    }

    $.ajax({
        url: "/post-answers",
        type: "post",
        data: JSON.stringify(data),
        dataType: "json",
        contentType: 'application/json',
        success: function (result) {
            $("#result").html(result.result)
            $("#result_container").removeClass("hidden")
        },
        error: function (result) {
            alert(result.responseJSON.message)
        }
    })
})
```

	Now we can test the output.	



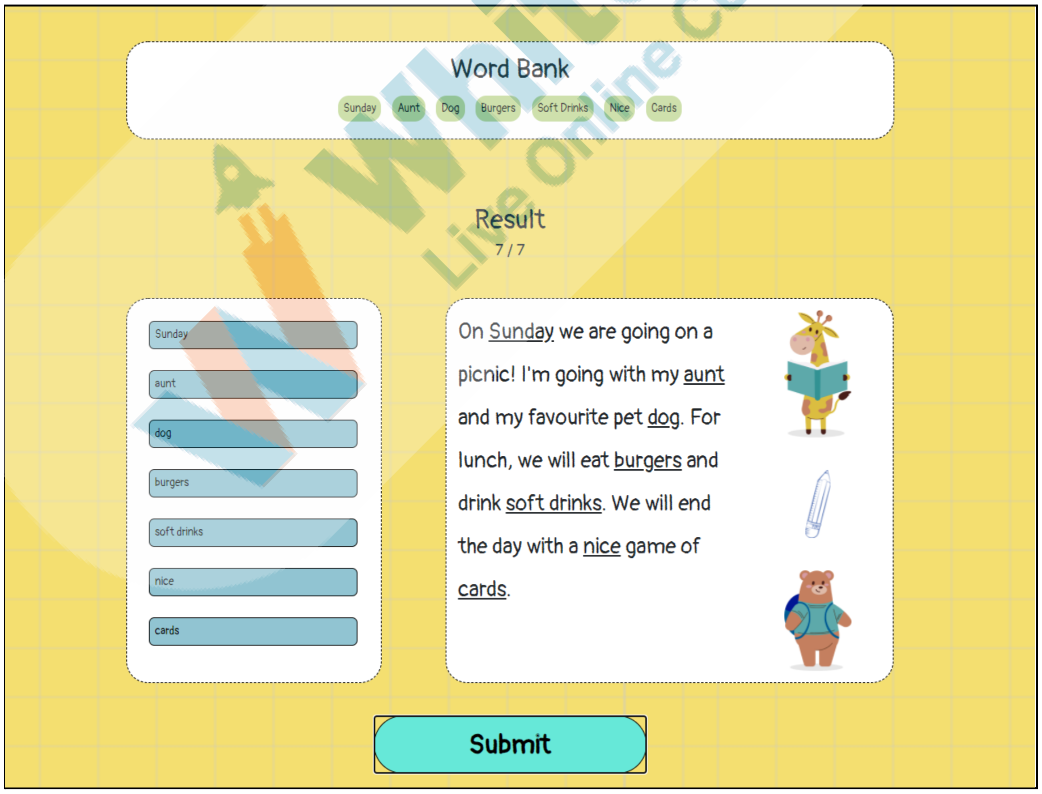
Picnic Time

Word Bank

Sunday Aunt Dog Burgers Soft Drinks Nice Cards

Sunday
aunt
dog
burgers
soft drinks
nice
cards

On Sunday we are going on a picnic! I'm going with my aunt and my favourite pet dog. For lunch, we will eat burgers and drink soft drinks. We will end the day with a nice game of cards.



Word Bank

Sunday Aunt Dog Burgers Soft Drinks Nice Cards

Result

7 / 7

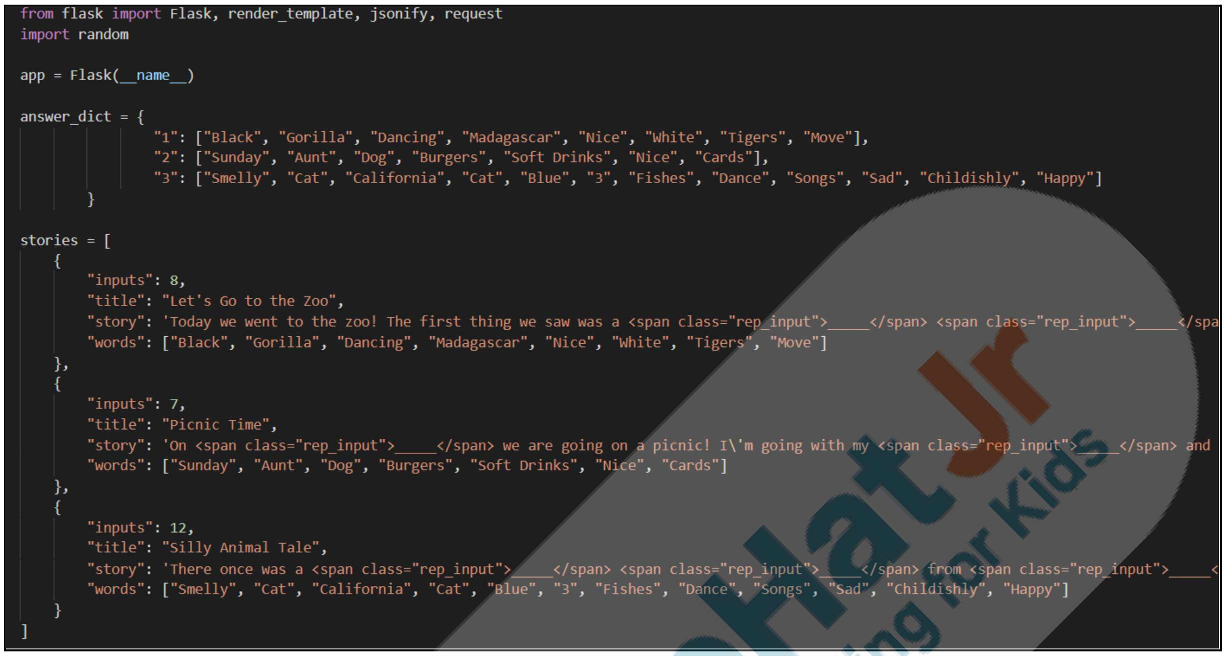
Sunday
aunt
dog
burgers
soft drinks
nice
cards

On Sunday we are going on a picnic! I'm going with my aunt and my favourite pet dog. For lunch, we will eat burgers and drink soft drinks. We will end the day with a nice game of cards.

WhiteHat Jr
Live online Coding for Kids

	That's interesting! We can see that "**Result"** gets updated when we click the "Submit" button! Now you are going to write a jQuery ajax() function which will help the user fetch() stories hosted from the server. Are you excited?	**ESR**: Yes!
Teacher Stops Screen Share		
	Now it's your turn. Please share your screen with me.	
Teacher Starts Slideshow **Slide 14 to 15** Refer to speaker notes and follow the instructions on each slide.		
We have one more class challenge for you. Can you solve it? Let's try. I will guide you through it.		
Teacher Ends Slideshow		
STUDENT-LED ACTIVITY - 20 mins		
● Ask the student to press the ESC key to come back to the panel. **● Guide the student to start screen share.** **● Teacher gets into fullscreen.**		

ACTIVITY ● **Write a jQuery ajax() function to add more Mad Libs' stories.**		
Step 3: Student-Led Activity (20 mins)	*The teacher guides the student to clone the code from Student Activity 1.* *[Student Activity 1]* ***Note**: The student will continue to add new functionality after the teacher activity.*	
	Let's look at the **index.js** file of the boilerplate code. Currently, we have the stories in our JavaScript code. The task is to move the stories to the Flask side and create an API to get a random story with the help of **AJAX** request.	
	To start with, we can move our **stories** variable from **"index.js"** to **"app.py"**	*The student moves the stories.*



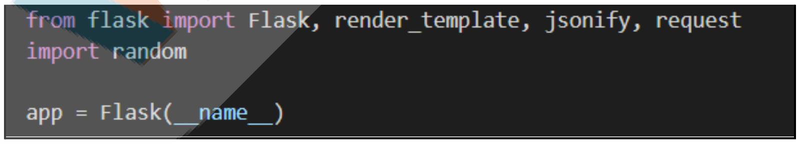
```
from flask import Flask, render_template, jsonify, request
import random

app = Flask(__name__)

answer_dict = {
                "1": ["Black", "Gorilla", "Dancing", "Madagascar", "Nice", "White", "Tigers", "Move"],
                "2": ["Sunday", "Aunt", "Dog", "Burgers", "Soft Drinks", "Nice", "Cards"],
                "3": ["Smelly", "Cat", "California", "Cat", "Blue", "3", "Fishes", "Dance", "Songs", "Sad", "Childishly", "Happy"]
            }

stories = [
    {
        "inputs": 8,
        "title": "Let's Go to the Zoo",
        "story": 'Today we went to the zoo! The first thing we saw was a <span class="rep_input">_____</span> <span class="rep_input">_____</spa
        "words": ["Black", "Gorilla", "Dancing", "Madagascar", "Nice", "White", "Tigers", "Move"]
    },
    {
        "inputs": 7,
        "title": "Picnic Time",
        "story": 'On <span class="rep_input">_____</span> we are going on a picnic! I\'m going with my <span class="rep_input">_____</span> and
        "words": ["Sunday", "Aunt", "Dog", "Burgers", "Soft Drinks", "Nice", "Cards"]
    },
    {
        "inputs": 12,
        "title": "Silly Animal Tale",
        "story": 'There once was a <span class="rep_input">_____</span> <span class="rep_input">_____</span> from <span class="rep_input">_____<
        "words": ["Smelly", "Cat", "California", "Cat", "Blue", "3", "Fishes", "Dance", "Songs", "Sad", "Childishly", "Happy"]
    }
]
```

To get the stories from the server, which HTTP request should we make?

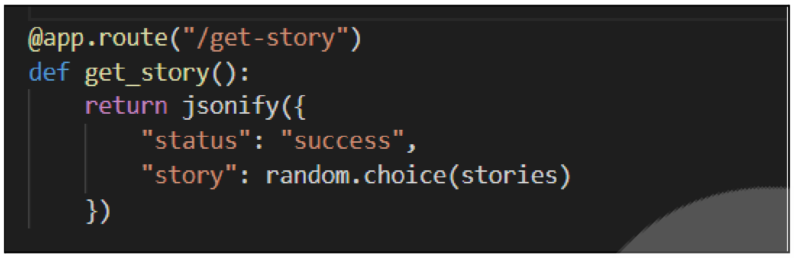
Let's write an API for the GET request in the **app.py** file.

Note that to select a random story, we can use **random.choice()** function and for that, we have to import the **random** module.

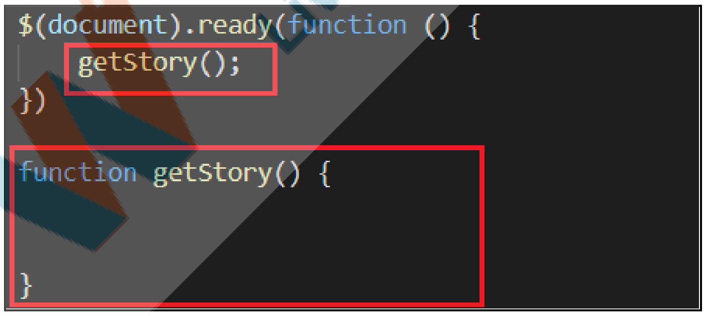
ESR: We should use the GET request.



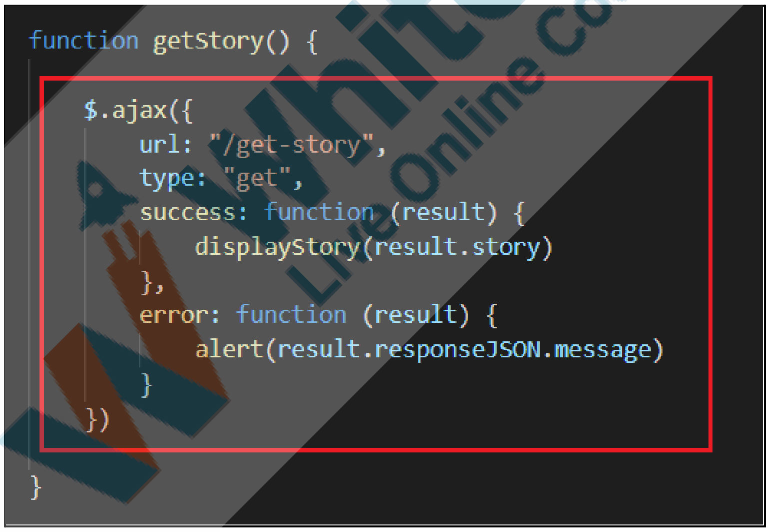
```
from flask import Flask, render_template, jsonify, request
import random

app = Flask(__name__)
```


```
@app.route("/get-story")
def get_story():
    return jsonify({
        "status": "success",
        "story": random.choice(stories)
    })
```

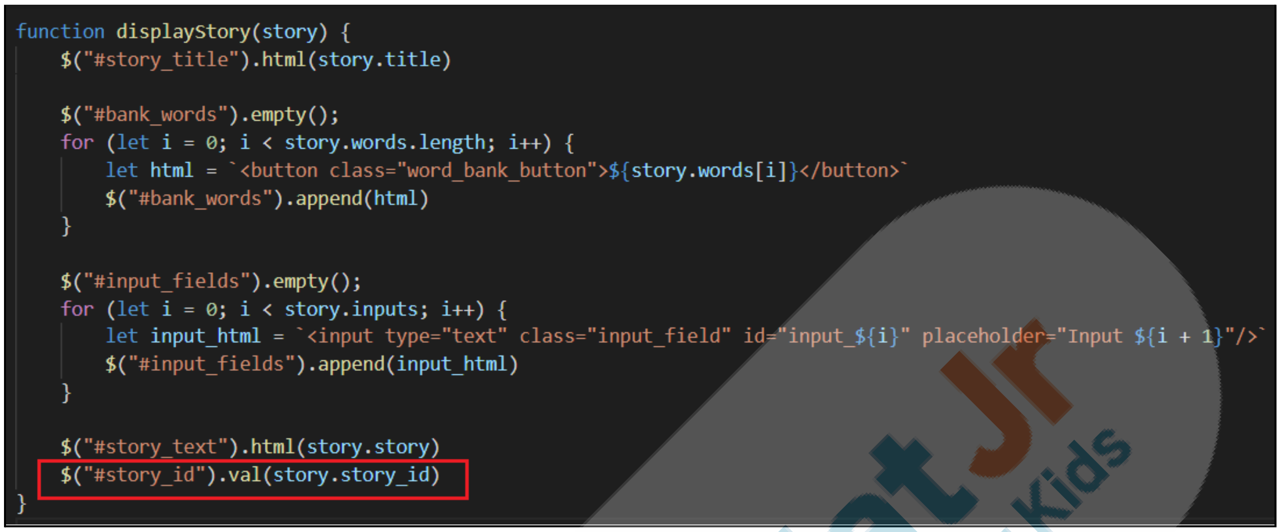
	We can use this route, "**/get-story**", to call the API and display all the stories. We are going to write a function, **getStory()** to call the API using jQuery ajax() method. ***Note***: *The* ***displayStory()*** *had been* ***removed*** *from the* ***.ready() event*** *function.* *The displayStory() will be called only after we get the successful API response inside the* ***getStory()*** *function.*	


```
$(document).ready(function () {
    getStory();
})

function getStory() {

}
```

	Remember how we use the ajax() method? Yes. Great!	**ESR**: We use $.ajax({}) **ESR**: We will need:

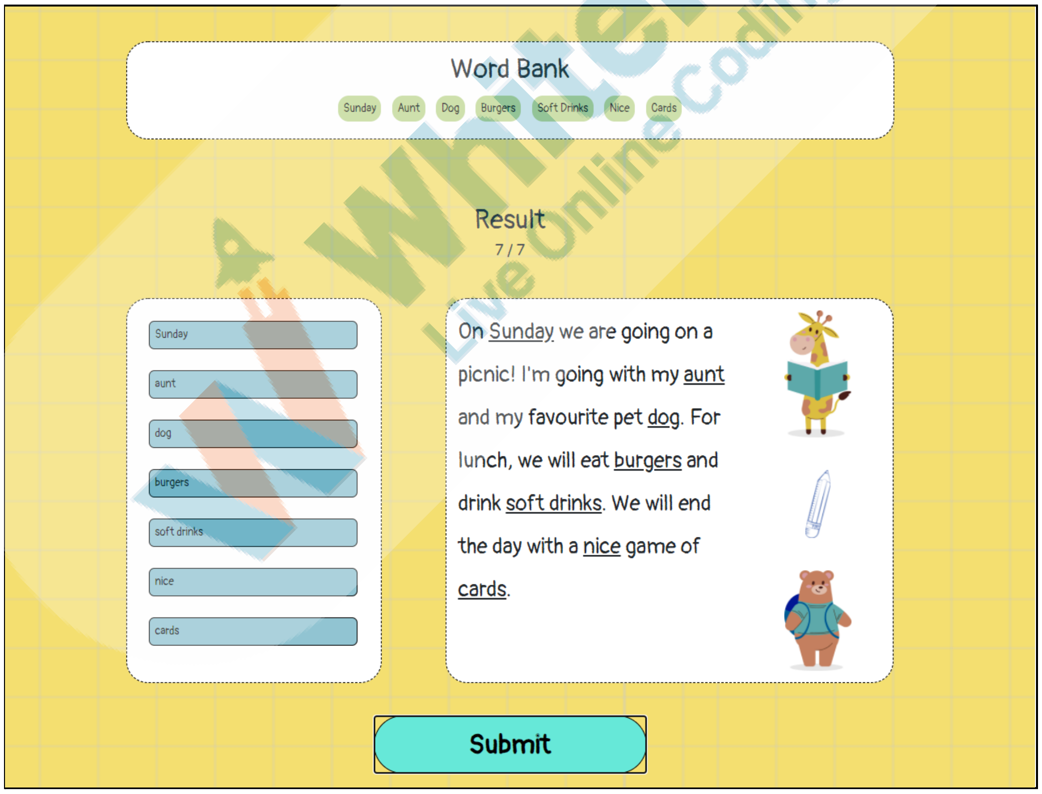
	And what all parameters do we need to make the GET ajax call? Superb! *Guide the student to set the parameters of the ajax() method:* ***url:** "/get-story"* ***type**: post* ***success**: a function to call **displayStory()*** ***error**:a function to send alert message*	url, type as get and success and error parameters for the ajax() method.
 `function getStory() { $.ajax({ url: "/get-story", type: "get", success: function (result) { displayStory(result.story) }, error: function (result) { alert(result.responseJSON.message) } }) }`		
	Also ensure to pass this **story** parameter in **displayStory()** function and update the code accordingly.	

```
function getStory() {
    $.ajax({
        url: "/get-story",
        type: "get",
        success: function (result) {
            displayStory(result.story)
        },
        error: function (result) {
            alert(result.responseJSON.message)
        }
    })
}
```



```
function displayStory(story) {
    $("#story_title").html(story.title)

    $("#bank_words").empty();
    for (let i = 0; i < story.words.length; i++) {
        let html = `<button class="word_bank_button">${story.words[i]}</button>`
        $("#bank_words").append(html)
    }

    $("#input_fields").empty();
    for (let i = 0; i < story.inputs; i++) {
        let input_html = `<input type="text" class="input_field" id="input_${i}" placeholder="Input ${i + 1}"/>`
        $("#input_fields").append(input_html)
    }

    $("#story_text").html(story.story)
    $("#story_id").val(story.story_id)
}
```

Guide the student to test the output.



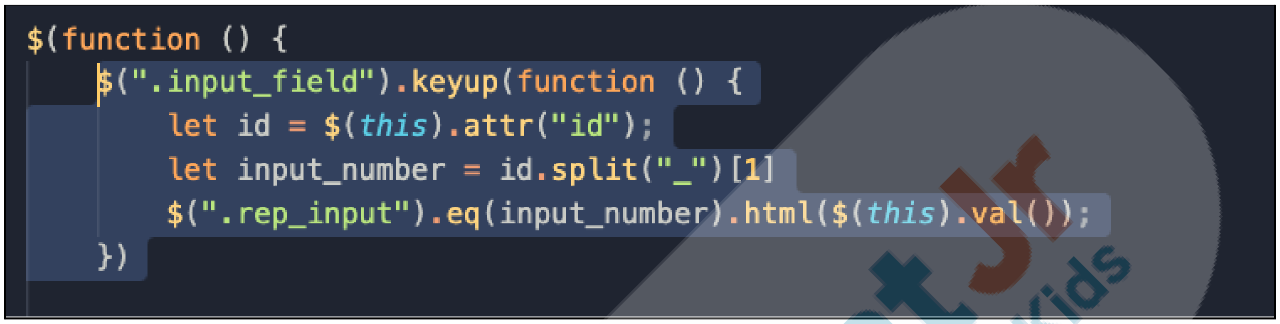
Sunday
aunt
dog
burgers
soft drinks
nice
cards

On Sunday we are going on a picnic! I'm going with my aunt and my favourite pet dog. For lunch, we will eat burgers and drink soft drinks. We will end the day with a nice game of cards.

	Awesome! Can you also try to run it in a new incognito browser?	*The student tries in the incognito browser and notices that the input fields are not getting updated suddenly.*
	Do you know why this happened? That's right! Why did it stop working suddenly?	**ESR:** Browser Caching. **ESR:** Varied.
	The reason why it's not working is because we are now using an **AJAX** request to get our stories and we are then creating the input fields. What's happening now is that when the browser reads the listeners in the **$(function())**, it doesn't find any input fields on the page since our **AJAX** request takes some time to first get the story data and then load it. Now since we were not getting the story through **AJAX** earlier, it was working fine because it was directly rendering the input fields and by the time our **$(function())** was getting initiated, the input fields were already there. To tackle this, one simple thing we can do is, we can move our **$(input).keyup()** listener from our **$(function)** to where we are creating the input fields. This way, our listener	

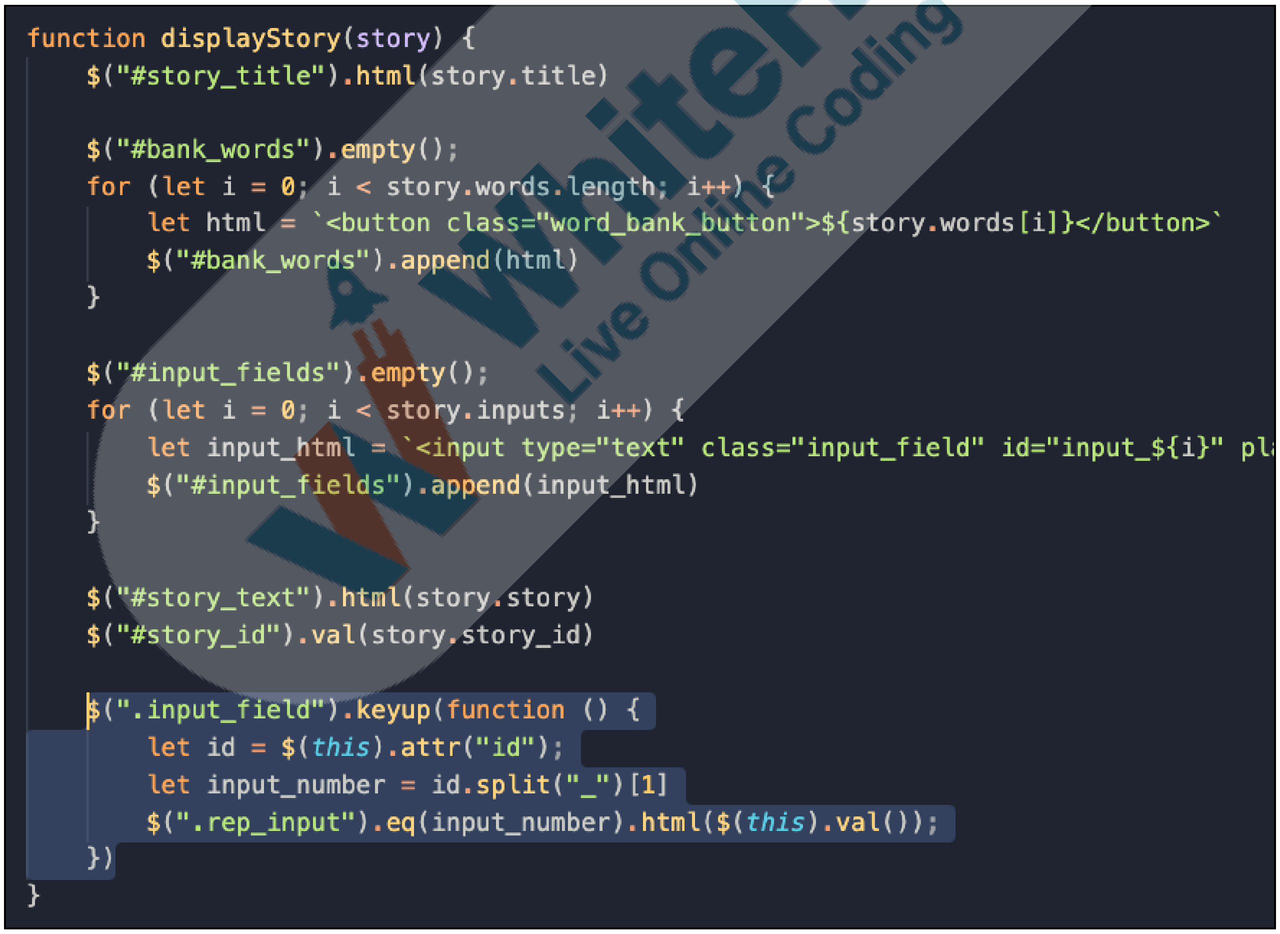
	will only get initiated after the input fields are created -	

Student moves the input keyup listener from here -



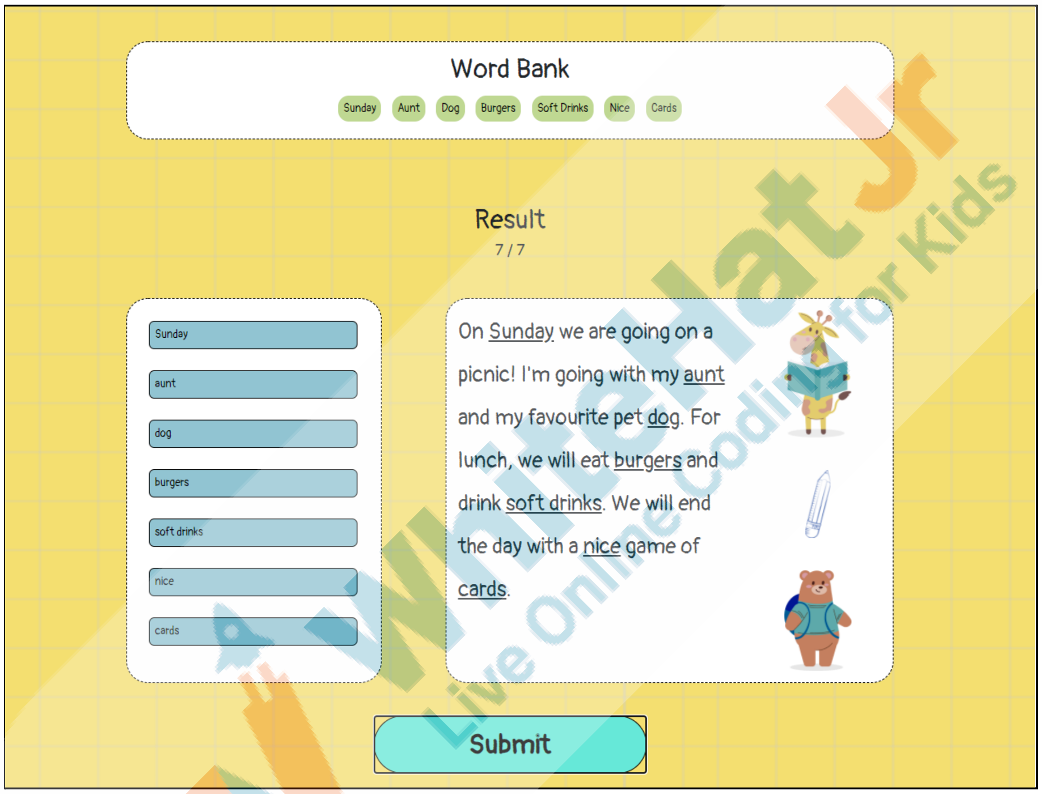
```
$(function () {
    $(".input_field").keyup(function () {
        let id = $(this).attr("id");
        let input_number = id.split("_")[1]
        $(".rep_input").eq(input_number).html($(this).val());
    })
```

To here -



```
function displayStory(story) {
    $("#story_title").html(story.title)

    $("#bank_words").empty();
    for (let i = 0; i < story.words.length; i++) {
        let html = `<button class="word_bank_button">${story.words[i]}</button>`
        $("#bank_words").append(html)
    }

    $("#input_fields").empty();
    for (let i = 0; i < story.inputs; i++) {
        let input_html = `<input type="text" class="input_field" id="input_${i}" pl
        $("#input_fields").append(input_html)
    }

    $("#story_text").html(story.story)
    $("#story_id").val(story.story_id)

    $(".input_field").keyup(function () {
        let id = $(this).attr("id");
        let input_number = id.split("_")[1]
        $(".rep_input").eq(input_number).html($(this).val());
    })
}
```

	Now let's close our incognito window and open it again to see if our code works fine now.	*The student tries again in a new incognito window and sees the output.*



On Sunday we are going on a picnic! I'm going with my aunt and my favourite pet dog. For lunch, we will eat burgers and drink soft drinks. We will end the day with a nice game of cards.

Teacher Guides Student to Stop Screen Share

WRAP UP SESSION - 5 mins

Teacher Starts Slideshow
Slide 16 to 19

Activity details
Following are the WRAP-UP session deliverables:

- Appreciate the student.

- Revise the current class activities.
- Discuss the quizzes.

WRAP-UP QUIZ
Click on In-Class Quiz

Continue WRAP-UP Session
Slide 20 to 25

Activity Details

Following are the session deliverables:
- Explain the facts and trivia
- Next class challenge
- Project for the day
- Additional Activity (Optional)

FEEDBACK
- **Appreciate and compliment the student for trying to learn a difficult concept.**
- **Get to know how they are feeling after the session.**
- **Review and check their understanding.**

Teacher Action	Student Action
You get Hats off for your excellent work!	*Make sure you have given at least 2 Hats Off during the class for:* Creatively Solved Activities +10 Great Question +10 Strong Concentration +10

PROJECT OVERVIEW DISCUSSION Refer the document below in Activity Links Sections		
	Teacher Clicks ✖ End Class	
Additional Activities	*Encourage the student to write reflection notes in their reflection journal using markdown.* Use these as guiding questions: ● What happened today? ○ Describe what happened. ○ The code I wrote. ● How did I feel after the class? ● What have I learned about programming and developing games? ● What aspects of the class helped me? What did I find difficult?	*The student uses the markdown editor to write their reflections in a reflection journal.*

Activity	Activity Name	Links
Teacher Activity 1	Boilerplate Code	https://github.com/whitehatjr/PRO-C177-Boilerplate-Teacher
Teacher Activity 2	Final Reference Code	https://github.com/whitehatjr/PRO-C177-Code-Ref
Student Activity 1	Boilerplate Code	https://github.com/whitehatjr/PRO-C177-Boilerplate-Student
Teacher Reference 1	Project Document	https://s3-whjr-curriculum-uploads.whjr.online/fdc8a36e-7a9f-42d1-b06c-09e9ba8432e1.pdf

Teacher Reference 2	Project Solution	https://github.com/whitehatjr/PRO-C177-Project-Solution
Teacher Reference 3	Visual-Aid	https://s3-whjr-curriculum-uploads.whjr.online/0a614ddf-c365-499c-99cf-889df0badff0.html
Teacher Reference 4	In-Class Quiz	https://s3-whjr-curriculum-uploads.whjr.online/515edf0d-1a9e-4d87-8c40-73bb22d795ce.pdf