DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING (ISE)

Module-5: jQuery and AJAX Integration in Django

5.1 Ajax

Ajax (Asynchronous JavaScript and XML) is not a standalone technology but rather a combination of existing technologies to enable dynamic content updates on web pages without requiring a full page reload. The term "overlaid function" means that Ajax utilizes and coordinates several different technologies to achieve its functionality. These include:

1. **HTML/CSS**: For structuring and styling the web page.
2. **JavaScript**: For handling the dynamic behaviour and making asynchronous requests to the server.
3. **XML/JSON**: For data exchange between the client and server.

5.2 Java Script

- JavaScript is the main engine for Ajax, enabling dynamic, asynchronous interactions on web pages.
- JavaScript works with various technologies, including:
  - **XMLHttpRequest**: For making server requests.
  - **HTML/XHTML/HTML5**: For structuring web pages.
  - **DOM**: For manipulating HTML elements.
  - **CSS**: For styling web pages.
  - **XML/JSON**: For data interchange.

JavaScript Challenges

1. **Language Design Issues**:
   - **Variable Scope**: Functions can interfere with each other's variables unless explicitly scoped with var.

```
function outer() {
   var result = 0;
   for (var i = 0; i < 100; ++i) {
      result += inner(i);
   }
```

```
    return result;
}

function inner(limit) {
    var result = 0;
    for (var i = 0; i < limit; ++i) {
        result += i;
    }
    return result;
}
```

- **Unexpected Behavior**: Inconsistent scoping can lead to bugs.

2. **Cross-Browser Inconsistencies:**
    - **XMLHttpRequest**: Different browsers handle XMLHttpRequest differently, leading to compatibility issues.
    - **Testing**: Developers must test across multiple browsers to ensure consistent behavior.

The Role of JavaScript Libraries

- **Simplifying Development**: Libraries like jQuery provide consistent interfaces for handling XMLHttpRequest, DOM manipulation, and more.
    - **Example**: Using jQuery for an Ajax request:

```
$.ajax({
    url: '/weather',
    method: 'GET',
    data: { city: 'New York' },
    success: function(response) {
        $('#weatherInfo').html(`
            <h2>Weather in ${response.city}</h2>
            <p>Temperature: ${response.temperature}°C</p>
            <p>Condition: ${response.condition}</p>
        `);
    }
});
```

JavaScript: Strengths and Perceptions

- **Misunderstood Language**: Initially considered a "toy" language, JavaScript has matured and gained recognition for its strengths.
- **Comparison to Python**: Like Python, JavaScript is multiparadigm and supports dynamic, object-oriented programming.
    - **Duck Typing**: Both languages use duck typing, allowing flexible and interchangeable objects.
    - **Prototype-Based**: JavaScript's object-oriented model is based on prototypes, allowing dynamic method and property attachment.

Evolution and Acceptance

- **Changing Reputation**: JavaScript is increasingly seen as a powerful and versatile language.
- **jQuery's Role**: jQuery simplifies JavaScript programming, making it more accessible and efficient for developers.

5.3 XHTML Http Request and Response

- The XMLHttpRequest object is crucial for developing complex games using Ajax technologies, such as massive multiplayer online role-playing games (MMORPGs).
- It enables games that rely on network connectivity, such as "Ajax chess," which allows human players to compete against each other over the network instead of playing against a computer engine.

- This object is also fundamental for applications like Gmail, Google Maps, Bing Maps, and Facebook, realizing Sun Microsystems' vision that "the network is the computer."
- Using an XMLHttpRequest object typically involves creating or reusing the object, specifying a callback event handler, opening the connection, sending data, and then having the callback handler retrieve and act on the response once the network operation completes.
- A basic XMLHttpRequest object includes essential methods and properties for handling these operations.

Key Methods of XMLHttpRequest

1. **abort()**:
    - Cancels any active request.
2. **getAllResponseHeaders()**:
    - Returns all HTTP response headers sent with the response.
3. **getResponseHeader(headerName)**:
    - Returns the requested header if available, or a browser-dependent false value if the header is not defined.
4. **open(method, URL, [asynchronous], [username], [password])**:
    - Initializes a request. The method can be GET, POST, HEAD, etc.
    - The URL is the target URL for the request.
    - asynchronous defaults to true. When set to false, it can lock up the visitor's browser.
    - username and password are optional for HTTP authentication.
5. **send(content)**:
    - Sends the request with optional content, which can be a string or a reference to a document.

Key Properties of XMLHttpRequest

1. **onreadystatechange**:
    - A property that holds a reference to the callback function to be executed when the state changes.
2. **readyState**:
    - An integer representing the state of the request:
        - 0: Uninitialized
        - 1: Open
        - 2: Sent

        - 3: Receiving
        - 4: Loaded
3. **responseText**:
    - Contains the response data as a string when the request is complete.
4. **responseXML**:
    - Contains the response data as an XML document.
5. **status**:
    - The HTTP status code returned by the server (e.g., 200 for OK).
6. **statusText**:
    - The HTTP status message returned by the server (e.g., "OK").

Example Usage

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4) {
    if (xhr.status === 200) {
      console.log(xhr.responseText);
    } else {
      console.error('Error: ' + xhr.statusText);
    }
  }
};

xhr.open('GET', 'https://example.com/api/data', true);
xhr.send();
```

Compatibility and Cross-Browser Issues

- Different browsers have various implementations and levels of support for XMLHttpRequest, leading to cross-browser compatibility issues.
- Modern libraries like jQuery provide a consistent interface for Ajax requests, abstracting away these differences.

5.4 HTML

HTML and XHTML are the foundational markup languages for the web. JavaScript and CSS were introduced to complement HTML, although JavaScript has since gained recognition as a standalone language, used in interpreters like SpiderMonkey and Rhino. HTML was the original web markup language, and other web technologies have evolved around it. Even when HTML is re-implemented as XHTML, it retains its core purpose while being more parser-friendly.

```
<HEADER>
<TITLE>The World Wide Web project</TITLE>
<NEXTID N="55">
</HEADER>
<BODY>
```

<H1>World Wide Web</H1>The WorldWideWeb (W3)
is a wide-area
<A NAME=0 HREF="WhatIs.html">
hypermedia</A> information retrieval
initiative aiming to give universal
access to a large universe of documents.<P>
Everything there is online about
W3 is linked directly or indirectly
to this document, including an
<A NAME=24 HREF="Summary.html">executive
summary</A> of the project,
<A NAME=29 HREF="Administration/Mailing/Overview.html">
Mailing lists</A> ,
<A NAME=30 HREF="Policy.html">Policy</A> , November's
<A NAME=34 HREF="News/9211.html">W3 news</A> ,
<A NAME=41 HREF="FAQ/List.html">Frequently Asked Questions
</A> .
<DL>
...

5.5 CSS

- ✓ **Cascading Style Sheets (CSS)** introduced new ways to style web pages, but much of the presentation was already possible before its arrival.
- ✓ CSS didn't just add new styling capabilities; it brought better engineering to the process.
- ✓ The core idea of CSS is to separate presentation from content, making it easier to create well-designed web pages.
- ✓ One of the benefits of CSS is the ability to rebrand websites easily. By changing images and a single stylesheet, you can reskin a website without altering the HTML/XHTML markup.
- ✓ For both Ajax and the broader web, the best practice is to use semantic and structural markup. Then, apply styles through a stylesheet rather than inline, ensuring that elements with the appropriate class or ID have the desired appearance.
- ✓ Tables are still useful and not deprecated, but they should be used for displaying tabular data where using table cells (<td>) and headers (<th>) makes sense. It is discouraged to use tables for positioning content that is not actually tabular data.

5.6 JSON

- ✓ JSON (JavaScript Object Notation) is a simple and effective data format.

- ✓ Unlike formats like XML and ReStructuredText, which require additional parsers in each language, JSON leverages the syntax of JavaScript objects.
- ✓ It makes a few adjustments to ensure cross-browser compatibility and simplicity. JSON's clarity and conciseness have made it popular not only in JavaScript but also in other programming languages.
- ✓ Although other languages need a parser for JSON, it is still preferred for inter-language communication.
- ✓ In JavaScript, while you can't use eval() directly due to security concerns, JSON remains a powerful tool for object declaration and data exchange.

5.7 iFrames

Iframes (short for inline frames) are HTML elements that allow you to embed another HTML document within the current document. They are often used to display content from another source within a web page, such as including a YouTube video, an advertisement, or content from another website.

Basic Syntax

```
<iframe src="https://example.com" width="600" height="400"></iframe>
```

- **src**: The URL of the page to embed.
- **width**: The width of the iframe.
- **height**: The height of the iframe.

Key Attributes

1. **src**: Specifies the URL of the document to be embedded.
2. **width** and **height**: Define the size of the iframe.
3. **name**: Allows the iframe to be targeted by links and forms.
4. **frameborder**: Sets the width of the border around the iframe. A value of "0" means no border.
5. **scrolling**: Controls the appearance of the scrollbars. It can take the values "yes", "no", or "auto".
6. **allowfullscreen**: Allows the iframe to be displayed in fullscreen mode.
7. **sandbox**: Adds extra restrictions on the content in the iframe, such as disallowing scripts or forms.

5.8 Settings of Java Script in Django

When developing a Django project, serving static content such as images, CSS, and JavaScript files is essential for proper functionality and aesthetics. While a different approach is recommended for production, the following steps outline how to set up static content for development purposes.

Steps:

1. **Create a Static Directory**

First, create a directory named static within your Django project. This directory will hold all your static files.

mkdir static

Inside the static directory, create subdirectories for CSS, images, and JavaScript files:

mkdir static/css static/images static/js

2. **Modify settings.py**

Open your project's settings.py file and add the following configuration:

- **Add the DIRNAME variable after importing os:**

```
import os

DIRNAME = os.path.abspath(os.path.dirname(__file__))
```

- **Set the MEDIA_ROOT and MEDIA_URL to point to your static directory:**

```
MEDIA_ROOT = os.path.join(DIRNAME, 'static/')
MEDIA_URL = '/static/'
```

3. **Serve Static Files During Development**

At the end of the settings.py file, add the following snippet to serve static files when DEBUG is True:

```
if DEBUG:
    from django.conf.urls.static import static
    from django.conf import settings

    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

This code dynamically adds URL patterns to serve static files during development.

Example settings.py

```
import os

# Add this line to define DIRNAME
DIRNAME = os.path.abspath(os.path.dirname(__file__))

# Media settings
MEDIA_ROOT = os.path.join(DIRNAME, 'static/')
MEDIA_URL = '/static/'

# Other settings...

# Add this block at the end of the file
if DEBUG:
```

```
from django.conf.urls.static import static
from django.conf import settings

urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Serving Static Files in Production

For production, it is recommended to use a web server optimized for serving static content, such as Nginx or a stripped-down Apache. Here's a brief overview of how you can serve static files in production with Nginx:

1. **Collect Static Files**

   ```
   python manage.py collectstatic
   ```

2. **Configure STATIC_ROOT**

   In settings.py, add the STATIC_ROOT setting:

   ```
   STATIC_ROOT = os.path.join(DIRNAME, 'staticfiles/')
   ```

3. **Nginx Configuration**

   Configure Nginx to serve the static files from the STATIC_ROOT directory. Here's a sample Nginx configuration:

   ```
   server {
      listen 80;
      server_name your_domain.com;

      location /static/ {
         alias /path/to/your/project/staticfiles/;
      }
      location / {
         proxy_pass http://127.0.0.1:8000;
         proxy_set_header Host $host;
         proxy_set_header X-Real-IP $remote_addr;
         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
         proxy_set_header X-Forwarded-Proto $scheme;
      }
   }
   ```

Replace /path/to/your/project/staticfiles/ with the actual path to your STATIC_ROOT directory.

5.9 jQuery and Basic AJAX

The jQuery approach simplifies and streamlines JavaScript operations, particularly when working with the DOM and Ajax. It allows for clean, readable, and maintainable code by leveraging the power of wrapped sets and method chaining.

Example for Basic Ajax "Hello, World!" with JavaScript and jQuery

Traditional JavaScript Approach

In traditional JavaScript, making an Ajax request involves several steps:

1) **Creating an XMLHttpRequest object:**

```
var xhr = new XMLHttpRequest();
```

2) **Configuring the request with the HTTP method and the URL:**

```
xhr.open("GET", "/project/server.cgi?text=world");
```

3) **Defining a callback function to handle the server response:**

```
callback = function() {
    if (xhr.readyState == 4 && xhr.status >= 200 && xhr.status < 300) {
        document.getElementById("result").innerHTML = xhr.responseText;
    }
};
xhr.onreadystatechange = callback;
```

4) **Sending the request:**

```
xhr.send(null);
```

jQuery Approach

With jQuery, the process is much simpler and more concise. Here's how you can achieve the same result using jQuery:

1. Select the HTML element (a wrapped set):

```
$("#result")
```

2. Load content into the selected element from a server URL, optionally specifying a query string:

```
$("#result").load("/project/server.cgi", "text=world");
```

jQuery Wrapped Sets

A wrapped set in jQuery is an object that contains a set of DOM elements and provides a rich set of operations you can perform on these elements

examples of creating a wrapped set:

$("#result"): Selects the element with the ID result.
$("p"): Selects all paragraph elements.
$("p.summary"): Selects all paragraph elements with the class summary.

$("a"): Selects all anchor tags.
$("p a"): Selects all anchor tags inside paragraph elements.
$("p > a"): Selects all anchor tags whose immediate parent is a paragraph tag.
$("li > p"): Selects all p tags whose immediate parent is an li tag.
$("p.summary > a"): Selects all anchor tags whose immediate parent is a paragraph tag with class summary.
$("table.striped tr:even"): Selects even-numbered rows from all tables belonging to the class striped.

Key Features of Wrapped Sets:

1. Chaining Operations:

jQuery operations return the same wrapped set, allowing methods to be chained together.
Example:

```
$("table.striped tr:even").addClass("even").hide("slow").delay(1000).show("slow");
```

2. Operation Examples:

Select even-numbered table rows from tables having the class striped.

```
$("table.striped tr:even")
```

- **Add the class even to these rows.**

```
.addClass("even")
```

- **Slowly hide these rows.**

```
.hide("slow")
```

- **Wait for 1000 milliseconds.**

```
.delay(1000)
```

- **Slowly show these rows again.**

```
.show("slow")
```

5.10 jQuery Ajax Facilities

jQuery provides robust tools for making Ajax requests and handling responses efficiently.

1. $.ajax()

The core method for Ajax requests.

```
$.ajax({
  url: "/update-user",
  type: "POST",
  data: "surname=Smith&cartTotal=12.34",
  dataType: "text",
  success: function(data, textStatus, XMLHttpRequest) {
    try {
      updatePage(JSON.parse(data));
    } catch(error) {
      displayErrorMessage("Error updating your shopping cart. Please call customer service
    }
  },
  error: function(XMLHttpRequest, textStatus, errorThrown) {
    displayErrorMessage("An error has occurred: " + textStatus);
  }
});
```

Parameters in $.ajax()

- **context**: Allows access to variables in callbacks.
- **data**: Data sent with the request.
- **dataFilter**: Function to process raw response.
- **dataType**: Expected data type of response (text, json, html, etc.).
- **error**: Callback for handling errors.
- **success**: Callback for handling successful response.
- **type**: HTTP method (GET, POST, etc.).
- **url**: The URL to send the request to.

Example of context and closures

```
$.ajax({
  url: "/user-info",
  type: "POST",
  context: {
    name: prompt("What is your name?", ""),
    email: prompt("What is your email address?", "")
  },
  success: function(data, textStatus, XMLHttpRequest) {
    alert(this.name + ", your email address is " + this.email + ".");
    processData(data, this.name, this.email);
  }
});
```

2. $.ajaxSetup()

Sets default values for $.ajax() calls.

```
$.ajaxSetup({
  dataType: "text",
  type: "POST"
});
```

Convenience Methods: **$.get() and $.post()**

Simplify $.ajax() calls for GET and POST requests.

- **GET Example:**

```
$.get("/resources/update", function(data) {
  $("#result").html(data);
});
```

- **POST Example:**

```
$.post("/resources/update", { user: "jsmith", product_id: 112 }, function(data) {
  $("#result").html(data);
});
```

- **.load()**

Loads data from the server and places the returned HTML into matched elements.

Example:

```
$("#messages").load("/sitewide-messages");

$("#messages").load("/user-messages", "username=jsmith");

$("#hidden").load("/user-customizations", "username=jsmith", function(responseText, textStatus, XMLHttpRequest) {

  performUserCustomizations(responseText);

});
```

Key Points

- Convenience methods like $.get(), $.post(), and .load() are easier to use but lack detailed error handling.
- Error handling: Prefer $.ajax() for robust error handling or use global error handlers with convenience methods.
- Context and closures: Useful for maintaining state in callbacks.
- Chaining: Be cautious with chaining methods after .load() due to potential race conditions.

5.11 Using jQuery UI Autocomplete in Django

jQuery is a powerful and lightweight JavaScript library designed to simplify various web development tasks, such as:

- **HTML Document Traversal and Manipulation:** Allows for easy selection and manipulation of HTML elements.
- **Event Handling:** Simplifies attaching event listeners to elements, managing events like clicks, form submissions, and more.
- **Animation:** Enables creating dynamic and interactive effects.
- **Ajax Interactions**: Simplifies making asynchronous HTTP requests for a seamless user experience.

Key Features and Benefits of jQuery

1) **DOM Manipulation:**

- Ease of Use: Provides a straightforward API for selecting and manipulating elements within the Document Object Model (DOM).
- Flexibility: You can easily traverse the DOM tree, modify element attributes and content, and add or remove elements from the page.

2) **Event Handling:**

- Simplified Process: jQuery makes it simple to attach event listeners to HTML elements.

- User Interactions: Handles various user interactions such as clicks, hovers, form submissions, and more.

To integrate jQuery UI's Autocomplete widget with a Django application, follow these steps:

1. Set Up Django View: Create a Django view that will serve the data for the autocomplete suggestions. This view will return a JSON response.

```
from django.http import JsonResponse
from .models import YourModel

def autocomplete_view(request):
    if 'term' in request.GET:
        qs = YourModel.objects.filter(name__icontains=request.GET.get('term'))
        names = list()
        for name in qs:
            names.append(name.name)
        return JsonResponse(names, safe=False)
    return JsonResponse([], safe=False)
```

2. Configure URL: Map the view to a URL in your urls.py file.

```
from django.urls import path
from .views import autocomplete_view

urlpatterns = [
    path('autocomplete/', autocomplete_view, name='autocomplete'),
]
```

3. Include jQuery and jQuery UI in Your Template: Make sure to include the jQuery and jQuery UI libraries in your template. You can use CDN links for simplicity.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Autocomplete Example</title>
    <link rel="stylesheet" href="https://code.jquery.com/ui/1.12.1/themes/base/jquery-ui.cs
    <script src="https://code.jquery.com/jquery-3.5.1.min.js"></script>
    <script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js"></script>
</head>
<body>
    <input type="text" id="autocomplete" placeholder="Start typing...">
    <script>
        $(document).ready(function() {
            $("#autocomplete").autocomplete({
                source: "{% url 'autocomplete' %}"
            });
        });
    </script>
</body>
</html>
```

4. Testing the Autocomplete

Run your Django server and navigate to the page with the autocomplete input field. Start typing in the input field to see the autocomplete suggestions in action.