MODULE-5: JQUERY AND AJAX INTEGRATION IN DJANGO

Django, a popular web development framework, excels in building high-quality, scalable web applications. One of its standout features is its seamless handling of asynchronous requests, facilitated by Ajax (Asynchronous JavaScript and XML). Ajax allows web pages to update asynchronously without the need for a page reload. In this blog post, we'll delve into how to harness the power of Ajax within Django.

To integrate Ajax into Django, you'll first need to incorporate the necessary JavaScript libraries into your project. Several options exist for this purpose, including jQuery, Vanilla JavaScript, and ReactJS.

5.0 AJAX SOLUTION

Ajax (Asynchronous JavaScript and XML) is a web development technique that enables web pages to update asynchronously without requiring a full page reload. This allows for a smoother and more dynamic user experience by fetching and displaying data from the server in the background, without disrupting the current page.

In Django, Ajax is commonly used to enhance interactivity and responsiveness in web applications.

Client-Side Implementation (JavaScript):

- On the client-side (in the browser), JavaScript is used to make asynchronous requests to the server.
- Typically, libraries like jQuery are used to simplify Ajax calls, although it's also possible to use vanilla JavaScript or other libraries/frameworks like Axios or Fetch API.
- JavaScript code is written to handle events, such as button clicks or form submissions, and trigger Ajax requests to the Django server.

Server-Side Handling (Django Views):

- In Django, Ajax requests are handled by views, just like regular HTTP requests.
- Django views receive Ajax requests, process the data (if any), and return a response.
- The response can be in various formats, such as JSON, XML, HTML, or plain text, depending on the requirements of the application.

Communication between Client and Server:

- When a user interacts with a page (e.g., clicks a button), JavaScript code triggers an Ajax request.

- The Ajax request is sent to a specific URL, typically mapped to a Django view.
- The Django view processes the request, performs any necessary operations (such as database queries or computations), and generates a response.
- The response is sent back to the client, where it can be processed and used to update the DOM (Document Object Model) dynamically, without a full page reload.

Updating the DOM (Client-Side):

- Once the response is received from the server, JavaScript code on the client-side processes it.
- Depending on the content of the response, the DOM may be updated to reflect changes, display new data, or show error messages.
- This process typically involves manipulating HTML elements or updating the content of specific elements on the page.

Error Handling and Validation:

- Error handling is an essential aspect of Ajax development. Both client-side and server-side code should include mechanisms to handle errors gracefully.
- On the client-side, error callbacks can be used to handle situations such as network errors or server-side failures.
- On the server-side, Django views should validate input data, handle exceptions, and return appropriate error responses when necessary.

CSRF Protection:

- When using Ajax with Django for POST requests, it's crucial to protect against Cross-Site Request Forgery (CSRF) attacks.
- Django provides built-in CSRF protection mechanisms, such as {% csrf_token %} template tag or csrfmiddlewaretoken token in Ajax requests, to ensure the security of your application.

By leveraging Ajax in Django, developers can create more dynamic and interactive web applications, enhancing the overall user experience. However, it's essential to use Ajax judiciously and consider factors such as performance, accessibility, and security while implementing Ajax functionality.

Example:

let's create a simple example to demonstrate how to use Ajax in Django. In this example, we'll create a Django application that allows users to submit a form asynchronously using Ajax. When the form is submitted, the data will be sent to a Django view using Ajax, and the server will respond with a success message.

Create a Django Project and App:

If you haven't already, create a new Django project and an app within it. Let's call the project "ajax_example" and the app "ajax_app".

```
django-admin startproject ajax_example

cd ajax_example

python manage.py startapp ajax_app
```

Define URLs:

Configure the URL patterns to route requests to the appropriate views.

```
# ajax_example/urls.py

from django.urls import path

from ajax_app import views

urlpatterns = [

    path('', views.index, name='index'),

    path('ajax_submit/', views.ajax_submit, name='ajax_submit'),

]
```

Create Templates:

Create HTML templates for the index page and the success page.

```
<!-- ajax_app/templates/index.html -->

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Ajax Form Submission</title>

    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>

</head>
```

```
<body>
  <h1>Ajax Form Submission</h1>
  <form id="ajax-form">
    <label for="input-data">Enter Data:</label>
    <input type="text" id="input-data" name="input_data">
    <button type="submit">Submit</button>
  </form>
  <div id="result"></div>
  <script src="{% static 'ajax_app/js/main.js' %}"></script>
</body>
</html>
```

Write JavaScript for Ajax:

Write JavaScript code to handle the form submission using Ajax.

```
// ajax_app/static/ajax_app/js/main.js
$(document).ready(function() {
  $('#ajax-form').submit(function(event) {
    event.preventDefault(); // Prevent the default form submission
    var inputData = $('#input-data').val(); // Get the input value
    $.ajax({
      type: 'POST',
      url: '/ajax_submit/', // URL to submit the form data
      data: {'input_data': inputData}, // Form data to be submitted
      success: function(response) {
        $('#result').text(response.message); // Display the response message
      },
      error: function(xhr, errmsg, err) {
            $('#result').text('Error occurred while submitting the form.'); // Display error
message
          }
        });
    });
});
```

Define Views:

Define Django views to handle the form submission.

```
# ajax_app/views.py
from django.shortcuts import render
from django.http import JsonResponse
def index(request):
    return render(request, 'ajax_app/index.html')
def ajax_submit(request):
    if request.method == 'POST' and request.is_ajax():
        input_data = request.POST.get('input_data')
        # Process the input data (e.g., save to database)
        response_data = {'message': 'Form submitted successfully!'}
        return JsonResponse(response_data)
    else:
        return JsonResponse({'error': 'Invalid request'})
```

Run the Server:

Finally, run the Django development server to see the application in action.

```
python manage.py runserver
```

Now, when you navigate to the homepage (http://127.0.0.1:8000/), you should see a form where you can enter data. When you submit the form, the data will be sent asynchronously using Ajax to the Django server. The server will respond with a success message, which will be displayed on the page without a full page reload.

5.1 JAVASCRIPT

JavaScript is a versatile programming language commonly used for web development. It allows developers to add interactivity, dynamic behavior, and functionality to web pages.

Client-Side Scripting: JavaScript primarily runs on the client-side (in the user's web browser), unlike server-side languages like Python (used in Django). This means JavaScript code is executed on the user's device, allowing for real-time interaction without needing to communicate with the server for every action.

Dynamic Content: JavaScript enables the manipulation of HTML and CSS, allowing developers to dynamically update the content and styling of web pages based on user interactions or other events. This includes actions like showing/hiding elements, changing text or images, and animating elements.

Event Handling: JavaScript allows developers to respond to various events triggered by user interactions, such as mouse clicks, keyboard inputs, form submissions, and page load/completion. Event handlers can be attached to HTML elements to execute JavaScript code when the event occurs.

DOM Manipulation: The Document Object Model (DOM) is a programming interface that represents the structure of an HTML document as a tree of nodes. JavaScript can interact with and manipulate the DOM, enabling the addition, removal, or modification of elements and their attributes on the fly.

Asynchronous Programming: JavaScript supports asynchronous programming, allowing tasks to be executed concurrently without blocking the main execution thread. This is crucial for tasks such as making Ajax requests, handling timeouts/intervals, and processing data in the background.

Libraries and Frameworks: JavaScript has a vast ecosystem of libraries and frameworks that streamline web development tasks and provide additional functionality. Popular libraries like jQuery simplify DOM manipulation and Ajax requests, while frameworks like React, Angular, and Vue.js offer more structured approaches to building complex web applications.

Browser Compatibility: JavaScript code can run on most modern web browsers, including Chrome, Firefox, Safari, and Edge. However, developers need to consider browser compatibility and may need to use polyfills or feature detection to ensure consistent behavior across different browsers.

Security Considerations: As JavaScript executes on the client-side, it's important to consider security risks such as Cross-Site Scripting (XSS) attacks, where malicious scripts are injected into web pages. Proper input validation, output encoding, and secure coding practices are essential for mitigating these risks.

Overall, JavaScript is a fundamental technology for web development, empowering developers to create interactive and engaging web applications that enhance the user experience

Example:

```
<!DOCTYPE html>

<html>

<head>

  <title>JavaScript Example</title>

</head>

<body>

  <h1 id="heading">Hello, World!</h1>

  <button id="change-text">Change Text</button>

  <script>

    // JavaScript code to handle button click event and modify the heading text

    document.getElementById("change-text").addEventListener("click", function() {

      var heading = document.getElementById("heading");

      heading.textContent = "Text Changed!";

      heading.style.color = "red";

    });

  </script>

</body>

</html>
```

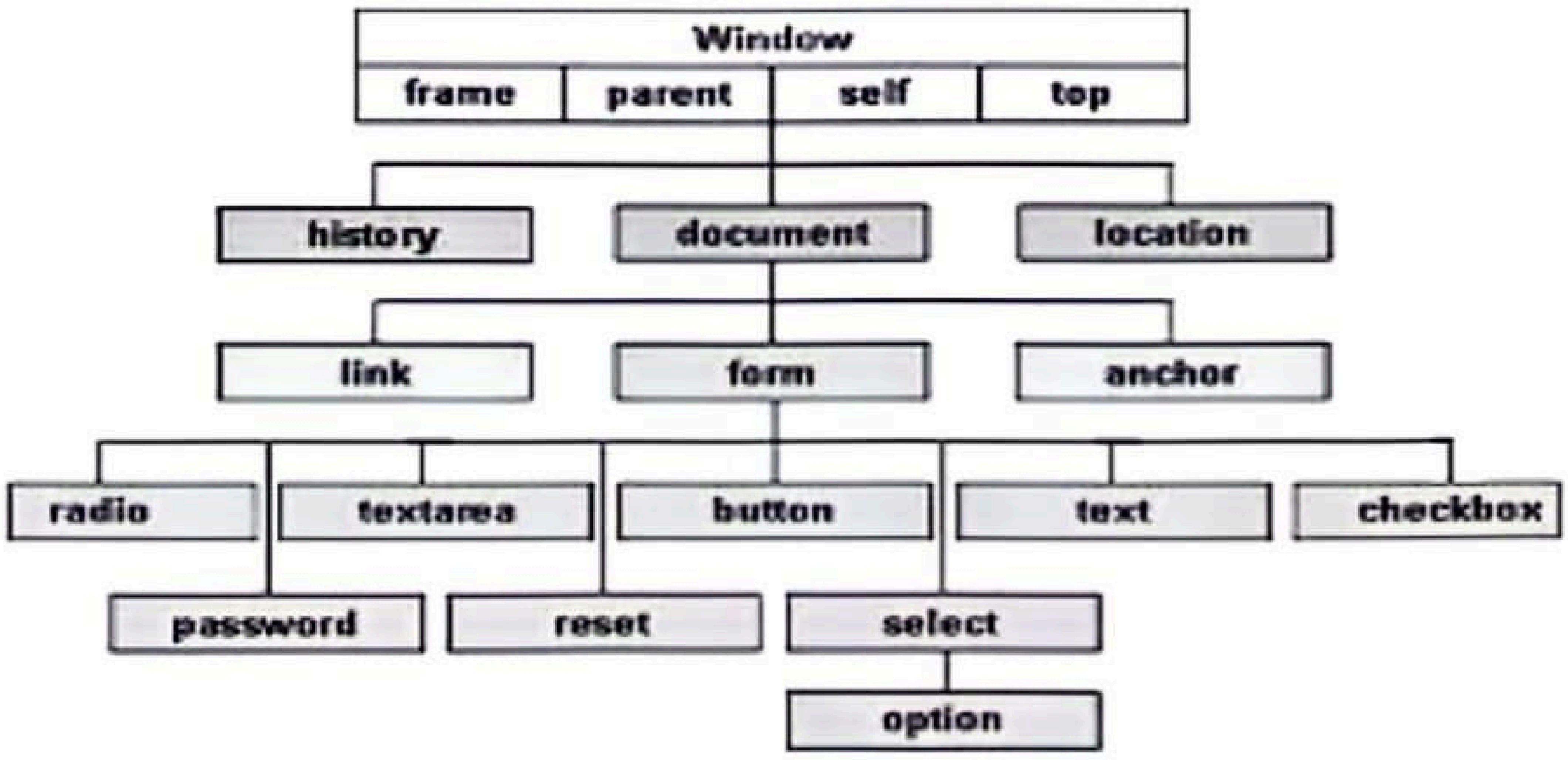
In this example:

- HTML defines a heading and a button.
- JavaScript code adds an event listener to the button, which changes the heading text and color when clicked.
- The textContent property is used to modify the text content of the heading element.
- The style property is used to modify the CSS style of the heading element.

Quickglance:

Introduction to JavaScript

- JavaScript is a scripting language primarily used for client-side web development.
- It was developed by Brendan Eich at Netscape Communications in 1995.
- JavaScript is used to add interactivity and dynamic behavior to web pages.



Basic Syntax and Data Types

Variables: var, let, const are used to declare variables.

var age = 30;

let name = "John";

const PI = 3.14;

Data types: strings, numbers, booleans, null, undefined.

var name = "John";

var age = 30;

var isStudent = true;

var car = null;

var test;

Control Structures

Conditional statements: if, else if, else.

var age = 18;

```
if (age >= 18) {
   console.log("You are an adult.");
} else {
   console.log("You are a minor.");
}
```

Loops: for, while, do-while.

```
for (var i = 0; i < 5; i++) {
   console.log(i);
}
```

Functions

Functions are blocks of reusable code.

```
function greet(name) {
   return "Hello, " + name + "!";
}
console.log(greet("John"));
```

Arrow functions:

```
var add = (a, b) => a + b;
console.log(add(2, 3)); // Output: 5
```

Arrays and Objects

Arrays: Ordered collections of values.

```
var fruits = ["apple", "banana", "orange"];
console.log(fruits[0]); // Output: apple
```

Objects: Key-value pairs.

```
var person = { name: "John", age: 30 };
console.log(person.name); // Output: John
```

DOM Manipulation

DOM represents the structure of HTML documents.

```
document.getElementById("myButton").addEventListener("click", function() {
  alert("Button clicked!");
});
```

Introduction to Asynchronous JavaScript

setTimeout: Executes a function after a specified time.

```
setTimeout(function() {
  console.log("Delayed message.");
}, 2000); // Execute after 2 seconds
```

Error Handling and Debugging

try-catch blocks: Handle errors gracefully.

```
try {
  // Code that may throw an error
} catch (error) {
  // Handle the error
}
```

ES6 Features

Template literals: Allows embedding expressions in strings.

```
var name = "John";
console.log(`Hello, ${name}!`);
```

Asynchronous Programming

Promises: Handle asynchronous operations.

```
var promise = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve("Data fetched successfully.");
  }, 2000);
});
promise.then(function(data) {
  console.log(data);
});
```

AJAX and Fetch API

Fetch API: Fetch data from a server asynchronously.

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

Local Storage

Store data locally in the browser.

```
localStorage.setItem("name", "John");
var name = localStorage.getItem("name");
console.log(name); // Output: John
```

Advanced DOM Manipulation

Event delegation: Handling events on dynamically created elements.

```
document.addEventListener("click", function(event) {
  if (event.target.matches("button")) {
    console.log("Button clicked.");
  }
});
```

Object-Oriented JavaScript

Prototypes and inheritance: Define methods shared among objects.

```
function Person(name) {
  this.name = name;
}
Person.prototype.greet = function() {
  return "Hello, " + this.name + "!";
};
var john = new Person("John");
console.log(john.greet());
```

Module Systems

ES6 modules: Organize and share code across files.

```
// math.js
export function add(a, b) {
  return a + b;
}
// app.js
import { add } from './math.js';
console.log(add(2, 3)); // Output: 5
```

Web APIs

Geolocation API: Retrieve the user's location.

```
navigator.geolocation.getCurrentPosition(function(position) {
  console.log("Latitude:", position.coords.latitude);
  console.log("Longitude:", position.coords.longitude);
});
```

Testing and Debugging

Jest: Unit testing framework for JavaScript.

```
function add(a, b) {
  return a + b;
}
test('adds 1 + 2 to equal 3', () => {
  expect(add(1, 2)).toBe(3);
});
```

Performance Optimization

- Minification: Reduce file size by removing unnecessary characters.
- Code splitting: Split code into smaller chunks to improve loading times.

Security Best Practices

- Cross-Site Scripting (XSS) prevention: Sanitize user input to prevent script injection.
- Content Security Policy (CSP): Define policies to mitigate XSS attacks.

5.2 XML

In Django, XML (eXtensible Markup Language) can be used for various purposes, such as data interchange, configuration files, or representing structured data. While JSON (JavaScript Object Notation) is more commonly used for data interchange in web applications due to its lightweight and easy-to-read format, Django does provide facilities for handling XML data when needed.

XML Parsing and Generation:

Django provides modules for parsing and generating XML data. The xml.etree.ElementTree module in Python's standard library is often used for XML parsing and generation in Django projects.

Integration with Django Models:

XML data can be integrated with Django models for data import/export or synchronization with external systems. You can use Django's ORM (Object-Relational Mapping) to query database records and serialize them into XML format.

XML Rendering in Views:

Django views can render XML responses using Django's HttpResponse class. This allows you to serve XML data to clients making requests to your Django application.

Django Rest Framework (DRF):

If you're building a RESTful API with Django using Django Rest Framework (DRF), you have the flexibility to serialize data into XML format alongside JSON. DRF's serializers support XML rendering and parsing out of the box.

XML Configuration Files:

Django projects often use XML files for configuration purposes, such as defining URL patterns in the urls.py file or configuring settings for third-party apps.

Third-party Libraries:

While Django provides basic support for handling XML data, you can also leverage third-party libraries for more advanced XML processing tasks. Libraries like lxml offer powerful XML parsing and manipulation capabilities.

XML-RPC and SOAP:

Django can be used to implement XML-RPC (Remote Procedure Call) and SOAP (Simple Object Access Protocol) APIs. These protocols use XML for data exchange between clients and servers.

XML Schema Validation:

Django does not natively support XML Schema validation out of the box, but you can integrate third-party XML schema validation libraries into your Django project if needed.

Overall, while JSON is more prevalent in web development, Django provides sufficient support for handling XML data when required. Whether it's integrating XML with Django models, rendering XML responses in views, or processing XML data in APIs, Django offers the flexibility to work with XML effectively within its framework.

EXAMPLE

Let's create a simple example demonstrating how to integrate XML handling in a Django project. In this example, we'll create a Django app that reads data from a database and serializes it into XML format. We'll then create a view that serves this XML data as a response.

Assuming you have a Django project set up with a Django app named xml_example, here's how you can implement it:

Model Definition:

Define a Django model in models.py representing the data you want to serialize.

```
# xml_example/models.py

from django.db import models

class Book(models.Model):

  title = models.CharField(max_length=100)

  author = models.CharField(max_length=100)

  published_date = models.DateField()

  def __str__(self):

    return self.title
```

View Implementation:

Create a Django view in views.py that fetches data from the database and serializes it into XML format.

```
# xml_example/views.py

from django.http import HttpResponse

from django.core.serializers import serialize

from .models import Book

def books_xml(request):

    books = Book.objects.all()

    xml_data = serialize('xml', books)

    return HttpResponse(xml_data, content_type='application/xml')
```

URL Configuration:

Map the view to a URL in urls.py.

```
# xml_example/urls.py

from django.urls import path

from .views import books_xml

urlpatterns = [

    path('books/xml/', books_xml, name='books_xml'),

]
```

Database Population:

Populate the database with some sample data using Django's admin interface or Django shell.

Accessing the XML Data:

Start the Django development server (python manage.py runserver) and navigate to http://localhost:8000/books/xml/ to access the XML data representing the books stored in the database.

This example demonstrates how to serialize Django model data into XML format and serve it as a response using a Django view. You can extend this example by customizing the XML serialization, adding more fields to the model, or incorporating XML parsing for handling XML data received from clients.

5.3 HTTPREQUEST AND RESPONSE

HTTP (Hypertext Transfer Protocol) requests and responses are the foundation of communication between clients (such as web browsers) and servers. In Django, HttpRequest and HttpResponse are classes used to handle incoming requests from clients and send responses back to them, respectively.

Let's explore HttpRequest and HttpResponse in Django:

HttpRequest:

HttpRequest represents an incoming HTTP request from a client to the Django server.

It contains metadata about the request, such as headers, request method, URL, query parameters, POST data, etc.

Attributes:

- method: HTTP method used for the request (GET, POST, PUT, DELETE, etc.).
- path: Path portion of the requested URL.
- GET: Dictionary-like object containing query parameters from the URL.
- POST: Dictionary-like object containing POST data sent in the request body.
- META: Dictionary containing metadata about the request (headers, IP address, user agent, etc.).

Usage:

HttpRequest objects are passed as the first argument to Django view functions.

Views access request data through attributes like GET, POST, and META.

HttpResponse:

HttpResponse represents an HTTP response sent from the Django server to the client.

It contains the response content, status code, and headers.

Attributes/Methods:

- content: Content of the response (HTML, JSON, XML, etc.).
- status_code: HTTP status code of the response (200 for OK, 404 for Not Found, etc.).
- set_cookie(): Method to set cookies in the response.
- delete_cookie(): Method to delete cookies from the response.
- headers: Dictionary-like object representing response headers.

Usage:

Django views return HttpResponse objects to send responses back to clients.

HttpResponse objects can be customized with response content, status code, and headers.

Example:

let's create a simple Django project from scratch and implement a view that handles an HTTP request and sends back an HTTP response.

First, ensure you have Django installed. If not, you can install it via pip:

```
pip install django
```

Now, let's create a new Django project and app:

```
django-admin startproject myproject

cd myproject

python manage.py startapp myapp
```

Next, let's define a view that handles an HTTP request and sends back a simple HTTP response.

Open myapp/views.py and add the following code:

```
from django.http import HttpResponse

def hello_world(request):

   return HttpResponse("Hello, World!")
```

This view function, hello_world, takes an HttpRequest object as an argument and returns an HttpResponse object with the content "Hello, World!".

Now, we need to define a URL pattern that maps to this view.

Open myproject/urls.py and add the following code:

```
from django.urls import path

from myapp.views import hello_world

urlpatterns = [

   path('hello/', hello_world, name='hello_world'),

]
```

This maps the URL /hello/ to the hello_world view function we defined earlier.

Finally, let's run the Django development server and test our view.

Run the following command:

python manage.py runserver

Now, open your web browser and navigate to http://127.0.0.1:8000/hello/. You should see the text "Hello, World!" displayed in the browser, indicating that our view is successfully handling the HTTP request and sending back the HTTP response.

5.4 HTML

let's provide a quick glance at how HTML is used in conjunction with Django to create dynamic web pages:

Template System:

Django comes with a powerful template engine that allows you to build HTML templates with placeholders for dynamic data.

HTML Template Files:

HTML templates in Django are regular HTML files with additional template tags and filters provided by the Django template engine.

Template Tags:

Template tags are enclosed in {% %} and allow you to add logic and control flow to your templates. For example, {% if %}, {% for %}, {% include %}, etc.

Template Filters:

Template filters are enclosed in {{ }} and allow you to modify the output of template variables. For example, {{ variable|default:"No data" }}, {{ variable|date:"Y-m-d" }}, etc.

Context Data:

Context data is passed from views to templates and contains dynamic data that will be rendered in the HTML. Views render templates with context data using the render() function.

Static Files:

Static files such as CSS, JavaScript, images, etc., can be included in Django templates using the {% static %} template tag. These files are served by Django's static file server during development.

Forms:

Django provides form handling functionalities that generate HTML form elements in templates. Forms can be rendered manually or by using Django's form rendering helpers.

URLs and Links:

Django's template engine provides the {% url %} template tag to generate URLs for views. This allows you to create links dynamically in your HTML templates.

Inheritance:

Django templates support template inheritance, allowing you to define a base template with common layout and structure, and then extend it in child templates to override specific blocks.

HTML Escaping:

Django automatically escapes HTML special characters in template variables to prevent XSS (Cross-Site Scripting) attacks. Use the safe filter to mark a string as safe HTML if necessary.

Example:

Let's create a simple example to illustrate how HTML is used with Django templates:

Template File (myapp/templates/index.html):

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <title>{{ title }}</title>

</head>

<body>

  <h1>Welcome to {{ title }}</h1>


 <ul>

  {% for item in items %}

    <li>{{ item }}</li>
    {% endfor %}
    </ul>
    <p>Today's date is {{ current_date|date:"F d, Y" }}</p>
    <a href="{% url 'about_page' %}">About</a>
    <img src="{% static 'images/logo.png' %}" alt="Logo">
</body>
</html>
```

View Function:

```
from django.shortcuts import render
from datetime import datetime
def index(request):
    context = {
        'title': 'My Django App',
        'items': ['Item 1', 'Item 2', 'Item 3'],
        'current_date': datetime.now(),
    }
    return render(request, 'index.html', context)
```

URL Configuration (urls.py):

```
from django.urls import path
from myapp.views import index
urlpatterns = [
    path('', index, name='index'),
]
```

Static Files:

Place static files (e.g., logo.png) in the myapp/static/ directory.

Link to About Page (myapp/templates/about.html):

```
<!DOCTYPE html>
```

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>About Us</title>
</head>
<body>
  <h1>About Us</h1>
  <p>This is the about page of our Django app.</p>
  <a href="{% url 'index' %}">Back to Home</a>
</body>
</html>
```

In this example, we have a base template index.html that renders dynamic data such as the title, a list of items, the current date, and a link to the about page. We use template tags like {% for %}, {% url %}, and {% static %} to generate dynamic content and links. The view function retrieves data and renders the template with the context data.

5.5 CSS

Cascading Style Sheets (CSS):

- CSS is a stylesheet language used to style the appearance of HTML elements on web pages.
- It allows web developers to control the layout, colors, fonts, and other visual aspects of a website.

Key Concepts:

- Selectors: Used to target HTML elements for styling.
- Properties: Define the visual characteristics of the selected elements.
- Values: Specify the desired settings for the properties.

Example:

```
/* CSS code */
h1 {
  color: blue;
  font-size: 24px;
```

```
    text-align: center;
}
<!-- HTML code -->
<h1>This is a Heading</h1>
```

CSS Selectors and Box Model

Selectors:

- Element Selector: Targets HTML elements by their tag name.
- Class Selector: Targets elements with a specific class attribute.
- ID Selector: Targets a single element with a unique ID attribute.
- Descendant Selector: Targets elements that are descendants of a specified parent.
- Pseudo-classes: Targets elements based on their state or position.

Box Model:

- Content: The actual content of the element.
- Padding: Space between the content and the border.
- Border: The border surrounding the padding.
- Margin: Space outside the border, separating the element from other elements.

Example:

```
/* CSS code */
.box {
   width: 200px;
   height: 100px;
   padding: 20px;
   border: 2px solid black;
   margin: 10px;
}
<!-- HTML code -->
<div class="box">Box Content</div>
```

CSS Flexbox and Grid Layout

Flexbox:

- Provides a flexible way to layout elements within a container.
- Allows for dynamic alignment and distribution of space among items.

Grid Layout:

- Defines a two-dimensional grid system for layout design.
- Allows precise positioning and alignment of elements in rows and columns.

Example:

```
/* CSS code */

.container {

  display: flex;

  justify-content: center;

  align-items: center;

}

<!-- HTML code -->

<div class="container">

  <div>Item 1</div>

  <div>Item 2</div>

  <div>Item 3</div>

</div>
```

CSS Animations and Transitions

Animations:

- Allows for the creation of dynamic, interactive effects on web pages.
- Keyframes define the intermediate steps of the animation.

Transitions:

- Smoothly animates the transition of an element's property from one state to another.
- Transition properties include duration, timing function, delay, and property to transition.

Example:

```
/* CSS code */

@keyframes slide {

  0% { transform: translateX(0); }

  100% { transform: translateX(100px); }

}

.box {

  width: 100px;

  height: 100px;

  background-color: blue;

  animation: slide 2s infinite alternate;

}

<!-- HTML code -->

<div class="box"></div>
```

Responsive Design and Media Queries

Responsive Design:

- Ensures that web pages render well on various devices and screen sizes.
- Utilizes flexible grids, images, and CSS media queries.

Media Queries:

Allows for the adaptation of styles based on the characteristics of the device, such as screen width, height, and orientation.

Example:

```
/* CSS code */

@media screen and (max-width: 600px) {

  .container {

    flex-direction: column;

  }
}
<!-- HTML code -->
<div class="container">
  <div>Item 1</div>
  <div>Item 2</div>
  <div>Item 3</div>
</div>
```

Example: simple example of a phone directory web application using Django for the backend and HTML/CSS/JavaScript for the frontend.

Let's start with the Django backend:

First, make sure you have Django installed. You can install it via pip:

```
pip install django
```

Create a new Django project:

```
django-admin startproject phone_directory
```

Create a Django app within the project:

```
cd phone_directory
django-admin startapp directory
```

Define your model in directory/models.py:

```
from django.db import models
class Contact(models.Model):
  name = models.CharField(max_length=100)
  phone = models.CharField(max_length=20)
  def __str__(self):
    return self.name
```

Register your model in directory/admin.py:

```
from django.contrib import admin
from .models import Contact
```

```
admin.site.register(Contact)
```

Create a view in directory/views.py:

```
from django.shortcuts import render
from .models import Contact
def index(request):
    contacts = Contact.objects.all()
    return render(request, 'directory/index.html', {'contacts': contacts})
```

Create a template index.html in directory/templates/directory:

```
html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Phone Directory</title>
    <link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css">
</head>
<body>
    <div class="container">
        <h1>Phone Directory</h1>
        <ul>
            {% for contact in contacts %}
            <li>{{ contact.name }} - {{ contact.phone }}</li>
            {% endfor %}
        </ul>
    </div>
```

```
</body>
</html>
```

Now, let's move to the frontend part:

Create a CSS file styles.css in phone_directory/static/css:

```css
/* styles.css */
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    margin: 0;
    padding: 0;
}

.container {
    max-width: 800px;
    margin: 20px auto;
    padding: 20px;
    background-color: #fff;
    border-radius: 5px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h1 {
    color: #333;
}

ul {
    list-style-type: none;
    padding: 0;
}

li {
    margin-bottom: 10px;
}
```

Link the CSS file in your HTML template:

html

```
<link rel="stylesheet" href="{% static 'css/styles.css' %}">
```

Now, you can run your Django server:

```
Python manage.py runserver
```

And you should be able to see your phone directory application running at http://127.0.0.1:8000/. You can then add more features like adding new contacts, editing existing contacts, etc., based on your requirements.

5.6 JSON

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is based on a subset of the JavaScript programming language, but it's language-independent, meaning it can be used with most programming languages.

JSON is commonly used for transmitting data between a server and a web application as an alternative to XML. It's widely used in web development for APIs (Application Programming Interfaces) because it's simple, easy to understand, and lightweight.

JSON data is represented as key-value pairs, similar to Python dictionaries or JavaScript objects. The keys are strings, and the values can be strings, numbers, arrays, objects, booleans, or null.

Here's an example of JSON data representing information about a person:

```
{
    "name": "Mallikarjuna",
    "age": 30,
  "is_student": false,
  "address": {
    "street": "123 Main St",
    "city": "Mysuru",
    "state": "CA"
  },
  "hobbies": ["reading", "hiking", "coding"]
}
```

In this example:

- "name", "age", and "is_student" are key-value pairs with string keys and string or boolean values.
- "address" is a key-value pair where the value is another object containing keys "street", "city", and "state".
- "hobbies" is a key-value pair where the value is an array containing strings.

JSON data can be parsed and converted into native data types in most programming languages, making it easy to work with in a variety of contexts.

simple example:

Let's say we want to create a simple JSON API for managing contacts.

Model Setup:

```
# models.py
from django.db import models
class Contact(models.Model):
  name = models.CharField(max_length=100)
  phone = models.CharField(max_length=20)

  def to_json(self):
    return {'name': self.name, 'phone': self.phone}
```

Serializer Creation (Optional):

```
# serializers.py

from rest_framework import serializers

from .models import Contact

class ContactSerializer(serializers.ModelSerializer):

    class Meta:

        model = Contact

        fields = ['name', 'phone']
```

Views:

```
# views.py

from django.http import JsonResponse

from .models import Contact

def get_contacts(request):

    contacts = Contact.objects.all()

    data = [contact.to_json() for contact in contacts]

    return JsonResponse(data, safe=False)
```

URL Configuration:

```
# urls.py

from django.urls import path

from .views import get_contacts

urlpatterns = [

    path('contacts/', get_contacts, name='get_contacts'),

]
```

With this setup, when you visit http://127.0.0.1:8000/contacts/, you'll receive a JSON response containing all the contacts in the database.

Make sure to install Django Rest Framework (pip install djangorestframework) if you choose to use serializers from it. Also, don't forget to include the app in your Django project's INSTALLED_APPS setting and set up your database.

5.7 USING JQUERY UI AUTOCOMPLETE IN DJANGO

Query is a fast, small, and feature-rich JavaScript library. It simplifies various tasks like HTML document traversal and manipulation, event handling, animation, and Ajax interactions for web development. jQuery was created by John Resig in 2006 and has since become one of the most popular JavaScript libraries used by developers worldwide.

Some key features and benefits of jQuery:

- DOM Manipulation: jQuery provides an easy-to-use API for selecting and manipulating HTML elements in the Document Object Model (DOM). With jQuery, you can easily traverse the DOM tree, modify element attributes and content, and add or remove elements from the page.
- Event Handling: jQuery simplifies event handling by providing methods for attaching event listeners to HTML elements. You can handle user interactions such as clicks, mouse movements, keyboard inputs, and more with ease.
- AJAX Support: jQuery simplifies asynchronous HTTP requests (Ajax) by providing a set of methods for making requests to the server and handling server responses. This allows you to load data from the server without refreshing the entire web page, leading to a more responsive user experience.
- Animation Effects: jQuery includes built-in animation effects and methods for creating custom animations. You can animate CSS properties, show/hide elements with various effects, and create complex animations with ease.
- Cross-browser Compatibility: jQuery abstracts away many browser inconsistencies and provides a unified interface that works across different web browsers. This helps developers write code that behaves consistently across various browser environments.
- Extensibility: jQuery is highly extensible, allowing developers to create plugins to extend its functionality further. There is a vast ecosystem of jQuery plugins available for various purposes, ranging from UI components to complex data visualization tools.

Overall, jQuery simplifies JavaScript development and makes it easier to create dynamic, interactive, and responsive web applications. However, with the advancement of modern web technologies and improvements in browser APIs, some developers prefer using native JavaScript or modern frameworks/libraries like React, Vue.js, or Angular for new projects

Example:

let's create a simple Django project from scratch and integrate jQuery step by step.

Create a Django Project:

```
django-admin startproject myproject
```

```
cd myproject
```

Create a Django App:

```
python manage.py startapp myapp
```

Define a View:

In myapp/views.py, define a simple view that renders a template.

```
from django.shortcuts import render
def index(request):
    return render(request, 'myapp/index.html')
```

Create a Template:

Create a directory named templates in the myapp directory. Inside templates, create a file named index.html.

```
html
<!-- myapp/templates/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Django with jQuery</title>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
  <script>
    $(document).ready(function() {
      // Example: Alert message on button click
      $('#myButton').click(function() {
        alert('Button clicked!');
      });
    });
  </script>
```

```
</head>

<body>

   <button id="myButton">Click me</button>

</body>

</html>
```

Define URLs:

In myproject/urls.py, define a URL pattern to map to the view.

```
from django.urls import path

from myapp.views import index

urlpatterns = [

   path('', index, name='index'),

]
```

Run the Server:

```
python manage.py runserver
```

Visit http://127.0.0.1:8000/ in your browser, and you should see a button. When you click the button, it should display an alert message, demonstrating the use of jQuery.

This setup demonstrates the integration of jQuery with Django from scratch. You can further expand the project by adding more views, templates, and jQuery functionality as needed for your application.