INSTITUTO TECNOLÓGICO SUPERIOR DEL SUR DE GUANAJUATO

AJAX y Django

Programación Web II

7°A
Ing. Sistemas Computacionales

Elaborada por:

América Citlalli López Lemus, S22120161

Profesor:

Gustavo Ivan Vega

Uriangato, Gto. a 16 de Octubre del 2025

Síntesis Digital de las Lecturas

AJAX (Asynchronous JavaScript and XML) permite solicitudes asíncronas al servidor para actualizar partes específicas de la página, mejorando la experiencia del usuario en escenarios como validaciones, envíos de formularios o carga dinámica de datos. En Django, se integra fácilmente con vistas que responden en JSON, verificando solicitudes AJAX mediante el header X-Requested-With: XMLHttpRequest (ya que request.is_ajax() está deprecado desde Django 3.1). Se enfatiza el manejo de CSRF en métodos POST/PUT/DELETE y el uso de JsonResponse para respuestas serializadas.

Formas Principales de Implementar AJAX con Django

Basado en los artículos, las implementaciones se dividen en cliente y servidor, con énfasis en simplicidad y escalabilidad.

- Fetch API (JavaScript Nativo): Usa la API nativa del navegador para solicitudes GET/POST/PUT/DELETE. Incluye headers como X-Requested-With y X-CSRFToken. Ejemplo: fetch(url, {method: 'POST', headers: {...}, body: JSON.stringify(data)}).then(response => response.json()). No requiere librerías externas; interfaz limpia y moderna.
- jQuery AJAX: Librería para simplificar solicitudes asíncronas. Ejemplo: $.ajax({url: url, type: 'POST', data: data, success: function(response){...}}. Fácil manejo de eventos y serialización; ideal para principiantes.
- Vistas Basadas en Funciones (FBV): Verifica AJAX en la vista, procesa datos (e.g., json.loads(request.body) para POST), valida con forms y responde con JsonResponse. Ejemplo: Para GET, serializar QuerySet con list(queryset.values()). Simple y directo para operaciones CRUD.
- Vistas Basadas en Clases (CBV): Hereda de View y define métodos get()/post(). Similar a FBV pero más estructurado. Mejor organización para lógica compleja; reutilizable.
- Manejo de Formularios y Validación: Usa ModelForm o Form para validación backend. Envía datos con FormData o serialize(). Responde errores como {"errors": form.errors}. Seguridad y usabilidad: previene envíos inválidos en tiempo real.
- Envío Asíncrono de Emails/Archivos: Usa hilos (threading.Thread) para tareas no bloqueantes. Adjunta archivos con EmailMultiAlternatives. Mejora rendimiento: usuario recibe feedback inmediato.

Para la configuración inicial de AJAX en Django, es esencial incluir librerías como jQuery o la Fetch API directamente en los templates mediante la etiqueta {% static %} o un CDN externo. Un aspecto crítico es el manejo de tokens CSRF, que se incorpora siempre en los headers de las solicitudes mediante funciones como getCookie('csrftoken'), garantizando la seguridad en operaciones sensibles como POST o PUT.

En cuanto a cuándo utilizar AJAX, se recomienda en escenarios que mejoran la interactividad del usuario, como autocompletado de campos, filtros dinámicos, validaciones en tiempo real o encuestas progresivas. Sin embargo, se aconseja evitarlo en aplicaciones de página única (SPAs)

complejas, donde opciones como WebSockets o frameworks frontend como React ofrecen mayor escalabilidad y rendimiento.

Las mejores prácticas incluyen verificar el método de solicitud y el origen AJAX en las vistas del servidor, utilizando códigos de estado HTTP como 200 para éxitos y 400 para errores, lo que facilita el manejo de respuestas en el cliente. Para serializar objetos complejos, como QuerySets, se sugiere el uso de serializers.serialize, y los artículos destacan la disponibilidad de repositorios en GitHub con ejemplos prácticos que permiten replicar y adaptar el código fácilmente.

Finalmente, entre las limitaciones destacadas, se menciona la necesidad de configurar CORS (Cross-Origin Resource Sharing) para solicitudes entre dominios, y se prioriza la Fetch API sobre jQuery en proyectos modernos por su integración nativa y menor peso, promoviendo un desarrollo más eficiente y actualizado.

Reporte de la practica

Esta práctica surge de la síntesis de lecturas sobre AJAX en Django, donde se exploraron métodos para integrar solicitudes asíncronas en aplicaciones web. AJAX permite actualizar partes específicas de una página sin recargas completas, ideal para formularios interactivos. El ejemplo elegido simula un formulario de selección geográfica: al elegir un estado de la República Mexicana, se cargan sus municipios correspondientes vía AJAX, evitando submits tradicionales.

El objetivo principal de esta práctica es demostrar la integración efectiva de AJAX en un entorno Django para lograr interacciones dinámicas. Específicamente, se busca implementar una vista en Django que responda con datos en formato JSON a solicitudes AJAX, utilizar la Fetch API en el frontend para manejar eventos de cambio en elementos de formulario, y probar la funcionalidad completa en un entorno de desarrollo local mientras se documenta el flujo de ejecución detallado. El alcance se limita a un proyecto minimalista sin una base de datos real, utilizando datos hardcodeados y enfocándose en solicitudes GET para mantener la simplicidad.

Para el desarrollo de esta práctica, se empleó Django en su versión 4.x como framework backend, instalado mediante el comando pip install django. En el frontend, se utilizó HTML5 junto con JavaScript nativo a través de la Fetch API, evitando librerías externas para mantener la ligereza. El entorno de ejecución fue un servidor de desarrollo local iniciado con python manage.py runserver. Los datos se basaron en un diccionario hardcodeado en el archivo de vistas, incluyendo ejemplos de estados y municipios como Aguascalientes, Baja California y Sonora. La estructura del proyecto consistió en una app principal llamada lugares dentro del proyecto Municipios, con templates ubicados en templates/lugares/ y URLs configuradas de manera jerárquica.

El desarrollo comenzó con la configuración inicial del proyecto Django, creando el proyecto base mediante django-admin startproject Municipios y luego generando la app con python manage.py startapp lugares. Posteriormente, se actualizó el archivo settings.py para incluir 'lugares' en la lista INSTALLED_APPS, y se definieron las URLs principales en Municipios/urls.py para integrar las rutas de la app de forma modular.

En el backend, el desarrollo de las vistas en lugares/views.py incluyó la creación de la vista index, que renderiza el template inicial pasando la lista de estados como contexto, y la vista get_municipios, que verifica el header X-Requested-With para confirmar solicitudes AJAX, extrae el parámetro estado de la consulta GET y responde con un JsonResponse conteniendo la lista de municipios en caso de éxito, o un mensaje de error con código de estado 400 si no se encuentra el estado. Se aplicaron decoradores como @require_http_methods(["GET"]) para restringir las operaciones a métodos GET seguros. Los datos se almacenaron en un diccionario simple llamado ESTADOS_MUNICIPIOS, que es fácilmente expandible.

Para el frontend, en el archivo templates/lugares/index.html, se diseñaron dos elementos <select>: el primero para estados, poblado dinámicamente mediante etiquetas de template como {% for %}, y el segundo para municipios, inicialmente deshabilitado con un mensaje placeholder. El script JavaScript inline incorpora un event listener en el evento change del select de estados, que captura el valor seleccionado y utiliza la Fetch API para enviar una petición GET a la URL /ajax/municipios/?estado=..., incluyendo el header necesario para identificarla como AJAX. El

manejo de la respuesta involucra la conversión a JSON, la actualización del DOM para agregar opciones al select de municipios, la visualización de un indicador "Cargando..." durante la espera, y el tratamiento de errores mediante alert() y logs en consola.

Finalmente, las rutas se definieron en lugares/urls.py para mapear index y get_municipios, y las pruebas se realizaron ejecutando el servidor, accediendo a http://127.0.0.1:8000/ y monitoreando las interacciones en la pestaña Network de la consola del navegador.

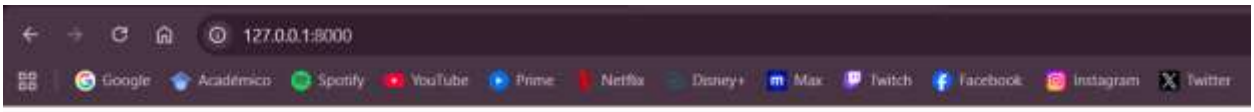
La funcionalidad principal se observó con éxito en la carga dinámica: al seleccionar un estado como "Aguascalientes", el select de municipios se actualiza de inmediato con opciones como "Aguascalientes", "Asientos" y "Calvillo", habilitándose para futuras selecciones. Lo mismo ocurre con otros estados como Baja California o Sonora. En cuanto al manejo de errores, si se intenta acceder a un estado inválido modificando manualmente la URL, el sistema responde con un alert mostrando "Estado no encontrado" y un código de estado 400, manteniendo la integridad de la interfaz. El rendimiento fue óptimo, con respuestas locales inferiores a 100 milisegundos y ausencia total de recargas de página. Durante la depuración, la consola del navegador registró logs detallados de errores en caso de fallos, mientras que la terminal de Django capturó las solicitudes GET sin generar advertencias.

Las evidencias de la ejecución incluyen una descripción visual clara: inicialmente, el select de municipios aparece deshabilitado con un mensaje indicativo, y tras la selección de un estado, muestra brevemente "Cargando..." antes de poblarse con las opciones correctas. En los logs de la consola del navegador, por ejemplo, se visualiza una petición GET a /ajax/municipios/?estado=Aguascalientes con respuesta 200 OK y contenido JSON.

Esta práctica concluye con éxito al demostrar cómo AJAX en Django puede transformar un formulario estático en uno altamente interactivo, alineándose directamente con los conceptos de las lecturas sintetizadas, como el uso de JsonResponse para respuestas dinámicas y la Fetch API para operaciones CRUD asíncronas. Se valida la simplicidad de las vistas basadas en funciones para prototipos, logrando una experiencia de usuario fluida que resalta la eficiencia de las solicitudes asíncronas en aplicaciones web modernas.

Selecciona un Estado

Estado: -- Selecciona -- Municipio: -- Primero selecciona estado --



Selecciona un Estado

Estado: Guanajuato Municipio: -- Selecciona --

-- Selecciona --
Abasolo
Acámbaro
Apaseo el Alto
Apaseo el Grande
Atarjea
Celaya
Comonfort
Coroneo
Cortazar
Cuerámaro
Moroleon
Uriangato
Yuriria