ChatGPT

Guía técnica para el Portal Inmobiliario SkyTerra

1. Buenas prácticas en el backend (Django)

- **Estructura de modelos:** Definir los modelos con nombres en singular y campos en `snake_case`. Por ejemplo, use nombres sencillos y concisos para los modelos (p.ej. `Property`, `UserProfile`) [1] y campos con minúsculas y guiones bajos para evitar problemas de mayúsculas en distintos motores de BD [2]. Utilice características de los modelos como `choices` para campos con valores fijos (mejorando la consistencia de datos) y active `db_index=True` o `unique=True` en campos muy consultados o únicos [3], lo que acelera las consultas. En SkyTerra se añadió `order_by('-created_at')` a los querysets para evitar advertencias de ordenamiento indefinido [4], una práctica recomendada al paginar resultados. Para campos geoespaciales (como polígonos), emplee `JSONField` para almacenar coordenadas en formato GeoJSON (evitando importaciones innecesarias como `ArrayField` si no se usa PostgreSQL).
- **Vistas y viewsets:** Use vistas basadas en clases o ViewSets de DRF para aprovechar autenticación, permisos y paginación integrados. Separe responsabilidades: las vistas sólo deben orquestar solicitudes, no contener lógica compleja. Por ejemplo, en SkyTerra el `AISearchView` Django procesa `current_query` y `conversation_history`, llama al asistente IA, y retorna un JSON con `assistant_message`, filtros sugeridos y `recommendations` [5]. Siempre filtre o ordene los querysets desde el modelo (`queryset = Property.objects.all().order_by(...)`) en la declaración del ViewSet para evitar errores de paginación indefinida [4].
- **Serializadores (DRF):** Para cada modelo principal tenga un `ModelSerializer`. Separe serializers de lectura y escritura si hay datos anidados pesados: por ejemplo, use un `PropertyListSerializer` ligero para listados y un `PropertyDetailSerializer` completo para detalle [6]. Marque campos de sólo lectura (como `id`, `created_at`) con `read_only=True` o en `Meta.read_only_fields` para optimizar las operaciones de escritura [7]. Implemente validaciones personalizadas usando `validate_<campo>` o `validate(self, data)` y lance `serializers.ValidationError` con mensajes claros para cada campo [8]. Esto convierte automáticamente la excepción en una respuesta HTTP 400 con detalles por campo [9]. Al salvar datos anidados o relaciones, considere los campos `source` y `depth` de DRF o serializers anidados explícitos para controlar cómo se incluyen los datos relacionados [10] [11].
- **Gestión de errores:** No exponga información sensible. Asegúrese de que nunca se incluyan claves secretas o stack traces en la respuesta JSON al cliente. Por ejemplo, en SkyTerra se modificó el manejo de errores en `AISearchView` para evitar que la API key de Gemini apareciera en un mensaje de error [12]. Use el manejo de excepciones de DRF: por defecto las excepciones `Http404` o `PermissionDenied` devuelven respuestas con estatus 404/403 y un campo `"detail"`, y `ValidationError` regresa 400 con detalles de campo [9]. Para respuestas personalizadas, implemente un `EXCEPTION_HANDLER` en DRF según la documentación [13]. Nunca retorne el `traceback` de Python ni errores internos en producción, ya que revelan la estructura interna de la app [14]. Emita mensajes genéricos o user-friendly (p.ej. “Error inesperado, intente más tarde”) para el usuario, y registre el detalle real del error en logs internos.

2. Buenas prácticas en el frontend (React)

- **Estructura de componentes y directorios:** Organice la carpeta `src` con subdirectorios claros (p.ej. `components/`, `pages/`, `services/`, `styles/`). Mantenga los componentes pequeños y enfocados en una única responsabilidad [15]. Por ejemplo, en SkyTerra cada barra o panel (como `AISearchBar`, `AIChatAssistant`) es un componente separado. Use nombres descriptivos en mayúscula inicial (p.ej. `PropertyCard`, `FilterPanel`) para facilitar la lectura [15]. Evite duplicación de código reutilizando componentes genéricos para formularios, botones o tarjetas (DRY). Si un componente crece demasiado, divídalo: p.ej. un componente padre podría manejar el estado de filtrado y delegar la presentación de la lista de propiedades a componentes hijos sin estado [16].
- **Estado y gestión global:** Decida claramente qué datos van en estado local de un componente y qué datos son globales. El estado local (`useState`) sirve para valores de UI específicos. Para filtros o datos compartidos (p.ej. filtros globales `globalFilters` en SkyTerra), use un estado global con Context o librerías como Redux/MobX según complejidad. No pase props innecesariamente por varios niveles: si muchos componentes necesitan el mismo estado, ubíquelo en un contexto superior [17]. Por ejemplo, SkyTerra usa callbacks (`onFiltersSuggested`) para elevar cambios desde el asistente IA al estado global. Asegúrese de reinicializar o limpiar el estado cuando sea necesario (por ejemplo, limpiar chat al cerrar el asistente). Use `useEffect` con dependencias precisas para evitar bucles infinitos.
- **Integración con APIs:** Interactúe con el backend mediante `fetch` o librerías como `axios`. Encapsule las llamadas en funciones reutilizables (p.ej. `api.getProperties(params)`) y maneje asíncrono con `async/await`. Siempre maneje errores de red con `try/catch`; muestre mensajes de error amigables o reintente la operación. Por ejemplo, al llamar a la API de filtro, muestre un loading spinner hasta recibir datos o indique «Cargando...»; si falla, muestre «No se pudieron cargar los resultados». Valide respuestas: verifique que los datos tienen la forma esperada antes de usarlos (p.ej. que `response.data` no sea `null`). En SkyTerra, se agregó manejo de nulos para evitar el error `Cannot read properties of null` al leer `priceRange` [18]. Use variables de entorno (`process.env.REACT_APP_*` o `VITE_*`) para configurar URL de API y llaves (no incruste valores en el código).
- **UI/UX y accesibilidad:** Diseñe una interfaz limpia y coherente. Utilice frameworks de diseño como Material UI, Bootstrap o librerías de componentes para ahorrar tiempo, pero asegúrese de personalizar estilos para alinearse con la imagen de SkyTerra. Mantenga la interfaz responsive (media queries, flexbox) para móviles y desktop. Agregue retroalimentación visual: botones deshabilitados mientras carga, mensajes de éxito/error tras acciones del usuario. Estructure el código JSX separando lógica de datos de la presentación: por ejemplo, un componente “contenedor” puede encargarse de cargar propiedades (`useEffect`) y pasar datos a un componente “presentacional” que solo las muestre [16]. Use PropTypes o TypeScript para validar props entrantes [19], lo que ayuda a detectar errores de uso de componentes en tiempo de desarrollo. Aplique buenas prácticas de UX: navegación clara, formularios etiquetados correctamente, enfoque de teclado, contrates accesibles (ARIA) y colores legibles.

3. Asistente conversacional IA (Gemini) – Integración y escalabilidad

- **Arquitectura de integración:** El asistente "Sky" se comunica vía el backend Django: el frontend envía `current_query` y `conversation_history` al endpoint `AISearchView` y este llama a la API de Gemini. Use un **prompt de sistema** claro que defina el rol del asistente ("como *Sky*, un asistente conversacional inmobiliario que..."). Incluya la conversación previa para contexto: según la API de Gemini, es crucial enviar los mensajes anteriores en el campo `parts` del JSON, no solo en "history", para que retenga el contexto [20]. Por ejemplo:

```
{
  "contents": [
    {"parts": [{"text": "Usuario: Hola"}, {"text": "Usuario: ¿Cuál es mi mensaje anterior?"}]}
  ]
}
```

  Esto permite que Gemini recuerde y responda apropiadamente [20]. Reciba y procese la respuesta JSON con campos como `assistant_message`, filtros sugeridos (`suggestedFilters`) y `recommendations`, y actualice el historial de conversación. Mantenga una estructura de datos consistente del historial (e.g. lista de pares usuario/ayudante).
- **Adaptación de prompt y casos de uso:** Para distintos escenarios, ajuste el prompt del sistema o de inicio del chat. Por ejemplo, para consultas generales ("¿Qué propiedades hay en X barrio?") vs. refinamiento de filtros, cambie el sistema para priorizar la búsqueda de propiedades sobre dar consejos generales. Use *few-shot prompts* si desea ejemplos de interacción previa. Si integra funciones adicionales (e.g. llamadas a mapas o calculadoras), especifique en el prompt qué funciones puede usar. Monitoree la longitud del prompt y del historial: considere resumir o truncar conversaciones muy largas para no exceder límites de tokens. Para escalabilidad, procese cada petición de Gemini asincrónicamente y registre el uso de tokens para controlar costos.
- **Escalabilidad y rendimiento:** A medida que crezca el número de usuarios, optimice llamando a la API de manera asíncrona (por ejemplo, con `asyncio` en Django o workers) y cacheando respuestas frecuentes si aplican (p.ej. ubicaciones comunes). Controle la carga usando herramientas de colas (Celery) si las consultas tardan mucho. Limite la frecuencia de solicitudes de IA para usuarios no autenticados o en planes gratuitos.
- **Depuración de problemas comunes:** Verifique la configuración de la clave de API y permisos (como ocurrió en SkyTerra) [21]. Por ejemplo, asegúrese de que la variable de entorno para la API KEY esté correctamente nombrada (en React/Vite debe empezar con `REACT_APP_` o `VITE_`) y reiniciar el servidor después de cambiarla [22]. Si recibe errores 400 "API key not valid", revise los permisos en Google Cloud Console y la validez de la clave [21] [22]. Use logs detallados en el backend para registrar la entrada/salida de Gemini y detectar fallos de parsing o de formato JSON. En el frontend, use DevTools para ver las peticiones de red y respuestas. Maneje excepciones del asistente (tiempos de espera, respuestas no JSON) con mensajes genéricos e intente recuperarse ("Lo siento, no entendí. ¿Puedes reformular?").
- **Mejora iterativa:** Ajuste el prompt según el feedback real: por ejemplo, si el asistente da respuestas irrelevantes, refina las instrucciones del sistema. Agregue validaciones en el backend (p.ej. que `assistant_message` siempre exista antes de enviarlo al usuario). Cuando ofrezca "filtros

sugeridos", verifíquelos con lógica de negocio (p.ej. que un rango de precios sea válido). Pruebe distintos modelos ("Gemini-1.5-flash" vs. versiones mayores) y compare resultados. Documente ejemplos de prompts exitosos para distintos tipos de consultas.

4. Depuración y debugging (Backend & Frontend)

- **Backend (Django/DRF):** Utilice el **shell de Django** (`python manage.py shell`) para inspeccionar objetos de la base de datos y probar queries manualmente [23]. Emplee herramientas como *Django Debug Toolbar* en desarrollo para medir tiempos de consulta y detectar cuellos de botella. Registre errores con `logging` configurado (o servicios externos como Sentry) para capturar excepciones en producción. Por ejemplo, integrar Sentry permite monitorear en tiempo real fallos en Django [24]. Para errores de serialización, compare los datos entrantes con los esperados: un error de serialización suele deberse a campos faltantes o tipo incorrecto, así que agregue validaciones o ajuste `fields` en el serializer. En migraciones, cuide las importaciones: si aparece un error de `makemigrations` por un campo no soportado (como importación de `ArrayField` sin Postgres), elimínelo. En caso de conflictos de migraciones, puede usar `python manage.py makemigrations --merge` o borrar migraciones intermedias y volver a generarlas con datos controlados. Use pruebas unitarias para validar migraciones críticas (crear y migrar datos de prueba).
- **Frontend (React):** Para errores de ejecución en JS, abra la consola del navegador. Identifique `TypeError: cannot read property 'x' of null/undefined` y trace de dónde viene el valor `null` (en SkyTerra se solucionó un fallo en `AISearchBar` añadiendo chequeos de nulos para `priceRange` [18]). Configure *React Developer Tools* para inspeccionar el estado (`state`) y las props de cada componente. Use *error boundaries* de React (`componentDidCatch` o `<ErrorBoundary>`) para capturar excepciones en la UI sin romper toda la app. Al integrar APIs, verifique las respuestas HTTP: capture códigos 4xx/5xx y maneje cada caso (p.ej. mostrar "Servicio no disponible" en 503). Use linters (ESLint) y TypeScript/PropTypes para detectar errores de tipado tempranamente en tiempo de compilación. Para problemas gráficos, compruebe diferencias en DevTools (p.ej. elementos ocultos por CSS). En general, corra la aplicación localmente con `npm start` en modo de desarrollo, donde React muestra advertencias detalladas.
- **Estrategias generales de debugging:** Documente y reproduzca los errores encontrados (pasos para replicar el bug). Agregue logs temporales (`console.log` o `logger.debug`) para entender el flujo de datos y descartarlos antes de producción. Use herramientas de monitoreo de cliente (p.ej. Sentry para JS) que capturan excepciones de usuarios reales. Revise el código fuente cercano al error: a veces el problema está en un dato previo (p.ej. un filtro mal aplicado antes de renderizar). Finalmente, la escritura de tests que reproduzcan fallos conocidos asegura que no regresen en el futuro.

5. Estrategias de escalabilidad y calidad

- **Organización del código:** Divida la aplicación en módulos bien definidos. En Django, separe apps (p.ej. `properties`, `users`, `chat`) y dentro de cada una use submódulos (`serializers.py`, `views.py`, `urls.py`) para claridad. En React, organice componentes por funcionalidad (p.ej. `Chat/`, `Map/`, `Auth/`). Mantenga un estilo de código consistente (use linters/formatters como ESLint, Prettier, Black) para facilitar la colaboración. Documente las APIs REST usando herramientas como Swagger o DRF Spectacular, para que otros desarrolladores comprendan los endpoints.

- **Testing (unitario e integración):** Implemente **pruebas unitarias** para cada pieza crítica. En Django use el framework de tests integrado o pytest-django para probar modelos, serializers y vistas por separado (por ejemplo, crear instancias de `Property` y verificar que el serializer produce la JSON correcta). Para testing **de integración**, pruebe flujos completos: p.ej. crear una propiedad vía API y luego consultarla. En React, use **Jest** con React Testing Library para probar componentes en aislamiento (asegúrese de probar interacciones clave: clics en botones, cambios de formulario, recepción de props). Además, configure pruebas **end-to-end** con Cypress o Selenium: simule al usuario usando el portal completo (login, búsqueda de propiedades, uso del chat) y valide el resultado. Integrar estas pruebas en el pipeline de CI garantiza que nuevos cambios no rompan funcionalidades.
- **Logging y monitoreo:** Configure un sistema de logging centralizado. En Django, use la librería `logging` con distintos niveles (INFO, ERROR) y rote archivos de log o envíe eventos a un servicio externo (Logstash, Cloud Logging). Monitoree métricas de rendimiento: uso de CPU/RAM, tiempos de respuesta, tasa de errores con herramientas como Prometheus/Grafana o soluciones de nube (Cloud Monitoring). Al menos en producción, habilite alertas (por ejemplo, notificar si hay más de X errores por minuto) para actuar rápidamente. Para seguimiento de excepciones, integre **Sentry** tanto en backend como en frontend, obteniendo rastros de errores con contexto usuario-navegador.
- **Despliegue continuo (CI/CD):** Use un repositorio Git en la nube (GitHub/GitLab) con ramas claras (por ejemplo, `main` para producción, `develop` para integración, `feature/*` para cada tarea). Configure pipelines automáticos: al hacer push a ciertas ramas o crear un *pull request*, ejecute tests unitarios, linteo, y despliegue a entornos de prueba. Herramientas como GitHub Actions, GitLab CI o Travis pueden orquestar esto. Al aprobar un PR, despliegue automático a **staging**; tras validarlo, promover a **producción**. Use migraciones automáticas (ej. `manage.py migrate`) en el despliegue backend. Documente pasos manuales si los hay (p.ej. reiniciar workers) y, de ser posible, automatícelos (scripts de despliegue).
- **Separación de entornos:** Mantenga entornos aislados: desarrollo local, *staging* (pruebas) y producción. Use archivos de configuración o variables de entorno distintos en cada caso (.env local, secretos en servidor). No comparta servicios entre entornos; por ejemplo, una base de datos separada por entorno. Active `DEBUG=False` en producción para no exponer errores. Use versiones específicas de dependencias (requirements.txt o Pipfile) para reproducibilidad. Considere contenedores Docker para igualar entornos entre máquinas.

6. Mapas interactivos para gestión de propiedades

Para dibujar y mostrar polígonos de propiedades, se puede integrar un mapa interactivo:
- **Leaflet vs Google Maps:** Leaflet es una librería de código abierto muy popular y liviana para mapas interactivos. Soporta directamente datos GeoJSON (por ejemplo, `L.geoJSON(datosGeoJSON).addTo(map)` para visualizar polígonos)[25]. No requiere clave (solo necesita un proveedor de tiles como OpenStreetMap). Google Maps API ofrece un mapa con más datos (satélites, Street View, puntos de interés), pero requiere una API Key y puede tener costos por uso excesivo. En Leaflet no hay gastos por llamada de API, pero en Google Maps puede haber cuotas. (Ver tabla comparativa más abajo.)
- **Dibujar polígonos:** En React, use librerías como **react-leaflet-draw** (basada en Leaflet Draw) o Google Maps Drawing Manager. Por ejemplo, react-leaflet-draw permite habilitar un `EditControl` para que el administrador dibuje un polígono sobre el mapa[26]. Al guardar, capture el evento `onCreated` o `onEdited` y obtenga las coordenadas del polígono (por ejemplo, con `layer.getLatLngs()` o convirtiendo a GeoJSON). Guarde ese array de coordenadas (o el GeoJSON) en el campo

`boundary_polygon` JSON del backend [27].
- **Mostrar polígonos guardados:** Al cargar una propiedad existente, obtenga el GeoJSON del servidor y páselo al mapa: por ejemplo, `L.geoJSON(property.boundary_polygon).addTo(map)`. Ajuste el estilo (color, opacidad) para distinguirlo. Si necesita edición, permita mover/vizualizar vértices. Asegúrese de convertir entre coordenadas Leaflet (lat, lng) y el formato esperado en la base de datos (GeoJSON estándar).
- **Comparativa de herramientas:**

Librería / API	Libre/ Gratuita	Necesita API Key	Ventajas principales	Consideraciones
Leaflet	Sí (open source) [28]	No (*solo opcional si usa capas pagas*)	Ligero, extensible (muchos plugins), soporta GeoJSON nativo [28]. Sin costo adicional.	Depende de proveedor de tiles (p.ej. OSM). No ofrece datos de negocios como Google Places.
Google Maps	No (limite gratuito, luego pago)	Sí	Mapas detallados, geocoding integrado, StreetView. Amplia documentación.	Requiere registro en Google Cloud, puede generar costos. Limitaciones de cuota.
Mapbox (bonus)	No (Plan free limitado)	Sí	Mapas vectoriales personalizables, velocidad.	Requiere clave, plan gratuito con límites.

Ejemplo de uso: SkyTerra planea usar Leaflet para el panel de administración: el admin dibujará el polígono con `draw` y el frontend lo enviará al backend en formato JSON (tal como se definió `boundary_polygon = JSONField` en `models.py` [27]).

7. Seguridad y gestión de claves/API

- **Claves de API:** Nunca incluya claves secretas en el código cliente ni en los mensajes de error. Use variables de entorno para todas las credenciales (por ejemplo, `.env` para Django y `REACT_APP_*` para React [22]). En el frontend, **no almacene la API key** de servicios en el bundle; en su lugar, haga que el backend maneje las llamadas sensibles o exponga sólo lo mínimo. Restrinja las claves usando permisos: en Google Cloud Console limite la clave de Gemini sólo al proyecto necesario. Si un atacante ve un stack trace o error largo que contiene la clave (por ejemplo, un `NullPointerException` con ruta de clase [14]), podría explotar la información. Por esto, configure Django/DRF para que en producción devuelva errores genéricos.
- **Configuración de entorno:** Proteja los `.env` añadiéndolos a `.gitignore` y no compartiéndolos. Use gestores de secretos en la nube (AWS Secrets Manager, Vault, etc.) si es posible. Para entornos de cliente (React), use variables que empiecen con `VITE_` o `REACT_APP_`, y no exponga secretos de prod en Git. Siempre reinicie el servidor tras cambiar `.env` para que tome las variables (como sucedió con la key de Gemini) [22].
- **Seguridad general:** Asegure las APIs Django con HTTPS y use tokens/autenticación. Configure permisos en Django REST (p.ej. JWT, OAuth) según corresponda. Revise que los usuarios sólo puedan acceder o modificar las propiedades permitidas. En manejo de errores, no incluya datos internos (detalles de la BD, rutas de archivos) en las respuestas para evitar dar pistas a posibles atacantes [14]. Aplique buenas prácticas de seguridad web: proteja contra CSRF en formularios, valide toda entrada

del usuario en el servidor, y mantenga dependencias actualizadas para evitar vulnerabilidades conocidas.

8. Herramientas recomendadas

- **Control de versiones (Git):** Use Git para todo el código. Siga un flujo de trabajo con ramas (`feature/`, `develop`, `main`), y escriba mensajes de commit claros. Por ejemplo, use *GitHub* o *GitLab* para repositorios remotos. Esto facilita revisiones de código y colaboración.
- **Entorno virtual:** Para Python/Django, use `venv` o herramientas como *pipenv/Poetry* para aislar dependencias. Active el virtualenv antes de instalar paquetes (`pip install -r requirements.txt`). En Node/React, administre versiones de Node con *nvm* y use `package.json` con dependencias bloqueadas.
- **Linters y formateadores:** En Django, ESLint/flake8 para Python, Black o Autopep8 para formateo; en React, ESLint + Prettier para JS/JSX. Esto mejora la calidad del código y previene errores de estilo.
- **Testing frameworks:** Para Django use el módulo de `unittest` integrado (ejecutando `python manage.py test`) o pytest con `pytest-django` para sintaxis más concisa. Para React, use **Jest** + **React Testing Library** para pruebas unitarias y de componentes, y **Cypress** o **Playwright** para pruebas end-to-end de la aplicación completa. Ejecute estas pruebas en el pipeline CI para detectar regresiones.
- **Otras herramientas:** Considere *Docker* para contenerizar la aplicación y facilitar despliegues. Use *Postman* o *HTTPie* para probar manualmente las APIs. Para monitoreo de bases de datos, herramientas como *pgAdmin* (Postgres) o *MySQL Workbench* si aplica.

En resumen, siga principios de arquitectura limpia, pruebe exhaustivamente cada parte, y aprenda de los problemas previos (como manejo de nulos o exposición de claves). Aplicando estas prácticas (ver citas relevantes), el portal inmobiliario SkyTerra será más robusto, seguro y escalable [24] [6].

Fuentes: Documentación oficial de Django/DRF, blogs técnicos y ejemplos de la práctica (véanse referencias usadas) [9] [17].

---

[1] [2] [3] Working with Django Models in Python: Best Practices | Django Stars
https://djangostars.com/blog/django-models-best-practices/

[4] [5] [12] [18] [21] [27] skyterra_progress_log.txt
file://file-7HMgGVh4n2tCZtbRWZncr8

[6] [7] [8] [10] [11] Effectively Using Django REST Framework Serializers | TestDriven.io
https://testdriven.io/blog/drf-serializers/

[9] [13] Exceptions - Django REST framework
https://www.django-rest-framework.org/api-guide/exceptions/

[14] Why you should never return stack traces in API errors | Bruno Araujo posted on the topic | LinkedIn
https://www.linkedin.com/posts/bruno-kiau_api-design-why-you-should-never-return-activity-7320823771454423043-Jzdl

[15] [16] [17] [19] 10 buenas prácticas en React - Syntonize
https://www.syntonize.com/10-buenas-practicas-en-react/

20 curl - How to Include Chat History When Using Google Gemini's API - Stack Overflow
https://stackoverflow.com/questions/78534769/how-to-include-chat-history-when-using-google-geminis-api

22 javascript - My Gemini API key is not working properly - Stack Overflow
https://stackoverflow.com/questions/77900701/my-gemini-api-key-is-not-working-properly

23 24 Debugging a Django Application | Product Blog • Sentry
https://blog.sentry.io/debugging-a-django-application/

25 28 Using GeoJSON with Leaflet - Leaflet - a JavaScript library for interactive maps
https://leafletjs.com/examples/geojson/

26 leaflet.draw - React-Leaflet-Draw: accessing a polygon's array of coordinates on save - Stack Overflow
https://stackoverflow.com/questions/47905239/react-leaflet-draw-accessing-a-polygons-array-of-coordinates-on-save