DESARROLLO DE SOFTWARE

TRABAJO PRACTICO FINAL

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL RESISTENCIA

GRUPO N°: 12

- AGUIRRE, JULIAN JULIIAN.A@HOTMAIL.COM
- MUÑOZ, ALAN
- OJEDA, JUAN

DOCENTES:

- ING. CAROLINA ORCOLA
- ING. LUIS EIMAN

AÑO:

- 2024

INGENIERIA EN SISTEMAS DE INFORMACION

Tabla de contenido

INTRODUCCION

Nuestro trabajo práctico final, se baso en el desarrollo de una aplicación web para la Bienal Internacional de Escultura del Chaco, donde como grupo tomamos la decisión de comenzar definiendo los casos de uso de negocio (CUN). Lo que nos permitió comprender las necesidades específicas del proyecto y los actores involucrados.

Con esta base, diseñamos un modelo relacional para la base de datos (DER), que nos ayudó a visualizar de las relaciones entre los diferentes elementos clave, como escultores, esculturas y visitantes. E

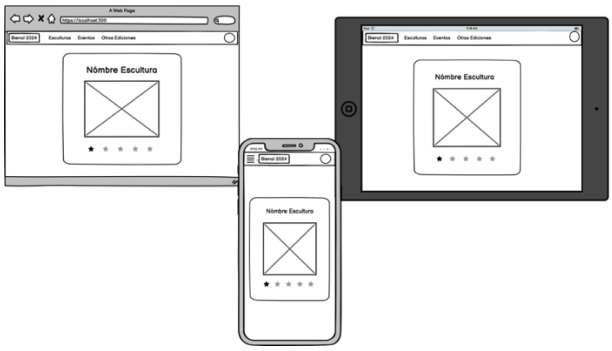
Además, para asegurar un diseño centrado en el usuario, desarrollamos wireframes en equipo, tomando como referencia el diseño de la página actual de la Bienal. Nos enfocamos en los componentes principales a tener en cuenta en este proyecto, como la de inicio, la de selección de esculturas, la de votación y el perfil del votante.

Para gestionar el código de manera organizada, utilizamos un repositorio Git compartido, donde realizábamos actualizaciones , ya sea trabajando en equipo o de manera individual. Permitiéndonos tener sincronizado el desarrollo , realizar revisiones conjuntas y resolver conflictos de integración de forma eficiente.

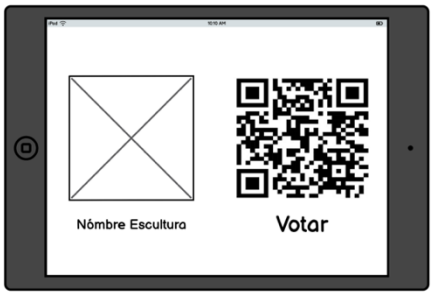
Finalmente, seleccionamos React para el frontend y Django para el backend, ya nos permitia cumplir con los requisitos funcionales como con los no funcionales establecidos por la cátedra.

WIREFRAMES RESPONSIVE PROPUESTOS

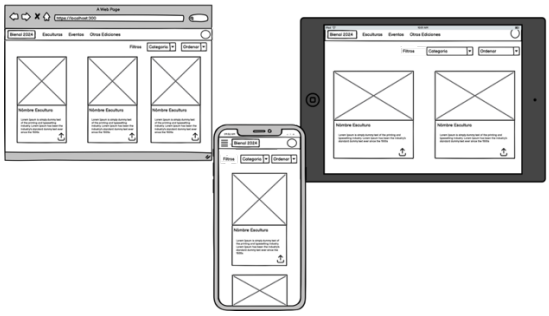
Sección Votación



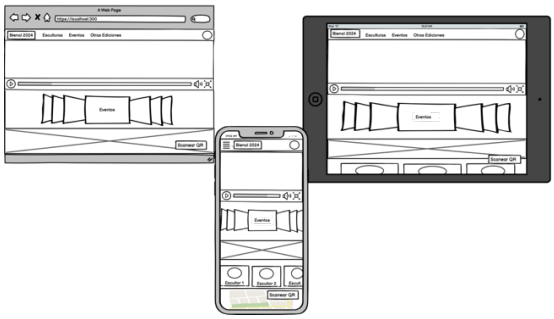
Sección QR dinámico



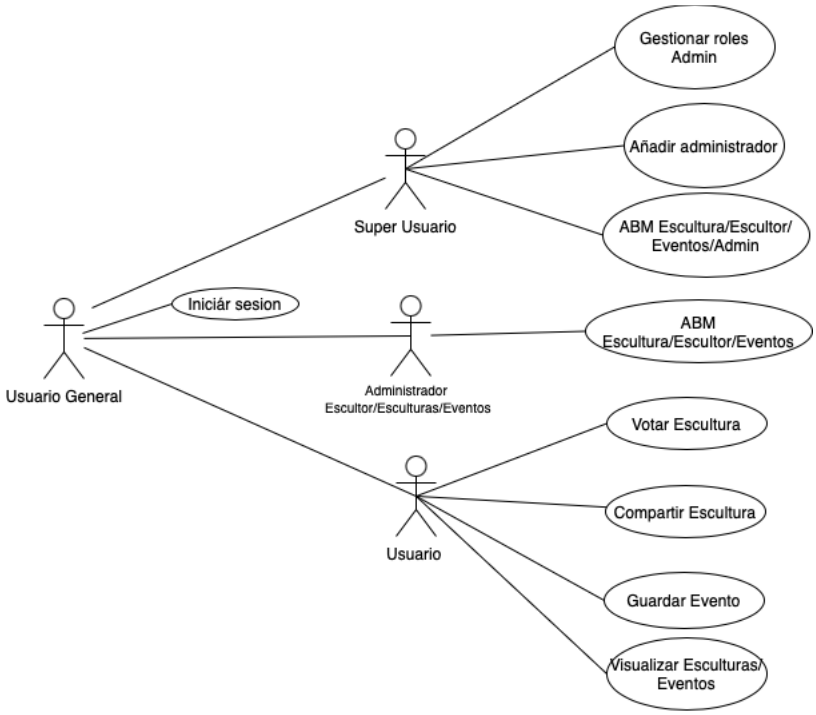
Seccion Esculturas



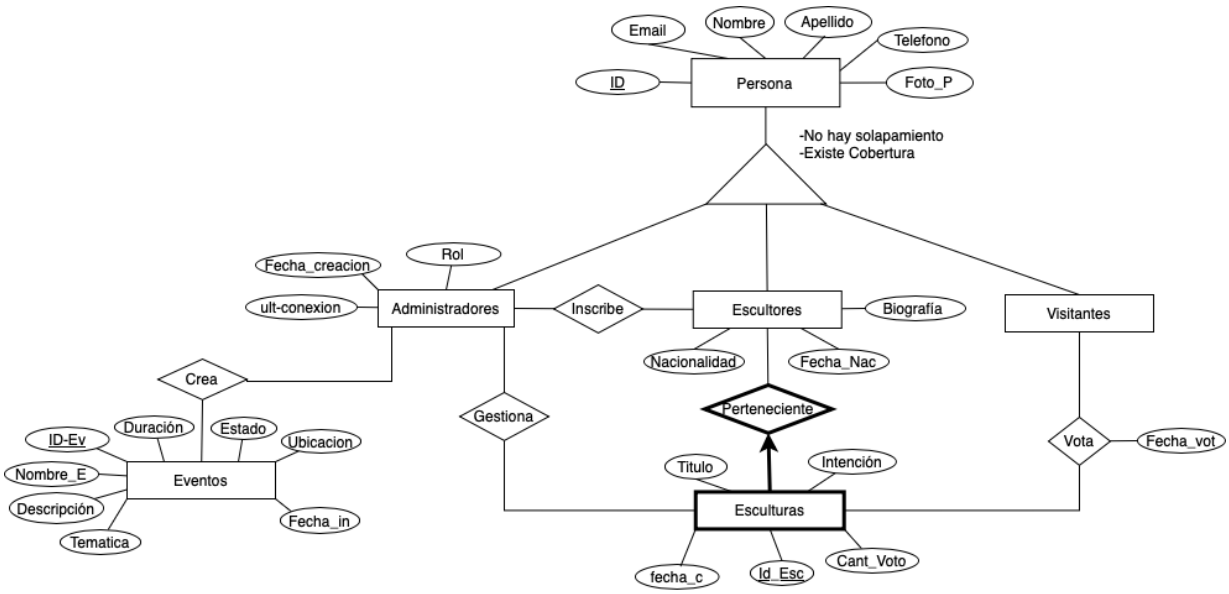
Seccion Home



MODELO DE CASO DE USO



MODELO DE DIAGRAMA ENTIDAD RELACION



INFORME DETALLADO

El proyecto utiliza React 18.3.1 para estructurar la lógica del frontend, ya que es una biblioteca ampliamente soportada y modular, lo que facilita el desarrollo y mantenimiento. Para la navegación entre las secciones de la aplicación, como la gestión de eventos o escultores, se eligió React Router DOM 6.27.0, que permite un manejo eficiente de rutas y facilita la implementación de un sistema de navegación basado en URL.

Para realizar peticiones HTTP y conectar el frontend con el backend, se optó por Fetch debido a su integración nativa en los navegadores modernos, eliminando la necesidad de librerías externas y permitiendo control detallado sobre las solicitudes y respuestas.

La interfaz de usuario está diseñada con Chakra UI 2.10.4, seleccionada por su conjunto de componentes preconstruidos y estilizados que son responsivos de manera predeterminada, lo que reduce el tiempo de desarrollo. Para implementar animaciones y transiciones en los componentes clave, se decidió usar Framer Motion 11.11.15 debido a su integración directa con React y su capacidad para manejar animaciones declarativas. FontAwesome 6.6.0 se eligió para los íconos porque proporciona un amplio catálogo necesario para representar acciones como compartir o votar, sin tener que diseñar íconos personalizados.

Para garantizar la seguridad en la autenticación de usuarios y restringir las votaciones a una por visitante, se implementó Auth0 React 2.2.4, que simplifica la integración de autenticación OAuth2 y provee herramientas de gestión de usuarios robustas. La generación de códigos QR dinámicos se realizó con React QR Code 2.0.15, ya que permite generar códigos en tiempo real de manera eficiente. Para manejar la validez de los códigos y evitar manipulación, se empleó bcrypt 5.1.1 para generar hashes únicos que identifican cada QR y permiten invalidarlos tras un periodo determinado.

Para la gestión de imágenes, se utilizó React Dropzone 14.3.5, ya que permite implementar fácilmente una interfaz de carga de archivos compatible con diferentes navegadores. Las imágenes se almacenan en Firebase 10.14.1, que proporciona una API sencilla para la carga y recuperación de archivos, devolviendo un enlace único que se guarda en la base de datos para futuras consultas.

Para implementar carruseles en la página principal, se eligieron React Slick 0.30.2 y Slick Carousel 1.8.1 debido a su capacidad para manejar contenido dinámico y su compatibilidad con React. Para asegurar que la aplicación funcione sin conexión, se configuró como una PWA utilizando React Scripts 5.0.1. Finalmente, para garantizar tiempos de carga rápidos y monitorear el rendimiento, se incluyó Web Vitals 2.1.4, que permite medir métricas esenciales como tiempo de interacción y velocidad de renderizado.

ESTRUCTURA DEL PROYECTO

Directorio Principal:

El proyecto está dividido en carpetas que representan las distintas funcionalidades y aspectos importantes de la aplicación. Esto incluye la parte del frontend y varios módulos específicos para la lógica de negocio.

Frontend: La carpeta frontend contiene todo lo relacionado con la interfaz de usuario, que fue desarrollada usando React. Dentro de esta carpeta tenemos:

- Src: contiene los archivos y subcarpetas principales del código fuente de la aplicación.
  - Assets: Carpeta dedicada a los recursos estáticos, como imágenes, íconos o archivos multimedia que se utilizan en la aplicación.
  - Layouts: Incluye los diseños generales de las páginas
    - private_sesion: incluye todos los componentes privados al publico general
      - ❖ Event_management: Manejo de eventos
        - ➢ Delete_event.js: permite buscar, visualizar y eliminar eventos de una lista obtenida desde un backend. Permite filtrar los eventos por nombre y confirma la eliminación solicitando un PIN a través de un modal interactivo.
        - ➢ Edit_event.js: permite filtrar y editar eventos de una lista obtenida desde un backend. Permite a los usuarios seleccionar un evento, editar sus detalles en un modal interactivo y guardar los cambios directamente en el servidor
        - ➢ Register_event.js permite registrar nuevos eventos mediante un formulario interactivo con campos como nombre, temática, ubicación y descripción. Utiliza validación básica para asegurar que todos los campos estén completos antes de enviar.
      - ❖ Sculptor_management: Manejo de escultores
        - ➢ Delete_sculptor.js: permite buscar, visualizar y eliminar escultores de una lista obtenida desde un backend y confirma la eliminación solicitando un PIN a través de un modal interactivo.
        - ➢ Edit_sculptor.js: permite filtrar y editar información de escultores de una lista obtenida desde un backend. Editando sus detalles en un modal interactivo y guardar los cambios directamente en el servidor
        - ➢ Sculptor_register.js: permite registrar nuevos escultores mediante un formulario completo, incluyendo campos como nombre, apellido, DNI, nacionalidad, teléfono,

email y biografía. También incluye la funcionalidad de cargar y previsualizar una foto del escultor.

- Sculpture_management: Manejo de esculturas
  - Delete_sculptore.js: permite buscar, visualizar y eliminar escultora de una lista obtenida desde un backend , junto con su relación al escultor y confirma la eliminación solicitando un PIN a través de un modal interactivo.
  - Edit_sculpture.js: permite filtrar y editar información de las esculturas de una lista obtenida desde un backend. Editando sus detalles en un modal interactivo, con opción para añadir nuevas fotos y guardar los cambios directamente en el servidor
  - Sculptor_register.js: permite registrar esculturas asociadas a escultores y eventos, completando información como título, intención y material principal. Incluye una vista previa de los datos ingresados mediante un modal interactivo antes de confirmar el envío.
- Allsculptures.js: muestra una lista de esculturas del escultor al que se encuentran asociada y permite navegar a una vista detallada en formato tablet de cada escultura mediante un botón que redirige a una ruta dinámica basada en el ID de la escultura.
- Panel.js :es un panel de administración que permite gestionar escultores, esculturas y eventos mediante botones interactivos. Al seleccionar una opción, se abre un modal que ofrece acciones específicas como agregar, editar o eliminar elementos, redirigiendo a las rutas correspondientes.
- Preview.js: es una vista previa diseñada para mostrar los detalles de una escultura antes de confirmar su registro. Presenta información como el título, material principal (temática), intención artística, y fechas relevantes. También incluye un espacio reservado para mostrar una imagen asociada a la escultura.
- Qrcoding.js: genera un código QR dinámico que se actualiza automáticamente cada cierto intervalo definido (por defecto, 10 segundos). Combina un timestamp codificado en base64 con datos proporcionados, como un ID opara crear una URL única que se representa como un QR que una vez superado el tiempo,el mismo redirije a otra pagina.
- Uploader.js: permite a los administradores cargar una imagen mediante arrastrar y soltar o seleccionar un archivo desde su dispositivo. Admite formatos .png, .jpg y .jpeg con un tamaño máximo de 5MB.

- public_sesion: incluye todos los compontes públicos para el sitio de web publico
    - ❖ events: carpeta que incluye todo aquello relacionado con la visualización de eventos
        - ➢ Allevents.js: renderiza una lista de eventos en un diseño de cuadrícula adaptable. Recibe los datos de los eventos desde el estado de la ubicación (useLocation) y utiliza el componente Eventcard para mostrar la información de cada evento en tarjetas individuales
        - ➢ Eventcard.js: muestra la información de un evento dentro de una tarjeta. Incluye un encabezado con el nombre del evento, una sección con detalles adicionales, y una imagen representativa. En el pie de la tarjeta, hay botones interactivos para acciones como "Like", "Compartir".
    - ❖ main: carpeta que incluye todo aquellos componentes que son renderizados a la hora de ingresar al home de la pagina
        - ➢ Main.js: organiza y combina varios subcomponentes principales en la página inicial, mostrando videos, un mapa interactivo, una lista de eventos y una lista de escultores
    - ❖ sculptors: carpeta que contiene componentes relacionados con la gestión y visualización de información de escultores.
        - ➢ SculptorCardMain.js: muestra una tarjeta con información básica de un escultor. Incluye un avatar, el nombre completo del escultor también ofrece un botón para ver más detalles, que redirige a una página específica utilizando React Route
        - ➢ ScuptorProfile.js: presenta un perfil detallado de un escultor, mostrando su foto, nombre, apellido, nacionalidad y biografía.
        - ➢ SculptorslistMain.js: es un carrusel que muestra una lista de escultores de forma interactiva y automática. Utiliza react-slick para crear un deslizador adaptable, configurado para mostrar diferentes cantidades de tarjetas según el tamaño de la pantalla. Carga los datos de los escultores desde una API y, mientras los datos están siendo obtenidos, muestra un spinner de carga.
    - ❖ Sculptures: carpeta que contiene componentes relacionados con la gestión y visualización de información de esculturas.
        - ➢ Cardlist.js: genera una lista de tarjetas para mostrar información de esculturas.
        - ➢ CardVote.js: permite a los usuarios votar por una escultura específica. Carga los datos de la escultura desde un backend usando su ID y muestra su título, imagen, y un sistema de calificación mediante estrellas (StarRating). Una vez que el usuario vota, se actualiza la cantidad de votos en el backend

- FiltterBar.js: es una barra de filtros que permite a los usuarios aplicar criterios de selección y ordenamiento en una lista o galería
- Qrvotes.js: El componente extrae un ID de escultura desde la URL utilizando la función extractIdFromUrl y, si encuentra un ID válido, renderiza el componente CardVote para permitir la votación de esa escultura; de lo contrario, muestra un mensaje indicando que no se encontró un ID válido en la URL.
- Sculpturelist.js: muestra una lista de esculturas obtenidas Mientras se cargan los datos, se muestran tarjetas como skeleto. Una vez que los datos están disponibles, cada escultura se presenta en una tarjeta que incluye su imagen, título y descripción breve. Además, cada tarjeta ofrece un botón para compartir la escultura utilizando la API de compartir del navegador.
- Starranking.js: permite a los usuarios calificar entre 1 y 5 estrellas de forma interactiva, mostrando etiquetas descriptivas y ejecutando un callback con el puntaje seleccionado.

- Footer.js: pie de página responsivo que incluye un mapa embebido, información de contacto, enlaces a redes sociales
- Navbar.js: barra de navegación que incluye enlaces, gestión de autenticación con Auth0, un menú de usuario dinámico con opciones de perfil y cierre de sesión
- Swippers.js: componete que contiene funciones de carruseles interactivos para mostrar galerías de imágenes y listas de elementos

Backend:

El proyecto backend utiliza Django como framework principal para desarrollar la API REST. Django es un framework robusto y flexible que permite organizar el código en "aplicaciones" o "apps". Cada aplicación dentro del proyecto se encarga de una parte específica de la funcionalidad del sistema. Esta estructura modular permite una mejor organización del código, facilita la reutilización y mejora el mantenimiento del proyecto.

Base de Datos

El backend utiliza Firebase Realtime Database. La configuración incluye:

- Un archivo de credenciales JSON para autenticar la conexión con Firebase.
- URL de la base de datos especificada mediante una variable de entorno (`FIREBASE_DATABASE_URL`).

Configuración del archivo:

```python
from pathlib import Path
from urllib.parse import urlparse
from dotenv import load_dotenv
import os

import firebase_admin
from firebase_admin import credentials

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
FIREBASE_CREDENTIALS_PATH = os.path.join(BASE_DIR, 'serviceAccountKey.json')
firebase_database_url = os.getenv('FIREBASE_DATABASE_URL')


if not firebase_admin._apps:
    cred = credentials.Certificate(FIREBASE_CREDENTIALS_PATH)
    firebase_admin.initialize_app(cred, {
        'databaseURL': 'https://imagenes-url-default-rtdb.firebaseio.com/'
    })
```

Middleware Utilizado

El proyecto utiliza varios middleware para la seguridad y el manejo de solicitudes:

- `corsheaders.middleware.CorsMiddleware`: Permite el intercambio de recursos entre dominios.
- `django.middleware.security.SecurityMiddleware`: Manejo de la seguridad HTTP.
- `django.middleware.csrf.CsrfViewMiddleware`: Prevención de ataques CSRF.
- Otros middleware de sesiones, autenticación y manejo de mensajes.

¿Qué son las aplicaciones en Django?

En Django, una "aplicación" (app) es una unidad de funcionalidad dentro del proyecto. Cada aplicación está diseñada para realizar tareas específicas, y puede interactuar con otras aplicaciones en el mismo proyecto. Por ejemplo, tenemos aplicaciones para gestionar usuarios, eventos, esculturas, autenticación, entre otras.

Ventajas de Usar Aplicaciones Separadas en Django

Modularidad: Al separar la lógica de negocio en aplicaciones distintas, puedes gestionar mejor cada aspecto del proyecto. Cada aplicación puede enfocarse en una tarea específica (como autenticación, administración de eventos, gestión de esculturas, etc.), lo que facilita el desarrollo y el mantenimiento.

Reutilización: Las aplicaciones en Django son reutilizables. Esto significa que puedes tomar una aplicación (por ejemplo, una de gestión de usuarios) y utilizarla en otros proyectos si fuera necesario, sin tener que duplicar el código.

Escalabilidad: Cuando el proyecto crece, la arquitectura modular facilita la escalabilidad. Si necesitas añadir nuevas funcionalidades, puedes agregar nuevas aplicaciones sin que esto afecte demasiado al resto del proyecto.

Desarrollo en equipo: La organización en aplicaciones facilita el trabajo en equipo. Diferentes miembros del equipo pueden trabajar en distintas aplicaciones sin interferir en el trabajo de otros, lo que mejora la productividad.

Estructura del proyecto: Aplicaciones

```
├── bienal2024/
│   ├── bienal2024/
│   ├── gestionEscultores
│   ├── gestionEsculturas
│   ├── gestionEventos
│   ├── gestionUsuarios
│   └── gestionVotos
```

Descripción de Aplicaciones (gestionEventos)

```
├── gestionEventos/
│   ├── models.py
│   ├── Serializers.py
│   ├── Urls.py
│   ├── views.py
│   ├── __init__.py
```

Models.py:
El archivo models.py define el modelo de datos de los eventos. Aquí es donde se especifican los campos que representarán cada evento en la base de datos (en este caso, Firebase Realtime Database) y su estructura.

```python
from django.db import models

class Evento(models.Model):
    nombre = models.CharField(max_length=255)
    descripcion = models.TextField()
    tematica = models.CharField(max_length=255)
    ubicacion = models.CharField(max_length=255)
    img_evento = models.URLField(max_length=500)
    fecha_inicio = models.CharField(max_length=10)
    fecha_fin = models.CharField(max_length=10)
```

Serializers.py:
El archivo serializers.py es responsable de transformar los objetos de Django (como los modelos) en formato JSON para ser enviados y recibidos a través de la API. También valida que los datos recibidos en las solicitudes estén correctamente formateados

Urls.py:
El archivo urls.py define las rutas de la API relacionadas con los eventos. Utiliza el router de Django REST Framework para crear rutas automáticamente para las operaciones CRUD de los eventos.

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import EventoViewSet

router = DefaultRouter()
router.register(r'eventos', EventoViewSet, basename='evento')

urlpatterns = [
    path('', include(router.urls)),
]
```

Views.py:
El archivo views.py contiene la lógica de negocio para manejar las operaciones CRUD sobre los eventos. Se utiliza Django REST Framework para crear una API RESTful que interactúa con Firebase Realtime Database.

EventoViewSet: Este es un viewset que proporciona la lógica para manejar las solicitudes HTTP relacionadas con los eventos.

list: Devuelve una lista de todos los eventos almacenados en Firebase.
retrieve: Recupera un evento específico usando su id.
create: Permite crear un nuevo evento y almacenarlo en Firebase.
update: Actualiza un evento específico con nuevos datos.
partial_update: Actualiza parcialmente los datos de un evento.
destroy: Elimina un evento específico de Firebase.

Flujo de Funcionamiento de las Apps (usando de ejemplo gestionEventos)

Imaginemos que el cliente (frontend o cualquier consumidor de la API) realiza una solicitud HTTP GET para obtener una lista de eventos. Vamos a ver qué pasa en cada paso del proceso.

1. Solicitud HTTP:
El cliente realiza una solicitud GET a la URL del servidor backend, como por ejemplo:
GET http://localhost:8000/eventos/
2. Enrutamiento de la Solicitud (Paso 1):
La solicitud llega al servidor de Django en el puerto 8000. En Django, las solicitudes entrantes se enrutan a través del archivo urls.py de la raíz del proyecto (el archivo principal).
El archivo urls.py del proyecto principal redirige la solicitud al archivo urls.py de la aplicación gestionEventos a través de include('gestionEventos.urls'). Esto asegura que cualquier solicitud relacionada con los eventos se maneje en la app correcta.

3. Enrutamiento de la Solicitud (Paso 2):
Una vez que la solicitud llega al archivo urls.py de la aplicación gestionEventos, Django utiliza los routers de Django REST Framework para manejar la solicitud y mapearla al método adecuado dentro del ViewSet.

```python
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import EventoViewSet

router = DefaultRouter()
router.register(r'eventos', EventoViewSet, basename='evento')

urlpatterns = [
    path('', include(router.urls)),
]

```

El DefaultRouter crea rutas automáticamente para todas las operaciones CRUD del EventoViewSet, como GET /eventos/, POST /eventos/, PUT /eventos/{id}/, y DELETE /eventos/{id}/.

4. Procesamiento de la Solicitud en views.py:
La solicitud ahora llega al archivo views.py de la aplicación gestionEventos, donde Django REST Framework se encarga de procesarla utilizando el ViewSet adecuado.

```python
from rest_framework import viewsets, status
from rest_framework.response import Response
from firebase_admin import db
from .Serializers import EventoSerializer

ref = db.reference('eventos')

class EventoViewSet(viewsets.ViewSet):


    def list(self, request):

        try:
            eventos_ref = ref.get()
            eventos = []
            if eventos_ref:
                for evento_id, evento_data in eventos_ref.items():
                    evento_data['id'] = evento_id
                    eventos.append(evento_data)
            return Response(eventos, status=status.HTTP_200_OK)
        except Exception as e:
            return Response({'error': str(e)}, status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

list: Cuando el cliente hace una solicitud GET a http://localhost:8000/eventos/, el método list de EventoViewSet es el que se ejecuta. Este método obtiene todos los eventos almacenados en Firebase y devuelve los resultados en formato JSON.

La respuesta HTTP es un 200 OK con la lista de eventos.

5. Respuesta al Cliente:
El servidor Django genera la respuesta HTTP basada en la lógica del método de la vista que se ejecutó. En este caso, se devuelve una lista de eventos en formato JSON.

Ejemplo de respuesta a un GET:

Este JSON es enviado como respuesta al cliente.

Flujo para Otros Métodos (POST, PUT, DELETE)

POST (/eventos/): Cuando el cliente realiza una solicitud POST para crear un nuevo evento, el router registra la solicitud y la redirige al método create del EventoViewSet, donde se validan los datos y se almacenan en Firebase.

PUT (/eventos/{id}/): Una solicitud PUT actualizaría un evento específico. El router redirige la solicitud al método update del EventoViewSet, que valida y luego actualiza el evento en Firebase.

DELETE (/eventos/{id}/): Una solicitud DELETE eliminaría un evento. La solicitud se maneja en el método destroy del EventoViewSet, que elimina el evento de Firebase.