UNIVERSIDAD TECNOLOGICA NACIONAL
FACULTAD REGIONAL RESISTENCIA

INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Materia: Desarrollo de Software

Nivel: 3

Ciclo Lectivo: 2024 – 2do. Cuatrimestre

Integrantes:

- Coschiza, Enzo
  - enzocoschiza.isi@gmail.com
- Diez, Chiara
  - diezchiara@gmail.com
- Drose, Juan Ignacio
  - juanidrose@gmail.com
- Fidanza, Felipe
  - ffidanzar@gmail.com
- Irala, Damian Agustin
  - agustinirala240@gmail.com
- Jerez, Fabricio
  - fabrijerez1@gmail.com

Grupo: 3

Docentes:

- P.h.D Villaverde, Jorge
- I.S.I Fernandez, José Alejandro

Introducción

La Bienal Internacional de Escultura del Chaco representa uno de los eventos culturales más importantes de la región, congregando escultores de renombre y promoviendo la participación activa del público. Ante la necesidad de modernizar y optimizar los procesos de gestión y promoción del evento, nuestro equipo fue convocado para desarrollar un sistema integral que facilite el registro de eventos, la gestión de escultores y esculturas, y la interacción de los visitantes mediante comentarios y votaciones.

Este proyecto no solo busca satisfacer las necesidades administrativas de la organización, sino también ofrecer una experiencia enriquecedora a los visitantes mediante una aplicación web progresiva (PWA), accesible desde dispositivos móviles, tabletas y computadoras.

El sistema ha sido diseñado utilizando Django REST Framework para el backend y Next.js con TypeScript para el frontend, asegurando escalabilidad, rendimiento y facilidad de mantenimiento. A lo largo de este informe, se detallarán las decisiones de diseño, las soluciones implementadas y los resultados obtenidos durante el desarrollo del proyecto.

Requerimientos Funcionales

Gestión de Eventos:

1. Crear y registrar eventos futuros.
2. Cargar información de eventos pasados para mantener un historial.
3. Posibilidad de agregar, ver, modificar y eliminar información de cada evento.
    - **Detalles:** Fecha, lugar, descripción y temática.

Gestión de Escultores:

1. Registrar y administrar la información de los escultores.
2. Visualizar y actualizar un perfil detallado que incluya:
    - Nombre, biografía, datos de contacto y obras previas.

Gestión de Esculturas:

1. Registrar esculturas y asociarlas con eventos o escultores.
2. Incluir detalles como la temática, fecha de creación y descripciones.

Gestión de Imágenes:

1. Subir imágenes relacionadas con esculturas en diferentes etapas (creación, exposición, etc.).
2. Visualización optimizada para dispositivos móviles y de escritorio.

Sistema de Votación:

1. Permitir que los visitantes voten por sus esculturas favoritas (de 1 a 5 estrellas).
2. Implementar un sistema de votación mediante:
    - Botón en la aplicación web pública.
    - QR dinámico generado cada minuto, accesible desde tablets/pantallas en el predio.
3. Autenticación para asegurar que cada visitante pueda votar una única vez.

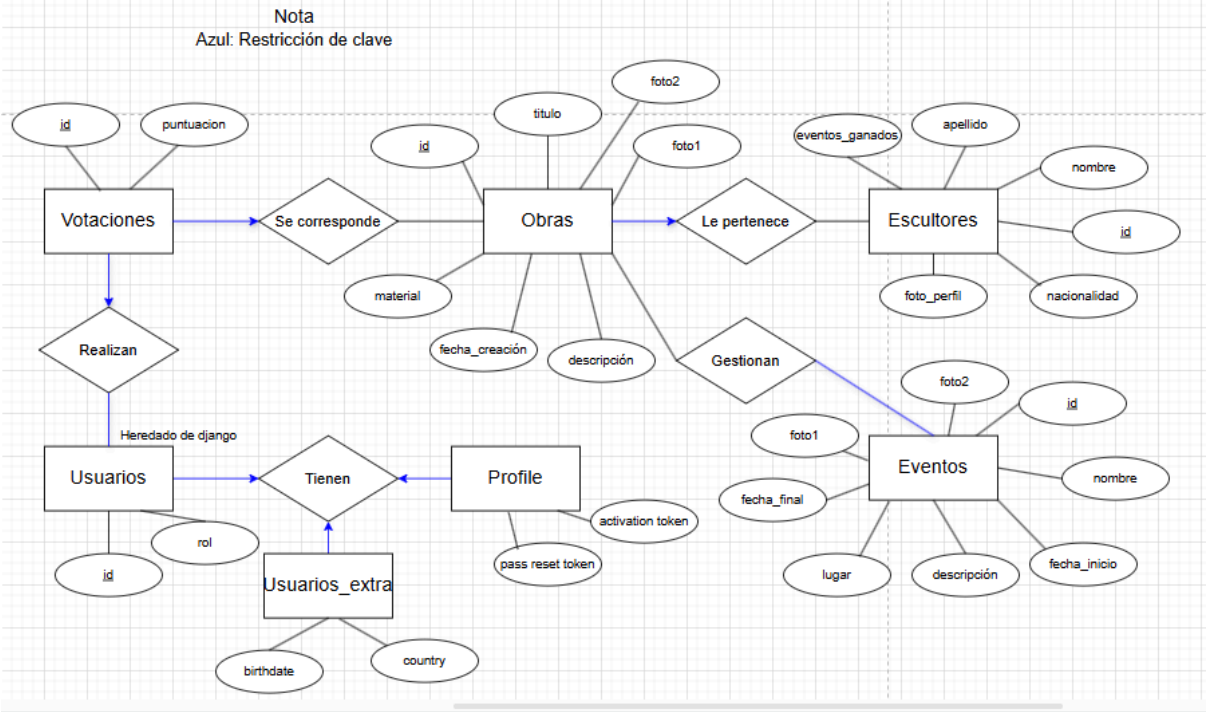
Requerimientos No Funcionales

1. Interfaz Adaptable: La UI debe ser responsiva y funcional en dispositivos móviles, tablets y PC.
2. Seguridad: Protección de datos mediante autenticación segura y encriptación.
3. Optimización de Imágenes: Las fotos cargadas deben ser ligeras sin comprometer la calidad.
4. Tiempo de Respuesta: El sistema debe responder en menos de 2 segundos en promedio para solicitudes comunes.
5. Accesibilidad: Diseño que cumpla estándares para personas con discapacidades (WCAG).

Decisiones de diseño

Estructura del proyecto

Teniendo en cuenta el escenario y los requerimientos solicitados en la consigna, primero realizamos el diagrama entidad - relación para guiarnos en cómo teníamos que estructurar el proyecto, sobre todo desde el backend:



La arquitectura que creímos que mejor se adapta a nuestro proyecto es de tipo Cliente - Servidor, en nuestro caso utilizamos un modelo cliente ligero. Esto porque toda la lógica del negocio, como las votaciones, los registros y los controles, se realizan del lado del Backend, mientras que del lado del Frontend simplemente se presenta la información.

Herramientas y tecnologías

- Para el Frontend: Next.js, React, Tailwind CSS, Shadcn, Vercel AI (v0), Librerías de React para formularios y modales, Zod, Zustand
- Para el Backend: Python con su framework Django Rest
- Servicios: Cloudinary, Render, Xata.
- Colaboración grupal: Trello (diagrama de Kanban), GitHub, Discord.

Introducción al Backend

Para poder cumplir con todos los requisitos, debimos tomar decisiones acerca de servicios y tecnologías que íbamos a usar para el desarrollo del backend.
Dentro de las decisiones fundamentales que tuvimos al realizar el proyecto se encuentran:

- Que framework y lenguaje íbamos a usar
- Como íbamos a almacenar la imágenes de los modelos que la necesiten
- Que base de datos íbamos a utilizar

- A la hora de plantear un deploy, qué servicio gratuito y efectivo podíamos usar
- Como íbamos a tratar la seguridad y los tokens para la misma

Todas estas dudas nos llevaron a investigar y sumergirnos en el tema, logrando como grupo el aprendizaje de muchas herramientas, tecnologías y servicios que se encuentran disponibles en la actualidad.

Optamos por el uso de *Python* y su framework *Django Rest*, nos decidimos por este ya que varios del grupo tenían familiaridad con este lenguaje y este framework, además notamos que es bastante robusto y completo.
Nos decidimos a usar *Cloudinary* para el almacenamiento de imágenes, ya que nos da bastante libertad en el servicio gratuito, es fácil su uso en django rest y además es efectivo y performante.
Decidimos hacer un despliegue de la API, así los programadores frontend del equipo, no debían estar preocupándose por tener todas las dependencias y el entorno virtual de los backends.
Usamos el servicio gratuito de *Render* para desplegar “Web Services”. Este servicio ya nos brinda un dominio y es fácil de configurar y crear un entorno para la correcta ejecución de nuestro proyecto.
Además, debimos usar un servicio gratuito de bases de datos, en un principio nos decidimos por una base de datos Postgres en Render, pero esta tenía la particularidad que se vencia cada un mes. Por lo que debimos afrontar el cambio y buscar un nuevo servicio gratuito, en donde encontramos el que usamos actualmente que es *Xata.io*, esta también es una base de datos en Postgres, de uso gratuito con ciertas limitaciones de request, pero que nos sobraban para realizar nuestro proyecto.
Respecto a la seguridad implementada, nos decidimos a usar una librería de django rest, que fue muy fácil de complementar con nuestro proyecto llamada TokenAuthentication. Esta no llega a ser una JWT pura, pero cumplía con nuestros requisitos.
Para enviar los correos electrónicos creamos un gmail dedicado al proyecto y creamos una función usando las librerías: smtplib y email.mime para manipular en cada vista el envío de correos.
Cuando creamos el correo tuvimos que aprender a configurarlo usando claves de aplicaciones.

Implementación del Backend

Documentación de Modelos Django Rest

1. Modelo `Escultores`

Representa a los escultores participantes de los eventos.

Atributos:

- `nombre` *(CharField)*: Nombre del escultor (máximo 50 caracteres, obligatorio).

- `apellido` *(CharField)*: Apellido del escultor (máximo 100 caracteres, obligatorio).
- `fecha_nacimiento` *(DateField)*: Fecha de nacimiento del escultor, obligatorio.
- `nacionalidad` *(CharField)*: Nacionalidad del escultor (máximo 50 caracteres, obligatorio).
- `eventos_ganados` *(CharField)*: Eventos en los que el escultor ha resultado ganador.
- `foto_perfil` *(CloudinaryField)*: Foto del perfil del escultor almacenada en Cloudinary.

Métodos:

- `save`: Verifica si el perfil ya existe para eliminar imágenes anteriores al actualizar.
- `delete`: Elimina la imagen del perfil de Cloudinary al borrar el escultor.
- `__str__`: Retorna el nombre completo del escultor.

2. Modelo `Eventos`

Representa los eventos de la bienal.

Atributos:

- `nombre` *(CharField)*: Nombre del evento (máximo 100 caracteres, obligatorio).
- `fecha_inicio` *(DateField)*: Fecha de inicio del evento, obligatorio.
- `fecha_final` *(DateField)*: Fecha de finalización del evento, obligatorio.
- `lugar` *(CharField)*: Lugar donde se desarrolla el evento (máximo 75 caracteres, obligatorio).
- `descripcion` *(CharField)*: Descripción del evento (máximo 500 caracteres, obligatorio).
- `foto1` *(CloudinaryField)*: Imagen principal del evento.
- `foto2` *(CloudinaryField)*: Imagen secundaria del evento.

Métodos:

- `save`: Elimina imágenes previas al actualizar fotos.
- `delete`: Elimina las imágenes del evento de Cloudinary.
- `evento_en_transcurso`: Indica si el evento está "En curso", "Finalizado" o "Por iniciar".
- `__str__`: Retorna el nombre del evento.

3. Modelo `Obras`

Representa las esculturas creadas por los escultores y asociadas a eventos.

Atributos:

- `titulo` *(CharField)*: Título de la obra (máximo 100 caracteres, obligatorio).
- `fecha_creacion` *(DateField)*: Fecha de creación de la obra, obligatorio.
- `descripcion` *(CharField)*: Descripción de la obra (máximo 500 caracteres).
- `material` *(CharField)*: Materiales utilizados para la construcción de la obra (máximo 200 caracteres).
- `id_escultor` *(ForeignKey)*: Relación con el escultor creador de la obra.
- `id_evento` *(ForeignKey)*: Relación con el evento donde participa la obra.
- `foto1` y `foto2` *(CloudinaryField)*: Imágenes de la obra en diferentes etapas.

Métodos:

- `save`: Elimina imágenes previas al actualizar fotos.
- `delete`: Elimina las imágenes de la obra de Cloudinary.
- `es_qr_valido`: Determina si el QR generado para la obra sigue vigente.
- `votacion_en_transcurso`: Indica si las votaciones asociadas a la obra están activas, basado en el evento.
- `__str__`: Retorna el título de la obra con el nombre del escultor.

4. Modelo `UsuariosExtra`

Extiende el modelo de usuarios para incluir información adicional.

Atributos:

- `birthdate` *(DateField)*: Fecha de nacimiento del usuario.
- `country` *(CharField)*: País del usuario (máximo 50 caracteres).
- `user` *(OneToOneField)*: Relación uno a uno con el modelo de usuarios de Django.

Métodos:

- `__str__`: Retorna el nombre del usuario relacionado.

5. Modelo `Votaciones`

Registra las votaciones realizadas por los usuarios para las obras.

Atributos:

- `puntuacion` *(IntegerField)*: Puntuación dada (entre 1 y 5).
- `id_usuario` *(ForeignKey)*: Relación con el usuario que realizó la votación.

- `id_obra` *(ForeignKey)*: Relación con la obra votada.

Restricciones:

- `unique_together`: Evita que un usuario vote más de una vez por la misma obra.

Métodos:

- `__str__`: Retorna un string con la puntuación, obra y usuario relacionados.
- `resultados_evento`: Calcula los resultados totales de las votaciones para las obras de un evento específico.

6. Modelo `Profile`

Amplía el modelo de usuario con tokens de activación y recuperación de contraseña.

Atributos:

- `user` *(OneToOneField)*: Relación con el modelo de usuarios de Django.
- `password_reset_token` *(CharField)*: Token para restablecer la contraseña.
- `activation_token` *(CharField)*: Token para la activación de la cuenta.

Métodos:

- `__str__`: Retorna el nombre de usuario relacionado.

Observaciones generales de los modelos

- Todos los modelos están diseñados para integrarse con Cloudinary para el manejo de imágenes, asegurando su optimización y disponibilidad en la nube.
- Se han implementado métodos `save` y `delete` personalizados para garantizar la eliminación correcta de imágenes antiguas en Cloudinary.
- La lógica para determinar el estado de eventos y votaciones asegura una experiencia de usuario coherente y alineada con los requisitos funcionales.
- Cada modelo tenía un serializador para poder ser tratado y comunicado.

Endpoints y vistas dispuestas en la API

1. Escultores

- **Listar todos los escultores**

    - **URL:** /api/escultores/
    - **Método:** GET
    - **Descripción:** Devuelve una lista de todos los escultores.
    - **Filtros:** id, eventos_ganados, nacionalidad, nombre, apellido
    - **Buscar:** nombre, apellido, nacionalidad
- **Crear un nuevo escultor**
    - **URL:** /api/escultores/
    - **Método:** POST
    - **Descripción:** Crea un nuevo escultor. Solo los usuarios administradores pueden realizar esta acción.
- **Actualizar un escultor**
    - **URL:** /api/escultores/<int:pk>/
    - **Método:** PUT
    - **Descripción:** Actualiza la información de un escultor existente. Solo los usuarios administradores pueden realizar esta acción.
- **Eliminar un escultor**
    - **URL:** /api/escultores/<int:pk>/
    - **Método:** DELETE
    - **Descripción:** Elimina un escultor existente. Solo los usuarios administradores pueden realizar esta acción.

2. Eventos

- **Listar todos los eventos**
    - **URL:** /api/eventos/
    - **Método:** GET
    - **Descripción:** Devuelve una lista de todos los eventos.
    - **Buscar:** nombre, lugar, descripción.
- **Crear un nuevo evento**
    - **URL:** /api/eventos/
    - **Método:** POST
    - **Descripción:** Crea un nuevo evento. Solo los usuarios administradores pueden realizar esta acción.
- **Actualizar un evento**
    - **URL:** /api/eventos/<int:pk>/
    - **Método:** PUT
    - **Descripción:** Actualiza la información de un evento existente. Solo los usuarios administradores pueden realizar esta acción.
- **Eliminar un evento**
    - **URL:** /api/eventos/<int:pk>/
    - **Método:** DELETE
    - **Descripción:** Elimina un evento existente. Solo los usuarios administradores pueden realizar esta acción.

3. Obras

- **Listar todas las obras**

    - **URL:** /api/obras/
    - **Método:** GET
    - **Descripción:** Devuelve una lista de todas las obras.
    - **Filtros:** id, titulo, id_escultor, id_evento.
    - **Buscar:** titulo, material, descripción.
- **Crear una nueva obra**
    - **URL:** /api/obras/
    - **Método:** POST
    - **Descripción:** Crea una nueva obra. Solo los usuarios administradores pueden realizar esta acción.
- **Actualizar una obra**
    - **URL:** /api/obras/<int:pk>/
    - **Método:** PUT
    - **Descripción:** Actualiza la información de una obra existente. Solo los usuarios administradores pueden realizar esta acción.
- **Eliminar una obra**
    - **URL:** /api/obras/<int:pk>/
    - **Método:** DELETE
    - **Descripción:** Elimina una obra existente. Solo los usuarios administradores pueden realizar esta acción.

4. Usuarios

- **Listar todos los usuarios**
    - **URL:** /api/usuarios/
    - **Método:** GET
    - **Descripción:** Devuelve una lista de todos los usuarios.
- **Crear un nuevo usuario**
    - **URL:** /api/usuarios/
    - **Método:** POST
    - **Descripción:** Crea un nuevo usuario.
- **Actualizar un usuario**
    - **URL:** /api/usuarios/<int:pk>/
    - **Método:** PUT
    - **Descripción:** Actualiza la información de un usuario existente.
- **Eliminar un usuario**
    - **URL:** /api/usuarios/<int:pk>/
    - **Método:** DELETE
    - **Descripción:** Elimina un usuario existente.

5. Escultores con sus obras

- **Listar escultores con sus obras**
    - **URL:** /api/escultores-obras/
    - **Método:** GET
    - **Descripción:** Devuelve una lista de todos los escultores con sus respectivas obras.

- **Filtro:** id_escultor (opcional) - Devuelve solo el escultor específico con sus respectivas obras.

6. Generar token para votación

- **Generar token**
  - **URL:** /generate-token/
  - **Método:** POST
  - **Descripción:** Genera un token para la votación.

7. Votar por una obra

- **Votar por una obra**
  - **URL:** /vote/<int:obra_id>/<str:token>/
  - **Método:** POST
  - **Descripción:** Permite votar por una obra específica utilizando un token.

8. Restablecer contraseña

- **Solicitar restablecimiento de contraseña**
  - **URL:** /password-reset/
  - **Método:** POST
  - **Descripción:** Solicita un restablecimiento de contraseña.
- **Confirmar restablecimiento de contraseña**
  - **URL:** /password-reset-confirm/
  - **Método:** POST
  - **Descripción:** Confirma el restablecimiento de contraseña.

9. Resultados de votaciones

- **Ver resultados de votaciones**
  - **URL:** /api/resultados/<int:evento_id>/
  - **Método:** GET
  - **Descripción:** Devuelve los resultados de votaciones para un evento específico.

10. Autenticación y registro

- **Iniciar sesión**
  - **URL:** /login
  - **Método:** POST
  - **Descripción:** Permite a los usuarios iniciar sesión.
- **Registrar un nuevo usuario**
  - **URL:** /register
  - **Método:** POST
  - **Descripción:** Permite a los usuarios registrarse.
- **Verificar email**

    - **URL:** /verify-email/<str:token>/
    - **Método:** GET
    - **Descripción:** Verifica el email del usuario utilizando un token.

11. Perfil de usuario

- **Ver perfil de usuario**
    - **URL:** /profile/
    - **Método:** GET
    - **Descripción:** Devuelve la información del perfil del usuario autenticado.
- **Actualizar perfil de usuario**
    - **URL:** /profile/
    - **Método:** PUT
    - **Descripción:** Actualiza la información del perfil del usuario autenticado.

12. Votaciones del usuario

- **Ver votaciones del usuario**
    - **URL:** /api/votaciones/
    - **Método:** GET
    - **Descripción:** Devuelve las votaciones realizadas por el usuario autenticado.

Documentación de la API

- **Swagger UI:** /
- **Redoc:** /redoc/

Notas

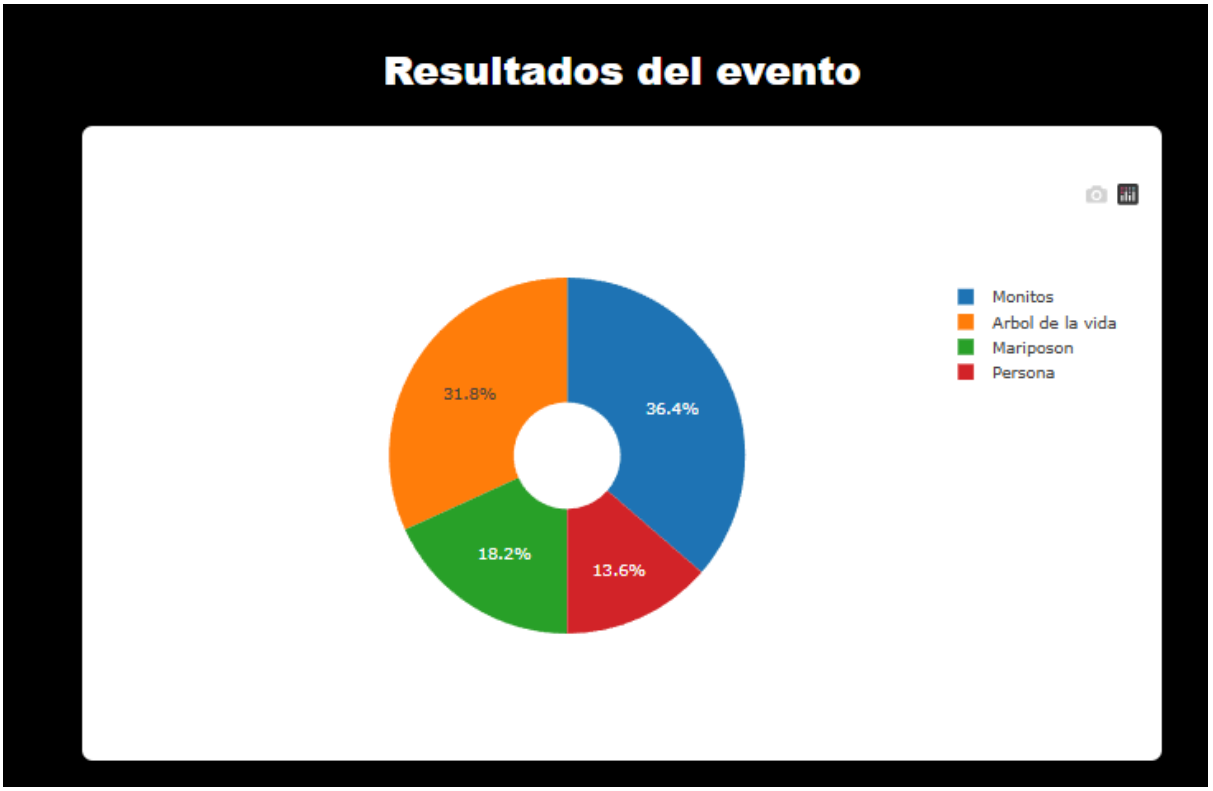
- **Autenticación:** La mayoría de los endpoints requieren autenticación mediante TokenAuthentication.
- **Permisos:** Algunos endpoints requieren permisos de administrador (IsAdminUser).

Extras

WebAssembly

Luego de la clase de Web Assembly, como grupo decidimos investigar del tema y tratar de implementar un WASM dentro del proyecto.
Indagando en el tema encontramos que python tiene un WASM algo adaptado. Básicamente es un WASM que ejecuta python pero con ayuda de Rust.
Decidimos que sería buena idea implementar gráficos con esta tecnología, para ello usamos Plotly, una librería matemática de python para generar gráficos y realizar otros tipos de cálculos.

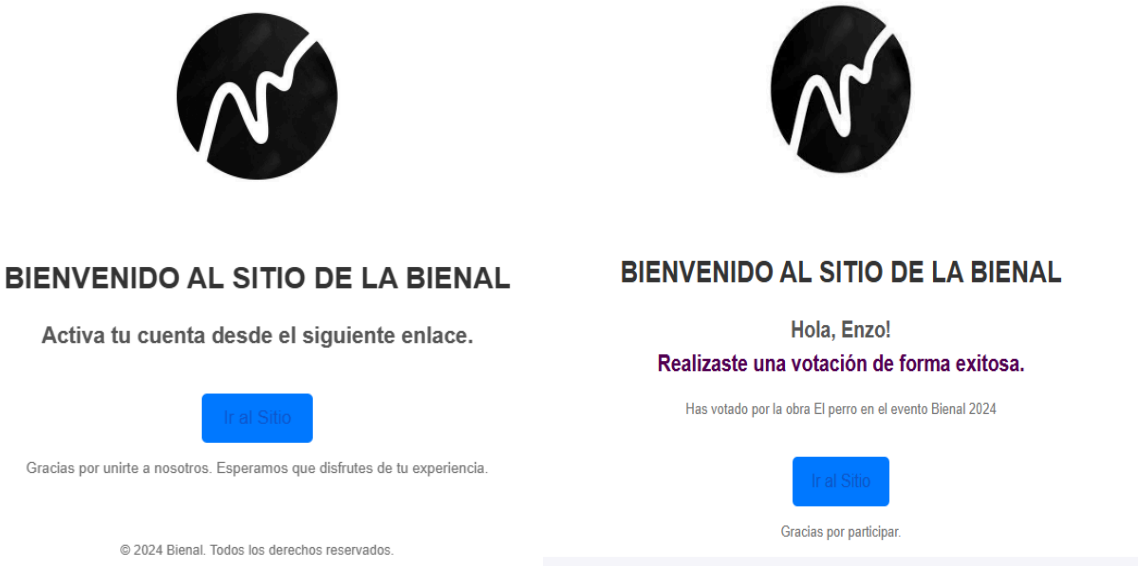
En el apartado de resultados de eventos fue donde lo incluimos, generando un gráfico de torta para ver la distribución de votos en cada obra por evento.



Este gráfico es typescript mezclado con la potencia de WASM de python plotly, para generar el gráfico, se realiza un fetch a la api para obtener los datos de un evento y se brinda a plotly esta información y características del gráfico, para que este se encargue de construirlo.

Envío de emails

Añadimos envíos de email para confirmar el registro, cambio de contraseña y además para notificar cuando realizamos una votación.



Implementación API Web Scrapping

Para nuestro proyecto, también decidimos agregar una funcionalidad extra dentro del Backend. Implementamos así una API aparte para extraer información de otras páginas relacionadas con el arte, como bienales de otros sitios, para así plasmar en nuestra página noticias acerca de las mismas. Ésta la encontramos en una sección aparte dentro del Home de nuestra página.

Para realizarlo, también se utilizó el Django Rest Framework de Python. Se crearon los modelos relacionados a la información a almacenar, en este caso las tablas de *Noticias, Subastas* y *Artistas*. La API cuenta con algoritmos que realizan Web Scrapping, una técnica que consiste en extraer código HTML de un sitio Web, renderizar, y manipularlo como una cadena de texto. De esta forma, recolectamos información como los títulos de las noticias, las descripciones, las imágenes vinculadas con cada noticia y la URL del sitio del cual proviene la noticia.

La información es almacenada dentro de una base de datos PostgreSQL desplegada en Render, al igual que la API. La información es servida al Frontend de nuestra página en formato JSON a través de la API. De esta forma, el único método que permite es el GET para obtener la información recolectada por los algoritmos de Scrapping. Cabe aclarar acá, que al ser el GET el único método permitido, no hace falta realizar controles de seguridad acerca de quién puede realizar un POST en la API por ejemplo, ya que esto se genera automáticamente con los algoritmos.

Por último, se intentó automatizar el proceso de Scrapping a los diferentes sitios Web, de forma tal de tener las noticias de último momento de todas éstas. Para ello se usó el Celery Beat de Django. Éste es un servicio de Django Celery que se encarga de las tareas

calendarizadas. Celery es una tecnología de Python que permite a las aplicaciones implementar colas de tareas para muchos trabajadores.

Modelos

1. News (title, description, image, url)
2. Auctions (title, author, price, image, url)
3. Artists (title, about, date, image, url)

API desplegada en Render

https://informacionexterna.onrender.com

Endpoints

1. /api/news
2. /api/auctions
3. /api/artists

Sitios visitados

1. News: https://www.infodesigners.eu/ - https://www.labiennale.org/en
2. Auctions: https://www.christies.com/en/browse?filterids=%7CCoaCategoryValues%7BAll%2Bother%2Bcategories%2Bof%2Bobjects%7D%7C&sortby=relevance
3. Artists: https://www.artnews.com/c/art-news/artists/

Frontend

Descripción

Este proyecto consiste en el desarrollo del frontend de la página web oficial de la **Bienal del Chaco**, diseñada para ofrecer una experiencia gráfica intuitiva y atractiva. Utilizamos el framework **Next.js** para crear una interfaz dinámica y optimizada, conectando el frontend con una API proporcionada por el equipo de backend (disponible en un repositorio independiente).

La plataforma está dirigida a todos los asistentes y seguidores de la Bienal que deseen participar activamente en la votación de esculturas, así como a los entusiastas del arte en general, quienes podrán explorar una sección de noticias relacionadas con eventos y actividades artísticas.

Además, el sistema incluye un **panel de administración** desarrollado para el personal de la organización, permitiéndoles gestionar y actualizar el contenido del sitio de manera ágil y eficiente.

Tecnologías utilizadas

Utilizamos Next.js como framework principal, el cual se basa en React para el desarrollo del frontend, permitiéndonos crear aplicaciones universales de manera eficiente. Para los estilos, empleamos Tailwind CSS, una librería que agiliza el diseño mediante el uso de clases utilitarias predefinidas.

Además, integramos varias otras herramientas para mejorar la experiencia de desarrollo. Por ejemplo, Shadcn nos proporciona una librería de componentes de React que facilita la creación de interfaces de usuario personalizadas. Utilizamos también la IA de Vercel v0 para generar componentes de forma automatizada a partir de prompts, lo que acelera el desarrollo y la personalización.

En cuanto al manejo de formularios y modales, nos apoyamos en diversas librerías de React, optimizando la interacción del usuario. Zod se utiliza para la validación de los campos en los formularios, garantizando que los datos ingresados sean correctos antes de su procesamiento.

Por último, para gestionar el estado global del usuario registrado, optamos por Zustand, que permite manejar el estado de manera sencilla y eficiente, asegurando que la experiencia del usuario sea fluida.

Estructura del proyecto

- **`src/`:**
  Contiene todo el código fuente de la aplicación. Es el directorio raíz donde se organiza el proyecto de manera modular y estructurada.
  - **`app/`:**
    Este directorio incluye las diferentes páginas de la aplicación y sus respectivas rutas. Cada carpeta representa una ruta específica y contiene componentes, lógica, y archivos relacionados con esa página.
    Ejemplo: Dentro de `app/` se podría tener rutas como `escultores/` y `admin/eventos/` con sus respectivas páginas y subcomponentes.
  - **`components/`:**
    Almacena todos los componentes reutilizables de la aplicación. Se organiza en subcarpetas según la funcionalidad o la sección a la que pertenecen.
    Ejemplo: Carpetas como `Escultores/` para los componentes de la sección de escultores.
  - **`services/`:**
    Contiene los servicios de la aplicación, encargados de interactuar con APIs o manejar lógica relacionada con el manejo de datos externos. Por

ejemplo, podrías tener funciones como `getSculptors()` para obtener escultores desde una API.

- **`store/`:**
  Este directorio se utiliza para gestionar el estado global de la aplicación.
- **`utils/`:**
  Incluye funciones utilitarias o helpers que se pueden usar en varias partes de la aplicación.
- **`styles/`:**
  Contiene los estilos globales de la aplicación. Configuraciones como archivo `globals.css` o temas personalizados.

- **`public/`:**
  Contiene archivos estáticos que se sirven directamente al navegador. Estos archivos son accesibles desde la raíz del dominio, como imágenes, íconos, y otros recursos estáticos.
  Ejemplo:
- **`favicon.ico`: Ícono del sitio web.**
- **`images/`: Carpeta para imágenes usadas en la aplicación.**

Instalación

Para instalar y configurar, sigue los siguientes pasos:

1. **Clonar el repositorio:**
   `git clone https://github.com/tu-usuario/tu-repositorio.git`

2. **Navegar al repositorio del proyecto:**
   `cd nombre-repositorio`

3. **Instala las dependencias:**
   `npm install`

Uso

Para ejecutar la aplicación en un entorno de desarrollo, utiliza el siguiente comando:

```
npm run dev
```

Contribución

Si deseas contribuir a este proyecto , por favor sigue los siguientes pasos:

1. **Hacer fork del repositorio (utilizando la interfaz gráfica de *github*).**
2. **Crear rama nueva para nueva funcionalidad:**
   `git checkout -b nombre-de-la-rama`

3. **Realizar cambios y hacer commit de ellos:**
   `git commit -m "descripción de los cambios"`
4. **Subir cambios:**
   `git push origin nombre-de-tu-rama`
5. **Abrir un *pull request* en el repositorio original (utilizando la interfaz gráfica de *github*)**

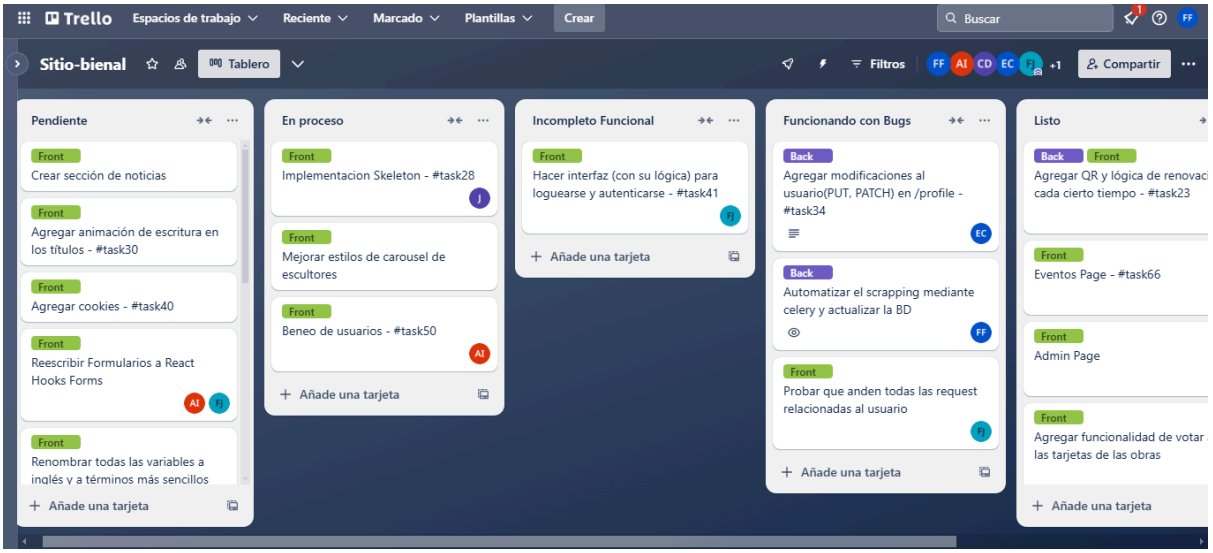
Anexo

Trello

Para la organización del trabajo en grupo, decidimos organizarnos mediante el sitio Web de Trello. Éste proporciona una herramienta tipo diagrama de Kanban, el cual indica las tareas pendientes, las tareas en proceso y las finalizadas, además de los responsables de realizar cada una. La idea fue hacer pequeñas releases del proyecto conforme iban pasando los meses y así avanzar juntos agregando funcionalidades de a poco.

Enlace al sitio: https://trello.com/b/PANRnOxN/sitio-bienal

Imagen ejemplo del tablero:



Links relevantes

Github backend:

https://github.com/EnzoCoschiza/tp-final-bienal

Github frontend:
https://github.com/fabriJerezz/sitio-bienal

API desplegada:
https://tp-final-bienal.onrender.com/

Web Scrapping desplegado:
https://informacionexterna.onrender.com/

Escenario planteado por la cátedra:
https://frre-ds.github.io/trabajos-practicos/tp-final/

Documentación framework Django Rest:
https://www.django-rest-framework.org/

Trello usado:
https://trello.com/b/PANRnOxN/sitio-bienal

Conclusión

El presente desarrollo ha permitido consolidar un sistema integral para la gestión de escultores, obras, eventos y votaciones, aplicando principios clave del diseño de software en el marco de la ingeniería en sistemas de información. A través de la utilización de Django como framework principal, combinado con herramientas modernas como Cloudinary para la gestión de medios, se logró implementar un conjunto de modelos robustos que cubren tanto los requerimientos funcionales como los no funcionales del proyecto.

La estructura de datos creada no solo asegura la integridad relacional mediante restricciones como claves foráneas y unicidad, sino que también incorpora funcionalidades avanzadas, como la eliminación y actualización eficiente de recursos en la nube. Esto garantiza una gestión óptima de los recursos multimedia, un aspecto esencial en sistemas orientados a eventos visuales.

Asimismo, los métodos personalizados integrados en los modelos permiten realizar cálculos y consultas dinámicas, como el estado de eventos y la generación de resultados de votaciones en tiempo real, mejorando la experiencia de usuario y facilitando la toma de decisiones informadas.

En conclusión, este proyecto refleja el impacto de las buenas prácticas en desarrollo de software, destacando la importancia de la modularidad, la escalabilidad y la integración de herramientas modernas. Además, demuestra cómo un enfoque bien planificado puede

transformar una idea en un sistema funcional, eficiente y listo para su implementación en escenarios reales.