Desarrollo de una Plataforma Web Desacoplada para la Gestión de Torneos Deportivos Universitarios

David Alexander Oliva Valdivia
Escuela Profesional de Ingeniería de Sistemas
Universidad Nacional de San Agustín
Arequipa, Perú
doliva@unsa.edu.pe
ORCID: 0009-0005-2479-6247

Edson Fabricio Subia Huaicane
Escuela Profesional de Ingeniería de Sistemas
Universidad Nacional de San Agustín
Arequipa, Perú
esubiahu@unsa.edu.pe
ORCID: 0009-0001-6836-5300

***Abstract*—La gestión manual de eventos deportivos a nivel universitario presenta desafíos significativos que resultan en una elevada carga administrativa y una alta incidencia de errores de datos. Este artículo detalla el diseño, desarrollo y despliegue de una plataforma web de arquitectura desacoplada para automatizar la administración de torneos de fútbol. El sistema integra un backend robusto desarrollado con Django REST Framework y desplegado en Vercel, un frontend reactivo construido con Angular y alojado en Netlify, y una base de datos PostgreSQL gestionada a través de Supabase. La plataforma soporta un flujo de trabajo completo, incluyendo el registro de usuarios con verificación por correo electrónico, inscripción de jugadores a equipos, y la generación automática de calendarios de competición (fixtures). Se discuten las decisiones arquitectónicas clave, como el uso de autenticación basada en JSON Web Tokens (JWT) y la implementación de una API RESTful, que garantizan la modularidad para futuras expansiones, seguridad mediante comunicación cifrada y una escalabilidad horizontal facilitada por la naturaleza serverless de la infraestructura.**

***Index Terms*—Arquitectura Desacoplada, Django REST Framework, Angular, Supabase, Vercel, Netlify, JSON Web Tokens, Gestión Deportiva, Aplicación Web.**

I. Introducción

La organización de torneos deportivos en el ámbito universitario es una tarea compleja que tradicionalmente ha dependido de procesos manuales, tales como hojas de cálculo, comunicación por correo electrónico y grupos de mensajería instantánea [1]. Estos métodos, si bien funcionales a pequeña escala, son propensos a errores de consistencia de datos, retrasos en la comunicación y una carga administrativa significativa para los organizadores. La falta de una fuente centralizada de verdad genera confusión entre los participantes y dificulta el seguimiento del progreso del torneo.

Aunque existen soluciones comerciales de gestión deportiva, estas suelen ser sistemas monolíticos, costosos o poco flexibles para las necesidades específicas del entorno universitario. Este trabajo se diferencia al proponer una solución a medida, basada en una arquitectura moderna y desacoplada, llenando un vacío en las herramientas disponibles para este nicho.

Para abordar esta problemática, se ha desarrollado una plataforma web integral denominada ”TorneoPro”. El objetivo principal de este proyecto es ofrecer una solución centralizada y automatizada que simplifique la gestión de torneos de fútbol, desde la inscripción inicial de equipos y jugadores hasta la visualización del calendario de partidos. La arquitectura del sistema se basa en un enfoque moderno de servicios desacoplados (headless), donde el backend y el frontend son entidades independientes que se comunican a través de una API RESTful [2].

La selección tecnológica fue un pilar fundamental del proyecto, priorizando herramientas modernas, escalables y con un ecosistema de desarrollo robusto:

- **Backend:** Se utilizó Django, un framework de alto nivel de Python, en conjunto con Django REST Framework (DRF) para la construcción de una API segura y eficiente. El despliegue se realizó en la plataforma *serverless* Vercel, que facilita la integración continua y la escalabilidad automática.
- **Frontend:** Se desarrolló una Single Page Application (SPA) con Angular, un framework de JavaScript mantenido por Google, que permite crear interfaces de usuario dinámicas y reactivas. El despliegue del frontend se gestionó a través de Netlify, optimizando la entrega de contenido a nivel global.
- **Base de Datos:** Se optó por Supabase como Backend as a Service (BaaS), que provee una base de datos PostgreSQL gestionada, simplificando la administración de la infraestructura de datos y garantizando su persistencia y seguridad.

El sistema implementa un flujo de autenticación seguro basado en JSON Web Tokens (JWT), con verificación de cuentas por correo electrónico para garantizar la validez de los usuarios. Además, una de las funcionalidades clave es la generación automática de fixtures mediante un algoritmo round-robin, cuyo resultado es accesible para los usuarios tanto en la plataforma web como en formato PDF descargable.

Este artículo presenta en detalle la arquitectura del sistema, el modelo de datos, la implementación de los endpoints de la API, y el proceso de despliegue en un entorno de producción. Finalmente, se exponen las conclusiones obtenidas y se delinean futuras líneas de trabajo para expandir las capacidades de la plataforma.

Este artículo se estructura de la siguiente manera: en la Sección II se describe la arquitectura general del sistema y

los componentes utilizados; la Sección III detalla el proceso de implementación, incluyendo modelado de datos y endpoints relevantes; en la Sección IV se discute el despliegue y las pruebas realizadas; finalmente, la Sección V presenta las conclusiones y propuestas de mejora a futuro.

II. Arquitectura del Sistema

La plataforma fue diseñada siguiendo un patrón de arquitectura de microservicios desacoplados, donde cada componente principal tiene una responsabilidad única y se comunica a través de interfaces bien definidas. Esta estructura, ilustrada en la Fig. 1, ofrece ventajas significativas en términos de mantenibilidad, escalabilidad y despliegue independiente de los componentes [3]. A continuación, se justifica la elección de cada componente tecnológico.

Fig. 1. Diagrama de la arquitectura desacoplada del sistema.

A. Componente de Backend (Django en Vercel)

El núcleo de la lógica de negocio reside en el backend. Se seleccionó **Django** por su filosofía "Batteries Included", que proporciona de serie un ORM robusto, un panel de administración y un sistema de seguridad probado, acelerando significativamente el desarrollo del núcleo de la aplicación. El despliegue en **Vercel** se eligió sobre alternativas como Heroku o AWS EC2 por su integración nativa con Git para CI/CD y su modelo de coste basado en el uso (pay-as-you-go), ideal para proyectos con tráfico variable como un torneo universitario. Las responsabilidades del backend incluyen:

- **Gestión de Datos:** Interacción con la base de datos PostgreSQL en Supabase a través del ORM de Django para realizar operaciones CRUD (Crear, Leer, Actualizar, Borrar) sobre los modelos de 'Facultad', 'Equipo', 'Jugador' y 'Fixture'.
- **Autenticación y Autorización:** Se implementó un sistema de autenticación basado en JWT con la librería 'djangorestframework-simplejwt'. Se crearon vistas personalizadas para permitir el registro de usuarios, la verificación de cuentas por correo electrónico y el login mediante 'email' en lugar de 'username'. Los permisos se gestionan de forma dinámica para diferenciar entre usuarios normales y administradores ('IsAuthenticated' vs. 'IsAdminUser').
- **Lógica de Negocio:** Se implementaron acciones específicas como la generación automática de fixtures, una operación compleja que se ejecuta a través de un endpoint protegido.

B. Componente de Frontend (Angular en Netlify)

El frontend es una Single Page Application (SPA) responsable de la interacción con el usuario. La elección de **Angular** se basó en su naturaleza como un framework completo y opinionado, que incluye un potente sistema de enrutamiento, inyección de dependencias y un cliente HTTP integrado. Esto contrasta con librerías como React, que requerirían la integración de múltiples paquetes de terceros para lograr la misma funcionalidad. Sus características principales son:

- **Reactividad:** Construido con Angular + Tailwind, el estado de la interfaz se actualiza dinámicamente en respuesta a las acciones del usuario y los datos recibidos de la API.
- **Manejo de Estado Centralizado:** Se utilizó un 'AuthService' con observables de RxJS ('BehaviorSubject') para gestionar el estado de autenticación del usuario de forma global, permitiendo que la interfaz reaccione en tiempo real a eventos como el login o logout.
- **Comunicación con la API:** Todas las peticiones al backend se manejan a través del 'HttpClient' de Angular y son interceptadas por un 'HttpInterceptor' que adjunta automáticamente el token JWT a las cabeceras, centralizando la lógica de autenticación de las peticiones.

C. Capa de Persistencia (Supabase)

Frente a la opción de gestionar una instancia de PostgreSQL manualmente, **Supabase** fue seleccionado por abstraer la complejidad del mantenimiento de la base de datos, incluyendo backups automáticos y escalado. La conexión se realiza a través de una URL estándar, desacoplando la lógica de la aplicación de la infraestructura de la base de datos y permitiendo un cambio de proveedor a futuro si fuese necesario [4].

III. Implementación

La fase de implementación se centró en la construcción de los componentes de backend y frontend, asegurando una comunicación fluida y segura entre ambos. A continuación, se detallan los aspectos más relevantes.

A. Modelo de Datos en Django

El diseño de la base de datos fue el punto de partida, definiendo las entidades principales del dominio del problema. Se utilizaron los modelos de Django para crear el esquema en la base de datos PostgreSQL de Supabase. La relación clave es el `OneToOneField` entre el modelo de `User` de Django y el modelo `Player`, lo que impone la regla de negocio de un solo jugador por cuenta de usuario.

```python
# tournament/models.py

from django.db import models
from django.conf import settings

class Team(models.Model):
    name = models.CharField(max_length=100)
    faculty = models.ForeignKey(Faculty, ...)

class Player(models.Model):
    user = models.OneToOneField(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        null=True, blank=True
    )
```

```
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()
    team = models.ForeignKey(Team, ...)
```

Listing 1. Definición simplificada de los modelos Player y Team.

El campo `user` se configuró como opcional a nivel de base de datos (`null=True`) para manejar datos heredados, pero su creación se fuerza a nivel de la API, garantizando que todo nuevo jugador esté asociado a un usuario.

B. API RESTful con Django REST Framework

Se desarrollaron *endpoints* para cada entidad del modelo de datos utilizando `ModelViewSet` de DRF, lo que simplificó la creación de operaciones CRUD. Los permisos se asignaron de forma dinámica para proteger las acciones sensibles.

1) Endpoint de Registro de Jugadores: La vista `PlayerViewSet` fue personalizada para controlar la lógica de creación. Se sobrescribió el método `perform_create` para asociar automáticamente al nuevo jugador con el usuario autenticado y para verificar que dicho usuario no tuviera ya un jugador registrado.

```python
# tournament/views.py

class PlayerViewSet(viewsets.ModelViewSet):
    # ...
    def get_permissions(self):
        if self.action == 'create':
            # Permite a cualquier usuario logueado crear.
            permission_classes = [IsAuthenticated]
        else:
            # El resto de acciones son para administradores.
            permission_classes = [IsAdminUser]
        return [p() for p in permission_classes]

    def perform_create(self, serializer):
        if Player.objects.filter(user=self.request.user).exists():
            raise ValidationError('Ya tienes un jugador registrado.')
        serializer.save(user=self.request.user)
```

Listing 2. Lógica de permisos y creación en PlayerViewSet.

2) Endpoint de Login Personalizado: Para permitir la autenticación mediante correo electrónico, se implementó una vista de login personalizada, `LoginView`, que hereda de `APIView`. Esta vista utiliza un *backend* de autenticación a medida que comprueba las credenciales contra el campo `email` del modelo `User`. Si la autenticación es exitosa, se generan manualmente los tokens JWT utilizando la librería 'simple-jwt'.

```python
# tournament/views.py

class LoginView(APIView):
    def post(self, request, *args, **kwargs):
        email = request.data.get('email')
        password = request.data.get('password')

        user = authenticate(username=email, password=password)

        if user is None:
            return Response(
                {'error': 'Credenciales invalidas.'},
                status=status.HTTP_401_UNAUTHORIZED
            )

        refresh = RefreshToken.for_user(user)
        data = {
            'refresh': str(refresh),
            'access': str(refresh.access_token),
        }
        return Response(data, status=status.HTTP_200_OK)
```

Listing 3. Vista de Login personalizada para autenticación con email.

C. Gestión de Estado en Angular

La reactividad de la interfaz de usuario se logró mediante un manejo de estado centralizado en el servicio de autenticación, `AuthService`. Se utilizó un `BehaviorSubject` de la librería RxJS para emitir el estado de autenticación actual ('true' o 'false').

```typescript
// src/app/services/auth.service.ts

export class AuthService {
  private authStatus = new BehaviorSubject<boolean>(false);
  public isAuthenticated$ = this.authStatus.asObservable();

  login(credentials: any): Observable<any> {
    return this.http.post(...).pipe(
      tap((tokens: any) => {
        this.setTokens(tokens.access, tokens.refresh);
        this.authStatus.next(true); // Emite el nuevo estado
      })
    );
  }
  // ...
}
```

Listing 4. Gestion de estado reactivo en AuthService.

Los componentes, como la cabecera principal, se suscriben a este observable (`isAuthenticated`) para actualizar dinámicamente la interfaz, mostrando, por ejemplo, los botones de "Login" o "Logout" según corresponda, sin necesidad de recargar la página.

IV. Despliegue y Pruebas

El proceso de despliegue se diseñó para ser lo más automatizado posible, utilizando un flujo de trabajo de Integración Continua y Despliegue Continuo (CI/CD) proporcionado por Vercel y Netlify.

A. Despliegue del Backend en Vercel

El backend de Django fue configurado para su despliegue en Vercel. Se creó un archivo `vercel.json` en la raíz del proyecto para definir la configuración de construcción y las rutas. Todas las variables de entorno sensibles, como la `SECRET_KEY` de Django, la URL de la base de datos y las credenciales de correo electrónico, se gestionaron de forma segura a través del panel de Vercel.

Un desafío clave fue la gestión de las dependencias de Python, que se resolvió manteniendo un archivo `requirements.txt` actualizado. La configuración de `ALLOWED_HOSTS` y `CORS_ALLOWED_ORIGINS` se realizó de forma dinámica leyendo las variables de entorno, permitiendo que el mismo código base funcionara tanto en desarrollo local como en producción.

B. Despliegue del Frontend en Netlify

El frontend de Angular se desplegó en Netlify, conectándolo directamente al repositorio de GitHub. Los ajustes de construcción en Netlify se configuraron de la siguiente manera:

- **Directorio base:** `Frontend/torneo-app/`
- **Comando de construcción:** `ng build`
- **Directorio de publicación:** `Frontend/torneo-app/dist/torneo-app`

Un paso crucial para el correcto funcionamiento de la SPA fue la creación de un archivo `_redirects` en el directorio `src` del proyecto Angular. Este archivo contiene una regla de reescritura (‘/* /index.html 200‘) que dirige todo el tráfico al `index.html` principal, permitiendo que el router de Angular gestione las rutas del lado del cliente [6].

C. Pruebas de Integración

Una vez desplegados ambos componentes, se realizaron pruebas de integración de extremo a extremo para verificar el flujo completo de la aplicación. Las pruebas se centraron en:

1) El ciclo de registro de usuario y verificación por correo electrónico.
2) El proceso de login y la correcta emisión y uso de tokens JWT.
3) La inscripción de un jugador por un usuario autenticado y la validación de la regla de ”un jugador por usuario”.
4) La generación y descarga de fixtures en PDF por parte de usuarios con los permisos adecuados.

Durante esta fase, se identificaron y solucionaron problemas de configuración de CORS y de sincronización de variables de entorno, como la `SECRET_KEY`, entre el entorno de desarrollo y el de producción.

V. Recomendaciones

Basado en la experiencia del desarrollo, se extraen las siguientes recomendaciones para proyectos futuros con arquitecturas similares:

- **Definición de Contratos de API (API First):** Se recomienda definir rigurosamente los endpoints, formatos de petición/respuesta y códigos de estado de la API antes de iniciar el desarrollo del frontend. Herramientas como OpenAPI/Swagger pueden formalizar este contrato, minimizando la fricción y permitiendo un desarrollo paralelo más eficiente.
- **Gestión Centralizada de Estado:** Para aplicaciones que escalen más allá de la simple autenticación, se debe considerar la adopción de una librería de gestión de estado más robusta como NgRx o Akita en Angular. Esto previene la dispersión de la lógica de estado a través de múltiples servicios y componentes.
- **Estrategia de Carga Perezosa (Lazy Loading):** Implementar lazy loading para los módulos de Angular desde el inicio del proyecto. Esto mejora drásticamente el tiempo de carga inicial de la aplicación, un factor clave en la retención de usuarios, al descargar el código de cada sección solo cuando es necesario.

VI. Conclusiones y Trabajo Futuro

El proyecto ha demostrado con éxito la viabilidad de construir una plataforma de gestión deportiva robusta, escalable y moderna utilizando una arquitectura de servicios desacoplados. La separación del backend y el frontend permitió un desarrollo paralelo y simplificó el proceso de despliegue, aprovechando las fortalezas de plataformas especializadas como Vercel y Netlify.

La automatización de tareas clave, como la generación de fixtures y la inscripción de jugadores, representa un avance significativo sobre los métodos de gestión manuales, ofreciendo un valor tangible tanto para los organizadores como para los participantes del torneo. El uso de Supabase como BaaS agilizó considerablemente el desarrollo al abstraer la complejidad de la administración de la base de datos.

Como trabajo futuro, se proponen las siguientes mejoras para expandir las capacidades de la plataforma:

- **Gestión de Resultados en Vivo:** Implementar endpoints y una interfaz para que los administradores puedan registrar los resultados de los partidos en tiempo real.
- **Tabla de Posiciones Dinámica:** Crear una nueva sección que calcule y muestre automáticamente la tabla de posiciones del torneo basándose en los resultados de los partidos.
- **Perfiles de Usuario y Jugador:** Desarrollar perfiles más detallados donde los usuarios puedan ver sus estadísticas o editar la información de su jugador.
- **Sistema de Notificaciones:** Integrar notificaciones (vía email o en la aplicación) para recordar a los equipos sobre sus próximos partidos.
- **Implementación de Refresco de Tokens:** Añadir lógica de refresco automático de tokens JWT en el frontend para mejorar la experiencia de usuario en sesiones largas, como se describe en la especificación RFC 7519.

En conclusión, la plataforma ”TorneoPro” sienta una base sólida que no solo resuelve la problemática inicial, sino que también ofrece un camino claro para su evolución hacia una solución de gestión deportiva aún más completa.

References

[1] J. Doe and J. Smith, “Challenges in manual event management in educational institutions,” *Journal of University Administration*, vol. 15, no. 2, pp. 45–52, 2022.

[2] R. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[3] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O’Reilly Media, 2015.

[4] The Twelve-Factor App, “III. Config,” 2012. [Online]. Available: https://12factor.net/config
[5] Django REST framework, “Authentication,” [Online]. Available: https://www.django-rest-framework.org/api-guide/authentication/
[6] Angular, “Understanding components,” [Online]. Available: https://angular.dev/guide/components
[7] Vercel, “Serverless Functions,” [Online]. Available: https://vercel.com/docs/functions