CulturaViva

Miguel Granel - 798913

Juan Antonio Almodóvar - 839953

Alejandro Benedí - 843826

Elizabeth Lilai Naranjo - 840091

CulturaViva funciona como una red social para eventos culturales locales en Zaragoza, lo cual fomenta la participación en actividades culturales. Esta aplicación permite a los usuarios explorar lugares culturales y eventos en un mapa interactivo, acceder a su información detallada, interactuar con otros asistentes y dejar valoraciones y comentarios sobre ellos.

URLs de acceso

API (Swagger)

https://culturaviva-backend.onrender.com/api-docs/

Frontend

https://culturaviva-frontend.onrender.com/

Local

El .env necesario de ambos repositorios se ha enviado por correo electrónico a jfabra@unizar.es

Credenciales de acceso

Usuario común

- Correo: user@culturaviva.invalid
- Contraseña: 3-kUO0O6PT4E

Administrador

- Correo: admin@culturaviva.invalid

- Contraseña: noT-NtS0L184

Módulos empleados en back-end

El módulo más importante que se ha utilizado para desarrollar el BackEnd es Express, que es un Framework completo para la creación de un servidor y el manejo de rutas.

Dependencias principales

- **express:** Framework para Node.js que facilita la creación de servidores y manejo de rutas.
- **mongoose:** Object-Document Mapper para MongoDB que permite modelar datos con esquemas definidos.
- **jsonwebtoken:** Generación y validación de tokens JWT para autenticación basada en tokens.
- **passport:** Middleware de autenticación flexible con soporte para múltiples estrategias.
- **express-async-errors:** Permite manejar errores en funciones async sin bloques try-catch.
- **axios:** Cliente HTTP para realizar peticiones a otros servicios desde el backend.
- **node-cron:** Permite programar tareas recurrentes en intervalos definidos.
- **nodemailer:** Envío de correos electrónicos desde el servidor.
- **socket.io:** Comunicación en tiempo real (websockets), ideal para chats o notificaciones.
- **swagger-jsdoc:** Generación de documentación Swagger a partir de comentarios en el código.
- **swagger-ui-express:** Visualización interactiva de documentación Swagger en la web.
- **ajv:** Validación de esquemas de la API.

Dependencias secundarias

- **passport-google-oauth20:** Estrategia de Passport para autenticación con Google.

- **passport-github2:** Estrategia de Passport para autenticación con GitHub.
- **passport-oauth2:** Estrategia base de OAuth2 para Passport.
- **passport-jwt:** Estrategia de Passport para autenticación basada en JWT.
- **passport-facebook:** Estrategia de Passport para autenticación con Facebook. No se llegó a terminar de implementar por política de Meta.
- **@superfaceai/passport-twitter-oauth2:** Estrategia de Passport para autenticación con Twitter. No se llegó a terminar de implementar por ser una librería de comunidad con escasa documentación oficial.
- **dotenv:** Carga variables de entorno desde un archivo .env, útil para configuraciones sensibles.
- **bcrypt:** Encriptación de contraseñas para mayor seguridad.
- **crypto:** Módulo de Node.js para funciones criptográficas básicas.
- **ajv-formats:** Soporte adicional de formatos para ajv.
- **sanitize-html:** Sanitiza contenido HTML para evitar ataques XSS. Utilizado para limpiar etiquetas HTML del contenido de la API de Zaragoza.

Dependencias de desarrollo / debug

- **jest:** Framework de pruebas para test unitarios y de integración.
- **supertest:** Librería para testear endpoints HTTP de forma automatizada junto con Jest.
- **prettier:** Herramienta de formateo de código para mantener un estilo consistente.
- **nodemon:** Reinicia automáticamente el servidor al detectar cambios en el código fuente durante el desarrollo.
- **winston:** Logger personalizable para registrar eventos del sistema.
- **winston-daily-rotate-file:** Extensión de Winston que permite rotar archivos de log por día.
- **cors:** Habilita el intercambio de recursos entre distintos orígenes (CORS).

Tecnología del frontend

El frontend se ha desarrollado con React + Vite en typescript.

Módulos empleados

- **react-router:** Definición de rutas y navegación dentro de la app.
- **react-query:** Caching y sincronización asíncrona.
- **react-query-auth:** Flujos de auth con query.
- **react-error-boundary:** Captura de errores en la UI y despliegue de componentes de respaldo (fallback).
- **axios:** Llamadas a la API.
- **socket.io-client:** Cliente para WebSockets con reconexión y eventos.
- **jwt-decode:** Decodificación de JWT para manejar sesiones.
- **Bootstrap:** Framework CSS + componentes React.
- **sweetalert2:** Ventanas emergentes estilo moderno, integradas con React.
- **fortawesome:** Iconos SVG de Font Awesome como componentes React.
- **chart.js:** Gráficas.
- **date-fns:** Utilidades para manipular fechas.
- **react-datepicker:** Componente de selección de fecha.
- **react-leaflet:** Mapa interactivo.
- **leaflet-color-markers:** markers de colores para el mapa interactivo.
- **react-leaflet-markercluster:** Marcadores agrupables por áreas del mapa.
- **react-swipeable:** Detección de gestos de deslizamiento (swipe) en móviles y touchpads.
- **react-hook-form:** Manejo eficiente de formularios.
- **zod:** Integración de validación basada en esquemas.

Validación y pruebas realizadas

Se han realizado pruebas de todos los models y controllers en el back-end utilizando la librería 'jest'. Las pruebas son exhaustivas y prueban tanto los casos donde todo va bien, como los casos de error. Si se ejecuta el comando 'npm test --coverage', se genera en la raíz del proyecto una carpeta coverage con el reporte de todos los tests, midiendo la cobertura de tests.

En cuanto a la API, se han realizado pruebas manuales de los endpoints utilizando Swagger. Se han probado de forma exhaustiva los casos de éxito y de error.

Adicionalmente, se ha realizado un pequeño test end2end, que consiste en una colección de postman que prueba un flujo de un usuario ordinario. No se han desarrollado más por falta de tiempo, pero la API ha sido probada manualmente de forma exhaustiva. El fichero de colección de Postman y el entorno deben cargarse en Postman y ejecutar el test.

Respecto al frontend, no se han creado pruebas automáticas pero se ha probado todo exhaustivamente de forma manual.

Mejoras implementadas

1. **DevOps (Utilización de herramientas de CI en el desarrollo)**

   En el back end se ha realizado una pipeline de GitHub Actions (.github/workflows/CI.yml) que hace que, tras cada push a la rama principal del repositorio (main) se ejecuten los tests. Si falla alguno de ellos, fallará el pipeline.

   Además, tanto en back end como en front end, si se despliega un tag (git tag vX.Y.Z, git push origin vX.Y.Z), el proyecto se despliega automáticamente en render. Esto, para el caso del back end, sucede únicamente si pasan todos los tests con éxito (etapa anterior del pipeline).

   Adicionalmente, se han incluido badges en algunos proyectos. Para esto se ha integrado la organización con SonarQubeCloud. Hay un badge que es de GitHub y que mide si el pipeline CI.yaml está en estado 'passed' o si ha fallado. Se han incluido badges interesantes proporcionados por SonarQubeCloud, como el 'quality gate', 'bugs', 'code smells', 'duplicated lines (%)', 'lines of code', 'reliability', 'technical debt', 'maintainability' o 'vulnerabilities'.

2. **Utilización de analizadores de código (SonarQubeCloud)**

   Ambos proyectos se han integrado con SonarQubeCloud. Sus recomendaciones se han tenido en cuenta a lo largo del desarrollo, aunque al final no nos dio tiempo de

arreglar muchas de ellas. Se incluyen badges en el Readme generados por SonarQubeCloud, como ya se ha comentado en la subsección anterior.

3. **Login con al menos dos sistemas externos**

   Se permite login social con Google y con GitHub, además del tradicional usuario-contraseña. Se intentó realizar login social con Facebook y, de hecho, está implementado (aunque se eliminaron los endpoints), pero para que funcione hay que mandar papeleo a Facebook por lo que se descartó la opción. Se intentó realizar login social con Twitter/X, pero la librería utilizada debe estar deprecated y no la conseguimos hacer funcionar.

4. **Cobertura de tests**

   En el back end, se ha conseguido un test coverage de entre el 70% y el 80% — dependiendo la métrica que se mire — en tests unitarios. Esto se puede corroborar ejecutando "npm test —coverage" y mirando el informe generado en la carpeta /coverage en la raíz del proyecto.

Adicionalmente, se comenzó a implementar el despliegue en AWS utilizando un balanceador de carga y un grupo de auto-escalado. Los manifiestos Terraform necesarios se encuentran en el repositorio https://github.com/CulturaViva-Unizar/CulturaViva-Deployment

No se llegó a hacer que terminase de funcionar y el despliegue automático con GitHub Actions no se terminó de implementar, pero está en un estado bastante avanzado y se retomará en el futuro por satisfacción personal y aprendizaje de los integrantes del equipo.

Valoración global del proyecto

En líneas generales, CulturaViva ha cumplido con sus objetivos principales: ofrecer una plataforma social para difundir y dinamizar la oferta cultural local de Zaragoza. El equipo ha logrado:

- **Desarrollo full-stack coherente**: Se levantó un backend REST bien organizado con Express, MongoDB y autenticación JWT, y un frontend moderno con React+Vite en TypeScript, integrando mapas Leaflet, gráficos, chats y foros y flujos de autenticación social.

- **Funcionalidades clave implementadas**:

  - Mapa interactivo con clusters y marcadores de distintos colores.

    - Visualización de eventos con detalles, valoraciones y comentarios.
    - Chat/notificaciones en tiempo real vía Socket.IO.
    - Login tradicional y social (Google, GitHub).
- **Calidad de código y DevOps**:
    - Pipeline de CI en GitHub Actions ejecutando tests de backend y desplegando automáticamente en Render.
    - Badges de cobertura y calidad (SonarQubeCloud) integrados en los README.
    - Uso de linters, formateadores (Prettier) y manejo de logs con Winston.
- **Pruebas y validación**:
    - Backend cubierto con Jest y Supertest, tanto en rutas como en lógica de negocio.
    - API documentada y probada manualmente con Swagger.

Sin embargo, la especificación de endpoints y contratos entre frontend y backend podría haberse cerrado antes de empezar el desarrollo, para evitar retrabajos y malentendidos. Pero no se pudo debido a compromisos con otras asignaturas. Además, fijar entregables intermedios claros (por ejemplo, “endpoint X listo para tal fecha”) habría ayudado a mantener el ritmo y anticipar bloqueos.

Mejoras propuestas

- **TypeScript en el backend:** Adoptar TypeScript en Express/Mongoose para asegurar tipado estricto en modelos, controladores y rutas, lo que facilita el refactor y disminuye errores en tiempo de ejecución.
- **Testing completo en frontend:** Configurar Jest junto con React Testing Library para tests unitarios de componentes y hooks e integrar coverage thresholds en CI para garantizar un mínimo de cobertura.
- **Pruebas E2E robustas:** Emplear Cypress (o Playwright) para flujos críticos: registro/login, creación de evento, comentarios y chat. Incluir en pipeline de CI para validar regresiones antes de cada despliegue.

- **Infraestructura en AWS:** Contenerizar backend y frontend con Docker, desplegar en AWS ECS/EKS o Lambdas, usando RDS (MongoDB Atlas o DocumentDB) y S3 para assets y configurar CloudFront como CDN y Route 53 para gestión de dominios.

- **Notificaciones:** Configurar notificaciones visibles al recibir nuevos mensajes o si te responden a un comentario/review.

- **API-First y documentación colaborativa:** Diseñar la API con OpenAPI/Swagger desde el inicio, compartiendo el spec en un repo y usar herramientas como Stoplight o Redocly para revisión por el equipo antes de codificar.

- **Mejoras en observabilidad y calidad continua:** Completar las recomendaciones de SonarQube (bugs, code smells, vulnerabilidades), añadir monitoring (Prometheus/Grafana) y alertas en producción e incorporar linting y formateo automático en pre-commits con ESLint y Prettier.

- **Mejoras DevOps:** Ejecutar pipelines automáticos de auditorías de búsqueda de vulnerabilidades (SAST y SCA) o linting.

- **Metodología ágil y gestión de proyecto:** Emplear sprints de 1–2 semanas, con revisiones regulares y retrospectivas y establecer historias de usuario claras y priorizadas en un backlog (Trello, Jira o GitHub Projects).

- **Accesibilidad y rendimiento:** Auditar con Lighthouse y mejorar Core Web Vitals y asegurar accesibilidad WCAG en componentes críticos (mapa, formularios, alertas).