Zenith Outdoor/Indoor Experience

May 2025 – April 2027 (24M)

Project 2024-1-CY01-KA220-ADU-000256958

D4.1 System Architecture Design and Integration Requirements

Authors	VIR
Abstract	The present deliverable provides an overview of the planned activities, methodologies, and expected outcomes associated with the system architecture design and integration requirements for the ZOE project (Activity A4.1, M1-M12). Specifically, this document details the optimal approach to interconnect the platform's data between the outdoor Augmented Reality (AR) environment (WP3) and the indoor 3D tour environment (WP4). It outlines the implementation of a scalable, microservices-based cloud infrastructure, including the utilization of a database for rapid geo-powered data retrieval, a headless Content Management System (CMS) for secure authentication and role-based access control, and a WebSockets subsystem for real-time interactivity.

THE CONSORTIUM

No.	Abbreviation	Partner Name	Logo
1	**GEO**	Geoimaging ltd	
2	**AUTH**	Aristotle University of Thessaloniki	
3	**TUW**	Technische Universität Wien	
4	**VIR**	Virtual Solutions	

DISCLAIMER

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Education and Culture Executive Agency (EACEA). Neither the European Union nor EACEA can be held responsible for them.

Co-funded by the European Union

Contents

Contents

Contents

1. INTRODUCTION 5
2. ARCHITECTURE REQUIREMENTS 5
2.1. Functional Requirements 5
2.2. Non-Functional Requirements 7
2.2.1. Performance 7
2.2.2. Scalability 7
2.2.3. Availability and Reliability 8
2.2.4. Usability 8
2.2.5. Security 9
2.2.6. Maintainability 9
2.2.7. Interoperability 10
2.2.8. Portability 10
2.2.9. Data Integrity 10
3. ARCHITECTURE OVERVIEW 11
4. APPLICATION INFRASTRUCTURE 14
4.1. Boot Sequence 14
4.2. Router & Layout Mapping 14
4.3. Pinia Global State (useStore) 15
5. DUAL-MODE AUTHENTICATION FLOW 16
5.1. Registered Users (Admin/Editor Access) 16
5.2. Free Users (Viewer-Only Access) 16
6. END-TO-END USER & CREATOR FLOWS 16
6.1. Experience Exploration Catalog (/experiences) 16
6.2. Base Asset Configuration (/base_exp) 16
6.3. Authoring Canvas Studio (/creator/:id) 17
6.4. Immersive 3D Runtime Viewer (/exp_viewer/:id) 17
6.5. Leaderboard Architecture (/leaderboard) 18
7. API ENDPOINT INVENTORY 18
7.1. Auth Endpoints 18
7.2. Experience Operations 19
7.3. Base Resource Configurations 19

7.4. Save Progress Metrics 19
7.5. Media & Storage Assets 19
8. DATA MODELS (JSON SCHEMAS) 20
8.1. Experience Schema 20
8.2. Point of Interest (POI) Schema 21
8.3. Action Sequence Step Schema 23
8.4. Gamification Level Schema 24
9. COORDINATION & SECURITY RULES 25
9.1. Geo-Spatial Calculations 25
9.2. System Vulnerabilities & Data Safeguards 25
9.3. Deployment Metrics 25
10. CONCLUSIONS 26

GLOSSARY

GDPR	General Data Protection Regulation
AR	Augmented Reality
API	Application Programming Interface
CMS	Content Management System
RBAC	Role-Based Access Control
UI	User Interface
POI	Point of Interest
WP	Work Package
FR	Functional Requirements

1. INTRODUCTION

This deliverable provides an extensive overview of the system architecture design, integration requirements, and dynamic gameplay mechanics developed over Months 1 to 12 (Activity A4.1). It details the microservices-based cloud infrastructure, a database engineered for rapid geo-powered data retrieval, and a headless Content Management System (CMS) that coordinates secure, dual-mode role-based access control to ensure a scalable and secure deployment lifecycle

2. ARCHITECTURE REQUIREMENTS

This chapter formalises the functional and non-functional requirements that drive the architectural decisions of Project Zoe. Each requirement is derived from the system's purpose as an immersive 3D experience platform and is grounded in the actual implementation described in the preceding chapters.

2.1. Functional Requirements

The functional requirements define what the system must do. They are organised by domain area and written using the "shall" convention common in requirements engineering.

FR-01 — User Authentication. The system shall support two authentication modes. Registered Strapi users authenticate with an email address and password and receive a JSON Web Token (JWT) for subsequent API access. Free users authenticate with a numeric PIN and a password, without requiring a JWT. This dual-mode approach allows both administrative users and casual visitors to access the platform through a single login interface.

FR-02 — User Registration. The system shall allow anonymous visitors to create a free-user account by providing a name and a password. Upon registration, the backend shall auto-generate a unique numeric PIN that serves as the user's identifier for future logins. The user is presented with this PIN and prompted to copy it before being redirected to the login page.

FR-03 — Session Persistence. The system shall optionally persist authentication state across browser sessions. When the "Remember me" option is enabled, credentials (JWT tokens for registered users, or PIN and password for free users) are stored in the browser's local storage. On application startup, the system shall automatically attempt to re-authenticate using the stored credentials without requiring manual login.

FR-04 — Token Refresh. The system shall transparently refresh expired JWT tokens without requiring the user to re-authenticate. When an API call returns a 401 (Unauthorised) response, the system shall automatically request a new access token using the stored refresh token, retry the original request, and update the stored tokens — all without any visible interruption to the user.

FR-05 — Experience Browsing. Authenticated users shall be able to browse, search, and filter the list of available experiences. Admin users shall see all experiences (including

unpublished drafts), while free users shall see only those that have been explicitly published. The experience list shall support text-based search and category filtering (all, mine, published, unpublished).

FR-06 — Base Experience Management. Admin users shall be able to create and edit base experiences, which represent the foundational 3D scenes upon which interactive experiences are built. This includes uploading 3D model files (Gaussian Splats, GLTF/GLB, or LAS point clouds), setting GPS coordinates (latitude and longitude), adjusting positional offsets, parsing Google Maps URLs for coordinate extraction, and capturing thumbnail snapshots directly from the 3D canvas.

FR-07 — Experience Authoring. Admin users shall be able to create and edit interactive experiences through a full 3D authoring environment. This includes placing Points of Interest (POIs) in the 3D scene via mouse-click raycasting, repositioning them with drag handles, configuring their visual appearance (type, shape, colour, opacity), attaching media files (images, videos, audio), defining action sequences composed of 12 distinct step types, and assigning proximity-based triggers that fire when the user's camera enters or exits configurable distance thresholds.

FR-08 — Level and Gamification Design. Admin users shall be able to define multiple game levels within each experience. Each level has a configurable goal type — collecting a target number of POIs, reaching a score threshold, or finding a specific POI. Levels also include instructional text, a target amount or score, and a designated start position within the 3D scene.

FR-09 — Immersive 3D Viewing. End users shall be able to explore experiences through a real-time 3D viewer rendered in the browser. The viewer shall support an orbit camera mode for free exploration, a first-person walk mode with keyboard (WASD/arrow keys) and touch joystick input for mobile devices, and WebXR augmented reality for compatible devices. The walk mode shall include terrain-following behaviour and wall-collision prevention.

FR-10 — Proximity-Based Interaction. The system shall continuously monitor the distance between the user's camera and each POI during 3D scene navigation. When the user enters or exits a POI's configured proximity zone (near or far threshold), the system shall automatically execute the associated action chain. State tracking shall prevent repeated triggering — an action fires only once per zone entry and resets when the user leaves the zone.

FR-11 — Player Progress Persistence. The system shall save and restore each player's progress per experience. This includes level completion status, accumulated scores, and the set of collected POIs. Progress is auto-saved using a debounced mechanism (2-second delay) to avoid excessive API calls during rapid interactions. The save flow checks whether a data entry already exists before creating or updating, ensuring no duplicate entries per player-experience pair.

FR-12 — Leaderboard. The system shall aggregate player scores across all experiences and present a ranked global leaderboard. Scores are computed by summing each level's score

from the player's saved progress data. The leaderboard deduplicates entries, groups them by player identifier, sorts by total score in descending order, and visually highlights the current player's row.

2.2. Non-Functional Requirements

The non-functional requirements define the quality attributes that constrain how the functional requirements are implemented. They are grouped into nine categories.

2.2.1. Performance

The application shall achieve first meaningful paint within three seconds on broadband connections. This is accomplished through Vite-based builds with manual chunk splitting and lazy-loaded route components via dynamic imports, ensuring that only the code needed for the current page is loaded initially.

Gaussian Splat 3D models of up to approximately 30 MB shall begin rendering within five seconds. The system uses streaming loaders that progressively render the scene as data arrives, rather than waiting for the entire file to download before displaying any content.

The 3D viewer shall maintain a minimum of 30 frames per second during scene navigation on mid-range consumer hardware. This is achieved through selective post-processing (only applying effects like outline highlighting when needed), billboard-mode POI meshes that reduce geometric complexity, and careful management of draw calls.

Proximity-based triggers shall fire within 500 milliseconds of the user crossing a distance threshold. Because distance calculations are performed on every animation frame (inside the requestAnimationFrame loop), the effective check rate matches the display refresh rate, ensuring near-instantaneous response.

Player progress auto-saving shall not block or degrade the user interface. Save operations are debounced with a two-second timeout and executed as asynchronous background requests, so the 3D scene remains fully interactive during data persistence.

2.2.2. Scalability

The system is designed as a fully static single-page application with no server-side rendering in production. All 3D rendering, game logic, proximity calculations, and UI interactions run entirely on the client side, placing no per-user computational load on the server. API calls are stateless REST requests authenticated with JWT tokens, which allows the Strapi backend to be horizontally scaled behind a load balancer as the user base grows.

Experiences, POIs, actions, and player data are stored as JSON documents in Strapi's database. Media files — including Gaussian Splats, videos, images, and thumbnails — are served as static assets either from the Strapi server's local /uploads/ directory or from external content delivery networks. This CDN-first media strategy ensures that content growth does not create bottlenecks at the API layer.

The production bundle uses a manual Rollup chunking strategy that splits vendor libraries into separate files. Notably, @babylonjs/core is isolated into its own chunk so that pages relying solely on Three.js are not penalised with unnecessary Babylon.js code. The @loaders.gl dependencies are pre-bundled via Vite's dependency optimisation to reduce cold-start times.

2.2.3. Availability and Reliability

The system provides offline resilience through local caching. Leaderboard data is cached in the browser's local storage, allowing the leaderboard page to display previously fetched results even if the API is temporarily unreachable. Player progress is similarly cached locally before being synchronised to the server.

Graceful degradation is ensured throughout the application. If a 3D model fails to load (due to network errors or corrupt files), the viewer displays an error state rather than crashing. Media load failures for images and videos fall back to placeholder visuals. The leaderboard endpoint is called with both GET and POST as a fallback strategy, accommodating varying backend configurations.

Session recovery is handled automatically on application startup. The system attempts to re-authenticate using stored credentials. If the JWT has expired, the refresh-token flow re-establishes the session transparently. If stored free-user credentials are no longer valid, the system performs a clean logout and redirects to the login page.

2.2.4. Usability

The user interface is fully responsive, adapting from mobile to desktop through Quasar Framework's responsive grid system. Layout switches — such as replacing a horizontal navigation bar with a hamburger menu — are triggered reactively based on the current viewport width.

Mobile 3D navigation is supported through a virtual joystick component that provides touch-based directional input. All POI interactions (clicking, proximity detection) support both touch events and traditional mouse input, ensuring a consistent experience across device types.

The login process is designed with low barriers to entry. Two distinct authentication modes are presented side by side via a visible toggle: PIN-based login for casual users and email/password login for registered users. This allows first-time visitors to begin exploring experiences without going through a full registration process.

Visual feedback is provided for all user actions. Toast notifications confirm successful operations (login, save, deletion) and report errors. In the 3D scene, selected POIs are highlighted using a post-processing outline effect. Loading states are communicated through spinner animations and progress overlays.

The vue-i18n internationalisation plugin is configured in the boot layer, establishing the infrastructure for future translation of all UI strings without requiring code changes.

2.2.5. Security

Authentication uses short-lived JWT tokens paired with a refresh-token mechanism that extends sessions without exposing long-lived credentials. Tokens are stored in the browser's local storage — a deliberate trade-off that accepts XSS vulnerability risk in exchange for the simpler implementation that a single-page application requires compared to cookie-based flows.

Authorisation is enforced at two levels. On the client side, Vue Router navigation guards prevent unauthenticated users from accessing protected routes. On the server side, Strapi's permission system and custom controller logic enforce data-level access control, ensuring that free users can only read published content while administrative operations require a valid JWT with the appropriate role.

All API communication occurs over HTTPS. The Axios instance is configured with credentials: include to support cookie-based refresh-token exchange with the backend.

Input validation is performed both on the client (e.g., password-match checks during registration) and on the server (Strapi's content-type model validators). The CORS policy on the Strapi backend restricts API access to known frontend origins, and the credentials: include flag requires explicit origin whitelisting — wildcard origins are not permitted.

Bot protection infrastructure is in place through the vue-recaptcha-v3 dependency, though it is currently disabled. The integration point is prepared in the boot layer for future activation.

2.2.6. Maintainability

The frontend codebase is organised into clearly separated directories that reflect distinct areas of concern: pages/ for route-level views, components/ for reusable UI elements, components/editor/ for editor-specific panels, components/js/ for 3D utility modules, stores/ for state management, boot/ for application initialisation, and utils/ for conversion utilities. This structure allows developers to locate and modify functionality without navigating an unstructured monolith.

All shared application state flows through a centralised Pinia store, preventing scattered local-storage access patterns and enabling predictable state debugging through Vue DevTools. API communication is similarly centralised — all HTTP requests pass through a single Axios instance defined in the boot layer, so changes to the base URL, default headers, or interceptor logic require only a single-point modification.

The Babylon.js and Three.js rendering code is encapsulated in dedicated JavaScript modules, isolating complex 3D logic (scene creation, physics, splat loading, POI rendering, proximity management) from the Vue component layer. This separation ensures that changes to 3D engine behaviour do not inadvertently affect UI component logic.

2.2.7. Interoperability

The system supports multiple 3D model formats out of the box. Gaussian Splats (.splat, .ply, .ksplat) are loaded via the Spark.js library for Three.js scenes and via Babylon.js's native GaussianSplattingMesh for the object previewer. GLTF and GLB models are loaded through Three.js's GLTFLoader with Draco compression support. Format detection is automatic based on the file extension.

Geographic coordinate system compatibility is ensured through Proj4.js with dynamic EPSG definition loading from the epsg.io service. This allows the system to convert between any coordinate reference system — not just WGS84 — making it suitable for projects in any geographic region.

Media assets can be served from the local Strapi uploads directory or from external CDNs, supporting hybrid hosting configurations without requiring code changes. The viewer's WebXR integration follows the WebXR Device API standard, enabling augmented-reality experiences on compatible devices such as Android Chrome and Quest browsers. All backend endpoints follow REST conventions with stateless, JSON-based communication, enabling integration with third-party clients, mobile applications, or automated pipelines.

2.2.8. Portability

The production build targets ES2022, Firefox 115, Chrome 115, and Safari 14, covering all modern desktop and mobile browsers. The application is compiled as a fully static SPA (output in dist/spa/), making it deployable to any static file server, CDN, or containerised environment. While the current deployment script uses rsync over SSH, it can be replaced with any CI/CD pipeline without affecting the build process.

2.2.9. Data Integrity

All experience data — including POIs, actions, levels, location coordinates, and offsets — is serialised as a single JSON blob within the extra_data field and saved atomically via a single API request. This prevents partial-update corruption that could occur if different parts of the experience were saved independently.

The leaderboard aggregation pipeline deduplicates raw player-data entries by their document identifier before grouping them by player identifier, preventing inflated scores that could result from duplicate database records. The player progress save flow enforces idempotency by first checking whether an entry already exists for the given player-experience pair, then either creating a new entry or updating the existing one — never producing duplicates.

Media file integrity is maintained through Strapi's upload provider, which generates unique hashed filenames (e.g., v2_68a4397790.jpeg) to prevent filename collisions. For image uploads, multiple size variants (thumbnail, small, medium, large) are automatically generated, ensuring efficient delivery across different display contexts.

3. ARCHITECTURE OVERVIEW

In order to deliver seamless web-based graphics, we chose to use a three-tier setup that directly connects a Vue 3/Quasar client interface and a Three.js 3D pipeline to a secure Strapi v5 headless backend. In order to streamline file management, we chose to host and serve all 3D models (.glb/.obj), media files, and thumbnails directly from the backend's local storage provider.

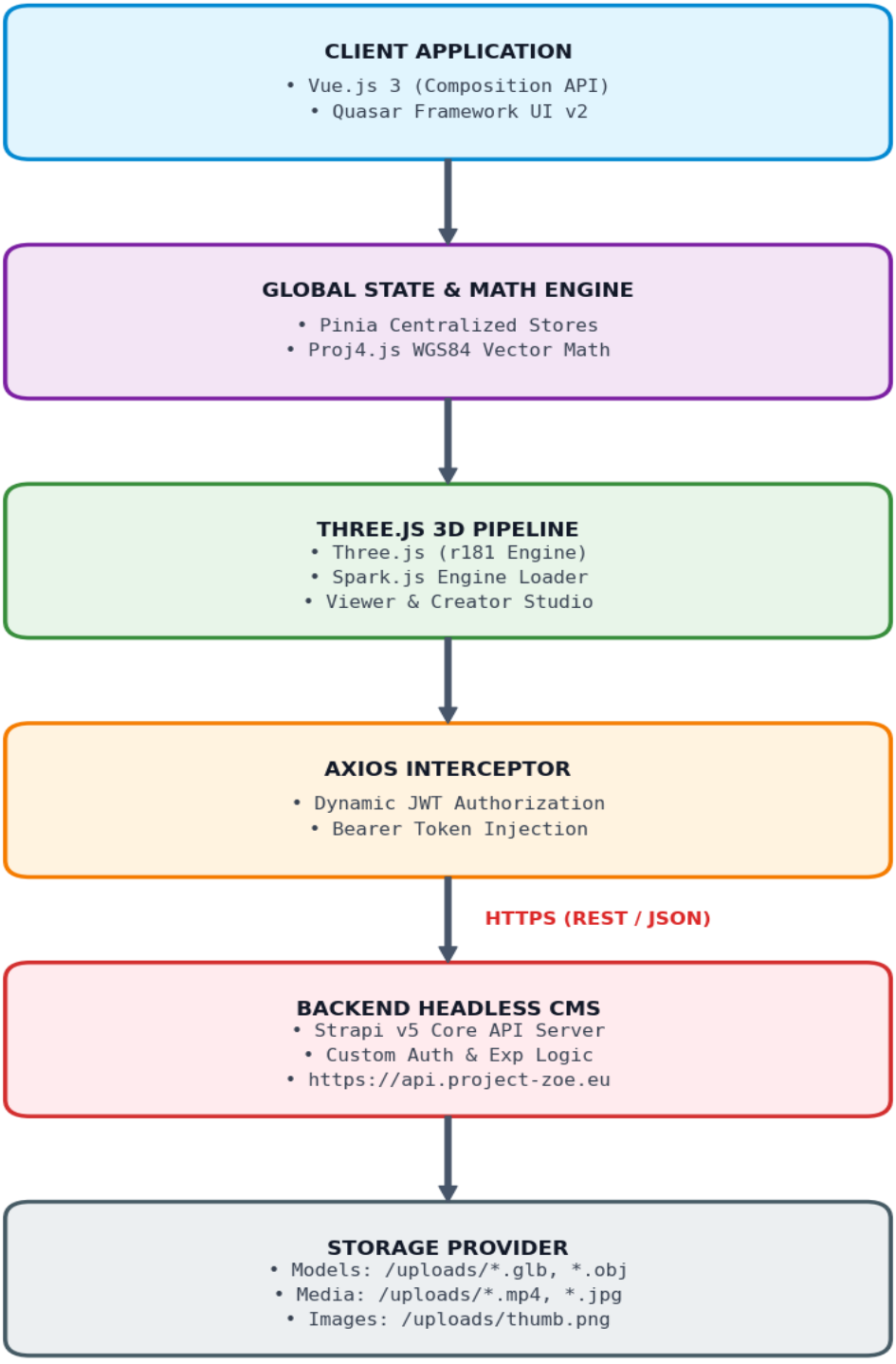


Figure 1: Architecture diagram of ZOE platform

Based on the architecture diagram provided in Figure 1, the image illustrates a **six-layer, top-down technical workflow** that manages data processing and system interactions within the ZOE platform:

- **Client Application:** The outermost frontend layer, built using Vue.js 3 (Composition API) and the Quasar Framework UI v2, which handles user interaction.
- **Global State & Math Engine:** The state management layer utilizing Pinia Centralized Stores and Proj4.js for handling WGS84 vector calculations.
- **Three.js 3D Pipeline:** The 3D rendering core driven by the Three.js (r181 Engine) and Spark.js Engine Loader to power the runtime Viewer and Creator Studio.
- **Axios Interceptor:** The communications security gateway that handles network traffic, performing dynamic JWT authentication and Bearer Token injection into API headers.
- **Backend Headless CMS:** The core server layer operating via a Strapi v5 Core API Server (hosted at api.project-zoe.eu), which processes custom authentication and experience logic.
- **Storage Provider:** The underlying database and file storage infrastructure where localized system assets—such as .glb/.obj 3D models, .mp4/.jpg media files, and thumb.png images—are directly hosted and retrieved.

The flow indicates that client-side interactions cascade down through the graphics and state engines, securely authenticate via HTTP requests (REST/JSON over HTTPS), and ultimately query or modify resources managed by the backend server and its local storage provider.

The users are distinguished into two different roles.

- **Administrators** upload geo-referenced 3D scans, place interactive Points of Interest (POIs) inside the scene, and build event-driven action sequences.
- **End Users** explore these environments in real-time via a web browser, engaging with proximity-triggered events, gamified levels, and a leaderboard.

The table below explains how the system's architecture is working.

Layer	Technology	Role / Implementation
Frontend UI	Vue 3 (<script setup>) + Quasar v2	Reactive interface and responsive layout utilities.
Viewer 3D Engine	Three.js (r181) + Spark.js	Real-time scene rendering, POI visualization, walk-mode.
Creator 3D Engine	Three.js (r181)	Full authoring environment with physics, gizmos, and tools.
State & HTTP	Pinia v3 + Axios	Centralized global state; API client equipped with JWT auto-refresh.
Backend CMS	Strapi v5 (Headless)	REST API, authentication, and media hosting via custom controllers.

Layer	Technology	Role / Implementation
Geo-Location Math	Proj4.js + WGS84 Ellipsoidal	Translates real-world GPS coordinates to local 3D space.
Splat Optimization	@playcanvas/splat-transform	Converts standard .splat files into optimized .sog formats.
Build & Deploy	Vite + rsync over SSH	Bundles the SPA for deployment to staging.project-zoe.eu.

4. APPLICATION INFRASTRUCTURE

In order to launch cleanly, we chose to use a streamlined Quasar boot process that initializes our HTTP client and our cross-component messaging layout right at startup. We chose to use HTML5 History Mode layouts to completely isolate full-screen 3D viewer pages from the standard admin management pages, while a global Pinia store manages state synchronization.

4.1. Boot Sequence

Quasar initializes core modules at launch in the following order (registered in quasar.config.js under the boot array):

1. axios.js: Establishes the core HTTP client with baseURL: https://api.project-zoe.eu/api. *Note: Authentication interceptors are attached dynamically later during login.*
2. EventBus.js: Provisions an Emittery instance for decoupled, cross-component messaging.

4.2. Router & Layout Mapping

The application relies on **Vue Router** in **HTML5 History Mode** split into two structural layouts:

Layout A: FlatLay (Immersive, Frameless Pages)

- / → IndexPage (Landing page)
- /login → LoginPage (Supports registered and free users)
- /registration → RegistrationPage (Generates viewer-only PIN profiles)
- /exp_viewer/:id → ExperienceViewer (The core interactive 3D scene)
- /base_experience_view/:id → BaseExperienceEditor (3D base asset configuration)
- /creator/:id → ExperienceCreationPage (Full 3D scene authoring toolkit)

Layout B: MainLay (Standard Navigation Layout)

- /experiences → ExperiencesPage (Searchable asset/experience catalog)

- /leaderboard → LeaderboardPage (Global user score rankings)
- /base_exp & /base_exp/:id → BaseExperience (Management lists for raw 3D models)

Navigation Security Guard:

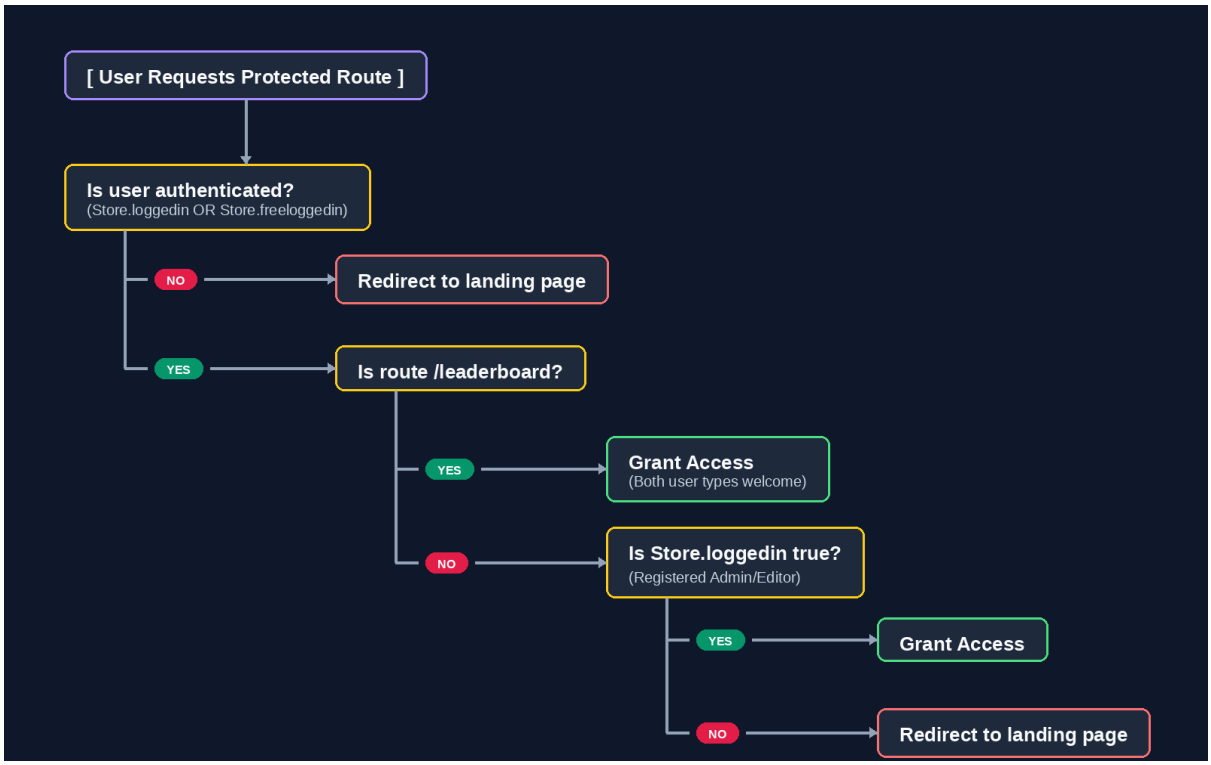


Figure 2: Navigation security guard map

To protect sensitive areas of the platform, we implemented a strict navigation guard that verifies each user's role and authentication status before allowing access. Users who do not possess the required administrator or viewer permissions are automatically redirected to the landing page, preventing unauthorized entry.

4.3. Pinia Global State (useStore)

- **Auth Flags**: loggedin (Admin/Editor status) | freeloggedin (PIN-based user status) | rememberMe (Persistence flag)
- **User Data**: profile (Tracks ID, name, PIN, role) | access_token (JWT string) | refresh_token (Renewal string)
- **Asset Lists**: experiences (Configured maps/games) | base_experiences (Raw 3D models)

Core Actions

- login(email, password): Requests /callback, mounts the Axios bearer interceptor, and downloads admin assets.
- freeUserlogin(credentials): Submits PIN & password to /get_my_data for viewer profile access.
- refreshAccessToken(): Uses refresh tokens to hit /token/refresh, seamlessly swapping expired JWTs.

- relogin(): Runs on app mount; reads LocalStorage tokens to quickly restore an interrupted session.
- logout(): Flushes memory stores and purges credentials from browser storage.

5. DUAL-MODE AUTHENTICATION FLOW

5.1. Registered Users (Admin/Editor Access)

1. **Request**: UI sends credentials to POST /api/callback.
2. **Response**: Strapi delivers a jwt and an automated refreshCookie.
3. **Setup**: The application caches tokens in LocalStorage and dynamically binds an **Axios interceptor** to inject Authorization: Bearer <jwt> into headers.
4. **Error Recovery**: If a request encounters a 401 Unauthorized response, the interceptor pauses execution, calls refreshAccessToken(), updates the key, and re-runs the original request seamlessly.

5.2. Free Users (Viewer-Only Access)

1. **Registration**: User posts a name and password to POST /api/freeuserregister. The backend auto-generates a unique access PIN and prints it on-screen.
2. **Login**: User inputs their PIN and password to POST /api/get_my_data.
3. **Session**: Stores profile data locally. No JWT is required; authorization privileges restrict their visibility to published maps via GET /api/get_published_exps.

6. END-TO-END USER & CREATOR FLOWS

6.1. Experience Exploration Catalog (/experiences)

Users land on a searchable card grid reading live from Store.experiences.

- **Admin Mode**: Retrieves all variations via GET /api/get_all_exp.
- **Standard Mode**: Retrieves visible variations via GET /api/get_published_exps.
- **Thumbnails**: Paths resolve dynamically via: https://api.project-zoe.eu + exp.extra_data.thumbnail.url.

6.2. Base Asset Configuration (/base_exp)

Before an experience can be customized, a raw 3D scan data source (Base Experience) must be processed:

- **Google Maps Parser**: Extracts standard latitude and longitude vectors instantly from raw pasted URLs.
- **Transform Offsets**: Modifies X, Y, Z and heading matrices to orient raw data correctly.

- **Thumbnail Snapshots**: Captures a viewport canvas image, uploading it on-the-fly via POST /api/upload.

6.3. Authoring Canvas Studio (/creator/:id)

The authoring workshop coordinates Three.js and a customized control panel sidebar.

1. **POI Creation**: Clicking the workspace fires a raycaster into the scene mesh to calculate 3D structural coordinates.
2. **Interactive Properties**: Designers adjust visibility, geometric indicator types (stars, boxes, nodes), and point-reward weights.
3. **Action Pipelines**: Utilizing a drag-and-drop sequencing panel, authors link reactive events (such as changing camera focuses or opening media players) to user triggers.

6.4. Immersive 3D Runtime Viewer (/exp_viewer/:id)

The engine instantiates structural components on page entry:

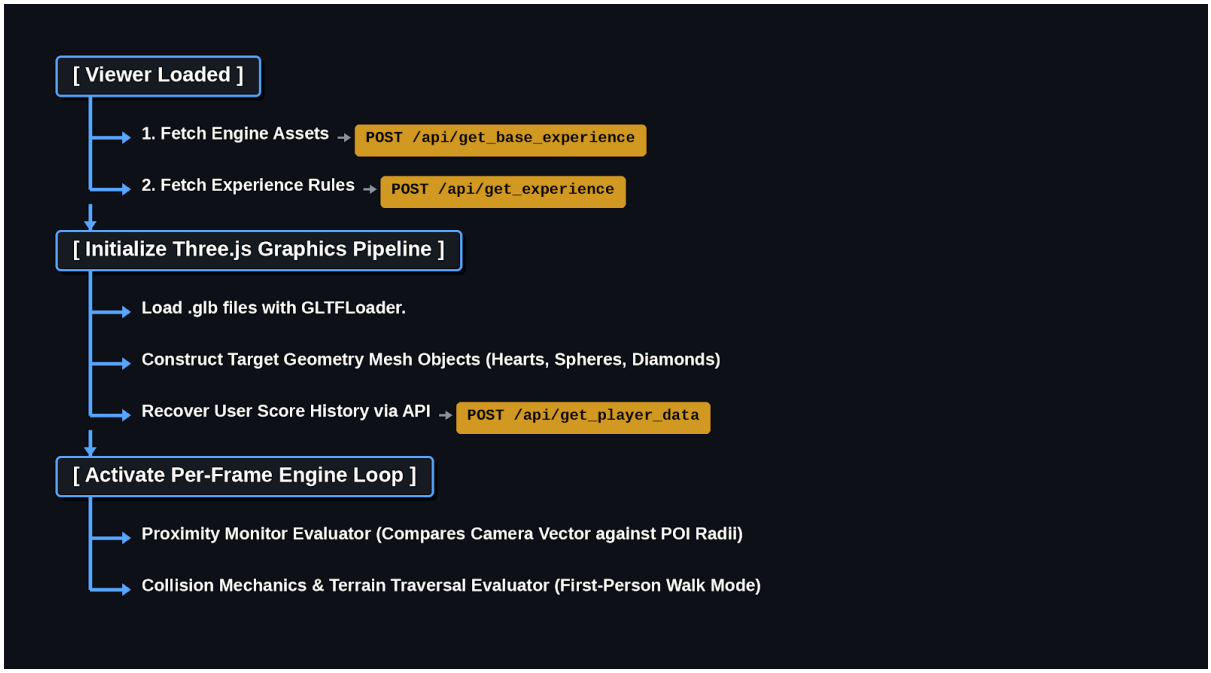


Figure 3: Engine instantiates structure

Camera Navigation Schemes

- **Orbit Mode**: Standard OrbitControls configuration for bird's-eye mouse dragging and rotation.
- **Walk Mode**: Custom first-person movement setup mapping WASD keys alongside mobile virtual joystick components. Features active wall-collision containment boundaries and ground-snapping height logic.
- **WebXR AR Mode**: Project layouts directly into immediate physical environments using device video tracking layouts and a virtual wrist overlay menu.

Proximity Engine Logic

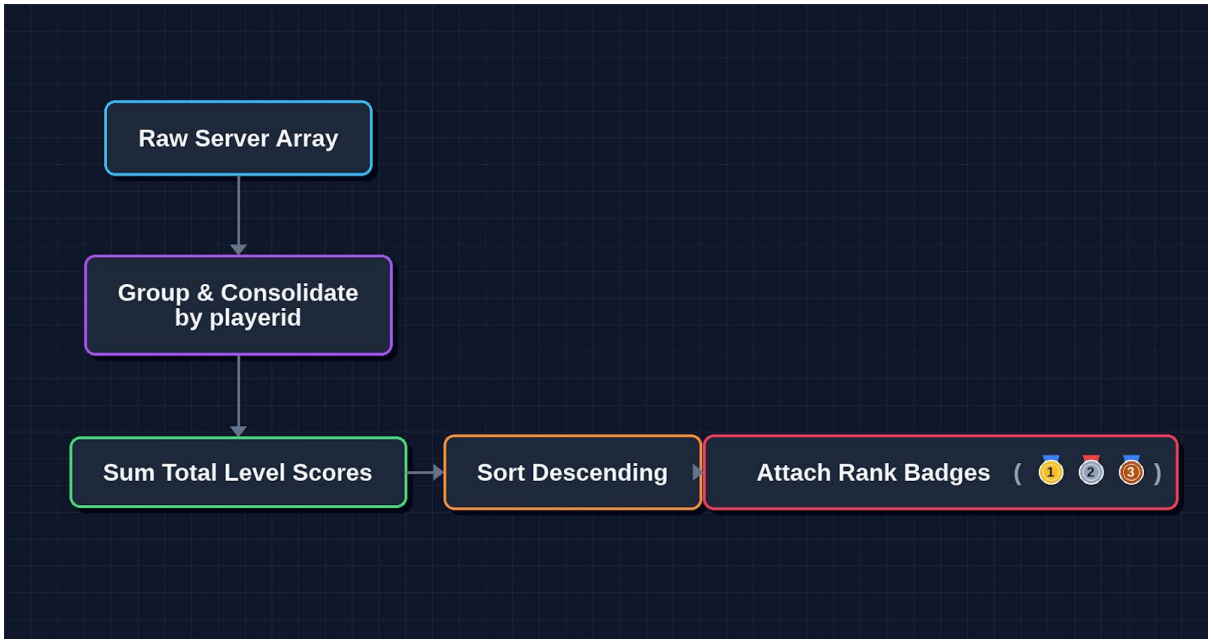
To prevent repeated firing cycles at zone boundaries, the engine utilizes a state-tracking loop updated every frame.

Gamification Loop

1. **Level Start**: Renders explicit instructional objectives (e.g., target scores or collection item counts).
2. **Collection Check**: Evaluates user positions dynamically against items. Collecting target items updates the layer score counter.
3. **Completion Event**: Reaching criteria fires a confetti layout, increments user profile points, and flushes progress data out via debounced (2-second) POST /api/updateentry updates.

6.5. Leaderboard Architecture (/leaderboard)

The server returns all historical entries via GET /api/get_players_data. The client transforms this information using the following sequence:



7. API ENDPOINT INVENTORY

7.1. Auth Endpoints

- POST /api/callback → Account log-in (Registered user validation).
- POST /api/get_my_data → PIN-based viewer profile entry authentication.
- POST /api/freeuserregister → Registers casual users and provisions authorization access numbers.
- POST /api/token/refresh → Swaps expired keys for fresh runtime tokens.

- GET /api/users/me?populate=role → Pulls profile definitions and security parameters.

7.2. Experience Operations

- GET /api/get_all_exp → Drops all configured instances (Editor access).
- GET /api/get_published_exps → Isolates standard publicly visible maps.
- POST /api/get_experience → Delivers configuration schemas based on an object ID.
- POST /api/new_exp → Commits new layout manifests to the server.
- POST /api/update_exp → Overwrites configuration profiles.
- POST /api/delete_exp → Removes targeted maps.
- GET /api/get_experience_data/:pin → Queries single maps via shared alpha-numeric codes.

7.3. Base Resource Configurations

- GET /api/get_all_base_exp → Catalog index of asset scans.
- POST /api/get_base_experience → Locates direct source links for targeted meshes.
- POST /api/update_base_exp → Updates coordinates, transformations, and asset details.

7.4. Save Progress Metrics

- POST /api/get_player_data → Downloads past save-states for an active map.
- POST /api/get_exp_data → Verifies whether a specific historical progress record exists.
- POST /api/createentry → Creates a new user log entry.
- POST /api/updateentry → Commits score updates.
- GET /api/get_players_data → Retransmits global data dumps for leaderboard processing.

7.5. Media & Storage Assets

- POST /api/upload → Processes multipart/form-data asset packages (Splat meshes, images, audio, video).
- Static paths scale across the following providers:
    - Local instances: https://api.project-zoe.eu/uploads/
    - Model storage CDNs: https://cdnalliance.virtualsolutions.gr/objects/
    - System thumbnail assets: https://cdn.alliance4xr.eu/

8. DATA MODELS (JSON SCHEMAS)

8.1. Experience Schema

To organize and manage a complete virtual tour experience, we use a master layout that stores key metadata such as GPS coordinates, orientation offsets, and walk-mode configurations. Serving as the central blueprint, it consolidates all related points of interest (POIs), action sequences, and game levels into a single, unified experience.


```
{
 "id": 17,
 "documentId": "gnazu2vzcmkxfripyqbqdm97",
 "title": "Vienna Walking Tour",
 "Published": true,
 "base_exp_id": 17,
 "extra_data": {
  "pois": [],
  "actions": [],
  "levels": [],
  "location": {
   "crs": "EPSG:4326",
   "x": 48.198255,
   "y": 16.37188,
   "cx": 16.37188,
   "cy": 48.198255
  },
```

```
  "offsets": {
    "x": 0, "y": 0, "z": 0,
    "heading": 0,
    "nudge_east": 0,
    "nudge_north": 0
  },
  "thumbnail": { "url": "/uploads/thumb.png", "formats": {} },
  "loading_position": {
    "position": { "x": 0, "y": 5, "z": 10 },
    "target": { "x": 0, "y": 0, "z": 0 }
  },
  "walk_mode": false,
  "base_doc_id": "gnazu2vzcmkxfripyqbqdm97",
  "createdBy": 2,
  "updatedBy": 2
 }
}
```


8.2. Point of Interest (POI) Schema

We chose to use this specific data structure in order to store everything related to an interactive 3D map marker, such as its shape style, visual color, and coordinate placement. It also assigns score point rewards and points directly to asset files like videos or audio clips that play when a user clicks or walks near the marker.


```
{
```

```
"id": 1770038430986,
"title": "Ancient Column",
"description": "A Roman-era column fragment",
"type": "video",
"visualType": "star",
"data": {
  "position": { "x": -54.24, "y": 216.81, "z": -39.37 },
   "lat": 48.19860,
   "lon": 16.37115,
   "epsg": "EPSG:4326"
},
"color": "#0000ff",
"ringColor": "blue",
"opacity": 0.3,
"isvisible": true,
"isClickable": true,
"score": 5,
"mediafile": { "url": "/uploads/video.mp4" },
"proximityNear": 9,
"proximityFar": 15,
"useProxNear": true,
```

```
"useProxFar": true,
"proximityNearAction": { "id": 123, "title": "Play video",
"steps": [] },
"proximityFarAction": { "id": 124, "title": "Pause video",
"steps": [] },
"onClickedAction": { "id": 125, "title": "Show info",
"steps": [] },
"useOnClicked": true
}
```

8.3. Action Sequence Step Schema

To create dynamic event timelines, we use a step-by-step workflow that sequences custom animations and events along a precise timeline. This system enables the platform to automatically trigger complex interactions—such as scaling a target POI or rotating the camera—as soon as a user enters a designated zone, delivering a seamless and responsive experience.

```
{
"id": 1770193897421,
"title": "Play video and award points",
"type": "near",
"steps": [
 {
  "typeId": 5,
  "typeLabel": "Play Video",
```

```
    "id": 1770117818075,
    "targetPoiId": 1770117759578,
    "waitForFinish": false,
    "focusPoi": false
  },
  {
    "typeId": 11,
    "typeLabel": "Score",
    "id": 1770193877661,
    "score": 20,
    "targetPois": { "label": "POI #3", "value": 1770117759578 }
  }
[
}
```


8.4. Gamification Level Schema

To create engaging, objective-based gameplay experiences, we use this game structure to define and manage individual levels within the 3D environment. It maintains mission objectives, gameplay rules, and scoring criteria, determining the conditions required for progression—such as achieving a target score or collecting a specified number of markers before advancing to the next level.


```
{
 "id": 1,
 "name": "Find the Temple",
 "goalType": "collect_count",
 "targetVisual": "star",
 "targetAmount": 5,
```

```
"targetScore": 100,
"instructions": "Collect all 5 star markers near the temple ruins",
"completed": false,
"startTime": null,
"endTime": null,
"score": 0,
"startPosition": {
  "x": -54.0, "y": 216.0, "z": -39.0,
  "lat": 48.19860, "lon": 16.37115
}
}
```


9. COORDINATION & SECURITY RULES

9.1. Geo-Spatial Calculations

The system targets coordinate translations via the standard **WGS84 Ellipsoid model** ($a=6,378,137.0$, $f=1/298.257223563$).

- Local configurations interpret coordinates along the following axes: ** X → East **, ** -Z → North **, and ** Y → Up **.
- A built-in detector auto-maps correct spatial definitions by pulling transformation structures from https://epsg.io/{code}.proj4 on demand.

9.2. System Vulnerabilities & Data Safeguards

- **Storage Vectors**: JSON Web Tokens and authentication profiles reside in browser storage components.
- **API Access Enforcements**: Strapi security permissions evaluate transactions on the backend. Client route configurations act as secondary structural gateways.
- **Cross-Origin Security**: Backend systems apply implicit CORS origin definitions, routing credentials using secure cookies (credentials: include).

9.3. Deployment Metrics

- Staging Interface environment: staging.project-zoe.eu
- Staging server target path: /var/www/staging.project-zoe.eu/html/
- Production compilation routine via Vite: quasar build
- App Interface environment: app.project-zoe.eu
- App server target path: /var/www/app.project-zoe.eu/html/

10. CONCLUSIONS

In conclusion, this document outlines the comprehensive System Architecture Design and Integration Requirements for the ZOE platform. By establishing a clear, three-tier framework connecting a Vue 3/Quasar client interface with a headless Strapi v5 backend, the architecture provides a highly scalable foundation for managing immersive digital environments. The integration of a database for rapid geo-powered data retrieval, advanced Three.js rendering pipelines, and automated WGS84 coordinate translations effectively bridges the platform's data between outdoor Augmented Reality (AR) environments and indoor 3D tour workspaces. Ultimately, these structured system specifications, JSON data models, and secure deployment guidelines fulfill the core objectives, ensuring an interconnected, performant, and secure lifecycle for the ZOE platform.