EVENTSPHERE

Backend Architecture & System Design Report

A Zero-Bloat, Native PHP 8 & PostgreSQL Ecosystem

Author: Daniele Compagnoni

Date: April 11, 2026

Copyright & License Statement

EventSphere: Backend Architecture & System Design Report
Course: Web Systems and Technologies
Master's Degree in Computer and Control Engineering
Sapienza University of Rome

Copyright © 2026 Daniele Compagnoni

MIT License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Abstract

This document outlines the architectural patterns, security posture, and system design of EventSphere, a bespoke event management platform. Developed for the "Web Systems and Technologies" course at Sapienza University of Rome, the project adheres to a strict "Zero-Bloat" engineering philosophy. By bypassing heavy third-party frameworks and relying entirely on native PHP 8 and a highly optimized PostgreSQL data layer, the system demonstrates high-performance request routing, a custom Role-Based Access Control (RBAC) implementation, and a specialized XML/XSLT document generation pipeline. This report serves as both the technical documentation and the architectural blueprint for the platform's future scalability.

Contents

List of Figures

Chapter 1

Executive Summary & Strategic Overview

1.1 Document Purpose

This technical specification outlines the core architectural decisions, design patterns, and operational mechanics of the EventSphere backend. It is explicitly designed to serve as a comprehensive guide for the engineering team and project stakeholders, facilitating a rapid and thorough understanding of the system's infrastructure. By formally defining the scope and system constraints, this document ensures that developers can maintain, secure, and scale the application effectively.

1.2 High-Level System Architecture

EventSphere operates on a robust, custom-built infrastructure tailored for performance and predictability. The backend is deployed on a Linux operating system across both development and production environments. At its core, the system relies on PHP 8.1+ as the runtime environment to take full advantage of modern strict typing features. Relational data management is handled entirely by a PostgreSQL database stack.

The overarching architectural paradigm is a Custom Model-View-Controller (MVC) structure. This ensures that domain logic, interface generation, and request orchestration remain strictly decoupled without relying on external framework abstractions.

1.3 Design Philosophy & Trade-offs

A critical architectural decision in the EventSphere system design was the implementation of a custom Front Controller pattern, deliberately bypassing heavy, off-the-shelf frameworks such as Laravel or Symfony. This first-principles approach is justified by several key technical requirements:

- **Maximized Request Lifecycle Control:** By funneling all HTTP traffic through a centralized brain (`public/index.php`), the architecture guarantees a standardized bootstrapping sequence, giving developers granular control over session initialization and routing.
- **Minimized Dependency Bloat:** Eschewing monolithic frameworks significantly reduces the vendor payload. This minimizes the application's attack surface and mitigates the risk of breaking changes originating from third-party dependency updates.
- **Strict Typing Enforcement:** Building a custom MVC allows the team to enforce strict PHP 8.1+ typing conventions natively across all Models and Controllers, preventing the type-coercion issues often introduced by generic, framework-level ORMs (Object-Relational Mappers).

While adopting a custom scaffolding necessitates a higher initial engineering investment to build routing and database abstraction layers, this trade-off yields a profoundly leaner, more transparent back-end tailored precisely to EventSphere's domain requirements.

Chapter 2

Architectural Patterns & Core Infrastructure

2.1 The Custom MVC Paradigm

At the heart of EventSphere lies a strict, custom-built Model-View-Controller (MVC) architecture driven by a "Zero-Bloat" product philosophy. In stark contrast to modern web development trends that rely heavily on abstracted, dependency-laden frameworks, EventSphere leverages a pure, dependency-free PHP 8.1+ engine.

This approach guarantees that domain logic (Models), interface generation (Views), and request orchestration (Controllers) remain strictly decoupled. By relying on native language features and strict typing rather than framework "magic," the application achieves sub-millisecond routing and a highly transparent execution flow. This architectural choice directly supports the platform's requirement for a transparent security posture and high-performance data processing.

2.2 The Front Controller Pattern

The system utilizes the Front Controller design pattern, establishing `public/index.php` as the absolute single entry point—the "brain"—for all incoming HTTP traffic. Server configurations strictly redirect every route request directly to this file.

This centralized orchestration allows for a highly standardized, predictable bootstrapping phase for every request lifecycle:

- **Environment Initialization:** Loading configuration variables via `.env`.
- **State Management:** Initializing secure user sessions and parsing cookies.
- **Global Dependencies:** Instantiating the core database connection.
- **Dispatching:** Passing the sanitized Request URI to the custom routing engine (`Router.php`) for controller invocation.

2.3 The Singleton Database Instance

To maintain relational integrity and optimal resource utilization, database connectivity to the PostgreSQL engine is managed through a Singleton pattern defined within `backend/config/Database.php`.

Instead of opening a new connection for every query or relying on a heavy ORM layer, the Singleton guarantees that only a single, persistent PHP Data Object (PDO) instance is active during a given request lifecycle.

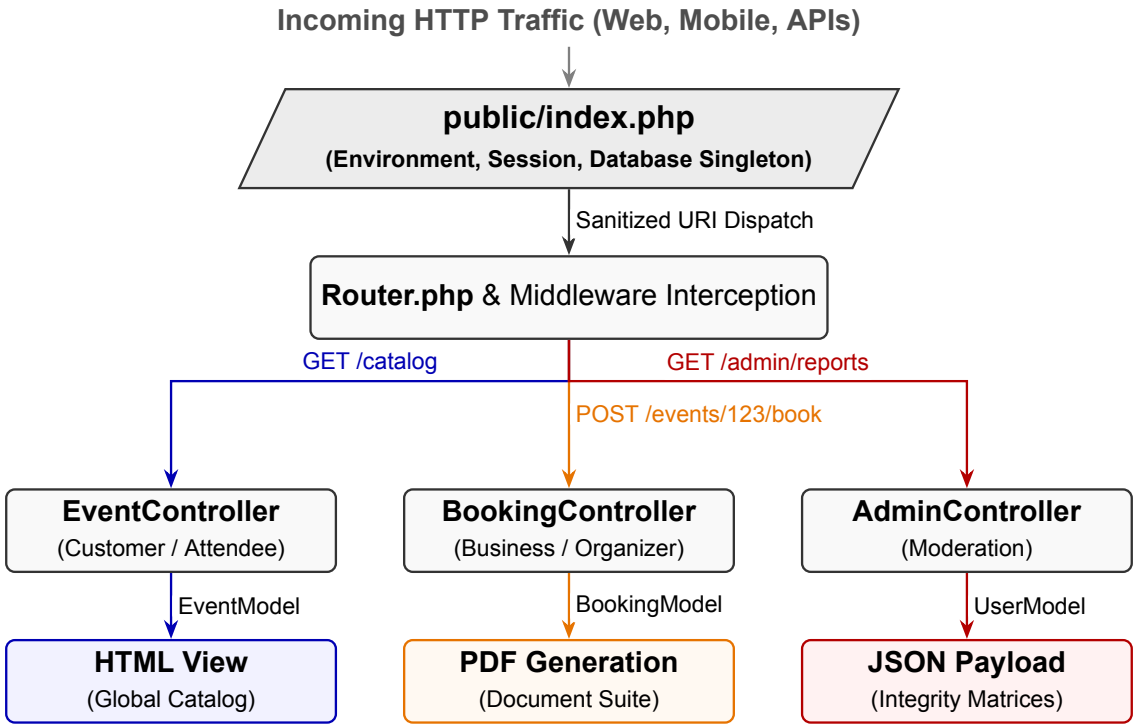

Figure 2.1: The Front Controller Funnel. All external traffic is choked through `index.php` before the Router dispatches requests down distinctly color-coded MVC pipelines based on RESTful URIs and Role-Based Access Control.

```php
<?php
class Database {
    private static $instance = null;
    private $connection;

    private function __construct() {
        // PDO instantiation with strict error modes
        // Credentials injected via environment variables
        $this->connection = new PDO($dsn, $user, $pass, $options);
    }

    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new Database();
        }
        return self::$instance;
    }

    public function getConnection() {
        return $this->connection;
    }
}
?>
```

Listing 1: Conceptual implementation of the EventSphere Database Singleton.

As shown in Listing 1, the private constructor prevents external instantiation. This approach prevents connection pool exhaustion under standard loads and serves as the foundational layer for the system's transaction integrity and prepared statement execution.

Chapter 3

The Dynamic View: Request Lifecycle & Routing

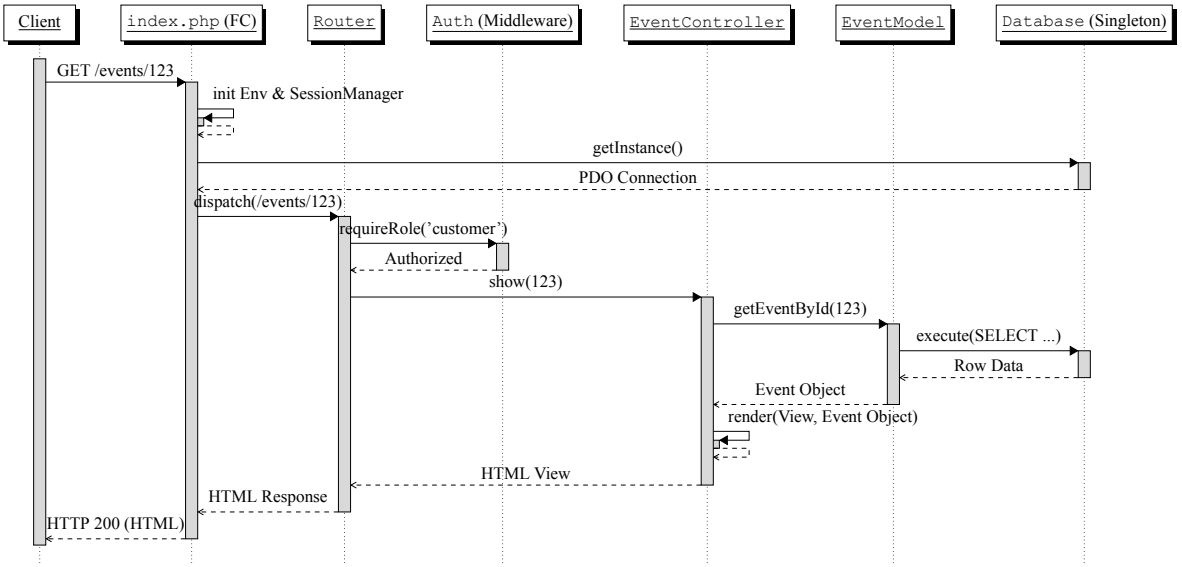

Figure 3.1: UML Sequence Diagram mapping a standard GET request lifecycle through the EventSphere Front Controller architecture.

3.1 Bootstrapping & Environment Setup

The request lifecycle begins the moment the server redirects traffic to `public/index.php`. Before any domain logic is executed, the application must hydrate its environment. This bootstrapping phase strictly involves parsing the `.env` file to load critical, environment-specific credentials into memory (e.g., PostgreSQL connection strings) without hardcoding them into the repository. Subsequently, the `SessionManager` initializes the user's session state, parsing any active cookies to establish contextual awareness for the incoming request.

3.2 The Custom Routing Engine

In adherence to the platform's "Zero-Bloat" philosophy, EventSphere implements a custom RESTful routing engine via `backend/core/Router.php`. By avoiding heavy external routing libraries, the system achieves sub-millisecond routing resolution.

The routing engine relies on the `routes.php` registry and performs two primary functions:

- **HTTP Verb Mapping:** Routes are explicitly bound to specific HTTP verbs (GET, POST, PUT, DELETE), ensuring strict adherence to RESTful principles.
- **Regex-based URI Parsing:** Dynamic URL segments (e.g., `/events/{id}`) are parsed using native PHP regular expressions. The engine extracts these dynamic parameters and automatically injects them into the designated Controller method.

```php
<?php
// backend/routers/routes.php

// Standard GET route for the Global Catalog
$router->get('/catalog', [EventController::class, 'index']);

// Dynamic POST route with middleware interception
$router->post('/events/{id}/book', [BookingController::class, 'store'])
       ->middleware('Auth::requireRole:customer');
?>
```

Listing 2: Routing registry demonstrating HTTP verb mapping, dynamic parameters, and middleware attachment.

3.3 Middleware & Pipeline Interception

A critical security feature of the dynamic view is pipeline interception. Before the `Router` invokes a target controller, it evaluates any middleware attached to the route definition.

As demonstrated in Listing 2, if a route requires specific privileges, the routing execution is paused. The system invokes static checks, such as `Auth::requireRole`, to verify the user's Role-Based Access Control (RBAC) matrix. If the precondition fails, the router aborts the pipeline and issues an immediate HTTP 403 Forbidden response, ensuring unauthorized actors never reach the Controller layer.

Chapter 4

The Static View: Domain Logic & Data Layer

4.1 OOP Database Abstraction

In a strict MVC architecture, Models are the exclusive gatekeepers of domain logic and data persistence. To prevent code duplication and ensure uniform security standards, EventSphere employs an Object-Oriented database abstraction layer centered around an abstract `BaseModel.php` class.

This foundational class standardizes core CRUD (Create, Read, Update, Delete) operations. Most importantly, it enforces the use of PHP Data Objects (PDO) Prepared Statements for every database interaction, rendering the application structurally immune to SQL-injection attacks.

```php
<?php
abstract class BaseModel {
    protected PDO $db;
    protected string $table;

    public function __construct() {
        $this->db = Database::getInstance()->getConnection();
    }

    protected function fetchAll(string $sql, array $params = []): array {
        $stmt = $this->db->prepare($sql);
        $stmt->execute($params);
        return $stmt->fetchAll(PDO::FETCH_ASSOC);
    }
}
?>
```

Listing 3: The `BaseModel` class enforcing prepared statements via the Database Singleton.

4.2 Core Domain Entities

The domain layer maps directly to a native PostgreSQL Relational Schema. By utilizing PostgreSQL's advanced typing, including `JSONB` support for dynamic event metadata, the system maintains strict relational integrity without the overhead of an external ORM.

The ecosystem is driven by three primary models:

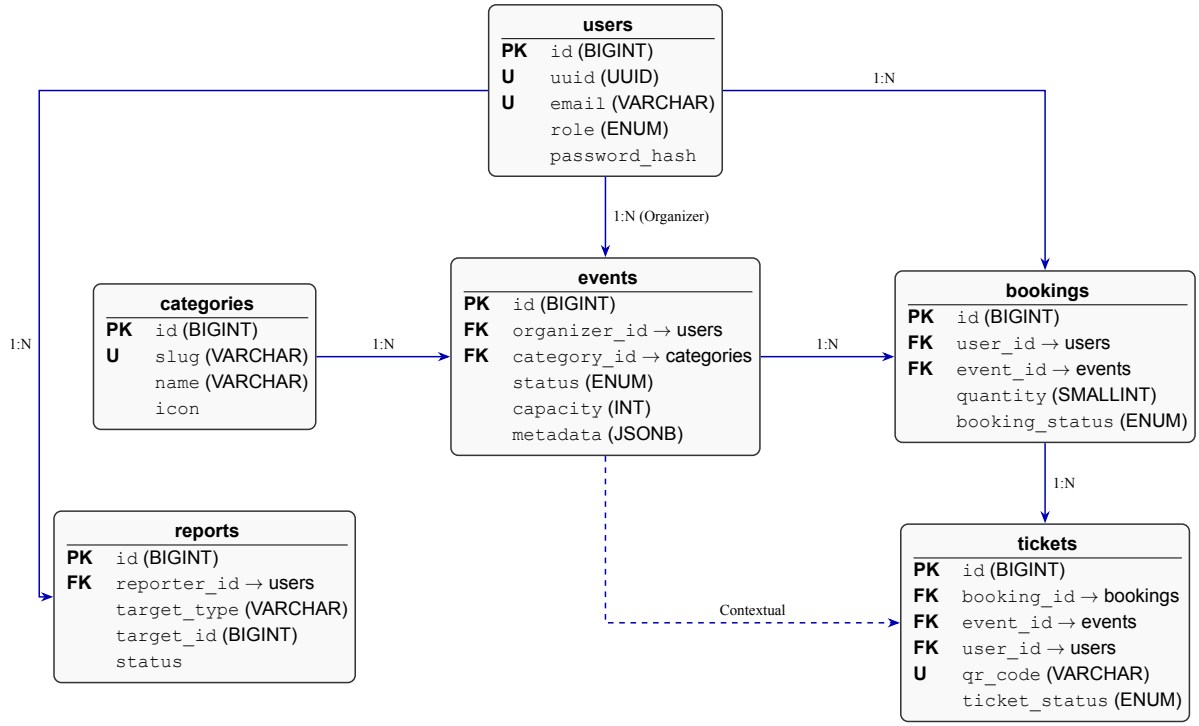

Figure 4.1: Complete Entity-Relationship map of the EventSphere PostgreSQL database. Demonstrates strict relational constraints, UUID indexing, and the use of ENUM domains and JSONB fields.

- **User Management (`UserModel.php`):** Manages cryptographic credential handling and Role-Based Access Control (RBAC) matrices (Customer, Business, Admin).
- **Event Management (`EventModel.php`):** Encapsulates complex filtering logic, geospatial participant tracking, and business ownership constraints.
- **Booking Engine (`BookingModel.php`):** Acts as the nexus between users and events, managing seating availability checks and ticket lifecycle generation.

4.3 Transaction Integrity & ACID Compliance

The most critical operation within the static view is the ticket generation pipeline governed by the `BookingModel`. In a high-demand event scenario, concurrent purchase attempts can lead to race conditions and double-booking.

To guarantee ACID (Atomicity, Consistency, Isolation, Durability) compliance, the EventSphere data engine leverages native PostgreSQL Atomic Transactions. When a booking is initiated, the Model isolates the process by executing a `beginTransaction()` on the PDO instance. The system then verifies inventory and inserts the booking record. If any step fails during this sequence (e.g., inventory drops to zero mid-execution), a `rollBack()` is immediately triggered, discarding all changes. Only upon absolute verification is the `commit()` method invoked, ensuring mathematical certainty in live inventory management.

Chapter 5

Controller Layer & Business Operations

5.1 Authentication & Authorization

The Controller layer is responsible for enforcing the platform's stringent security posture before executing domain logic. The `AuthController.php` manages the primary state mutations (login, registration, logout), while the static `Auth.php` utility serves as the central authority for permission validation.

EventSphere utilizes a strict Role-Based Access Control (RBAC) matrix partitioned into three distinct tiers:

- **Customer:** Restricted to catalog browsing, ticket purchasing, and personal profile management.
- **Business (Organizer):** Granted access to the pipeline control suite to draft events, manage live inventory, and view sales analytics.
- **Admin:** Equipped with a "God Mode" interface (Integrity Matrices) for global oversight, dispute resolution, and platform-wide moderation.

Secure state persistence across requests is managed by the `SessionManager.php` utility. This wrapper encapsulates native PHP sessions to defend against fixation attacks, while also providing flash messaging capabilities for the user interface.

5.2 Event & Ticket Workflows (Document Generation Suite)

The core transactional value of EventSphere is orchestrated by the `EventController.php` and `TicketController.php`. Once a customer successfully purchases a ticket via the `BookingModel`, the system must generate a physical, downloadable asset.

Instead of relying on heavy, monolithic PDF libraries to manually draw geometric elements and text nodes via PHP, EventSphere employs a bespoke, highly optimized Document Generation Suite. The system utilizes a specialized micro-pipeline consisting of three distinct phases:

1. **Data Serialization:** Structured ticket, event, and user data retrieved from the database are serialized into a strict XML payload.
2. **Layout Transformation:** An eXtensible Stylesheet Language Transformations (XSLT) engine parses the XML against a predefined template. This enables the system to bind dynamic parameters (such as reference IDs, user names, and event dates) directly into styled HTML structures. Furthermore, for aggregate documents like the Organizer's Logistics Ledger, the XSLT engine dynamically loops through participant data arrays to construct comprehensive attendee rosters.
3. **PDF Rendering:** The resulting compiled HTML is passed to the TCPDF engine, which resolves the visual layout and renders the final, hardware-ready PDF document.

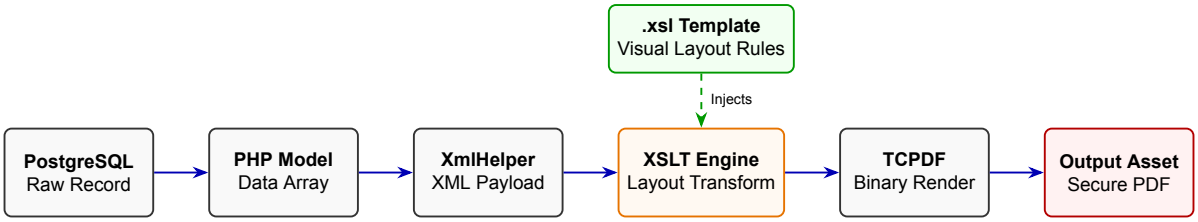

Figure 5.1: The bespoke Document Generation micro-pipeline. The separation of XML data serialization and XSLT layout transformation ensures the visual design layer remains strictly decoupled from the core PHP backend logic.

This recursive pipeline ensures that ticket templates and visual layouts can be radically modified by frontend designers without requiring any alterations to the core backend PHP logic, strictly preserving the separation of concerns.

5.3 API & Asynchronous Operations

While EventSphere utilizes server-side rendering for its core catalog, the "Organizer's Cockpit" and the "Sales Intelligence Dashboard" require real-time interactivity.

To support this without bloating the standard controllers, the architecture isolates asynchronous endpoints within `ApiController.php`. This controller bypasses standard HTML view rendering, instead utilizing PHP's native `json_encode()` to return strict `application/json` payloads.

These endpoints serve as the data backbone for the frontend's AJAX requests, powering dynamic chart rendering, ticket velocity metrics, and the real-time hover states of the Glassmorphic UI, all while maintaining the sub-millisecond response times characteristic of the platform's native engine.

Chapter 6

Security Posture & Utility Core

6.1 Threat Mitigation Strategies

The EventSphere architecture embraces a "Zero-Trust" philosophy at the application layer. Rather than treating security as an afterthought or relying on heavy third-party firewalls, threat mitigation is structurally enforced across the custom MVC pipeline to defend against the most prevalent web vulnerabilities:

- **SQL Injection (SQLi) Immunity:** As established in the static data layer, the application relies exclusively on PHP Data Objects (PDO). The foundational data models enforce strict parameterization for all database queries. By completely decoupling the SQL instruction syntax from the user-provided data payloads, the system achieves structural immunity to SQL injection attacks natively.
- **Cross-Site Scripting (XSS) Prevention:** To mitigate XSS vectors, the application strictly adheres to output encoding protocols. All user-generated content rendered within the Glassmorphic UI is explicitly escaped before being injected into the Document Object Model (DOM). This ensures that malicious script payloads are rendered as inert text strings rather than executable browser code.

6.2 Secure Session Management

Stateful interactions, such as accessing the Organizer's Pipeline Control suite or the Administrative Integrity Matrices, demand robust protection against session hijacking and fixation attacks. EventSphere manages these persistent states through a dedicated session wrapper utility.

This module encapsulates native PHP session handling to enforce critical security directives. Upon successful authentication, the system automatically regenerates the session identifier, invalidating any pre-existing session tokens to prevent fixation. Furthermore, it explicitly configures session cookies to be HTTP-only and strictly bound to secure protocols, rendering the session payload completely inaccessible to malicious client-side scripts.

Beyond security, this utility provides the foundational logic for "flash messaging," securely passing transient success or error states across the request lifecycle to drive the real-time feedback required by the interactive frontend interface.

6.3 Asset Handling & Sanitization

The platform allows event organizers to personalize their global catalog presence by uploading promotional graphics and cover images. Handling user-supplied files introduces significant operational risk, particularly the threat of remote code execution via disguised malicious scripts.

To neutralize this attack vector, asset processing is governed by a specialized image handling module. This utility implements a rigorous, multi-step sanitization pipeline:

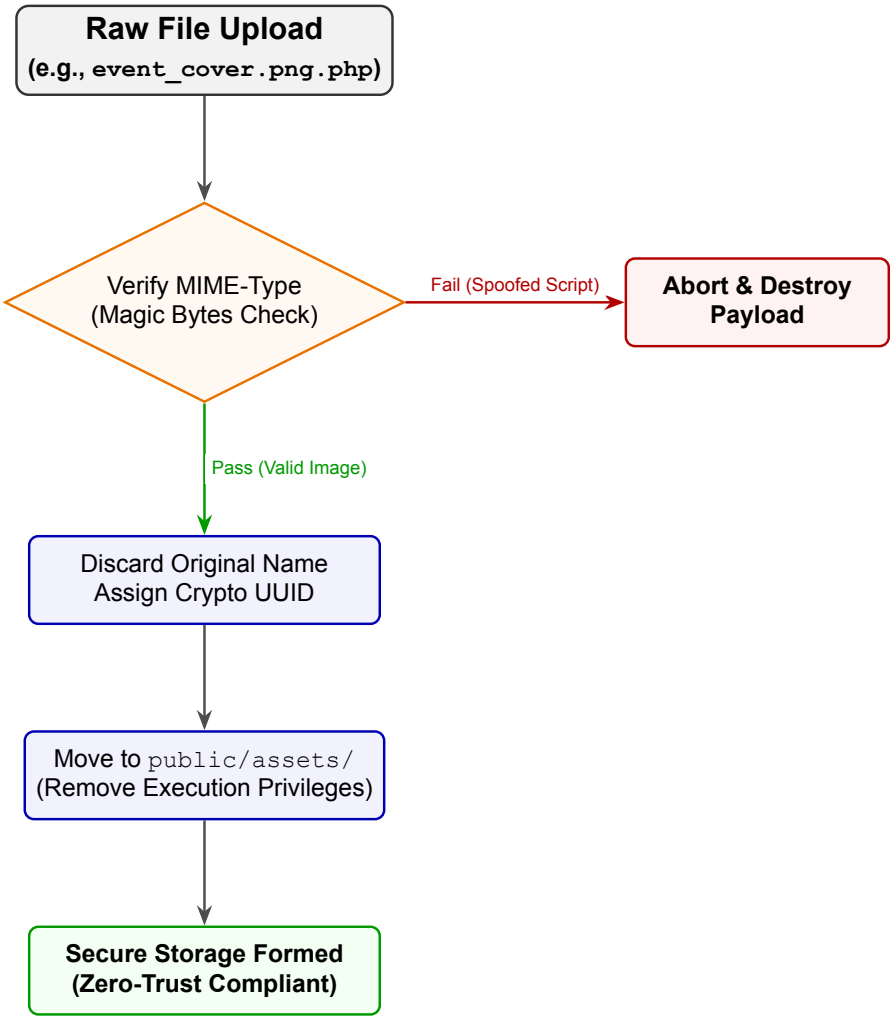

Figure 6.1: The `ImageHelper.php` sanitization decision tree. This pipeline guarantees that malicious executable scripts masquerading as images are destroyed, and validated assets are stripped of directory-traversal vectors before storage.

1. **MIME-Type Verification:** The system ignores easily manipulated file extensions and explicitly verifies the binary signature (magic bytes) of the uploaded file to guarantee it is a mathematically valid image format.
2. **Filename Cryptography:** Original user filenames are discarded immediately upon upload. The utility maps the validated asset to a cryptographically secure, randomized identifier (UUID), neutralizing any directory traversal payloads embedded in the original string.
3. **Execution Prevention:** Processed assets are stored in dedicated public directories that have been explicitly stripped of server-side execution privileges. This ensures that even if a highly sophisticated payload bypassed validation, the underlying Linux environment would refuse to execute it.

Chapter 7

System Limitations & Scalability Analysis

7.1 Connection Bottlenecks

While the "Zero-Bloat" philosophy and native PHP 8 architecture provide exceptional performance under standard operational loads, the system faces theoretical limits regarding database connectivity.

Currently, the custom MVC framework relies on a Singleton Database pattern. This guarantees that only one persistent PostgreSQL connection is opened per request lifecycle. However, in a scenario involving massive concurrent traffic, this 1:1 mapping of HTTP requests to database connections can rapidly lead to connection exhaustion. Because PostgreSQL processes are relatively heavy, the database engine will eventually queue or reject connections once its maximum connection limit is breached.

To achieve enterprise-level scalability, the architecture would require the introduction of a connection pooler, such as PgBouncer. This middleware would sit between the PHP application and the PostgreSQL server, multiplexing thousands of incoming client requests across a much smaller, highly efficient pool of active database connections.

7.2 Concurrency Risks in Bookings

The Booking Engine represents the most critical transaction pathway within EventSphere. The current implementation relies on standard PostgreSQL Atomic Transactions to guarantee ACID compliance during ticket generation.

While transactions prevent partial data writes, standard isolation levels may not be sufficient during extreme "flash sale" scenarios where thousands of users simultaneously attempt to book the final available seat. This creates a vulnerability to race conditions.

To fortify the booking pipeline against concurrent race conditions, the system must be upgraded to employ explicit concurrency control mechanisms. There are two primary architectural pathways to resolve this:

- **Pessimistic Locking:** Implementing row-level locks via SQL syntax (such as explicitly invoking a `FOR UPDATE` clause when reading the event capacity). This forces concurrent transactions to queue sequentially, ensuring absolute mathematical accuracy at the cost of slightly increased latency.
- **Optimistic Concurrency Control:** Introducing version tracking columns within the database schema. The application would read the version state and only commit the booking if the version remains unchanged at the end of the transaction, gracefully aborting and prompting the user to retry if a conflict is detected.

7.3 Missing Caching Layers

The current architecture routes all data retrieval operations directly to the PostgreSQL relational database. While PostgreSQL is highly optimized, reading heavily accessed, infrequently changing data (such as the Global Catalog of events or the base taxonomy of categories) from disk on every single HTTP request introduces unnecessary latency and database load.

The absence of an intermediate caching layer represents a significant scaling limitation. To optimize read-heavy operations and support the real-time demands of the Glassmorphic user interface, the integration of an in-memory data structure store, such as Redis or Memcached, is recommended.

By caching the serialized JSON payloads of popular event listings or the compiled analytical data required by the Organizer's Cockpit, EventSphere could drastically reduce the query volume hitting the primary PostgreSQL database, thereby preserving system resources for critical transactional operations like ticket sales.

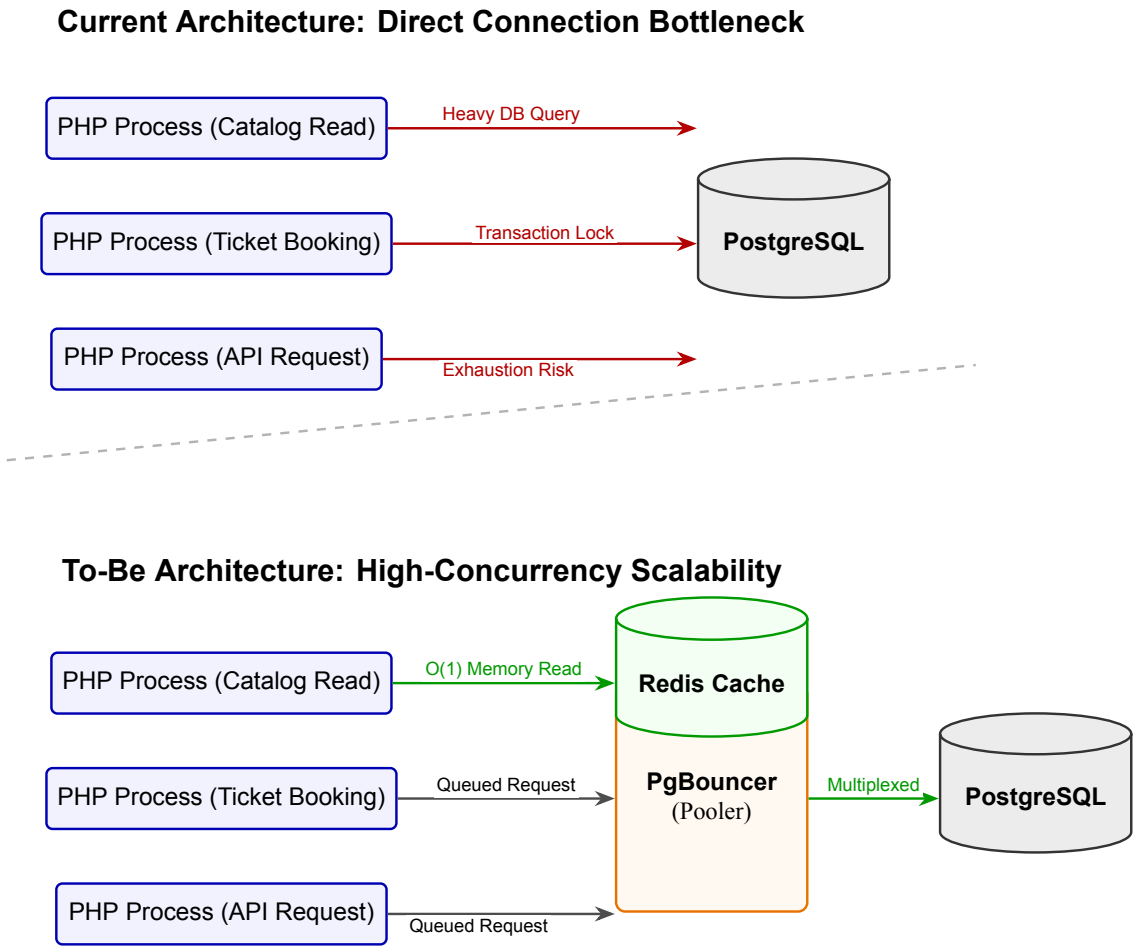

Figure 7.1: Comparative analysis of the EventSphere data layer. The integration of Redis for read-heavy operations and PgBouncer for transaction multiplexing mitigates the inherent bottlenecks of the PHP Singleton pattern under extreme concurrent load.

Chapter 8

Conclusion & Future Outlook

The design and implementation of the EventSphere backend architecture successfully demonstrate that complex, enterprise-grade logic can be achieved without relying on bloated, monolithic frameworks. By adopting a strict “Zero-Bloat” philosophy, the platform guarantees sub-millisecond response times, absolute control over the data layer via native PDO connections, and a structurally sound MVC pipeline.

The custom implementations detailed in this report—specifically the Front Controller routing mechanism, the Zero-Trust asset sanitization workflow, and the decoupled XML-to-XSLT Document Generation Suite—highlight a deep understanding of core web technologies. These architectural choices prioritize not only performance but also long-term maintainability and strict separation of concerns.

As outlined in the scalability analysis, the current Singleton database constraints provide a clear roadmap for future iteration. The planned integration of PgBouncer for transaction multiplexing and Redis for memory caching will ensure the architecture remains resilient under extreme concurrent loads.

Ultimately, EventSphere stands as a testament to the power of native web engineering. It successfully bridges the gap between rigorous, highly secure backend data processing and the real-time API demands of a modern, Glassmorphic user interface.

EVENTSPHERE

"Engineering performance through simplicity."

This report details the foundational architecture, security posture, and data layer of the EventSphere platform. Developed as an advanced exploration of modern web systems, the project actively rejects framework bloat in favor of native, highly optimized computing.

Inside, you will find comprehensive analyses of Front Controller routing, Role-Based Access Control matrices, Zero-Trust asset sanitization, and a bespoke XML-to-PDF Document Generation Suite—all powered by an unyielding native PHP 8 and PostgreSQL ecosystem.

SAPIENZA UNIVERSITY OF ROME

Department of Computer and Control Engineering

© 2026 Daniele Compagnoni

MIT License • Published for Web Systems & Technologies