
React + PHP: The Thin Stack

Il protocollo miniCMS per Web App moderne

Simone Pizzi

Marzo 2026 — Prima Edizione

Indice

La Visione	1
CAPITOLO 1: Manifesto	2
Perché Esiste Questo Protocollo	2
Il Principio Fondativo: La Separazione dei Piani	3
Cosa Non È Questo Protocollo	4
I Valori che Guidano Ogni Decisione	5
A Chi È Rivolto	6
Come Usare Questo Manuale	7
 L'Architettura	 10
CAPITOLO 2: Architettura e Struttura Progetto (v1.2 - ADVANCED)	11
1. Topologia delle Cartelle e Asset Separation	11
1.1 Struttura Fisica (Root)	11
1.2 Anatomia dell'Area API (public/api/)	11
2. Meccanismi di Sicurezza in Build (The Clean-Dist Logic)	11
3. Gestione del Routing e degli URL (SPA vs API)	12
4. Inizializzazione Dinamica del File System	12
5. Gestione degli Ambienti (Env Strategy)	13
6. Il Pattern Fork (FDCA / DISINTELLIGENZA)	13
 CAPITOLO 3: Database Strategy (v1.2 - ADVANCED)	 15
1. Architettura di Connessione (Agnostic PDO)	15
1.1 Configurazione Ottimale SQLite	15
1.2 Auto-Scaffolding della Cartella .data	15
2. Indicizzazione Strategica (Performance Engineering)	16
3. Ciclo di Vita delle Migrazioni (Safe-Schema Update)	17
4. Normalizzazione Dati (The "Round" Rule)	17
5. Manutenzione e Integrità	17
6. Quando Passare a MySQL	18
 CAPITOLO 4: Frontend Dependencies (v1.0)	 20
1. Il Core Stack (Ogni Progetto)	20
2. Tailwind CSS: v3 vs v4	20
SitoRuntime — Tailwind v3 (Setup Classico)	21

SimonePizziWebSite, DISINTELLIGENZA, FDCA — Tailwind v4	21
@tailwindcss/typography	21
3. Rendering del Contenuto	21
showdown — Markdown → HTML	21
dompurify — Sanitizzazione XSS	22
react-quill-new — Rich Text Editor WYSIWYG	22
4. Animazioni & Effetti Visivi	23
framer-motion	23
typewriter-effect	23
tailwindcss-animate	24
5. SEO & Meta Tag	24
react-helmet-async	24
6. Utilities	24
clsx + tailwind-merge	24
date-fns	25
7. Build-Time Optimization	25
sharp	25
8. Matrice delle Dipendenze per Sito	26
9. Regole per i Nuovi Progetti	26
I Componenti	30
CAPITOLO 5: Backend Logic (PHP) (v1.2 - ADVANCED)	31
1. Gestione degli Identificatori (Slug Logic)	31
1.1 Algoritmo Base	31
1.2 Algoritmo Avanzato (con Normalizzazione Accenti Italiani)	31
1.3 Strategia Anti-Collisione	32
2. Gestione del Fuso Orario (Timezone Forcing)	32
3. Gestione dei Metodi HTTP (RESTful Completo)	32
4. Il Pattern auth_helper.php (Autenticazione Inline)	33
5. Elaborazione Media e Ottimizzazione Immagini	34
5.1 Polimorfismo dei Percorsi	34
5.2 Auto-Resize & Trasparenza	34
6. Sicurezza dell'Input e Output	34
7. CORS Management	34
8. Buffer Management	35
CAPITOLO 6: Frontend Bridge (API.ts) (v1.1 - ADVANCED)	37
1. Architettura del Wrapper Universale	37
1.1 Il Pattern “Double Read” (Response Cloning)	37
2. Polimorfismo delle Chiamate	37
3. Sincronizzazione dello Stato Globale	38
3.1 Silent Auth Check	38

3.2 Hard Logout	38
4. Validazione Incrociata (HTTP vs Logic)	38
5. TypeScript Integration (Type Safety)	38
CAPITOLO 7: Media & Optimization (v1.2 - ADVANCED)	40
1. Architettura di Caching (The TTL Strategy)	40
1.1 Cache Paginata (Listings)	40
1.2 Invalidazione Intelligente	40
2. SEO & Social Pre-rendering	40
2.1 Il Meccanismo (Riferimento Completo: Capitolo 11)	40
2.2 Il Rebuild Cache SEO (Per Retrocompatibilità)	41
3. Ottimizzazione Media & Compression	41
3.1 Normalizzazione delle Immagini (PHP GD)	41
3.2 Sharp (Node.js) per Image Processing Build-Time	41
3.3 Cache Control sui File Statici	41
4. Manutenzione e Rebuilding	42
CAPITOLO 8: Advanced Content Editing & Media Integration (v1.1 - ADVANCED)	44
1. Il Media Center Centralizzato	44
1.1 Filtraggio Contestuale (Tab Strategy)	44
1.2 Gestione Asset Avanzata	44
2. Il Componente MediaPicker (Integration Layer)	44
3. Rich Text Editor & Sanitizzazione	45
4. Gestione degli URL e Portabilità	45
5. UX e Feedback Visivo	45
Il Flusso Operativo	48
CAPITOLO 9: Content Lifecycle (v1.2 - ADVANCED)	49
1. Stati Dinamici vs Stati Persistenti	49
1.1 Matrice degli Stati	49
2. Implementazione della “Programmazione Reale”	49
2.1 Logica Frontend (React)	49
2.2 Normalizzazione Data/Ora (Input Strategy)	50
3. Workflow Editoriale e Integrità	50
4. Gestione Multicategoria e Tagging	50
5. Security & Visibility (Bypass Logic)	50
5.1 Bypass per Singolo Articolo (via Slug)	51
5.2 Bypass per Lista (via Parametro Admin)	51
5.3 Accesso per ID (Editor Admin)	52
CAPITOLO 10: Security & Auth (v1.2 - ADVANCED)	54
1. Gestione della Sessione (Stateful-Sec)	54
1.1 Configurazione Server-Side	54

1.2 Endpoint check_auth	54
2. Architettura Protected Routes (React)	55
2.1 Bootstrapping della Sicurezza	55
2.2 Role-Based UI (RBAC)	55
3. Sicurezza Crittografica	55
4. Protezione del Database SQLite	55
5. Sanitizzazione dei Buffer di Output	55
6. Caso Reale: L'Attacco DDoS a Runtime Radio (Febbraio 2026)	56
6.1 Il Contesto: Due Crisi Sovrapposte	56
6.2 Il Vettore di Attacco: I Bot dei Social Media	56
6.3 La Risposta: Tre Livelli di Difesa	57
6.4 Le Persone	57
6.5 Le Lezioni da Portare in Ogni Progetto	57
CAPITOLO 11: SEO Pre-rendering con PHP Entry-Point (v1.0)	60
1. Il Problema e la Soluzione	60
Il Problema	60
La Soluzione: The Swap Trick	60
2. Il Meccanismo: Build Rename Strategy	60
In DISINTELLIGENZA (build script in package.json):	61
In SimonePizziWebSite (build script standard):	61
3. Il Codice di index.php (Analisi Completa)	61
4. Punti Critici del Pattern	63
4.1 Connessione Diretta al DB (No Include Esterno)	63
4.2 Sanitizzazione dello Slug	63
4.3 Fallback Silenzioso	64
4.4 Bypass Admin	64
4.5 Immagine Assoluta	64
5. Estensione a Più Entità	64
6. Configurazione Apache (.htaccess)	64
CAPITOLO 12: RSS Feed & Syndication (v1.0)	66
1. Struttura del Feed RSS Standard	66
2. Dettagli Tecnici Critici	67
2.1 Il Problema del Fuso Orario	67
2.2 Formato Data RFC 822	68
2.3 URL degli Articoli nel Feed	68
2.4 Enclosure per Immagini	68
2.5 GUID Univoco	68
3. Feed RSS per Podcast (Podcast App)	69
4. Registrazione del Feed nel Routing	69
5. Integrazione nel Frontend (Discovery)	69

CAPITOLO 13: Newsletter & Email System (v1.0)	71
1. Schema del Database	71
2. Architettura degli Endpoint	71
3. Iscrizione Pubblica	72
4. Disiscrizione GDPR-Compliant	73
5. Conteggio Iscritti (Admin)	74
6. Invio Newsletter (Admin)	74
6.1 Generazione HTML con Placeholder	74
6.2 Headers Email	75
6.3 Rate Limiting	75
6.4 Query degli Articoli (Ottimizzazione)	75
7. Struttura Email HTML (Pattern)	76
8. Considerazioni per la Scalabilità	77
 I Casi Reali	 80
CAPITOLO 14: Database Evolution - Da SQLite a MySQL (v1.0)	81
1. Quando e Perché Migrare	81
2. La Storia del WAL (Warning reale)	81
3. Architettura della Migrazione (Il Pattern a 3 Script)	82
3.1 db_credentials.php — Il File dei Segreti	82
3.2 db.php con MySQL — Il Connettore Agnostico	82
3.3 init_mysql.php — Creazione Schema	83
3.4 migrate_to_mysql.php — Il Trasloco Dati	83
4. Differenze Strutturali SQLite vs MySQL	84
5. Il Pattern di Credenziali (Sicurezza)	84
6. Checklist di Migrazione	84
 CAPITOLO 15: Portfolio & Projects Module (v1.0)	 87
1. La Differenza con il Modulo News/Articles	87
2. Schema Database	87
3. L'API projects.php (5 Metodi HTTP)	88
GET — Lista con Bypass Admin	88
POST — Creazione con Auto-Sort	88
PATCH — Aggiornamenti Parziali	88
4. Il Pattern dei Pulsanti CTA (button_a / button_b)	89
5. Il Pattern di Slug Avanzato (Normalizzazione Accenti Italiani)	89
6. React Frontend — Componenti Chiave	90
PortfolioGrid.tsx	90
ProjectEditor.tsx (Admin)	90
ProjectsList.tsx (Admin)	91
7. Strategie di Categoria	91
8. auth_helper.php — Il Pattern Minimale	91

CAPITOLO 16: Festival Logic - Iscrizioni e Workflow Approvazione (v1.0)	95
1. Il Workflow del Partecipante	95
1.1 Stati di Iscrizione	95
2. Automazione Email (Transaction Emails)	95
3. Gestione Asset Partecipanti	95
4. Newsletter Sync Strategy	95
CAPITOLO 17: Festival Logic - Votazioni e Protezione Anti-Frode (v1.0)	97
1. La Sessione di Voto	97
2. Meccanismi di Protezione (Anti-Abuso)	97
3. Gestione dei Round (Il “Palcoscenico” del Voto)	97
4. Master Switch di Votazione	98
CAPITOLO 18: Festival Logic - Dashboard Admin, Settings e Reporting (v1.0)	100
1. Master Switches (Global Settings)	100
2. Dashboard Gestionale (KPIs)	100
3. Workflow di Approvazione e Ranking	100
4. Reporting Automatico	100
Allegati	102
BOILERPLATE CHECKLIST: Avvio Nuovo Progetto miniCMS (v2.0)	103
Fase 1: Setup Ambiente e Sicurezza Iniziale	104
Fase 2: Configurazione Backend Core (PHP)	104
Fase 3: Configurazione Frontend Core (React)	104
Fase 4: Integrazione Media e Contenuti	105
Fase 5: SEO e Syndication	105
Fase 6: Ottimizzazione e Deploy	105
Fase 7: Specifico per Tipologia di Sito	105
Per Siti con Newsletter (SitoRuntime pattern)	105
Per Portfolio/Sito Personale (SimonePizziWebSite pattern)	106
Per Festival/Concorso (FDCA / DISINTELLIGENZA pattern)	106
Per Migrazione SQLite → MySQL (SitoRuntime pattern)	106

La Visione

CAPITOLO 1: Manifesto

Perché Esiste Questo Protocollo

Esiste una tensione irrisolta al centro dello sviluppo web moderno.

Da un lato, il frontend ha raggiunto una maturità estetica e funzionale straordinaria: React, TypeScript, Tailwind. Animazioni fluide, componenti riutilizzabili, type safety, hot reload. L'esperienza di sviluppo è diventata un piacere, e il prodotto finale — quando fatto bene — è visivamente e funzionalmente superiore a qualsiasi soluzione del passato.

Dall'altro lato, questa rivoluzione ha portato con sé una complessità infrastrutturale sproporzionata rispetto ai bisogni reali della maggioranza dei siti. Node.js, database cloud, container, pipeline CI/CD, micro-servizi, CMS headless con piani in abbonamento. L'overhead tecnico e il costo operativo sono diventati la norma anche per siti che potrebbero girare perfettamente su un hosting da 5 euro al mese.

Questo protocollo nasce da una domanda precisa: **è possibile avere la potenza estetica e tecnica di React senza abbandonare la semplicità, il controllo e l'economicità di un backend PHP con SQLite?**

La risposta, costruita su anni di lavoro reale su progetti reali, è sì.

Il Principio Fondativo: La Separazione dei Piani

Il Modello Universale non è una tecnologia. È un'architettura mentale.

Separa con nettezza due piani che spesso vengono confusi:

Il Piano della Presentazione appartiene a React. È il luogo della forma, dell'interazione, dell'animazione, della tipografia, della palette colori. È dove vive il talento estetico, dove Tailwind traduce l'intenzione visiva in CSS preciso, dove framer-motion aggiunge peso e respiro ai movimenti. Questo piano è compilato, ottimizzato, servito come asset statico.

Il Piano dei Dati appartiene a PHP e SQLite (o MySQL quando necessario). È il luogo della persistenza, della logica di business, della sicurezza, del ciclo di vita dei contenuti. Non è "il backend" nel senso pesante del termine — nessun framework, nessun ORM, nessuna dipendenza esterna. Solo PHP nativo, PDO, e un database file-based che non richiede configurazione server.

Questi due piani comunicano attraverso un contratto preciso: le API REST. Il frontend non sa niente del database. Il backend non sa niente di React. La loro separazione è la fonte di tutta la scalabilità e la manutenibilità del sistema.

Cosa Non È Questo Protocollo

Non è un framework. Non impone strutture di codice rigide, non richiede dipendenze specifiche, non vincola le scelte stilistiche.

Non è un CMS tradizionale. Non c'è un'interfaccia visual builder, non ci sono temi preconfezionati, non c'è un marketplace di plugin. Ogni sito costruito con questo protocollo è unico, fatto a mano, tailor-made per il suo scopo.

Non è una soluzione enterprise. Non è progettato per gestire milioni di utenti simultanei, flussi di dati complessi o architetture distribuite. È progettato per siti che devono essere eccellenti, veloci, sicuri e mantenibili da un team piccolo — o anche da una persona sola.

Non è per chi vuole un sito in 10 minuti. È per chi vuole capire cosa sta costruendo.

I Valori che Guidano Ogni Decisione

Controllo totale. Chi costruisce un sito con questo protocollo possiede ogni riga del suo stack. Nessun vendor lock-in, nessun aggiornamento forzato che rompe la produzione, nessuna dipendenza da un servizio esterno per la sopravvivenza del sito.

Leggerezza come principio, non come compromesso. SQLite non è la scelta “economica” rispetto a MySQL — è la scelta giusta per il 90% dei casi d’uso. La semplicità non è una limitazione da superare: è un obiettivo da raggiungere e difendere.

La sicurezza come architettura, non come patch. Le decisioni di sicurezza sono integrate nel design del sistema: database fuori dalla root pubblica, clean-dist.js che non lascia mai il database nel deploy, sessioni PHP con cookie httponly, password mai in chiaro. Non sono misure aggiunte dopo — sono la struttura stessa.

La documentazione come parte del codice. Un sistema che non si capisce è un sistema che non si può mantenere. Questo protocollo è documentato con la stessa cura con cui è costruito, perché la conoscenza deve rimanere accessibile anche quando il contesto cambia.

L’esperienza reale come unico validatore. Ogni pattern documentato in questo manuale è stato estratto da codice che gira in produzione. Le lezioni più importanti — il crash del WAL in produzione, l’attacco DDoS che ha messo in ginocchio Runtime Radio, la migrazione SQLite/MySQL forzata dal traffico crescente — sono incidenti reali, non scenari ipotetici. La teoria senza la cicatrice non insegna abbastanza.

A Chi È Rivolto

A chiunque voglia costruire un sito web che sia una cosa viva — non un template, non un WordPress personalizzato, non un sito generato da un builder.

Al developer che conosce React e vuole un backend senza dover imparare un framework intero.

Al freelance che deve consegnare un sito veloce, mantenibile e sicuro a un cliente che non ha budget per infrastrutture cloud.

All'autore, al musicista, al festival, alla radio che vuole una presenza digitale propria, controllata, indipendente dai capricci delle piattaforme.

A chiunque creda che il web possa essere ancora un posto fatto da persone, per persone — senza intermediari.

Come Usare Questo Manuale

Il manuale è organizzato in capitoli tematici indipendenti. Non è necessario leggerlo dall'inizio alla fine — ogni capitolo è una reference autonoma.

Per iniziare un nuovo progetto da zero, la **BOILERPLATE-CHECKLIST** è il punto di partenza pratico.

Per migliorare un progetto esistente, i capitoli specifici (Database Strategy, Security & Auth, SEO Pre-rendering) offrono pattern applicabili chirurgicamente.

Per imparare dalla storia, i capitoli con la voce esperienziale (WAL disaster, DDoS su Runtime Radio, migrazione MySQL) sono la lettura più onesta che questo manuale può offrire.

Il codice non mente. Le cicatrici nemmeno.

“La perfezione si raggiunge non quando non c’è più niente da aggiungere, ma quando non c’è più niente da togliere.” — Antoine de Saint-Exupéry

Prossimo Capitolo: Architettura e Struttura Progetto — dove le idee diventano cartelle.

L'Architettura

CAPITOLO 2: Architettura e Struttura Progetto

(v1.2 - ADVANCED)

Questa sezione definisce l'architettura fisica e logica del sistema, garantendo sicurezza dei dati, portabilità e scalabilità.

1. Topologia delle Cartelle e Asset Separation

Il Modello Universale impone una separazione netta tra i file sorgente (Build-time) e i file di runtime (Persistent-data).

1.1 Struttura Fisica (Root)

```
/
├── public/                # Contenuto pubblico e Entry Point API
│   ├── .htaccess          # Routing SPA (Cruciale per React Router)
│   ├── index.php          # SEO Engine (Capitolo 11) – entry-point PHP
│   └── api/               # Core Backend PHP (Vedi 1.2)
├── src/                  # Frontend React 19 / TS
├── scripts/              # Utility Tools (Build, Clean, Migration)
├── .env.local             # Configurazione locale (VITE_API_URL)
├── package.json           # Gestione dipendenze e Script di automazione
└── clean-dist.js          # Script di sanitizzazione post-build (SECURITY)
```

1.2 Anatomia dell'Area API (public/api/)

La cartella `api /` deve essere configurata per gestire la persistenza senza dipendenze esterne: - **.data/**: Contiene il database SQLite. Deve contenere un file `.htaccess` con `Deny from all`. - **.cache/**: File JSON generati dinamicamente per le performance. Puliti automaticamente ad ogni scrittura. - **uploads/**: Asset caricati dall'utente (immagini, audio). Deve essere esclusa dai backup del codice sorgente.

2. Meccanismi di Sicurezza in Build (The Clean-Dist Logic)

Uno dei rischi maggiori è il "Database Overwrite" durante il deploy. Il sistema implementa uno script `clean-dist.js` eseguito post-build che: 1. Analizza la cartella `dist/api/`. 2. Rimuove ricorsi-

vamente ogni file con estensione `.sqlite`, `.sqlite3`, `.db` o `.bak`. 3. Notifica l'operatore con log di sicurezza (☒ SECURITY: Removed...). 4. **Regola Progettuale**: La cartella `dist/` generata deve essere "database-free". Il DB deve essere inizializzato sul server o migrato manualmente, mai sovrascritto dalla build automatica.

Il pattern esatto del build script varia per sito: - **SimonePizziWebSite**: `"postbuild": "node clean-dist.js"` (hook automatico npm) - **DISINTELLIGENZA**: `"build": "tsc -b && vite build && node clean-dist.js && move dist\\index.html dist\\index_react.html"` (rinomina `index.html` per abilitare il SEO Engine PHP, vedi Capitolo 11) - **SitoRuntime**: `"build": "tsc -b && vite build && node scripts/remove-db-from-dist.js"` (script dedicato in cartella `scripts/`)

3. Gestione del Routing e degli URL (SPA vs API)

Per permettere a React Router di funzionare in armonia con le API PHP su server Apache, lo standard prevede un `.htaccess` nella root pubblica:

```
<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /
  # Se la richiesta punta a un file o cartella reale, servilo direttamente (permette
  ↪ alle API di funzionare)
  RewriteCond %{REQUEST_FILENAME} -f [OR]
  RewriteCond %{REQUEST_FILENAME} -d
  RewriteRule ^ - [L]
  # Altrimenti, reindirizza tutto a index.html (o index.php se presente)
  RewriteRule ^ index.html [L]
</IfModule>
```

Nota: Apache serve `index.php` prima di `index.html` per priorità predefinita. Questo è il meccanismo che permette al SEO Engine (Capitolo 11) di intercettare le richieste senza modificare le regole di rewrite.

4. Inizializzazione Dinamica del File System

Il backend PHP non deve dare per scontato l'esistenza delle cartelle di runtime. Ogni script di inizializzazione (`init_db.php`) o la classe Database devono implementare la logica di **Auto-Scaffolding**:

```
// Logica di creazione cartelle con permessi corretti (0755)
$paths = [__DIR__ . '/.data', __DIR__ . '/.cache', __DIR__ . '/uploads'];
foreach ($paths as $path) {
  if (!is_dir($path)) {
    mkdir($path, 0755, true);
    // Protezione immediata se è la cartella .data
```

```
if (basename($path) === '.data') {  
    file_put_contents($path . '/.htaccess', "Order allow,deny\nDeny from  
    ↪ all");  
}  
}  
}
```

In **SimonePizziWebSite** questo auto-scaffolding è integrato direttamente in `db.php` (connessione lazy), mentre in altri siti è in `init_db.php`. Entrambi i pattern sono validi; il primo è più robusto perché si attiva ad ogni connessione.

5. Gestione degli Ambienti (Env Strategy)

- **Local Dev:** React punta a `http://localhost/tuo-progetto/public/api`.
- **Production:** React punta a `/api` (percorso relativo, garantendo la compatibilità con SSL e domini diversi).
- **Vite Config:** Utilizzo del proxy in fase di sviluppo per evitare errori CORS continui.

6. Il Pattern Fork (FDCA / DISINTELLIGENZA)

FDCA e DISINTELLIGENZA condividono **identica struttura PHP** (stessi file, stessa logica). Questo è un pattern deliberato: quando due progetti condividono la stessa base funzionale (es. festival con votazioni), si parte da un fork del progetto più maturo.

Vantaggi: Nessuna dipendenza condivisa — ogni progetto evolve indipendentemente. **Rischio:** Bug-fix e miglioramenti vanno applicati manualmente a entrambi i fork. Per questo è fondamentale mantenere una documentazione (questo manuale) come fonte di verità comune.

Prossimo Capitolo: Database Strategy - Dettagli avanzati su lock, indici, migrazioni e la vera storia del WAL.

CAPITOLO 3: Database Strategy (v1.2 - ADVANCED)

Il database è il cuore pulsante del miniCMS. Questa sezione definisce le strategie di ottimizzazione, integrità e migrazione per garantire performance “zero-latency” anche su hosting condivisi.

1. Architettura di Connessione (Agnostic PDO)

Il sistema utilizza un wrapper PDO che astrae il motore sottostante.

1.1 Configurazione Ottimale SQLite

Per evitare il problema dei “file lock” comuni su Apache/PHP, il Modello Universale impone: - **Journal Mode: DELETE**: Più lento di WAL ma infinitamente più stabile su hosting condivisi dove il locking del file system è imprevedibile. - **Busy Timeout**: Impostato a 5000ms per gestire tentativi di scrittura simultanei (es. Newsletter + Admin).

```
// In db.php
self::$pdo = new PDO("sqlite:" . $dbPath);
self::$pdo->exec("PRAGMA journal_mode=DELETE;");
self::$pdo->exec("PRAGMA busy_timeout=5000;");
self::$pdo->exec("PRAGMA foreign_keys = ON;");
```

❗ **Perché DELETE e non WAL**: SitoRuntime ha tentato di usare `journal_mode=WAL` in produzione per migliorare le performance sotto carico. Il WAL ha causato problemi gravi su hosting condiviso Apache: il file di lock `.sqlite-wal` rimaneva “appeso” corrompendo le letture. È stato necessario uno script di emergenza (`emergency_revert_wal.php`) per tornare a DELETE. **Questa è la lezione: usare sempre DELETE mode su hosting condiviso.**

1.2 Auto-Scaffolding della Cartella .data

La classe Database in **SimonePizziWebSite** integra la creazione automatica della cartella `.data/` e del relativo `.htaccess` di protezione direttamente nella connessione lazy:

```
public static function connect() {
    if (self::$pdo === null) {
        $dbPath = __DIR__ . '/.data/database.sqlite';
        $dir = dirname($dbPath);
```

```
// Crea la cartella se non esiste
if (!is_dir($dir)) {
    mkdir($dir, 0777, true);
}

// Protezione immediata con .htaccess
$htaccessPath = $dir . '/.htaccess';
if (!file_exists($htaccessPath)) {
    file_put_contents($htaccessPath, "Require all denied\n");
}

try {
    self::$pdo = new PDO('sqlite:' . $dbPath);
    self::$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    self::$pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
    self::$pdo->exec('PRAGMA journal_mode = DELETE;');
    self::$pdo->exec('PRAGMA foreign_keys = ON;');
} catch (PDOException $e) {
    http_response_code(500);
    echo json_encode(['status' => 'error', 'message' => 'Database connection
    ↪ failed']);
    exit;
}
return self::$pdo;
}
```

2. Indicizzazione Strategica (Performance Engineering)

Le query di lettura devono essere istantanee. Il Modello Universale impone la creazione dei seguenti indici:

Tabella	Colonna	Tipo	Scopo
news/articles	slug	UNIQUE	Ricerca articolo via URL (SEO)
news/articles	published_at	DESC	Ordinamento cronologico veloce
news/articles	status	INDEX	Filtraggio Bozza/Pubblicato
speakers	sort_order	ASC	Ordinamento manuale trascinabile
podcasts	slug	UNIQUE	Accesso rapido alle serie

Tabella	Colonna	Tipo	Scopo
projects	category	INDEX	Filtraggio per categoria portfolio
projects	sort_order	ASC	Ordinamento manuale portfolio

Comando di Manutenzione: Periodicamente (o dopo caricamenti massivi) deve essere eseguito `ANALYZE` ; per ricalcolare le statistiche di accesso e ottimizzare il piano di esecuzione delle query.

3. Ciclo di Vita delle Migrazioni (Safe-Schema Update)

Le modifiche allo schema devono essere atomiche e reversibili. - **Atomicità:** Ogni script di migrazione deve utilizzare le transazioni (`beginTransaction`). - **Idempotenza:** Lo script deve verificare l'esistenza di colonne o tabelle prima di tentare la creazione (`IF NOT EXISTS, PRAGMA table_info`). - **Protezione:** Gli script di migrazione (`update_db_vX.X.X.php`) devono essere protetti da controllo sessione admin per evitare esecuzioni non autorizzate. - **Nomenclatura:** Il pattern di naming dei siti reali: `update_db_v0.4.2.php`, `update_db_v0.5.4.php`. Numerazione semantica (Major.Minor.Patch) allineata alla versione del progetto in `package.json`.

4. Normalizzazione Dati (The “Round” Rule)

Per evitare errori di precisione nei calcoli (es. durata podcast o bitrate), il sistema impone la normalizzazione lato backend prima del salvataggio: - **Date:** Formato ISO 8601 (`Y-m-d H:i:s`) per compatibilità SQL e JS. - **Numerici:** Interi puri o arrotondati a zero decimali per evitare bug di virgola mobile tra PHP e SQLite. - **Booleani:** Sempre `INTEGER` in SQLite (0 o 1). In MySQL `TINYINT(1)`.

5. Manutenzione e Integrità

- **VACUUM:** Da eseguire mensilmente o post-cancellazione massiva per ricostruire il file database e ridurne il peso fisico.
- **Backup:** Ogni operazione di migrazione deve essere preceduta da una copia fisica del file `.sqlite` in una cartella di backup protetta.
- **optimize_db.php:** SitoRuntime include uno script dedicato alla manutenzione (`VACUUM`, `ANALYZE`, verifica integrità) eseguibile manualmente dall'admin.

6. Quando Passare a MySQL

Vedi Capitolo 14 per la storia completa e il processo di migrazione. In sintesi: rimani su SQLite finché il traffico è gestibile (< 50 scritture/ora) e non ci sono vincoli di hosting. La migrazione è documentata con script reali, testati in produzione.

Prossimo Capitolo: Frontend Dependencies - La matrice delle dipendenze, le regole di scelta e il costo di ogni libreria.

CAPITOLO 4: Frontend Dependencies (v1.0)

Le dipendenze frontend sono scelte architetture, non solo dichiarazioni in `package.json`. Ogni libreria aggiunta ha un costo — dimensione del bundle, superficie di aggiornamento, complessità di integrazione — e deve guadagnarsi il suo posto. Questo capitolo documenta le scelte fatte nei 4 siti di riferimento, con le motivazioni che le guidano.

1. Il Core Stack (Ogni Progetto)

Queste dipendenze sono presenti in tutti e 4 i siti senza eccezioni:

Libreria	Versione	Ruolo
<code>react+react-dom</code>	<code>^19.2.x</code>	Framework UI — componenti, stato, rendering
<code>react-router-dom</code>	<code>^7.x</code>	Client-side routing, React Router v7 (API stabile)
<code>typescript</code>	<code>~5.x</code>	Type safety a compile time
<code>vite</code>	<code>^7.x</code>	Build tool — dev server HMR, bundle ottimizzato
<code>@vitejs/plugin-react</code>	<code>^5.x</code>	Plugin Vite per JSX/TSX transform
<code>lucide-react</code>	<code>^0.5xx</code>	Libreria icone SVG — tree-shakeable, React-native

React 19 introduce significativi miglioramenti al rendering concorrente e alle Server Components. Per il Modello Universale (solo SPA client-side), la differenza pratica con v18 è minima, ma mantenere l'ultima versione stabile è la scelta corretta per nuovi progetti.

2. Tailwind CSS: v3 vs v4

I 4 siti usano versioni diverse di Tailwind, con configurazione significativamente diversa.

SitoRuntime — Tailwind v3 (Setup Classico)

```
// devDependencies
"tailwindcss": "^3.4.17",
"autoprefixer": "^10.4.22",
"postcss": "^8.5.6"
```

Richiede `tailwind.config.js` e `postcss.config.js` espliciti. Il file CSS di ingresso:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

SimonePizziWebSite, DISINTELLIGENZA, FDCA — Tailwind v4

```
// devDependencies
"tailwindcss": "^4.x",
"@tailwindcss/vite": "^4.x" // o "@tailwindcss/postcss"
```

Tailwind v4 usa il plugin Vite nativo — nessun `postcss.config.js` separato, nessun `tailwind.config.js` obbligatorio. Il file CSS:

```
@import "tailwindcss";
```

Quando usare v4: Per tutti i nuovi progetti. La migrazione da v3 a v4 richiede attenzione alle classi rinominate (es. `shadow-sm` → `shadow-xs`).

@tailwindcss/typography

Presente in tutti i siti come `devDependency`. Aggiunge la classe `prose` che applica stili tipografici curati a blocchi di HTML generato dinamicamente (come output Markdown o contenuto da rich text editor).

```
<div class="prose prose-invert max-w-none">
  {/* HTML generato da showdown o Quill */}
</div>
```

3. Rendering del Contenuto

showdown — Markdown → HTML

Usato in: SimonePizziWebSite, SitoRuntime, DISINTELLIGENZA (3/4 siti)

```
"showdown": "^2.1.0",
"@types/showdown": "^2.0.6"
```

Converte Markdown in HTML. Pattern di utilizzo:

```
import showdown from 'showdown';
const converter = new showdown.Converter({ tables: true, strikethrough: true });
const html = converter.makeHtml(markdownContent);
```

Non usare mai showdown senza dompurify — l'HTML generato da contenuto non trusted deve essere sanificato.

dompurify — Sanitizzazione XSS

Usato in: SimonePizziWebSite, SitoRuntime (2/4 siti, sempre accoppiato con showdown o Quill)

```
"dompurify": "^3.3.x",
"@types/dompurify": "^3.0.5"
```

DOMPurify rimuove script iniettati e attributi pericolosi dall'HTML:

```
import DOMPurify from 'dompurify';
import showdown from 'showdown';

const converter = new showdown.Converter();
const rawHtml = converter.makeHtml(markdownContent);
const safeHtml = DOMPurify.sanitize(rawHtml);

// Solo ora è sicuro usare dangerouslySetInnerHTML
<div dangerouslySetInnerHTML={{ __html: safeHtml }} />
```

Regola assoluta: showdown + dangerouslySetInnerHTML senza DOMPurify in mezzo è una vulnerabilità XSS. I due vanno sempre in coppia.

react-quill-new — Rich Text Editor WYSIWYG

Usato in: SitoRuntime (1/4 siti — solo dove serve editing visivo avanzato)

```
"react-quill-new": "^3.7.0",
"quill-image-drop-module": "^1.0.3",
"quill-magic-url": "^4.2.0"
```

react-quill-new è il fork mantenuto di react-quill, necessario perché la versione originale non supporta React 19. Output in HTML (non Markdown), quindi richiede comunque DOMPurify lato rendering.

Il modulo `quill-image-drop-module` aggiunge drag & drop di immagini nell'editor. `quill-magic-url` converte automaticamente URL incollati in link cliccabili.

Quando usarlo: Solo quando il progetto richiede un editor visivo tipo Word per i redattori — es. radio web, sito editoriale. Per un portfolio o blog tecnico, un'area textarea con Markdown è più semplice e altrettanto efficace.

4. Animazioni & Effetti Visivi

framer-motion

Usato in: tutti e 4 i siti

```
"framer-motion": "^12.x"
```

La libreria di animazioni declarative per React. Pattern fondamentali nel Modello Universale:

```
import { motion } from 'framer-motion';

// Fade-in all'ingresso del componente
<motion.div
  initial={{ opacity: 0, y: 20 }}
  animate={{ opacity: 1, y: 0 }}
  transition={{ duration: 0.4 }}
>
  {/* contenuto */}
</motion.div>

// Animazioni su lista di elementi (staggered)
<motion.ul variants={containerVariants} initial="hidden" animate="show">
  {items.map(item => (
    <motion.li key={item.id} variants={itemVariants}>...</motion.li>
  ))}
</motion.ul>
```

framer-motion è pesante (~100KB gzip). Giustificato quando le animazioni sono parte integrante dell'identità visiva del sito. Per siti dove le animazioni sono marginali, CSS transitions native sono la scelta corretta.

typewriter-effect

Usato in: SimonePizziWebSite (1/4 siti)

```
"typewriter-effect": "^2.22.0"
```

Effetto macchina da scrivere per headline animati — specifico per portfolio/presentazioni personali.

tailwindcss-animate

Usato in: DISINTELLIGENZA (1/4 siti)

```
"tailwindcss-animate": "^1.0.7"
```

Plugin Tailwind che aggiunge classi di animazione come `animate-in`, `fade-in`, `slide-in-from-bottom`. Alternativa leggera a `framer-motion` per animazioni semplici basate su CSS.

5. SEO & Meta Tag

react-helmet-async

Usato in: SimonePizziWebSite, DISINTELLIGENZA (2/4 siti)

```
"react-helmet-async": "^2.0.5"
```

Gestisce dinamicamente `<title>`, `<meta>`, `<link>` nel `<head>` del documento da qualsiasi componente React:

```
import { Helmet } from 'react-helmet-async';

<Helmet>
  <title>{article.title} - Runtime Radio</title>
  <meta name="description" content={article.excerpt} />
  <meta property="og:image" content={article.cover_image} />
</Helmet>
```

Perché `react-helmet-async` e non `react-helmet`: La versione originale (`react-helmet`) non è più mantenuta e ha problemi di memory leak con React 18+. `react-helmet-async` è il fork attivo e compatibile con React 19.

Limitazione importante: `react-helmet-async` gestisce i meta tag per gli utenti con JavaScript attivo (browser reali). I crawler dei social media e dei motori di ricerca richiedono il **PHP SEO Engine** (Capitolo 11) per ricevere meta tag corretti, perché non eseguono JavaScript.

SitoRuntime non usa `react-helmet-async` perché delega tutto il SEO all'engine PHP lato server.

6. Utilities

clsx + tailwind-merge

Usati in: SimonePizziWebSite, DISINTELLIGENZA

```
"clsx": "^2.1.1",  
"tailwind-merge": "^3.x"
```

`clsx` permette la composizione condizionale di `className`:

```
import { clsx } from 'clsx';  
const classes = clsx('base-class', isActive && 'active', error && 'text-red-500');
```

`tailwind-merge` risolve i conflitti tra classi Tailwind (es. `text-sm text-lg` → `text-lg`):

```
import { twMerge } from 'tailwind-merge';  
const cn = (...inputs) => twMerge(clsx(inputs)); // Pattern helper classico
```

date-fns

Usato in: DISINTELLIGENZA (1/4 siti)

```
"date-fns": "^4.1.0"
```

Utilità per la formattazione e manipolazione delle date. Alternativa a `dayjs` e `moment.js` (quest'ultimo deprecato). Usato in DISINTELLIGENZA per formattare le date del festival e delle iscrizioni nel frontend.

Gli altri siti gestiscono la formattazione date lato PHP (con `date()` e `strtotime()`), riducendo la complessità frontend.

7. Build-Time Optimization

sharp

Usato in: SimonePizziWebSite (come dependency, usato solo in script post-build)

```
"sharp": "^0.34.5"
```

`sharp` è una libreria Node.js per la manipolazione di immagini ad alte prestazioni. Nel contesto del Modello Universale, viene usato nel `clean-dist.js` post-build per ottimizzare automaticamente le immagini nella cartella `dist/` prima del deploy.

```
// Esempio di utilizzo in clean-dist.js  
import sharp from 'sharp';  
await sharp('input.jpg')  
  .resize(1200)  
  .webp({ quality: 80 })  
  .toFile('output.webp');
```

È tecnicamente una dependency invece che devDependency perché il build script (`postbuild`) la richiede al momento della build — ma non viene mai inclusa nel bundle React (non è importata da nessun file `.tsx`).

8. Matrice delle Dipendenze per Sito

Libreria	SitoRuntime	DISINTELLIGENZA	FDCA	SimonePizziWebSite
React 19	✓	✓	✓	✓
react-router-dom v7	✓	✓	✓	✓
framer-motion	✓	✓	✓	✓
showdown	✓	✓	—	✓
dompurify	✓	—	—	✓
lucide-react	✓	✓	✓	✓
Tailwind v3	✓	—	—	—
Tailwind v4	—	✓	✓	✓
@tailwindcss/typography	✓	✓	✓	✓
react-helmet-async	—	✓	✓	✓
react-quill-new	✓	—	—	—
clsx + tailwind-merge	—	✓	✓	✓
sharp	—	—	—	✓
date-fns	—	✓	✓	—
typewriter-effect	—	—	—	✓
tailwindcss-animate	—	✓	✓	—

9. Regole per i Nuovi Progetti

1. **Parti dal minimo:** Core stack + framer-motion + lucide-react. Aggiungi solo quando la funzionalità è richiesta concretamente.
2. **showdown richiede sempre dompurify:** Mai usare l'uno senza l'altro quando il contenuto è user-generated o proviene dal DB.
3. **react-quill-new solo per editor WYSIWYG:** Per semplici aree di testo Markdown, un `<textarea>` è sufficiente e mantiene il bundle piccolo.
4. **react-helmet-async per SEO client-side, PHP engine per SEO crawler:** I due si complementano — non sono alternativi.

5. **Tailwind v4 per nuovi progetti:** La configurazione è più semplice. Verificare la compatibilità delle classi se si migra da v3.

Capitoli correlati: Cap 2 (Struttura Progetto) per la configurazione Vite, Cap 7 (Media & Optimization) per sharp nel dettaglio, Cap 11 (SEO Pre-rendering) per il rapporto tra react-helmet-async e PHP engine.

Prossimo Capitolo: Backend Logic (PHP) - CRUD unificato, gestione dei buffer e sanitizzazione degli input.

I Componenti

CAPITOLO 5: Backend Logic (PHP) (v1.2 - ADVANCED)

Il backend del miniCMS funge da motore di elaborazione dati e guardiano della sicurezza. Questa sezione definisce gli standard per la manipolazione delle stringhe, l'elaborazione dei file e la gestione delle risposte.

1. Gestione degli Identificatori (Slug Logic)

Gli URL devono essere parlanti (SEO-friendly) e univoci. Il Modello Universale impone una strategia di generazione e collisione lato server.

1.1 Algoritmo Base

```
function createSlug($string) {  
    // Minuscolo, trim e rimozione caratteri non alfanumerici (eccetto trattino)  
    $slug = strtolower(trim(preg_replace('/[^A-Za-z0-9-]+/', '-', $string)));  
    return $slug;  
}
```

1.2 Algoritmo Avanzato (con Normalizzazione Accenti Italiani)

Il pattern base produce slug malformati con parole italiane che contengono accenti (es. "caffè" → "caff-"). Il pattern avanzato, implementato in **SimonePizziWebSite**, risolve il problema con una mappa esplicita:

```
function generateSlug($title, $pdo) {  
    // Mappa esplicita accenti italiani e francesi comuni  
    $accents = ['à', 'è', 'é', 'ì', 'ò', 'ù', 'À', 'È', 'É', 'Ì', 'Ò', 'Ù',  
               'â', 'ê', 'î', 'ô', 'û', 'ä', 'ë', 'ï', 'ö', 'ü'];  
    $replacements = ['a', 'e', 'e', 'i', 'o', 'u', 'a', 'e', 'e', 'i', 'o', 'u',  
                    'a', 'e', 'i', 'o', 'u', 'a', 'e', 'i', 'o', 'u'];  
  
    $title = str_replace($accents, $replacements, $title);  
    $slug = strtolower(trim(preg_replace('/[^A-Za-z0-9-]+/', '-', $title)));  
  
    // Anti-collisione
```

```
$stmt = $pdo->prepare("SELECT COUNT(*) FROM articles WHERE slug = ?");
$stmt->execute([$slug]);
if ($stmt->fetchColumn() > 0) {
    $slug .= '-' . time();
}
return $slug;
}
```

Usare sempre il pattern avanzato per siti con contenuto italiano.

1.3 Strategia Anti-Collisione

In fase di creazione, il backend deve verificare l'esistenza dello slug. Se presente, viene aggiunto un suffisso temporale per garantire l'unicità senza errori di database.

```
$stmt = $pdo->prepare("SELECT count(*) FROM news WHERE slug = ?");
$stmt->execute([$slug]);
if ($stmt->fetchColumn() > 0) {
    $slug .= '-' . time(); // Aggiunge timestamp per unicità assoluta
}
```

2. Gestione del Fuso Orario (Timezone Forcing)

Su hosting internazionali (es. server a Los Angeles), `date()` e `time()` usano il timezone del server. Questo causa problemi nella logica `published_at <= NOW()` per siti italiani.

Soluzione standard (implementata in `SimonePizziWebSite` e `SitoRuntime`):

```
// All'inizio di ogni endpoint con logica temporale
date_default_timezone_set('Europe/Rome');
$ita_now_str = date('Y-m-d H:i:s'); // Ora italiana per confronti SQL
$ita_now_time = time();             // Timestamp Unix italiano
```

Questa istruzione va inserita in ogni endpoint che filtra per `published_at` o che genera timestamp di creazione.

3. Gestione dei Metodi HTTP (RESTful Completo)

Il Modello Universale supporta tutti i metodi HTTP standard per API RESTful:

- **GET**: Lettura. Supporta `page`, `limit`, `slug`, `id`, `category`, `admin=true`.
- **POST**: Creazione di nuove risorse.
- **PUT**: Sostituzione completa di una risorsa esistente (tutti i campi).

- **PATCH:** Aggiornamento parziale (singolo campo: es. toggle `is_visible`, aggiornamento `sort_order`).
- **DELETE:** Eliminazione di una risorsa.

```
$method = $_SERVER['REQUEST_METHOD'];

if ($method === 'GET') { /* ... */ }
elseif ($method === 'POST') { Auth::check(); /* ... */ }
elseif ($method === 'PUT') { Auth::check(); /* ... */ }
elseif ($method === 'PATCH') { Auth::check(); /* ... */ }
elseif ($method === 'DELETE') { Auth::check(); /* ... */ }
```

Nota: In progetti più vecchi (DISINTELLIGENZA, SitoRuntime prima del refactor), tutte le operazioni di modifica usavano POST con un campo `act_i` nel body. Il pattern RESTful con metodi separati è più leggibile e permette al frontend TypeScript di essere più espressivo.

4. Il Pattern `auth_helper.php` (Autenticazione Inline)

SimonePizziWebSite introduce un pattern più snello per la gestione dell'autenticazione: un file `auth_helper.php` che incapsula `session_start()`, l'header `Content-Type` e la classe `Auth` in un unico include:

```
<?php
// auth_helper.php
require_once 'db.php';
session_start();
header('Content-Type: application/json');

class Auth {
    public static function check() {
        if (!isset($_SESSION['user_id'])) {
            http_response_code(401);
            echo json_encode(['status' => 'error', 'message' => 'Non autorizzato']);
            exit;
        }
    }
}
```

Utilizzo in ogni endpoint:

```
require_once 'auth_helper.php'; // Gestisce session_start() e Content-Type per tutti

// Nelle route protette:
Auth::check();
```

Vantaggio: `session_start()` e `header()` vengono chiamati una volta sola nel file helper — meno rischi di “headers already sent” dovuti a spazi o BOM nel file PHP.

5. Elaborazione Media e Ottimizzazione Immagini

Il caricamento di un file non è un semplice spostamento (`move_uploaded_file`), ma un processo di trasformazione.

5.1 Polimorfismo dei Percorsi

Il sistema deve mappare il `type` di upload a percorsi fisici e pubblici differenti, applicando logiche di sicurezza specifiche per ogni categoria: - **images/**: Accessibili pubblicamente, ridimensionamento automatico. - **audio/podcasts/**: Solo caricamento via Admin. - **audio/participants/**: Caricamento aperto (se abilitato), cartella isolata.

5.2 Auto-Resize & Trasparenza

Ogni immagine caricata dall'admin deve essere normalizzata (es. max 1920px) per preservare lo spazio su disco e la velocità di caricamento del frontend. - **PNG/WebP**: Il backend deve preservare il canale Alpha (`imagealphablending`, `imagesavealpha`). - **Qualità**: Standard fissato a 85% per JPEG/WebP per un bilanciamento ottimale peso/qualità.

6. Sicurezza dell'Input e Output

- **Sanitizzazione Nomi**: Ogni file caricato deve essere rinominato usando `uniqid()` e pulito da caratteri speciali per evitare vulnerabilità di esecuzione script.
- **JSON Integrity**: Ogni risposta deve essere preceduta da header (`'Content-Type: application/json'`). In caso di errore, deve essere inviato il codice HTTP corretto (400, 401, 403, 500) unito a un messaggio JSON descrittivo.
- **FILTER_SANITIZE_STRING deprecato** (PHP 8.1+): Usare `strip_tags(trim($var))` in alternativa.

7. CORS Management

SitoRuntime include un file dedicato `cors.php` per la gestione degli header CORS, utile quando il frontend è servito da un dominio diverso dall'API (es. sviluppo locale con Vite proxy):

```
<?php
// cors.php — da includere prima di qualsiasi output nelle API pubbliche
header("Access-Control-Allow-Origin: *");
header("Access-Control-Allow-Methods: GET, POST, OPTIONS");
header("Access-Control-Allow-Headers: Content-Type, Authorization,
↪ X-Requested-With");
header('Content-Type: application/json');
```

```
if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {  
    http_response_code(200);  
    exit(0); // Risponde al preflight e termina  
}
```

Nota: In produzione con frontend e API sullo stesso dominio, questo file non è necessario. Limitare Access-Control-Allow-Origin: * all'ambiente di sviluppo; in produzione specificare il dominio esatto.

8. Buffer Management

Per evitare che errori PHP (Notice/Warning) sporchino l'output JSON rendendolo invalido per il frontend, il Modello Universale suggerisce l'uso di `ob_start()` o la disattivazione dei log a schermo in produzione (`display_errors = 0`).

Prossimo Capitolo: Frontend Bridge (API.ts) - La connessione tipizzata tra React e PHP, il pattern Double Read.

CAPITOLO 6: Frontend Bridge (API.ts) (v1.1 - ADVANCED)

Il “Bridge” tra React e PHP è il punto di snodo critico per la stabilità dell’applicazione. Questa sezione definisce gli standard per la gestione delle chiamate asincrone, la cattura degli errori parlanti e la sincronizzazione dello stato.

1. Architettura del Wrapper Universale

Invece di chiamate `fetch` isolate, il sistema utilizza un oggetto `api` centralizzato che incapsula la logica di base degli URL e degli header.

1.1 Il Pattern “Double Read” (Response Cloning)

Per estrarre messaggi di errore dal server senza perdere la possibilità di gestire lo stato HTTP, il Modello Universale impone il clonaggio della risposta:

```
async function handleResponse(res: Response) {
  if (!res.ok) {
    let errorMessage = 'Errore imprevisto dal server';
    try {
      const json = await res.clone().json(); // Clona lo stream
      errorMessage = json.message || errorMessage;
    } catch (e) { /* Ignora errori di parsing */ }
    throw new Error(errorMessage);
  }
  return await res.json();
}
```

2. Polimorfismo delle Chiamate

Il Bridge deve adattarsi al tipo di dato inviato, distinguendo tra payload JSON e flussi binari (multi-part).

- **JSON Standard:** Utilizzato per configurazioni, news e login. Richiede `JSON.stringify(body)`.
- **FormData:** Obbligatorio per upload di file o form complessi (es. iscrizioni con allegati). Non deve essere stringhizzato; va passato direttamente alla `fetch`.

3. Sincronizzazione dello Stato Globale

3.1 Silent Auth Check

L'operazione di `checkAuth` deve essere “silenziosa”: non deve lanciare eccezioni in caso di fallimento, ma restituire `null`. Questo permette all'applicazione di decidere se reindirizzare l'utente o permettere la navigazione pubblica senza interrompere il flusso di rendering.

3.2 Hard Logout

Per garantire che nessun dato sensibile rimanga negli stati di React (Context, Redux, etc.), l'azione di logout deve concludersi con un `window.location.reload()`, resettando l'intero ambiente di esecuzione del browser.

4. Validazione Incrociata (HTTP vs Logic)

Alcune risposte del backend potrebbero essere tecnicamente corrette (HTTP 200) ma contenere fallimenti logici. Il Bridge deve validare entrambi i livelli:

```
const data = await res.json();  
if (data.status === 'error') throw new Error(data.message);  
return data;
```

5. TypeScript Integration (Type Safety)

Ogni metodo dell'oggetto API deve (dove possibile) essere tipizzato. Questo garantisce che il frontend conosca esattamente la struttura dei dati (es. `NewsArticle[]`, `UserRole`, `StatsResponse`) riducendo errori di runtime dovuti a proprietà mancanti o rinominate.

Prossimo Capitolo: Media & Optimization - Caching, ridimensionamento e SEO.

CAPITOLO 7: Media & Optimization (v1.2 - ADVANCED)

Per garantire un'esperienza utente istantanea e un'indicizzazione SEO perfetta, il Modello Universale adotta strategie di caching aggressivo e pre-rendering dei metadati.

1. Architettura di Caching (The TTL Strategy)

Il database non deve essere interrogato per query di sola lettura ripetitive. Il sistema implementa un caching basato su file JSON.

1.1 Cache Paginata (Listings)

Ogni richiesta di lista (es. `news.php?page=1`) genera un file univoco in `.cache/` (es. `news_p1_l10.json`). - **Validità:** Il file viene servito direttamente se creato meno di 300 secondi fa. - **Header di Tracciamento:** Il sistema deve inviare `X-Cache: HIT` o `X-Cache: MISS` per monitorare l'efficienza del sistema.

1.2 Invalidazione Intelligente

Ad ogni operazione di scrittura (POST), il backend deve pulire la cartella `.cache/` per garantire che il pubblico veda immediatamente le nuove modifiche, evitando il "vizio" dei dati obsoleti.

2. SEO & Social Pre-rendering

Le Single Page Application (SPA) hanno problemi di SEO. Il Modello Universale risolve questo problema con un **PHP Entry-Point** (non una cache file JSON separata come in approcci precedenti).

2.1 Il Meccanismo (Riferimento Completo: Capitolo 11)

L'`index.php` nella root pubblica intercetta tutte le richieste, estrae lo slug dall'URL, interroga direttamente il database SQLite, e inietta i meta tag `og:title`, `og:description` e `og:image` nell'HTML generato da Vite prima di servirlo. I bot di social e i motori di ricerca ricevono HTML completo con meta tag corretti.

Flusso: 1. Apache riceve richiesta `/categoria/mio-slug` 2. `.htaccess` → nessun file fisico trovato → passa a `index.php` 3. `index.php` interroga SQLite per `slug = 'mio-slug'` 4. Inietta meta tag nell'HTML di Vite via `str_replace('</head>', $seoBlock, $html)` 5. Serve HTML completo al browser/bot

2.2 Il Rebuild Cache SEO (Per Retrocompatibilità)

SitoRuntime include ancora `rebuild_seo_cache.php`, uno script di manutenzione che rigenera file JSON di cache SEO per entità pre-esistenti nel database. Utile per migrazioni e fix straordinari. Da eliminare dopo l'uso in produzione.

3. Ottimizzazione Media & Compression

3.1 Normalizzazione delle Immagini (PHP GD)

Non è permesso servire immagini “grezze” caricate dall'utente. Il backend PHP deve: - Ridimensionare a max 1920px (larghezza) o 1080px (altezza). - Applicare una compressione del 85% per JPEG/WebP. - Convertire (opzionale) in formati moderni come WebP per ridurre il peso del 30-50% rispetto al JPEG.

3.2 Sharp (Node.js) per Image Processing Build-Time

SimonePizziWebSite introduce sharp come dipendenza Node.js per il processing di immagini in fase di build o come utility script. Sharp è più performante di PHP GD per batch processing e supporta formati moderni (AVIF, WebP) con qualità superiore. Non è un'alternativa al backend PHP per upload live — è complementare per pre-processing di asset statici.

```
// package.json
"dependencies": {
  "sharp": "^0.34.5"
}
```

3.3 Cache Control sui File Statici

Tramite `.htaccess`, i file nella cartella `uploads/` devono essere serviti con header di cache a lungo termine (`Expires`, `Cache-Control: max-age=31536000`), poiché una volta caricati non cambiano mai nome (grazie all'`hash uniqid()`).

```
<IfModule mod_expires.c>
  ExpiresActive On
  <FilesMatch "\.(jpg|jpeg|png|gif|webp|svg|ico)$">
    ExpiresDefault "access plus 1 year"
```

```
Header set Cache-Control "max-age=31536000, public"  
</FilesMatch>  
</IfModule>
```

4. Manutenzione e Rebuilding

Il sistema deve includere script protetti (`rebuild_seo_cache.php`, `optimize_db.php`) per operazioni di manutenzione straordinaria: - **rebuild_seo_cache.php**: Rigenera i file JSON SEO per tutte le entità del database. Da eseguire dopo una migrazione o un cambio di struttura metadati. - **optimize_db.php**: Esegue VACUUM e ANALYZE sul database. Riduce il peso del file e ottimizza i piani di query.

Entrambi vanno eliminati dal server dopo l'uso — includono output HTML non JSON e possono esporre informazioni interne se accessibili pubblicamente.

Prossimo Capitolo: Advanced Content Editing & Media Integration - L'editor definitivo.

CAPITOLO 8: Advanced Content Editing & Media Integration (v1.1 - ADVANCED)

Il Modello Universale (miniCMS) eleva l'esperienza di editing trasformandola in un centro di controllo multimediale integrato. Questa sezione definisce gli standard per la gestione degli asset, l'editing del testo e l'integrazione fluida tra i due mondi.

1. Il Media Center Centralizzato

L'architettura dei media non è un semplice file browser, ma un gestore di stato con logiche di antepri-ma e azioni di massa.

1.1 Filtraggio Contestuale (Tab Strategy)

I media vengono organizzati in base alla loro natura e destinazione, filtrati dinamicamente tramite i percorsi del file system: - **Immagini**: Filtrate per mime-type `image/*`. - **Audio Partecipanti**: Isolati nella sottocartella `/participants/`. - **Audio Podcast**: Riservati alla cartella `/podcasts/` con permessi di scrittura limitati agli Admin.

1.2 Gestione Asset Avanzata

Il sistema deve implementare funzioni di manutenzione proattiva: - **Bulk Delete**: Selezione multipla e cancellazione atomica per la pulizia del server. - **Audio Engine**: Player integrato (`Audio API`) per il pre-ascolto istantaneo direttamente dalla griglia media. - **Visual Feedback**: Visualizzazione del peso dei file (`formatBytes`) per sensibilizzare l'editor sull'occupazione del disco.

2. Il Componente MediaPlayer (Integration Layer)

Il MediaPlayer è il ponte tra l'editor di news e la libreria media. - **Modal Logic**: Deve aprirsi in sovrapposizione (Overlay) senza perdere lo stato del form sottostante. - **Search & Filter**: Ricerca testuale in tempo reale sulla lista dei file e filtraggio automatico per il tipo di dato richiesto dal form (es. solo immagini per la cover, solo audio per il podcast). - **Selection Callback**: Restituzione dell'URL relativo al componente genitore, con chiusura automatica del modale.

3. Rich Text Editor & Sanitizzazione

L'editor di testo è il cuore dell'interfaccia. Per garantire la pulizia del codice e la coerenza del layout:

- **Paste Protection (Advanced):** Intercettazione dell'evento `paste` con rimozione di stili inline, classi CSS estranee e attributi non necessari (tipici di Word o Wikipedia). - **Markdown Support:** Convertitore integrato per trasformare automaticamente testi formattati in Markdown nel corrispondente HTML semantico durante l'incollaggio. - **State Reset (Key Strategy):** L'uso di `key={id}` nel componente React per garantire la pulizia dei buffer interni e l'inizializzazione corretta del cursore ad ogni cambio di contenuto.

4. Gestione degli URL e Portabilità

- **Relative Path Strategy:** Nel database vengono salvati solo percorsi relativi (es. `/api/uploads/file.jpg`).
- **Absolute Resolver:** In fase di condivisione o copia (pulsante "Copia URL"), il sistema trasforma il percorso in URL assoluto (`window.location.origin + path`) per garantire che i link funzionino anche al di fuori dell'applicazione.

5. UX e Feedback Visivo

- **Miniature (Lazy Loading):** Caricamento differito delle miniature nella griglia media per non appesantire il browser in librerie con centinaia di file.
- **Transizioni:** Uso di animazioni CSS (`animate-in`, `fade-in`) per rendere fluida l'apertura dei modali e il caricamento delle liste.

Conclusione: Il Modello Universale miniCMS è ora un sistema completo, sicuro e scalabile, pronto per essere impiegato come standard universale.

Prossimo Capitolo: Content Lifecycle - Il ciclo di vita dei contenuti, dalla bozza alla pubblicazione programmata, con il pattern di bypass admin.

Il Flusso Operativo

CAPITOLO 9: Content Lifecycle (v1.2 - ADVANCED)

Il Modello Universale trasforma un semplice database in un sistema editoriale dinamico. Questa sezione definisce la logica degli stati, la programmazione temporale e il feedback visivo necessario per una gestione professionale del palinsesto.

1. Stati Dinamici vs Stati Persistenti

Mentre il database salva uno stato “statico” (*status*), l'applicazione deve calcolare uno stato “dinamico” basato sul tempo.

1.1 Matrice degli Stati

DB Status	Data di Pubblicazione	Stato Reale (UI)	Descrizione
draft	<i>Qualsiasi</i>	BOZZA	Mai visibile al pubblico.
published	> NOW()	PROGRAMMATO	Visibile solo agli Admin, in attesa del momento giusto.
published	<= NOW()	PUBBLICATO	Visibile a tutti.

2. Implementazione della “Programmazione Reale”

La programmazione non richiede cron job. La visibilità è controllata dalla query SQL e riflessa nella logica React.

2.1 Logica Frontend (React)

Per fornire feedback immediato all'amministratore, la tabella di gestione deve calcolare lo stato al volo:

```
const isDraft = item.status === 'draft';
const isScheduled = item.status === 'published' && new Date(item.published_at) > new
  Date();
const isPublished = item.status === 'published' && new Date(item.published_at) <=
  new Date();
```

2.2 Normalizzazione Data/Ora (Input Strategy)

Il formato `datetime-local` del browser utilizza il separatore T, mentre il DB richiede uno spazio. Lo standard impone la conversione bidirezionale nel componente di editing:

```
// Da DB a UI
value={published_at.replace(' ', 'T').slice(0, 16)}

// Da UI a DB
onChange={e => setPublishedAt(e.target.value.replace('T', ' ') + ':00')}
```

3. Workflow Editoriale e Integrità

- **Auto-Slug:** Lo slug deve essere rigenerato solo alla creazione o se esplicitamente richiesto, per evitare di rompere i link esistenti (SEO integrity) in caso di modifica del titolo.
- **Rich Text Reset:** Durante il passaggio tra la modifica di due contenuti diversi, il componente editor deve essere forzatamente rimosso e reinserito (`key={item.id}`) per pulire i buffer interni e prevenire perdite di dati cross-articolo.
- **Anteprima Immediata:** Il form deve mostrare una miniatura (preview) dell'immagine di copertina selezionata, con possibilità di rimozione istantanea prima del salvataggio.

4. Gestione Multicategoria e Tagging

Il sistema deve supportare la categorizzazione fissa (dropdown) unita al tagging libero (input a virgola), permettendo al frontend di filtrare i contenuti con logiche di “Related Content” basate sui tag comuni.

5. Security & Visibility (Bypass Logic)

Gli endpoint API devono implementare una logica di “Bypass” per gli utenti autenticati: - **Pubblico:** `WHERE status = 'published' AND published_at <= NOW()` - **Admin:** nessun filtro — lista completa con bozze e articoli programmati nel futuro

Questo permette all'amministratore di testare la visualizzazione di un articolo programmato semplicemente navigando sul sito dopo aver effettuato il login, senza alcun meccanismo di preview separato.

5.1 Bypass per Singolo Articolo (via Slug)

L'endpoint recupera sempre l'articolo dal DB indipendentemente dallo stato. La decisione di mostrarlo o meno avviene *dopo* il fetch, in base alla sessione:

```
// GET /api/articles.php?slug=titolo-articolo
$stmt = $pdo->prepare("SELECT * FROM articles WHERE slug = ?");
$stmt->execute([$GET['slug']]);
$article = $stmt->fetch();

if ($article) {
    $is_admin = isset($_SESSION['user_id']); // Sessione attiva = admin

    // Articolo pubblico: status=published E data nel passato/presente
    $is_published = $article['status'] === 'published' &&
        (empty($article['published_at']) ||
        ↪ strtotime($article['published_at']) <= $ita_now_time);

    // Fingere il 404 – non 403 – per non rivelare l'esistenza di bozze a utenti non
    ↪ autenticati
    if (!$is_admin && !$is_published) {
        http_response_code(404);
        echo json_encode(['error' => 'Articolo non trovato']);
        exit;
    }

    echo json_encode($article); // Admin vede tutto: bozze, programmati, pubblicati
}
```

Principio chiave: il 404 simulato (anziché il 403) è una misura di sicurezza deliberata. Un 403 confermerebbe l'esistenza del contenuto; un 404 lo nega.

5.2 Bypass per Lista (via Parametro Admin)

Per la dashboard amministrativa, che deve mostrare *tutti* gli articoli incluse le bozze, si usa un parametro esplicito combinato con il controllo sessione:

```
// GET /api/articles.php?admin=true (solo da dashboard autenticata)
$is_admin_dashboard = isset($_SESSION['user_id']) &&
    isset($_GET['admin']) &&
    $_GET['admin'] === 'true';

if (!$is_admin_dashboard) {
    // Filtri pubblici: solo pubblicati e già usciti
    $conditions[] = "status = 'published'";
    $conditions[] = "(published_at IS NULL OR published_at = '' OR published_at <=
    ↪ ?)";
    $params[] = $ita_now_str; // Stringa datetime calcolata all'inizio dello script
}
// Se $is_admin_dashboard === true: nessun filtro, lista completa
```

Il doppio controllo (`session + parametro`) è fondamentale: il parametro `?admin=true` da solo sarebbe bypassabile da chiunque.

5.3 Accesso per ID (Editor Admin)

Un terzo pattern — fetch per ID anziché slug — serve esclusivamente al form di editing in dashboard. Richiede autenticazione obbligatoria:

```
// GET /api/articles.php?id=42 (solo admin editor)
if (isset($_GET['id'])) {
    Auth::check(); // Termina con 401 se non autenticato
    $stmt = $pdo->prepare("SELECT * FROM articles WHERE id = ?");
    $stmt->execute($_GET['id']);
    // ...
}
```

Questo endpoint bypassa il filtro sullo stato perché è progettato per caricare nel form anche le bozze mai pubblicate.

Prossimo Capitolo: Security & Auth - Gestione sessioni, ruoli e protezione avanzata.

CAPITOLO 10: Security & Auth (v1.2 - ADVANCED)

La sicurezza nel Modello Universale non è un'opzione, ma un'architettura stratificata che protegge l'integrità dei dati e la privacy degli utenti attraverso controlli sia lato client che lato server.

1. Gestione della Sessione (Stateful-Sec)

Il sistema utilizza sessioni PHP native, ma con parametri di sicurezza moderni per prevenire il dirottamento (Session Hijacking).

1.1 Configurazione Server-Side

Prima di ogni `session_start()`, il sistema deve configurare i cookie per essere inaccessibili agli script JavaScript (prevenzione XSS):

```
ini_set('session.cookie_httponly', 1);
ini_set('session.cookie_secure', 1); // Solo se HTTPS
ini_set('session.cookie_samesite', 'Lax');
session_start();
```

1.2 Endpoint check_auth

Il frontend non deve mai memorizzare la password o lo stato di login in `localStorage`. Deve invece interrogare periodicamente il server per confermare che la sessione sia ancora valida:

```
if (isset($_SESSION['user_id'])) {
    echo json_encode([
        'status' => 'success',
        'user' => [
            'username' => $_SESSION['username'],
            'role' => $_SESSION['role']
        ]
    ]);
}
```

2. Architettura Protected Routes (React)

L'accesso all'area Admin è protetto da un **Higher-Order Component** o un **Layout Wrapper** (`AdminLayout.tsx`).

2.1 Bootstrapping della Sicurezza

All'interno del `useEffect` principale, l'app verifica l'identità. Se il server risponde con `401 Unauthorized`, il frontend distrugge lo stato locale e reindirizza istantaneamente alla pagina di login, impedendo flash di contenuti sensibili.

2.2 Role-Based UI (RBAC)

La sidebar e le rotte admin vengono generate dinamicamente in base al ruolo ricevuto dal server: - **Admin**: Accesso a Gestione Utenti, Impostazioni di Sistema, Reset Database. - **Editor**: Accesso limitato a News, Media e Podcast.

3. Sicurezza Crittografica

- **Hashing**: Le password vengono salvate esclusivamente tramite `password_hash($pass, PASSWORD_DEFAULT)`.
- **Verifica**: In fase di login, si utilizza `password_verify($pass, $hash)`. Il sistema non conosce mai la password in chiaro dell'utente.
- **Brute Force Mitigation**: Gli endpoint di login devono implementare un ritardo artificiale (`sleep(1)`) in caso di errore per rallentare i tentativi di attacco automatizzato.

4. Protezione del Database SQLite

Il database fisico (`.sqlite`) deve essere protetto da accesso diretto tramite URL. - **Strategia 1**: Posizionamento del file in una cartella esterna alla root pubblica (es. `../data/`). - **Strategia 2**: Se all'interno della root, protezione tramite `.htaccess` (`Deny from all`) e scelta di nomi file non prevedibili.

5. Sanitizzazione dei Buffer di Output

Per evitare che errori PHP involontari (Notice/Warning) rompano l'output JSON e forniscano indizi ad un attaccante (Path Disclosure), lo standard impone: - `error_reporting(0)`; in produzione. - Cattura delle eccezioni via `try-catch` e restituzione di messaggi di errore "parlanti" ma generici.

6. Caso Reale: L'Attacco DDoS a Runtime Radio (Febbraio 2026)

“I grafici del traffico hanno iniziato a disegnare picchi anomali, simili a pareti verticali.”

Questa sezione documenta un incidente reale accaduto a **Runtime Radio (runtimeradio.com)** tra il 23 e il 27 febbraio 2026. Non è un caso di studio teorico: è una cicatrice viva nel codice e nell'architettura del progetto.

6.1 Il Contesto: Due Crisi Sovrapposte

La settimana del 23 febbraio 2026 ha visto due eventi distinti che si sono sovrapposti in modo devastante:

Crisi 1 — Collasso del Database (23 febbraio) Il traffico crescente sulla piattaforma aveva messo sotto stress il database SQLite per mesi. Il 23 febbraio il sistema ha ceduto: l'infrastruttura database, sottodimensionata rispetto al carico reale, ha smesso di rispondere. *“Il sistema aveva tenuto per mesi, poi è ceduto tutto insieme, e rapidamente.”* In meno di 24 ore il team ha completato la migrazione completa dei dati a MySQL — senza perdita di dati (vedi Capitolo 14 per il processo di migrazione).

Crisi 2 — L'Attacco DDoS (24-27 febbraio) Mentre l'infrastruttura era già in fase di stabilizzazione post-migrazione, è arrivato l'attacco. I server hanno iniziato a restituire errori 503 e 500. Il traffico mostrava picchi anomali, verticali, impossibili da spiegare con la crescita organica degli utenti.

6.2 Il Vettore di Attacco: I Bot dei Social Media

La vulnerabilità sfruttata era elegante e insidiosa. Runtime Radio, come tutti i siti costruiti con il Modello Universale, implementava il **SEO pre-rendering via PHP entry-point** (Capitolo 11): ogni richiesta in arrivo veniva processata da `index.php`, che interrogava il database per estrarre i meta tag OG corretti da servire ai crawler di Telegram, Facebook e X/Twitter.

Questa funzionalità — progettata per migliorare l'esperienza di condivisione dei link sui social — si è trasformata in un'arma. Migliaia di bot ostili, **simulando i crawler dei social media**, hanno bombardato il server con richieste continue. Ogni richiesta forzava una query al database. Il database — appena migrato, ancora in fase di stabilizzazione — ha ceduto sotto il peso.

Bot ostile → simula Telegram/Facebook crawler
→ richiesta a / o /news/slug
→ index.php → query MySQL → meta tag
→ risposta → bot scarta → ripete 1000 volte/secondo
→ MySQL sopraffatto → 503/500

La lezione: qualsiasi endpoint che interroga il database per rispondere a richieste non autenticate è un potenziale bersaglio. I bot dei social media sono riconoscibili dallo user-agent — e possono essere falsificati.

6.3 La Risposta: Tre Livelli di Difesa

La risoluzione è avvenuta in fasi, documentata nell'articolo “*Sopravvivere alla Tempesta*” (27 febbraio 2026):

Fase 1 — Spegnimento d’Emergenza con Degradazione Controllata Il sito principale è stato messo offline, ma lo **streaming audio è rimasto attivo** tramite una pagina statica di manutenzione. Principio fondamentale: il servizio core (la radio) non si interrompe mai, anche durante un attacco. La pagina di manutenzione non interrogava nessun database.

Fase 2 — Cache Precompilata per i Bot La soluzione definitiva ha separato il percorso delle richieste dei bot da quello degli utenti reali: - Le risposte per i crawler social (i tag OG, i meta SEO) vengono ora servite da **file JSON statici precompilati**, scritti nel momento in cui un articolo viene pubblicato o aggiornato. - Il bot riceve la risposta in millisecondi, **senza che il database venga mai interrogato**. - Solo le richieste di utenti reali (browser con JavaScript) ricevono la pagina React completa.

Fase 3 — Sistema Ibrido Leggero e Corazzato L'architettura risultante distingue esplicitamente tra: - **Richieste bot** (identificate dallo user-agent): servite da cache statica, zero DB access. - **Richieste umane**: percorso normale, React SPA + API.

```
// Pattern anti-DDoS: identificazione bot e servizio da cache statica
$userAgent = $_SERVER['HTTP_USER_AGENT'] ?? '';
$isSocialBot =
    ↪ preg_match('/facebookexternalhit|Twitterbot|TelegramBot|LinkedInBot|WhatsApp/i',
    ↪ $userAgent);

if ($isSocialBot && $slug) {
    $cacheFile = __DIR__ . '/api/.cache/seo_' . md5($slug) . '.json';
    if (file_exists($cacheFile)) {
        $seoData = json_decode(file_get_contents($cacheFile), true);
        // Serve meta tag da cache statica — nessuna query DB
        // ... iniezione nell'HTML ...
        exit;
    }
}
// Percorso normale per utenti reali
```

6.4 Le Persone

L'incidente non è stato risolto in solitudine. Carlo Santagostino, Walter Sbrano, Peppe Pugliese e Valerio Galano (del podcast *Pensieri in Codice*) hanno contribuito alla ricostruzione dell'infrastruttura. La comunità tecnica intorno al progetto si è dimostrata parte dell'architettura di resilienza — non solo il codice.

6.5 Le Lezioni da Portare in Ogni Progetto

1. **Ogni endpoint pubblico che interroga il DB è un bersaglio potenziale.** Valutare sempre se la risposta può essere servita da cache statica per le richieste non autenticate.

2. **I bot dei social media possono essere falsificati.** Non fare mai affidamento solo sullo user-agent per decisioni di sicurezza critiche — usarlo solo per ottimizzare le performance (cache bot), mai come gatekeeper di accesso.
3. **Il servizio core deve sopravvivere a qualsiasi crisi.** Progettare sempre una “modalità degradata” che mantenga attiva la funzione principale del sito (streaming, in questo caso) anche quando tutto il resto è offline.
4. **La migrazione del database non è mai un momento sicuro.** Un sistema appena migrato è fragile. Pianificare la migrazione in periodi di basso traffico e avere un piano di emergenza pronto prima di iniziare.
5. **La cache non è solo una ottimizzazione di performance — è un layer di sicurezza.** Un sistema che risponde alle richieste ripetitive senza interrogare il database è intrinsecamente più resiliente agli attacchi volumetrici.

Prossimo Capitolo: SEO Pre-rendering con PHP Entry-Point - Il motore SEO invisibile che trasforma una SPA in un sito indicizzabile.

CAPITOLO 11: SEO Pre-rendering con PHP

Entry-Point (v1.0)

Il problema fondamentale delle Single Page Application (SPA) React è che i bot di Google, Facebook, Twitter e LinkedIn vedono solo un `<div id="root"></div>` vuoto: JavaScript non viene eseguito al momento della scansione. Il Modello Universale risolve questo problema con un **PHP Entry-Point** che inietta i meta tag corretti nell'HTML prima che arrivi al bot, senza rinunciare alla potenza di React.

Questo capitolo documenta il pattern reale implementato in **SimonePizziWebSite (v1.4.0)** e menzionato in forma più semplice in SitoRuntime.

1. Il Problema e la Soluzione

Il Problema

```
Bot Google → richiede /videogiochi/mio-articolo
Server → serve index.html (solo <div id="root">)
Bot → "non c'è contenuto, ignoro"
```

La Soluzione: The Swap Trick

```
Bot Google → richiede /videogiochi/mio-articolo
Server → Apache → index.php (invece di index.html!)
index.php → query SQLite → estrae title, description, image dell'articolo
index.php → legge index.html compilato da Vite → inietta meta tag → serve HTML completo
Bot → "ho trovato contenuto, indexo"
```

2. Il Meccanismo: Build Rename Strategy

Il trick sta nel **rinominare** l'`index.html` generato da Vite in modo che `index.php` possa occupare quella posizione come entry-point principale.

In DISINTELLIGENZA (build script in package.json):

```
"build": "tsc -b && vite build && node clean-dist.js && move dist\\index.html  
↪ dist\\index_react.html"
```

Vite genera `dist/index.html`, lo script di build lo rinomina in `dist/index_react.html`. L'`index.php` nella cartella sorgente viene poi copiato nella `dist/` durante il deploy. In questo modo Apache serve `index.php` come file predefinito.

In SimonePizziWebSite (build script standard):

```
"build": "tsc && vite build",  
"postbuild": "node clean-dist.js"
```

Qui `index.php` è già nella cartella `public/` e Vite compila tutti gli asset statici nella stessa cartella (o la struttura è organizzata diversamente). Il risultato è identico: `index.php` è il file servito da Apache.

3. Il Codice di `index.php` (Analisi Completa)

```
<?php  
/**  
 * SEO ENGINE SERVER-SIDE (v1.4.0)  
 * Intercetta le richieste di React Router, interroga il database,  
 * popola i meta tag HTML (OpenGraph/Twitter) e passa il controllo a React.  
 */  
  
// 1. META DI DEFAULT (Fallback per homepage e rotte sconosciute)  
$protocol = (!empty($_SERVER['HTTPS']) && $_SERVER['HTTPS'] !== 'off' ||  
↪ $_SERVER['SERVER_PORT'] == 443)  
    ? "https://" : "http://";  
$metaTitle      = "Nome Sito - Tagline principale";  
$metaDescription = "Descrizione generale del sito per SEO";  
$metaImage      = $protocol . $_SERVER['HTTP_HOST'] . "/og-default.png";  
$currentUrl     = $protocol . $_SERVER['HTTP_HOST'] . $_SERVER['REQUEST_URI'];  
  
// 2. ESTRAZIONE SLUG DALL'URL  
$request_uri = trim($_SERVER['REQUEST_URI'], '/');  
$uri_parts  = explode('/', $request_uri);  
$slug = null;  
  
// Escludere l'area admin dal match SEO (la SPA gestisce il suo routing)  
if (isset($uri_parts[0]) && $uri_parts[0] === 'admin') {  
    $slug = null;
```

```
} elseif (count($uri_parts) == 2) {
    // URL tipo /categoria/mio-slug → prende l'ultimo segmento
    $slug = end($uri_parts);
} elseif (count($uri_parts) == 1 && $uri_parts[0] !== '') {
    // URL corto tipo /mio-slug
    $slug = $uri_parts[0];
}

// 3. QUERY DATABASE (Connessione diretta, senza includere api/db.php)
$dbPath = __DIR__ . '/api/.data/database.sqlite';

if ($slug && file_exists($dbPath)) {
    try {
        $pdo = new PDO('sqlite:' . $dbPath);
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        // Sanitizzazione: PHP 8.1+ ha deprecato FILTER_SANITIZE_STRING
        $cleanSlug = strip_tags(trim($slug));

        $stmt = $pdo->prepare(
            "SELECT title, excerpt, cover_image
            FROM articles
            WHERE slug = :slug AND status = 'published'
            LIMIT 1"
        );
        $stmt->execute([':slug' => $cleanSlug]);
        $article = $stmt->fetch(PDO::FETCH_ASSOC);

        if ($article) {
            $metaTitle = htmlspecialchars($article['title'], ENT_QUOTES, 'UTF-8') . "
            ↳ | Nome Sito";
            $rawDesc = $article['excerpt'] ?: "Leggi l'articolo completo.";
            $metaDescription = htmlspecialchars(strip_tags($rawDesc), ENT_QUOTES,
            ↳ 'UTF-8');

            if (!empty($article['cover_image'])) {
                $img = filter_var($article['cover_image'], FILTER_SANITIZE_URL);
                // Rendere il path assoluto se relativo (i bot richiedono URL assoluti)
                if (strpos($img, 'http') !== 0) {
                    $prefix = (substr($img, 0, 1) !== '/') ? '/' : '';
                    $metaImage = $protocol . $_SERVER['HTTP_HOST'] . $prefix . $img;
                } else {
                    $metaImage = $img;
                }
            }
        }
    } catch (PDOException $e) {
        // Fallback silenzioso: proseguiamo con i meta default
        error_log("SEO Engine DB Error: " . $e->getMessage());
    }
}

// 4. LETTURA E INIEZIONE NELL'HTML DI VITE
```

```
$htmlFile = __DIR__ . '/index_react.html'; // o index.html se non rinominato

if (!file_exists($htmlFile)) {
    die("Error: Production HTML not found. Eseguire 'npm run build'.");
}

$htmlContent = file_get_contents($htmlFile);

// Costruzione del blocco SEO da iniettare
$seoInjection = "
    <title>{$metaTitle}</title>
    <meta name=\"title\" content=\"{$metaTitle}\" />
    <meta name=\"description\" content=\"{$metaDescription}\" />
    <meta property=\"og:type\" content=\"website\" />
    <meta property=\"og:url\" content=\"{$currentUrl}\" />
    <meta property=\"og:title\" content=\"{$metaTitle}\" />
    <meta property=\"og:description\" content=\"{$metaDescription}\" />
    <meta property=\"og:image\" content=\"{$metaImage}\" />
    <meta property=\"twitter:card\" content=\"summary_large_image\" />
    <meta property=\"twitter:url\" content=\"{$currentUrl}\" />
    <meta property=\"twitter:title\" content=\"{$metaTitle}\" />
    <meta property=\"twitter:description\" content=\"{$metaDescription}\" />
    <meta property=\"twitter:image\" content=\"{$metaImage}\" />
</head>";

// Operazione chirurgica: rimuove il <title> di Vite e inietta il pacchetto SEO
$htmlContent = preg_replace('/<title>.*?</title>/s', '', $htmlContent, 1);
$htmlContent = str_replace('</head>', $seoInjection, $htmlContent);

echo $htmlContent;
?>
```

4. Punti Critici del Pattern

4.1 Connessione Diretta al DB (No Include Esterno)

`index.php` non deve fare `require_once 'api/db.php'`. Connette direttamente tramite PDO per due motivi: - Semplicità: evita dipendenze da path relativi che cambiano in base alla posizione del file. - Performance: connessione read-only, niente sessioni, niente CORS headers.

4.2 Sanitizzazione dello Slug

L'uso di `strip_tags(trim($slug))` al posto del deprecato `FILTER_SANITIZE_STRING` garantisce: - Compatibilità PHP 8.1+. - Rimozione di eventuali tag HTML iniettati via URL.

4.3 Fallback Silenzioso

Il blocco `try-catch` intorno alla query `NON` mostra errori all'utente. Se il DB non è raggiungibile, serve i meta default. Questo è fondamentale: un errore del SEO Engine non deve rompere l'intera applicazione.

4.4 Bypass Admin

Le rotte `/admin/*` non vengono processate dal motore SEO. L'area admin non ha bisogno di Open Graph e include già il proprio ciclo di autenticazione React.

4.5 Immagine Assoluta

I bot di social richiedono URL assoluti per `og:image`. La logica trasforma i path relativi (come `/api/uploads/copertina.jpg`) in `https://dominio.com/api/uploads/copertina.jpg`.

5. Estensione a Più Entità

Il pattern si estende facilmente a più tipi di contenuto. Basta aggiungere query successive:

```
// Prima: cercare tra gli articoli
$stmt = $pdo->prepare("SELECT title, excerpt, cover_image FROM articles WHERE slug=?
    ↳ AND status='published' LIMIT 1");

// Poi: se non trovato, cercare tra i progetti
if (!$article) {
    $stmt = $pdo->prepare("SELECT name as title, description as excerpt, cover_image
    ↳ FROM projects WHERE slug=? AND is_visible=1 LIMIT 1");
}
```

6. Configurazione Apache (.htaccess)

Il `.htaccess` del Modello Universale (Capitolo 2) gestisce già il routing. Apache serve `index.php` come documento predefinito se presente, prima di `index.html`. Non è necessario alcun aggiornamento delle regole `RewriteRule`.

Prossimo Capitolo: RSS Feed & Syndication - Come generare feed XML per aggregatori e podcast app.

CAPITOLO 12: RSS Feed & Syndication (v1.0)

Il feed RSS è il canale di distribuzione automatica dei contenuti verso aggregatori, lettori di notizie, podcast app e motori di ricerca. Tutti i siti del Modello Universale implementano almeno un feed: **SimonePizziWebSite** (rss.php), **SitoRuntime** (feed_news_rss.php), **DISINTELLIGENZA** e **FDCA** (feed.php). Questo capitolo consolida il pattern standard.

1. Struttura del Feed RSS Standard

Un feed RSS valido richiede un header HTTP preciso e una struttura XML rigorosa:

```
<?php
require_once 'db.php';

// Header fondamentale: senza questo, i lettori RSS rifiutano il feed
header('Content-Type: application/rss+xml; charset=utf-8');

$pdo = Database::connect();

// Recupero dinamico del protocollo e del dominio
$protocol = (isset($_SERVER['HTTPS']) && $_SERVER['HTTPS'] === 'on') ? 'https' :
    ↪ 'http';
$host = $_SERVER['HTTP_HOST'];
$base_url = $protocol . '://' . $host;

// Forzatura del fuso orario per siti su hosting internazionali (es. Los Angeles)
date_default_timezone_set('Europe/Rome');
$now_str = date('Y-m-d H:i:s');

define('RSS_FEED_LIMIT', 50); // Numero massimo di articoli nel feed

$site_title = "Nome del Sito";
$site_description = "Descrizione del sito per il feed RSS";

echo '<?xml version="1.0" encoding="UTF-8" ?>' . "\n";
echo '<rss version="2.0">' . "\n";
echo '<channel>' . "\n";
echo '  <title>' . htmlspecialchars($site_title) . '</title>' . "\n";
echo '  <link>' . htmlspecialchars($base_url) . '</link>' . "\n";
echo '  <description>' . htmlspecialchars($site_description) . '</description>' .
    ↪ "\n";
echo '  <language>it-IT</language>' . "\n";
```

```
try {
    $stmt = $pdo->prepare(
        "SELECT title, slug, excerpt, content, cover_image, category, published_at
        FROM articles
        WHERE status = 'published'
        AND (published_at IS NULL OR published_at = '' OR published_at <= :now)
        ORDER BY published_at DESC
        LIMIT " . RSS_FEED_LIMIT
    );
    $stmt->execute([':now' => $now_str]);
    $articles = $stmt->fetchAll();

    foreach ($articles as $article) {
        // Formato data secondo RFC 822 (obbligatorio per RSS 2.0)
        $pubDate = date(DATE_RSS, strtotime($article['published_at'] ?? 'now'));
        $item_url = $base_url . '/' . rawurlencode($article['category']) . '/' .
            rawurlencode($article['slug']);

        echo ' <item>' . "\n";
        echo ' <title>' . htmlspecialchars($article['title']) . '</title>' . "\n";
        echo ' <link>' . htmlspecialchars($item_url) . '</link>' . "\n";
        echo ' <description>' . htmlspecialchars($article['excerpt']) .
            "</description>" . "\n";
        echo ' <pubDate>' . $pubDate . '</pubDate>' . "\n";
        echo ' <guid>' . htmlspecialchars($item_url) . '</guid>' . "\n";

        // Enclosure per immagine (utile per aggregatori che mostrano preview)
        if (!empty($article['cover_image'])) {
            $img_url = str_starts_with($article['cover_image'], 'http')
                ? $article['cover_image']
                : $base_url . '/' . ltrim($article['cover_image'], '/');
            echo ' <enclosure url="' . htmlspecialchars($img_url) . '"
                type="image/jpeg" />' . "\n";
        }

        echo ' </item>' . "\n";
    }
} catch (Exception $e) {
    // Fallback silenzioso: il feed rimane valido anche senza articoli
}

echo '</channel>' . "\n";
echo '</rss>';
```

2. Dettagli Tecnici Critici

2.1 Il Problema del Fuso Orario

Su hosting internazionali (es. server fisicamente a Los Angeles), `date()` e `strtotime()` usano il timezone del server. Un articolo programmato alle **10:00 ora italiana** risulterebbe pubblicato alle

01:00 Pacific Time, rompendo la logica di filtro `published_at <= NOW()`.

Soluzione: `date_default_timezone_set('Europe/Rome')` all'inizio del file. Questa istruzione deve precedere qualsiasi uso di `date()` o `time()`. Pattern valido anche in `articles.php`, `news.php` e qualsiasi endpoint con logica temporale.

```
// All'inizio del file, prima di qualsiasi uso di date()
date_default_timezone_set('Europe/Rome');
$ita_now_str = date('Y-m-d H:i:s'); // Ora italiana corretta
$ita_now_time = time();           // Timestamp Unix corretto
```

2.2 Formato Data RFC 822

RSS 2.0 richiede date nel formato RFC 822. PHP fornisce la costante `DATE_RSS` che genera il formato corretto:

Mon, 24 Mar 2026 10:00:00 +0100

Non usare mai ISO 8601 (`Y-m-d\TH:i:s`) in un feed RSS: molti lettori lo rifiutano.

2.3 URL degli Articoli nel Feed

L'URL deve essere **assoluto e correttamente encodato**:

```
$item_url = $base_url . '/' . rawurlencode($article['category']) . '/' .
↳ rawurlencode($article['slug']);
```

`rawurlencode()` gestisce spazi, accenti e caratteri speciali nei category slug. Non usare `urlencode()` (sostituisce gli spazi con `+` invece che `%20`).

2.4 Enclosure per Immagini

Il tag `<enclosure>` permette agli aggregatori di mostrare l'immagine di anteprima dell'articolo nel feed. Richiede URL assoluto — i path relativi (`/api/uploads/...`) non funzionano:

```
$img_url = str_starts_with($article['cover_image'], 'http')
    ? $article['cover_image'] // Già assoluto
    : $base_url . '/' . ltrim($article['cover_image'], '/'); // Rendi assoluto
```

2.5 GUID Univoco

Il tag `<guid>` identifica univocamente ogni articolo nel feed. Usare l'URL dell'articolo come GUID garantisce univocità e stabilità nel tempo (l'URL non cambia dopo la pubblicazione, grazie alla logica di auto-slug che non rigenera per articoli esistenti — vedi Capitolo 9).

3. Feed RSS per Podcast (Podcast App)

SitoRuntime implementa anche un feed per podcast nel formato Apple Podcasts / Spotify. Il feed podcast ha requisiti aggiuntivi rispetto al feed news:

```
// Namespace iTunes obbligatorio per podcast app
echo '<rss version="2.0" xmlns:itunes="http://www.itunes.com/dtds/podcast-1.0.dtd">'
  ↪ . "\n";
echo '<channel>' . "\n";
echo '  <itunes:author>Nome Autore</itunes:author>' . "\n";
echo '  <itunes:category text="Technology" />' . "\n";

// Per ogni episodio:
echo '  <item>' . "\n";
echo '    <title>' . htmlspecialchars($sep['title']) . '</title>' . "\n";
echo '    <enclosure url="' . $audio_url . '" type="audio/mpeg" length="' .
  ↪ $sep['file_size'] . '" />' . "\n";
echo '    <itunes:duration>' . $sep['duration'] . '</itunes:duration>' . "\n";
echo '  </item>' . "\n";
```

4. Registrazione del Feed nel Routing

Il feed deve essere accessibile tramite URL dedicato. Con il `.htaccess` del Modello Universale (che reindirizza tutto a `index.php`), i file PHP fisici sono già serviti direttamente — non servono configurazioni aggiuntive per `rss.php` o `feed.php`.

URL consigliati: - `/api/rss.php` — feed notizie standard - `/api/feed.php` — alias alternativo - `/api/feed_news_rss.php` — naming esplicito SitoRuntime

5. Integrazione nel Frontend (Discovery)

Il feed RSS deve essere **annunciato** nell'<head> HTML per permettere ai browser e ai lettori RSS di scoprirlo automaticamente. Il blocco va aggiunto all'`index.php` (SEO engine) o all'`index.html` base:

```
<link rel="alternate" type="application/rss+xml"
      title="Nome Sito - Feed Notizie"
      href="/api/rss.php" />
```

Prossimo Capitolo: Newsletter & Email System - La lista email come asset posseduto, indipendente dalle piattaforme.

CAPITOLO 13: Newsletter & Email System (v1.0)

La newsletter è uno degli strumenti di fidelizzazione più diretti e indipendenti dalle piattaforme. A differenza dei social media, una lista email è un asset *posseduto* — non soggetta a algoritmi, shadowban o chiusure di account. Questo capitolo documenta il pattern completo implementato in SitoRuntime (Runtime Radio).

1. Schema del Database

La tabella `subscribers` è volutamente minimale. L'iscrizione richiede solo un'email valida — niente nome, niente dati aggiuntivi — in linea con il principio del minimo dato necessario per la conformità GDPR.

```
-- SQLite
CREATE TABLE IF NOT EXISTS subscribers (
  id          INTEGER PRIMARY KEY AUTOINCREMENT,
  email       TEXT UNIQUE NOT NULL,
  is_active   INTEGER DEFAULT 1,
  created_at  TEXT DEFAULT (datetime('now'))
);

-- MySQL (equivalente)
CREATE TABLE IF NOT EXISTS subscribers (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  email       VARCHAR(255) UNIQUE NOT NULL,
  is_active   TINYINT(1) DEFAULT 1,
  created_at  DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Il campo `is_active` permette la **disiscrizione morbida**: l'utente viene marcato come inattivo senza cancellare il record, preservando la storia dell'iscrizione e prevenendo re-iscrizioni accidentali.

2. Architettura degli Endpoint

Il file `newsletter.php` gestisce più azioni via query string `?action=`. La struttura ha un gate chiaro: le azioni pubbliche vengono servite immediatamente, quelle admin richiedono autenticazione.

GET/POST `?action=subscribe` → pubblico

GET ?action=unsubscribe → pubblico (via link email)
GET ?action=count → admin only
POST ?action=send → admin only

```
<?php
require_once 'cors.php';
require_once 'auth_utils.php';
session_start();

// Lazy DB Connection — caricata solo se necessario
function getDB() {
    static $pdo = null;
    if ($pdo === null) {
        require_once 'db.php';
        $pdo = Database::connect();
    }
    return $pdo;
}

$input = json_decode(file_get_contents('php://input'), true);
$action = $_GET['action'] ?? '';

// --- AZIONI PUBBLICHE (prima del gate admin) ---
// ...

// --- GATE ADMIN ---
if (!isLoggedIn() || $_SESSION['role'] !== 'admin') {
    http_response_code(403);
    echo json_encode(['success' => false, 'error' => 'Forbidden']);
    exit;
}

// --- AZIONI ADMIN ---
// ...
```

3. Iscrizione Pubblica

```
if ($action === 'subscribe') {
    $email = filter_var($input['email'] ?? '', FILTER_VALIDATE_EMAIL);

    if (!$email) {
        echo json_encode(['success' => false, 'error' => 'Email non valida']);
        exit;
    }

    try {
        $stmt = getDB()->prepare("INSERT INTO subscribers (email) VALUES (?)");
        $stmt->execute([$email]);
        echo json_encode(['success' => true, 'message' => 'Iscrizione completata']);
    }
```

```

    } catch (PDOException $e) {
        if ($e->getCode() == 23000) { // UNIQUE constraint violation
            echo json_encode(['success' => true, 'message' => 'Sei già iscritto!']);
        } else {
            echo json_encode(['success' => false, 'error' => 'Errore database']);
        }
    }
    exit;
}

```

Note implementative: - FILTER_VALIDATE_EMAIL è la validazione lato server — obbligatoria anche se il form frontend valida già. - Il codice di errore 23000 (UNIQUE constraint violation) viene gestito silenziosamente restituendo un successo: l'utente già iscritto non deve sapere che il suo indirizzo è già nel database (misura anti-enumeration minima). - La risposta è sempre JSON — questo endpoint è chiamato da React via `fetch()`.

4. Disiscrizione GDPR-Compliant

La disiscrizione avviene via link personalizzato nell'email. A differenza della subscribe (che riceve JSON da React), questa action restituisce HTML diretto — perché viene visitata dal browser dell'utente che clicca il link nell'email.

```

if ($action === 'unsubscribe') {
    $email = filter_var($_GET['email'] ?? '', FILTER_VALIDATE_EMAIL);

    if (!$email) {
        echo "Email non valida.";
        exit;
    }

    try {
        $stmt = getDB()->prepare("UPDATE subscribers SET is_active = 0 WHERE email = 
        ⇐ ?");
        $stmt->execute([$email]);
        echo "<h1>Disiscrizione completata</h1>
        ⇐ <p>Ti abbiamo rimosso dalla newsletter. Ci dispiace vederti
        ⇐ andare!</p>";
    } catch (PDOException $e) {
        echo "Errore durante la disiscrizione.";
    }
    exit;
}

```

La URL di disiscrizione nell'email ha questa forma:

`https://runtimeradio.com/api/newsletter.php?action=unsubscribe&email=mario%40example.com`

L'email è `urlencode()`-ata al momento della generazione dell'email (vedi Sezione 6).

5. Conteggio Iscritti (Admin)

```
if ($action === 'count') {
    try {
        $stmt = getDB()->query("SELECT COUNT(*) FROM subscribers WHERE is_active =
        ↳ 1");
        echo json_encode(['success' => true, 'count' => $stmt->fetchColumn()]);
    } catch (PDOException $e) {
        echo json_encode(['success' => false, 'error' => 'Database error']);
    }
    exit;
}
```

Questo endpoint viene tipicamente chiamato al caricamento della dashboard admin per mostrare la dimensione della lista. Solo iscritti attivi (`is_active = 1`).

6. Invio Newsletter (Admin)

L'azione più complessa: genera l'HTML dell'email, personalizza il link di disiscrizione per ogni destinatario, e invia con rate limiting.

6.1 Generazione HTML con Placeholder

L'email viene costruita una volta sola come stringa HTML con un placeholder `{EMAIL_PLACEHOLDER}` nel link di disiscrizione. Il placeholder viene poi sostituito per ogni destinatario al momento dell'invio:

```
// Costruzione HTML con placeholder
$html .= '<a
↳ href="https://runtimeradio.com/api/newsletter.php?action=unsubscribe&email={EMAIL_PLACEHOLDER}
    DISISCRIVITI QUI
  </a>';

// Al momento dell'invio, per ogni destinatario:
foreach ($subscribers as $to) {
    $personalHtml = str_replace('{EMAIL_PLACEHOLDER}', urlencode($to), $html);
    mail($to, $subject, $personalHtml, $headers);
}
```

Questo pattern evita di rigenerare l'intero HTML per ogni email — solo la sostituzione del placeholder è per-destinatario.

6.2 Headers Email

```
$headers = "MIME-Version: 1.0\r\n";
$headers .= "Content-type:text/html;charset=UTF-8\r\n";
$headers .= "From: Runtime Radio <no-reply@runtimeradio.com>\r\n";
$headers .= "Reply-To: runtimeradio@gmail.com\r\n";
$headers .= "Return-Path: no-reply@runtimeradio.com\r\n";
$headers .= "X-Mailer: PHP/" . phpversion() . "\r\n";
```

- From diverso da Reply-To: le risposte vanno all'indirizzo reale, non al no-reply.
- Return-Path: gestisce i bounce (email non consegnate) — usato dai server di posta.

6.3 Rate Limiting

```
$count = 0;
foreach ($subscribers as $to) {
    $personalHtml = str_replace('{EMAIL_PLACEHOLDER}', urlencode($to), $html);
    mail($to, $subject, $personalHtml, $headers);
    $count++;

    // Pausa ogni 10 email per non sovraccaricare il mail server
    if ($count % 10 === 0) {
        usleep(500000); // 0.5 secondi
    }
}

echo json_encode(['success' => true, 'message' => "Newsletter inviata a $count
↳ iscritti"]);
```

Il rate limiting con `usleep()` è fondamentale per hosting condivisi, che spesso impongono limiti di invio per-secondo. Senza questa pausa, il server SMTP rifiuta le email dopo un certo threshold, causando una consegna parziale invisibile.

6.4 Query degli Articoli (Ottimizzazione)

L'endpoint di invio riceve un array di ID articoli dal frontend. La query che li recupera è deliberatamente priva del campo `content`:

```
$placeholders = str_repeat('?', count($articleIds) - 1) . '?';
$stmt = getDB()->prepare(
    "SELECT id, title, slug, summary, cover_image, published_at
    FROM news WHERE id IN ($placeholders) ORDER BY published_at DESC"
);
$stmt->execute($articleIds);
```

Il campo `content` (HTML completo dell'articolo) non viene mai caricato nelle email: l'email include solo titolo, excerpt/summary e link "Leggi tutto". Questo riduce la dimensione del payload e mantiene le email snelle.

7. Struttura Email HTML (Pattern)

Le email HTML devono usare **CSS inline** — i client email (Outlook, Gmail app) non supportano fogli di stile esterni né `<style>` in `<head>`. Il pattern segue questa struttura:

```
[Header con logo/nome]
[Testo intro opzionale]
[Loop articoli: immagine + titolo + excerpt + link]
[Footer: informativa GDPR + link disiscrizione + copyright]
```

```
$html = '<!DOCTYPE html><html><body style="font-family: sans-serif; background:
↳ #0f172a;">';
$html .= '<div style="max-width: 600px; margin: 0 auto; background: #1e293b;
↳ padding: 20px;">';

// Header
$html .= '<h1 style="color: #2dd4bf; text-align: center;">Runtime Radio News</h1>';

// Intro (opzionale, inserito dall'admin)
if ($intro) {
    $html .= '<div style="background: #334155; padding: 15px; border-radius: 6px;">';
    $html .= nl2br(htmlspecialchars($intro)); // Preserva a-capo, escapa HTML
    $html .= '</div>';
}

// Loop articoli
foreach ($articles as $art) {
    $link = "https://runtimeradio.com/news/" . $art['slug'];
    $html .= '<h2><a href="' . $link . '" style="color: #fff;">'
        . htmlspecialchars($art['title']) . '</a></h2>';
    $html .= '<p>' . htmlspecialchars($art['summary']) . '</p>';
    $html .= '<a href="' . $link . '">Leggi tutto &rarr;</a>';
}

// Footer GDPR
$html .= '<p>I tuoi dati sono trattati in conformità al GDPR (UE 2016/679).
    <a href="https://runtimeradio.com/privacy-policy">Privacy Policy</a></p>';
$html .= '<a
↳ href="...?action=unsubscribe&email={EMAIL_PLACEHOLDER}">DISISCRIVITI</a>';
$html .= '</div></body></html>';
```

8. Considerazioni per la Scalabilità

Il pattern con `mail()` nativo di PHP funziona per liste fino a qualche migliaio di iscritti. Per volumi maggiori o per migliorare la deliverability (evitare lo spam folder), il passaggio a un servizio SMTP esterno richiede solo la sostituzione del blocco di invio:

- **Brevo** (ex Sendinblue) — API gratuita fino a 300 email/giorno
- **Mailgun** — API REST, ottima deliverability
- **Amazon SES** — costo bassissimo a volume alto

La struttura del codice rimane invariata: solo il metodo di consegna cambia.

Capitolo correlato: Cap 10 (Security & Auth) per la gestione della sessione admin che protegge gli endpoint di invio.

Prossimo Capitolo: Database Evolution - La migrazione da SQLite a MySQL, documentata ora per ora dalla notte reale di febbraio 2026.

I Casi Reali

CAPITOLO 14: Database Evolution - Da SQLite a MySQL (v1.0)

Questo capitolo documenta il percorso reale vissuto dal progetto **SitoRuntime (Runtime Radio)** nella migrazione da SQLite a MySQL, avvenuta il 24 febbraio 2026. È una storia concreta di sfide, soluzioni e lezioni apprese — non un manuale teorico.

1. Quando e Perché Migrare

SQLite è perfetto per progetti leggeri o in fase di sviluppo: zero configurazione, file unico, deployabile in pochi secondi. Ma esistono soglie oltre le quali inizia a mostrare i propri limiti su hosting condivisi Apache/PHP:

- **Traffico crescente:** più utenti simultanei = più richieste di scrittura = file lock frequenti.
- **Hosting restrittivi:** alcuni provider limitano o bloccano SQLite per policy interne.
- **Necessità di accesso remoto al DB:** tools di gestione visuale (es. phpMyAdmin) richiedono MySQL.
- **Performance su query complesse:** MySQL gestisce meglio join e aggregazioni su tabelle grandi.

Regola pratica: Finché il sito ha < 50 scritture/ora, SQLite è più che sufficiente. Sopra quella soglia, o in presenza di file lock ricorrenti in produzione, valutare MySQL.

2. La Storia del WAL (Warning reale)

Prima della migrazione definitiva, SitoRuntime aveva tentato di risolvere i problemi di file lock attivando il **WAL mode** (Write-Ahead Logging) di SQLite — teoricamente più veloce e meno soggetto a lock. In pratica su hosting condiviso Apache con PHP, il WAL ha causato problemi peggiori: il file di lock WAL (`.sqlite-wal`) rimaneva “appeso” e corrompeva le letture.

Esiste ancora in repo il file `emergency_revert_wal.php`, uno script d'emergenza che forza il ritorno al `journal_mode=DELETE`. Questa è la ragione per cui il **Capitolo 3 impone DELETE mode e non WAL**: la lezione è stata appresa a caro prezzo in produzione.

3. Architettura della Migrazione (Il Pattern a 3 Script)

La migrazione da SQLite a MySQL è stata eseguita con 3 script dedicati, ciascuno con un ruolo preciso:

3.1 db_credentials.php — Il File dei Segreti

```
<?php
// ATTENZIONE: Aggiungere questo file al .gitignore.
// NON committare credenziali nel repository.
return [
    'DB_HOST' => 'mysql.tuohoster.com',
    'DB_NAME' => 'nome_database',
    'DB_USER' => 'utente_mysql',
    'DB_PASS' => 'password_sicura',
    'DB_PORT' => 3306,
];
```

Principio: Le credenziali non devono mai essere hardcodate in db.php o in qualsiasi file committato. Il pattern require __DIR__ . '/db_credentials.php' carica un file separato, escluso dal version control tramite .gitignore.

3.2 db.php con MySQL — Il Connettore Agnostico

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

    public static function connect() {
        if (self::$pdo === null) {
            try {
                $config = require __DIR__ . '/db_credentials.php';

                $dsn = sprintf(
                    "mysql:host=%s;dbname=%s;port=%d;charset=utf8mb4",
                    $config['DB_HOST'],
                    $config['DB_NAME'],
                    $config['DB_PORT']
                );

                self::$pdo = new PDO($dsn, $config['DB_USER'], $config['DB_PASS'], [
                    PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
                    PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
                    PDO::ATTR_EMULATE_PREPARES => false, // Prepared statements
                    ↪ nativi MySQL
                    PDO::ATTR_TIMEOUT => 5,
                ]);
            } catch (PDOException $e) {
                // Gestione errore
            }
        }
    }
}
```

```

        PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8mb4 COLLATE
        ↳ utf8mb4_unicode_ci"
    ]]);

    } catch (PDOException $e) {
        http_response_code(503);
        die(json_encode(['error' => 'Database Connection Error']));
    }
}
return self::$pdo;
}
}

```

Differenze chiave rispetto a SQLite:

- `PDO::ATTR_EMULATE_PREPARES => false`: Usa prepared statements nativi di MySQL. In SQLite era impostazione di default, in MySQL va forzata esplicitamente.
- `charset=utf8mb4` nel DSN + `SET NAMES utf8mb4`: Garantisce il corretto encoding di emoji e caratteri speciali.
- `PDO::ATTR_TIMEOUT => 5`: Timeout connessione di 5 secondi. Con SQLite non serviva (accesso locale al file); con MySQL (rete) è essenziale.
- Niente `PRAGMA journal_mode` o `busy_timeout`: Queste sono direttive SQLite-only.

3.3 `init_mysql.php` — Creazione Schema

Script che crea tutte le tabelle nel database MySQL da zero. Va eseguito **una volta sola** prima della migrazione dati. Deve essere protetto da autenticazione o IP whitelist e va eliminato dopo l'uso.

3.4 `migrate_to_mysql.php` — Il Trasloco Dati

Script ONE-SHOT che: 1. Apre una connessione a MySQL (destinazione). 2. Apre il file SQLite caricato temporaneamente sul server (sorgente). 3. Migra tabella per tabella con `INSERT ... ON DUPLICATE KEY UPDATE` (idempotente: ri-eseguibile senza duplicati). 4. Stampa un riepilogo con conteggi di verifica.

Strategia di sicurezza:

- Lo script viene caricato sul server solo per il tempo strettamente necessario.
- Dopo l'esecuzione, sia lo script che il file `.sqlite` vengono eliminati dal server.
- Il `set_time_limit(120)` previene i timeout su database grandi.

```

// Esempio di migrazione idempotente (tabella news)
$stmt = $mysql->prepare("
    INSERT INTO news (id, slug, title, summary, content, cover_image, published_at,
    ↳ category)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?)
    ON DUPLICATE KEY UPDATE
        title=VALUES(title),
        summary=VALUES(summary),
        content=VALUES(content),
        updated_at=NOW()
");

```

4. Differenze Strutturali SQLite vs MySQL

Aspetto	SQLite	MySQL
Connessione	File locale	Rete (host:port)
PRAGMA	journal_mode, busy_timeout, foreign_keys	Non applicabili
Charset	Configurabile per file	utf8mb4 per emoji e accenti
AUTO_INCREMENT	INTEGER PRIMARY KEY AUTOINCREMENT	INT AUTO_INCREMENT PRIMARY KEY
Boolean	INTEGER (0/1)	TINYINT (1)
JSON	Stored as TEXT	Tipo nativo JSON disponibile
Prepared Statements	Emulati di default	Nativi con EMULATE_PREPARES=false
Concurrent writes	File lock (problematico)	Gestione nativa row-level locking
Backup	Copia del file .sqlite	mysqldump o tool dedicati

5. Il Pattern di Credenziali (Sicurezza)

Il file `db_credentials.php` deve essere aggiunto al `.gitignore` **prima del primo commit** del progetto:

```
# .gitignore
public/api/db_credentials.php
```

In produzione il file viene caricato manualmente sul server via FTP/SFTP, mai tramite pipeline automatiche o repository pubblici. Se per errore viene committato con credenziali reali, cambiare immediatamente la password del database.

6. Checklist di Migrazione

- ☐ Fare un backup completo del file `.sqlite` di produzione.
- ☐ Creare il database MySQL sul provider hosting.
- ☐ Caricare e eseguire `init_mysql.php` per creare lo schema.
- ☐ Caricare il `.sqlite` e `migrate_to_mysql.php` sul server.
- ☐ Eseguire `migrate_to_mysql.php` e verificare i conteggi nel riepilogo.

- ☐ Aggiornare `db.php` con la versione MySQL.
- ☐ Caricare `db_credentials.php` con le credenziali reali.
- ☐ Testare tutte le API in produzione.
- ☐ Eliminare `migrate_to_mysql.php` e il file `.sqlite` dal server.
- ☐ Aggiungere `db_credentials.php` al `.gitignore`.

Prossimo Capitolo: Portfolio & Projects Module - Il modulo universale per portfolio e showcase, con ordinamento drag-and-drop.

CAPITOLO 15: Portfolio & Projects Module (v1.0)

Il modulo Portfolio è un'entità distinta dalla News/Article, progettata per siti di tipo personale, agenzia o showcase. Documentato da **SimonePizziWebSite**, introduce pattern specifici: visibilità granulare, ordinamento manuale, pulsanti azione multipli e gestione per categorie. È il modello di riferimento per qualsiasi sito che debba esporre un catalogo di lavori, prodotti o progetti.

1. La Differenza con il Modulo News/Articles

Caratteristica	News/Articles	Projects/Portfolio
Identificatore URL	slug (testo parlante)	id (numerico)
Visibilità	status (draft/published)	is_visible (boolean)
Programmazione temporale	published_at	Non prevista
Rich Text body	Sì (HTML)	Opzionale (description breve)
Ordinamento	Per data (automatico)	sort_order (manuale)
CTA	—	button_a + button_b (URL esterni)
Categorizzazione	category + tags	Solo category

2. Schema Database

```
CREATE TABLE IF NOT EXISTS projects (  
  id          INTEGER PRIMARY KEY AUTOINCREMENT,  
  name        TEXT NOT NULL DEFAULT 'Nuovo Progetto',  
  description TEXT DEFAULT '',  
  category    TEXT NOT NULL DEFAULT 'progetti-software',  
  cover_image TEXT DEFAULT '',  
  button_a_label TEXT DEFAULT 'Scopri',  
  button_a_url  TEXT DEFAULT '',  
  button_b_label TEXT DEFAULT '',  
  button_b_url  TEXT DEFAULT '',  
  is_visible   INTEGER NOT NULL DEFAULT 1,    -- 1=visibile al pubblico, 0=nascosto  
  sort_order   INTEGER NOT NULL DEFAULT 0,    -- Ordinamento manuale per categoria  
  created_at   DATETIME DEFAULT (datetime('now'))
```

```
);  
  
CREATE INDEX IF NOT EXISTS idx_projects_category ON projects(category);  
CREATE INDEX IF NOT EXISTS idx_projects_sort_order ON projects(sort_order ASC);
```

3. L'API projects.php (5 Metodi HTTP)

Il modulo Projects usa **tutti e 5 i metodi HTTP**, incluso PATCH per aggiornamenti parziali — il pattern più pulito per operazioni di visibilità e riordinamento:

GET — Lista con Bypass Admin

```
$is_admin = isset($_SESSION['user_id']);  
$conditions = [];  
$params = [];  
  
if (!$is_admin) {  
    $conditions[] = "is_visible = 1"; // Il pubblico vede solo i visibili  
}  
  
if ($category) {  
    $conditions[] = "category = ?";  
    $params[] = $category;  
}  
  
// Ordinamento: per categoria poi per sort_order manuale  
$query .= " ORDER BY category ASC, sort_order ASC, created_at ASC";
```

POST — Creazione con Auto-Sort

Alla creazione, `sort_order` viene calcolato automaticamente come `MAX(sort_order) + 1` all'interno della stessa categoria. Questo garantisce che i nuovi progetti appaiano in fondo alla lista della loro categoria:

```
$stmtMax = $pdo->prepare("SELECT COALESCE(MAX(sort_order), 0) FROM projects WHERE  
↪ category = ?");  
$stmtMax->execute([$category]);  
$sort_order = (int)$stmtMax->fetchColumn() + 1;
```

PATCH — Aggiornamenti Parziali

PATCH è il metodo corretto per aggiornamenti atomici di un singolo campo. Non invia l'intero oggetto: solo il campo che cambia.

```
// Toggle visibilità singolo progetto
if (isset($data['is_visible'])) {
    $stmt = $pdo->prepare("UPDATE projects SET is_visible=? WHERE id=?");
    $stmt->execute([(int)$data['is_visible'], $id]);
}

// Aggiornamento ordinamento (da drag-to-sort frontend)
if (isset($data['sort_order'])) {
    $stmt = $pdo->prepare("UPDATE projects SET sort_order=? WHERE id=?");
    $stmt->execute([(int)$data['sort_order'], $id]);
}
```

Perché PATCH e non POST? Semantica HTTP: POST crea, PUT sostituisce l'intero oggetto, PATCH modifica parzialmente. Usare PATCH per toggle di visibilità e riordinamento è la scelta corretta e comunica chiaramente l'intento dell'operazione.

4. Il Pattern dei Pulsanti CTA (button_a / button_b)

Ogni progetto può avere fino a due pulsanti di azione che puntano a risorse esterne: - button_a: Pulsante principale (es. "Scopri", "Visita il sito", "Gioca ora") - button_b: Pulsante secondario opzionale (es. "GitHub", "Case Study", "App Store")

```
// Frontend TypeScript: rendering condizionale dei bottoni
{project.button_a_url && (
    <a href={project.button_a_url} target="_blank" rel="noopener noreferrer"
      className="btn-primary">
        {project.button_a_label || 'Scopri'}
    </a>
)}
{project.button_b_url && (
    <a href={project.button_b_url} target="_blank" rel="noopener noreferrer"
      className="btn-secondary">
        {project.button_b_label}
    </a>
)}
```

Il `rel="noopener noreferrer"` è obbligatorio per i link `target="_blank"`: previene l'accesso alla `window.open` della pagina madre da parte della pagina di destinazione (vulnerabilità tabnapping).

5. Il Pattern di Slug Avanzato (Normalizzazione Accenti Italiani)

Scoperto in `articles.php` di `SimonePizziWebSite`, questo pattern risolve un problema reale: le parole italiane con accenti generano slug malformati.

Problema:

```
// Input: "Il caffè di Genova è buono"
// Risultato naive: "il-caff-di-genova--buono" ← accenti rimossi male
```

Soluzione con mappa esplicita:

```
function generateSlug($title, $pdo) {
    // Mappa esplicita accenti italiani e francesi comuni
    $accents      = ['à','è','é','ì','ò','ù','À','È','É','Ì','Ò','Ù',
                    'â','ê','î','ô','û','ä','ë','ï','ö','ü'];
    $replacements = ['a','e','e','i','o','u','a','e','e','i','o','u',
                    'a','e','i','o','u','a','e','i','o','u'];

    $title = str_replace($accents, $replacements, $title);
    $slug  = strtolower(trim(preg_replace('/[^A-Za-z0-9-]+/', '-', $title)));

    // Anti-collisione: verifica unicità prima del salvataggio
    $stmt = $pdo->prepare("SELECT COUNT(*) FROM projects WHERE slug = ?");
    $stmt->execute([$slug]);
    if ($stmt->fetchColumn() > 0) {
        $slug .= '-' . time();
    }

    return $slug;
}
```

Risultato:

"Il caffè di Genova è buono" → "il-caffe-di-genova-e-buono" ✓

Questo pattern va applicato a **tutti i moduli che generano slug** in siti con contenuto italiano.

6. React Frontend — Componenti Chiave

PortfolioGrid.tsx

La griglia del portfolio filtra i progetti per categoria e li mostra in cards. Con dati provenienti dall'API, supporta: - Filtraggio lato client per category - Visualizzazione dell'immagine di copertina con lazy loading - Rendering condizionale dei due pulsanti CTA - Badge di categoria

ProjectEditor.tsx (Admin)

Il form di editing del progetto nell'area admin include: - Upload immagine via MediaPicker (vedi Capitolo 8) - Due coppie label + URL per i pulsanti - Toggle is_visible con switch UI - Selezione categoria da dropdown

ProjectsList.tsx (Admin)

La lista admin dei progetti espone: - **Drag-to-sort**: invio di PATCH request ad ogni riposizionamento - **Toggle visibilità**: PATCH istantaneo con cambio di icona (occhio aperto/chiuso) - **Filtro per categoria**: segmentazione visiva della lista

7. Strategie di Categoria

Le categorie sono stringhe fisse definite nel codice (non in database). Questo approccio è deliberato: le categorie di un portfolio cambiano raramente e la gestione da DB aggiungerebbe complessità non necessaria.

```
// Categorie fisse in constants.ts o direttamente nel componente
const PROJECT_CATEGORIES = [
  { value: 'progetti-software', label: 'Software' },
  { value: 'videogiochi',      label: 'Videogiochi' },
  { value: 'narrativa',        label: 'Narrativa' },
  { value: 'altro',            label: 'Altro' },
];
```

Per cambiare le categorie basta aggiornare questa lista nel codice frontend e, se necessario, aggiornare i record esistenti con un'unica query SQL.

8. auth_helper.php — Il Pattern Minimale

SimonePizziWebSite ha introdotto un pattern di autenticazione più snello rispetto agli altri siti: auth_helper.php è una classe Auth con un unico metodo statico check().

```
<?php
// auth_helper.php
require_once 'db.php';

session_start();
header('Content-Type: application/json');

class Auth {
    public static function check() {
        if (!isset($_SESSION['user_id'])) {
            http_response_code(401);
            echo json_encode(['status' => 'error', 'message' => 'Non autorizzato']);
            exit;
        }
    }
}
```

Utilizzo in ogni endpoint protetto:

```
// In projects.php, articles.php, ecc.  
require_once 'auth_helper.php'; // Include anche session_start() e gli header  
  
// Nelle route protette:  
Auth::check();
```

Il vantaggio rispetto al pattern `auth.php` standard: `session_start()` e gli header JSON vengono gestiti **una volta sola** nel file helper, non ripetuti in ogni endpoint. Riduce errori di “headers already sent”.

Conclusione: Con i capitoli 14-15 il Modello Universale copre i pattern di migrazione e showcase reali emersi dai progetti di produzione.

Prossimo Capitolo: Festival Logic - Iscrizioni e Workflow Approvazione - Il ciclo completo di gestione concorrenti per DISINTELLIGENZA e FDCA.

CAPITOLO 16: Festival Logic - Iscrizioni e Workflow Approvazione (v1.0)

Il Modello Universale include un sistema gestionale per concorsi e festival, focalizzato sull'acquisizione di talenti e la gestione del loro ciclo di vita.

1. Il Workflow del Partecipante

Ogni iscrizione passa attraverso una pipeline di validazione obbligatoria gestita dal backend.

1.1 Stati di Iscrizione

- **pending:** Stato iniziale. Il partecipante ha caricato i dati ma non è visibile sul sito.
- **approved:** Il partecipante è validato, riceve l'email di conferma e viene inserito automaticamente nella newsletter.
- **rejected:** Il partecipante viene scartato e riceve una notifica di cortesia.

2. Automazione Email (Transaction Emails)

Il backend deve gestire l'invio di email transazionali in tempo reale per ogni cambio di stato, utilizzando template HTML coerenti con il branding del festival. - **Email di Ricezione:** Conferma tecnica del caricamento file. - **Email di Esito:** Comunicazione formale (positiva o negativa).

3. Gestione Asset Partecipanti

I file audio o video caricati dagli utenti devono essere isolati (`api/uploads/audio/participants/`) e rinominati in modo univoco. L'admin deve poterli pre-ascoltare nel Media Center prima di procedere all'approvazione.

4. Newsletter Sync Strategy

Il sistema garantisce la crescita del database marketing inserendo l'indirizzo email dell'utente nella tabella `newsletter_subscribers` **solo nel momento dell'approvazione**. Questo assicura che il database contenga solo utenti reali e validati.

Prossimo Capitolo: Festival Logic - Votazioni e Protezione Anti-Frode.

CAPITOLO 17: Festival Logic - Votazioni e Protezione Anti-Frode (v1.0)

Il Modello Universale implementa un sistema di votazione pubblica robusto, progettato per prevenire manipolazioni e garantire l'equità del concorso.

1. La Sessione di Voto

L'utente può esprimere una o più preferenze (es. da 1 a 3) in una singola chiamata API. - **Validazione Server-Side:** Il backend verifica che i partecipanti votati siano effettivamente approvati e nel round attivo (`in_current_round = 1`). - **Transazione Atomica:** L'inserimento del voto e l'aggiornamento del contatore (`vote_count`) nel partecipante devono avvenire all'interno di una transazione SQL (`beginTransaction`).

2. Meccanismi di Protezione (Anti-Abuso)

Per impedire voti multipli dallo stesso utente o da bot: - **Client-Side Protection:** Impostazione di un cookie (`dis_voted`) con scadenza a 30 giorni dopo il voto. - **Server-Side Protection (IP Tracking):** Il backend registra l'indirizzo IP del votante e impedisce nuovi voti dallo stesso IP per le successive 24 ore. - **User Agent Check:** Ogni voto memorizza lo User Agent per permettere l'analisi post-voto in caso di comportamenti sospetti.

3. Gestione dei Round (Il “Palcoscenico” del Voto)

I partecipanti sono visibili nella pagina pubblica di voto solo se soddisfano due condizioni: 1. `status = 'approved'` 2. `in_current_round = 1`

Questo permette all'admin di attivare/disattivare interi gruppi di partecipanti (eliminatorie, semifinali, finale) semplicemente cambiando un interruttore booleano nella dashboard.

4. Master Switch di Votazione

La possibilità di votare è regolata da un interruttore globale (`voting_active`) nella tabella `settings`. Se disattivato, il backend deve restituire un errore `403 Forbidden` a chiunque tenti di inviare un voto.

Prossimo Capitolo: Festival Logic - Dashboard Admin, Settings e Reporting.

CAPITOLO 18: Festival Logic - Dashboard Admin, Settings e Reporting (v1.0)

Il controllo del Festival avviene tramite una dashboard centralizzata che permette all'admin di attivare/disattivare intere fasi dell'evento e monitorare i risultati.

1. Master Switches (Global Settings)

La tabella settings agisce come il “quadro elettrico” del festival: - **registration_active**: Abilita/Disabilita il form di iscrizione pubblico. - **voting_active**: Apre/Chiude la sessione di voto pubblico. - **current_round**: Identifica la fase attuale del concorso.

2. Dashboard Gestionale (KPIs)

L'area admin deve mostrare indicatori chiave di prestazione (KPI) in tempo reale: - **Totale Partecipanti**: Suddivisi per stato (Pending, Approved). - **Voti Totali**: Volume di voti ricevuti. - **Votanti Unici**: Stima degli utenti unici basata sull'IP e sul Cookie.

3. Workflow di Approvazione e Ranking

- **Approvazione**: L'admin visualizza i partecipanti in una tabella dedicata, ascolta l'audio e decide l'esito. L'approvazione è l'unica azione che scatena l'invio dell'email di conferma ufficiale.
- **Classifica (Ranking)**: Il sistema deve fornire una classifica ordinata per `vote_count`, permettendo all'admin di selezionare i finalisti da spostare nel round successivo.

4. Reporting Automatico

Alla chiusura della sessione di voto (quando `voting_active` viene portato a 0), il backend può innescare l'invio di un'email di report finale allo staff: - Riepilogo voti totali. - Top 20 dei partecipanti più votati. - Statistiche di partecipazione geografica (facoltativo, basato sugli IP).

Fine della Documentazione Avanzata del Modello Universale Festival miniCMS.

Allegati

BOILERPLATE CHECKLIST: Avvio Nuovo Progetto miniCMS (v2.0)

Questa checklist riassume i passi pratici per inizializzare un nuovo progetto Web (Sito o Web App) basato sugli standard definiti nel “Modello Universale miniCMS”. Per i dettagli implementativi, fare riferimento ai capitoli indicati.

Fase 1: Setup Ambiente e Sicurezza Iniziale

- ☐ Creare la struttura base delle cartelle (`public/api/`, `src/`, `scripts/`).
- ☐ Creare il file `public/.htaccess` per il routing della SPA (React Router). (Cap. 2)
- ☐ Eseguire lo scaffolding del database: cartella `public/api/.data/` con `.htaccess` → Deny from all. (Cap. 2)
- ☐ Creare la cartella media (`public/api/uploads/`) e la cache (`public/api/.cache/`).
- ☐ Configurare `vite.config.ts` con il proxy corretto per evitare CORS in locale.
- ☐ Creare `.env.local` con le variabili di sviluppo (`VITE_API_URL=http://localhost/...`).
- ☐ Aggiungere `db_credentials.php` e `.env.local` al `.gitignore` (mai committare credenziali). (Cap. 14)

Fase 2: Configurazione Backend Core (PHP)

- ☐ Implementare `db.php` con connessione Lazy a SQLite (`PRAGMA journal_mode=DELETE, busy_timeout=5000, PRAGMA foreign_keys=ON`). (Cap. 3)
- ☐ Includere Auto-Scaffolding in `db.php`: creazione cartella `.data/` e `.htaccess` se non esistono. (Cap. 3)
- ☐ Creare `init_db.php` per generare le tabelle e l'utente admin di default con `password_hash()`.
- ☐ Creare `auth_helper.php` o `auth.php` con classe `Auth::check()` che gestisce `session_start()` e header JSON. (Cap. 5)
- ☐ Configurare i cookie di sessione sicuri (`httponly`, `samesite=Lax`, `secure` se HTTPS). (Cap. 10)
- ☐ Applicare `date_default_timezone_set('Europe/Rome')` in tutti gli endpoint con logica temporale. (Cap. 5, 12)
- ☐ Nascondere gli errori in produzione (`display_errors = 0` o try-catch globale). (Cap. 10)

Fase 3: Configurazione Frontend Core (React)

- ☐ Implementare `src/api.ts` con il pattern “Double Read” per intercettare gli errori del server. (Cap. 6)
- ☐ Configurare `AdminLayout.tsx` con logica di auth check e “Hard Logout” (`window.location.reload()`). (Cap. 10)
- ☐ Abilitare il “Role-Based UI” per mostrare solo le voci consentite al ruolo connesso (Admin vs Editor). (Cap. 10)
- ☐ Usare `key={item.id}` nel componente editor per forzare il reset del rich text editor al cambio articolo. (Cap. 9)

Fase 4: Integrazione Media e Contenuti

- ☐ Implementare `upload.php` con ridimensionamento automatico GD e sanitizzazione nomi (`uniqid()`). (Cap. 5)
- ☐ Inserire il componente `MediaPicker` per il caricamento diretto di audio e immagini. (Cap. 8)
- ☐ Usare l'editor "Clean-HTML" con gestione sicura dell'evento "Paste". (Cap. 8)
- ☐ Implementare la logica slug con normalizzazione accenti italiani se il contenuto è in italiano. (Cap. 5)

Fase 5: SEO e Syndication

- ☐ Creare `public/index.php` come SEO Engine: query DB → iniezione meta tag nell'HTML di Vite. (Cap. 11)
- ☐ Aggiungere il rinomina di `index.html` → `index_react.html` nel build script se `index.php` è in `public/`. (Cap. 2, 11)
- ☐ Creare `api/rss.php` con feed RSS 2.0 valido (header corretto, date RFC 822, URL assoluti). (Cap. 12)
- ☐ Aggiungere tag `<link rel="alternate" type="application/rss+xml">` nell'<head> HTML. (Cap. 12)

Fase 6: Ottimizzazione e Deploy

- ☐ Configurare la "Programmazione Reale" tramite query SQL su `published_at`. (Cap. 9)
- ☐ Configurare la Cache JSON con TTL 300s per query di listing pesanti. (Cap. 7)
- ☐ Impostare `clean-dist.js` nel processo di build per rimuovere i file `.sqlite` dalla cartella `dist/`. (Cap. 2)
- ☐ Configurare header `Cache-Control: max-age=31536000` per i file in `uploads/` via `.htaccess`. (Cap. 7)

Fase 7: Specifico per Tipologia di Sito

Per Siti con Newsletter (SitoRuntime pattern)

- ☐ Creare tabella `subscribers` con `email` UNIQUE, `is_active`, `created_at`. (Cap. 13)
- ☐ Implementare `newsletter.php` con gate admin + azioni pubbliche (`subscribe`, `unsubscribe`). (Cap. 13)
- ☐ Usare il pattern `{EMAIL_PLACEHOLDER}` per il link di disiscrizione personalizzato. (Cap. 13)
- ☐ Abilitare rate limiting con `usleep(500000)` ogni 10 email nel ciclo di invio. (Cap. 13)

Per Portfolio/Sito Personale (SimonePizziWebSite pattern)

- ☐ Aggiungere tabella projects con sort_order, is_visible, button_a, button_b. (Cap. 15)
- ☐ Implementare projects.php con i 5 metodi HTTP incluso PATCH per toggle visibilità e riordinamento. (Cap. 15)
- ☐ Creare componenti PortfolioGrid.tsx, ProjectEditor.tsx, ProjectsList.tsx. (Cap. 15)

Per Festival/Concorso (FDCA / DISINTELLIGENZA pattern)

- ☐ Aggiungere tabelle participants, votes, settings con master switch registration_active e voting_active. (Cap. 16, 17)
- ☐ Implementare participants.php con workflow pending → approved/rejected. (Cap. 16)
- ☐ Implementare votes.php con protezione IP + cookie anti-frode. (Cap. 17)

Per Migrazione SQLite → MySQL (SitoRuntime pattern)

- ☐ Creare db_credentials.php separato (aggiungere al .gitignore). (Cap. 14)
- ☐ Aggiornare db.php con connessione PDO MySQL (utf8mb4, EMULATE_PREPARES=false). (Cap. 14)
- ☐ Eseguire init_mysql.php per creare lo schema MySQL sul server. (Cap. 14)
- ☐ Eseguire migrate_to_mysql.php per trasloco dati (ONE-SHOT, eliminare dopo). (Cap. 14)

Questa checklist è generata basandosi sui capitoli del Modello Universale miniCMS v2.0. Per i dettagli implementativi fare riferimento ai file .md corrispondenti.