

API REST in PHP

Creazione e Test di un'API CRUD con PHP e MySQL

Prof. Fedeli Massimo

IIS Fermi Sacconi Cpi di Ascoli Piceno

11 gennaio 2026

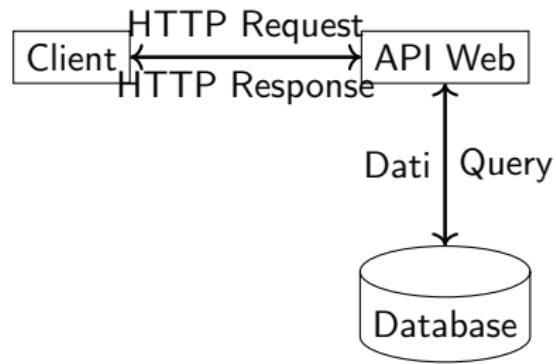
Indice

- 1 Introduzione alle API
- 2 Architettura REST
- 3 Implementazione in PHP
- 4 Validazione e Sicurezza
- 5 Best Practices e Conclusioni

Cos'è un'API?

- **API** = *Application Programming Interface*
È un'interfaccia software che definisce le regole con cui due componenti software possono interagire.
- Permette a programmi diversi — anche scritti in linguaggi differenti o eseguiti su macchine distinte — di **scambiare dati** o **richiedere servizi** in modo strutturato.
- Esistono due categorie principali:
 - **API locali**: usate all'interno di un'applicazione (es. metodi di una classe in PHP).
 - **Web API**: accessibili via rete (Internet/Intranet), tipicamente tramite protocollo HTTP/HTTPS.
- Alcuni modelli comuni di Web API:
 - **REST** – architettura leggera basata su HTTP e risorse (oggi la più diffusa).
 - **SOAP** – standard più rigido, basato su XML e con specifiche complesse.
 - **GraphQL** – sviluppato da Facebook, permette al client di richiedere esattamente i dati necessari.

Web API



Perché non connettersi direttamente al database?

- **Sicurezza:**

I database in cloud o su server remoti *non consentono connessioni dirette* da client esterni per motivi di sicurezza. Consentire l'accesso diretto esporrebbe credenziali, strutture dati e potenzialmente l'intero sistema ad attacchi (es. SQL injection, data leak).

- **Controllo e validazione:**

Un'API REST agisce come intermediario che *filtro, convalida e sanitizza* ogni richiesta prima di toccare il database. Questo evita operazioni non autorizzate o malformate.

- **Astrazione:**

L'API nasconde la complessità del database (tabelle, relazioni, query). Se la struttura del DB cambia, basta aggiornare l'API — i client non devono essere modificati.

- **Scalabilità e manutenzione:**

Attraverso l'API è possibile implementare facilmente:

- Logging delle richieste
- Rate limiting (limitare il numero di chiamate)
- Caching delle risposte
- Autenticazione e autorizzazione (es. token JWT)

Cos'è REST?

- **REST = Representational State Transfer**

È uno stile architettonico per la progettazione di sistemi distribuiti, introdotto da Roy Fielding nel 2000. Oggi è lo standard de facto per le Web API.

- **Basato su HTTP/HTTPS:**

Utilizza il protocollo HTTP (o HTTPS per connessioni sicure) come mezzo di comunicazione tra client e server. Tutte le operazioni avvengono tramite richieste e risposte HTTP.

- **Utilizza i verbi HTTP standard** per esprimere operazioni sulle risorse:

- GET – legge dati (Read)
- POST – crea nuovi dati (Create)
- PUT – aggiorna dati esistenti (Update)
- DELETE – rimuove dati (Delete)

Questi quattro metodi corrispondono alle operazioni **CRUD**.

- **Risorse identificate da URL:**

Ogni entità (es. un libro, un utente) è una "risorsa" accessibile tramite un URI univoco, ad esempio:

`https://miosito.com/api/books/123`

- **Formato dati tipico: JSON**

I dati scambiati tra client e server sono quasi sempre in formato **JSON** (JavaScript Object Notation), per la sua leggibilità, compattezza e supporto

I sei principi di REST

① Client-Server:

Separazione tra interfaccia utente (client) e logica di business/storage (server). Permette evoluzione indipendente dei due componenti.

② Stateless:

Ogni richiesta contiene tutte le informazioni necessarie. Il server non mantiene sessioni tra richieste diverse.

③ Cacheable:

Le risposte devono dichiarare se possono essere memorizzate in cache per migliorare le prestazioni.

④ Interfaccia uniforme:

Un'interfaccia standardizzata tra client e server semplifica l'architettura. Include identificazione delle risorse, manipolazione tramite rappresentazioni, messaggi auto-descrittivi e HATEOAS.

⑤ Sistema a strati:

L'architettura può essere composta da livelli gerarchici. Il client non sa se si connette direttamente al server finale o a un intermediario.

⑥ Code on Demand (opzionale):

Il server può estendere le funzionalità del client inviando codice eseguibile (es. JavaScript).

Stateless vs Stateful

Stateless (REST)

Ogni richiesta è indipendente e contiene tutte le informazioni necessarie (token, parametri). Il server non conserva informazioni sulla sessione del client.

Vantaggi: Scalabilità, semplicità, affidabilità

Svantaggi: Overhead per inviare informazioni ad ogni richiesta

Stateful (tradizionale)

Il server mantiene lo stato della sessione del client (es. via cookie di sessione in PHP). Le richieste successive si basano su questo stato.

Vantaggi: Meno dati trasmessi

Svantaggi: Complessità, difficoltà di scalabilità

- Nel paradigma REST, tutto è una **risorsa**
- Una risorsa è un'entità identificabile: utente, prodotto, ordine, ecc.
- Ogni risorsa è identificata da un **URI univoco**

Esempi di URI ben progettati

GET /api/books — Lista di tutti i libri

GET /api/books/5 — Dettaglio del libro con ID 5

POST /api/books — Creazione di un nuovo libro

PUT /api/books/5 — Aggiornamento del libro 5

DELETE /api/books/5 — Eliminazione del libro 5

Best Practice

Usa sostantivi plurali per le collezioni, non verbi.

Evita: /getBook, /createBook

Preferisci: /books con metodo HTTP appropriato

Metodi HTTP e CRUD

HTTP	CRUD	Descrizione
GET	Read	Recupera una risorsa o collezione
POST	Create	Crea una nuova risorsa
PUT	Update	Aggiorna completamente una risorsa
PATCH	Update	Aggiorna parzialmente una risorsa
DELETE	Delete	Elimina una risorsa

Idempotenza

GET, PUT, DELETE sono idempotenti: eseguirli più volte produce lo stesso risultato.

POST non è idempotente: ripeterlo crea risorse multiple.

- **2xx - Successo**

- 200 OK — Richiesta completata con successo
- 201 Created — Risorsa creata con successo
- 204 No Content — Successo senza corpo di risposta

- **4xx - Errori del client**

- 400 Bad Request — Richiesta malformata
- 401 Unauthorized — Autenticazione richiesta
- 403 Forbidden — Accesso negato
- 404 Not Found — Risorsa non trovata
- 405 Method Not Allowed — Metodo HTTP non supportato

- **5xx - Errori del server**

- 500 Internal Server Error — Errore generico del server
- 503 Service Unavailable — Servizio temporaneamente non disponibile

Formato dati: JSON

JSON (JavaScript Object Notation) è il formato standard per lo scambio di dati nelle API REST moderne. È leggero, leggibile dall'uomo e supportato da tutti i linguaggi di programmazione.

- Struttura basata su coppie chiave: valore
- Supporta array, oggetti annidati e tipi primitivi (stringhe, numeri, booleani, null)
- Più compatto e semplice rispetto a XML
- Nativo in JavaScript, ma facilmente parsabile anche in PHP
(`json_encode()`, `json_decode()`)

```
1  {
2      "Name": "John Wayne",
3      "Work": "Actor",
4      "Age": 52,
5      "Children": ["Lisa", "Thomas", "Knut"]
6  }
```

In una REST API PHP, i dati ricevuti o inviati al client vengono quasi sempre serializzati in JSON.

Headers HTTP importanti

- **Content-Type:** Specifica il formato del corpo della richiesta/risposta
 - application/json — Dati in formato JSON
 - application/xml — Dati in formato XML
 - multipart/form-data — Upload di file
- **Accept:** Il client comunica quale formato preferisce ricevere
- **Authorization:** Contiene le credenziali per l'autenticazione
 - Bearer <token> — Token JWT
 - Basic <credentials> — Basic Authentication
- **Cache-Control:** Gestione della cache
- **Access-Control-Allow-Origin:** Gestione CORS per richieste cross-origin

Ambiente LAMP

- **Linux** Sistema operativo
- **Apache** Web server
- **MySQL** Database
- **PHP** Linguaggio lato server

Ambiente LAMP

- **Linux** Sistema operativo
- **Apache** Web server
- **MySQL** Database
- **PHP** Linguaggio lato server

Alternative locali

XAMPP, WAMP, MAMP

Configurazione PHP necessaria

- PDO extension abilitata
- JSON extension abilitata (di default in PHP moderno)
- mod_rewrite abilitato in Apache (per URL puliti)

Struttura di una API REST in PHP

Architettura tipica

- **config.php**: Configurazione database e costanti
- **index.php**: Entry point, routing delle richieste
- **models/**: Classi per interagire con il database
- **controllers/**: Logica di business
- **utils/**: Funzioni helper (validazione, sanitizzazione)
- **.htaccess**: Riscrittura URL per routing

Principio MVC

Separare la logica in Model (dati), View (presentazione) e Controller (business logic) rende il codice più manutenibile e testabile.

Database: Tabella BOOK

```
1 CREATE TABLE BOOK (
2     BookId INT PRIMARY KEY AUTO_INCREMENT ,
3     Title VARCHAR(100) NOT NULL ,
4     Author VARCHAR(100) NOT NULL ,
5     Topic VARCHAR(100) NOT NULL ,
6     PublishedYear INT ,
7     ISBN VARCHAR(20) UNIQUE ,
8     CreatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP ,
9     UpdatedAt TIMESTAMP DEFAULT CURRENT_TIMESTAMP
10    ON UPDATE CURRENT_TIMESTAMP
11 );
12
```

Best Practice

- Usa chiavi primarie auto-incrementali
- Aggiungi timestamp per tracciare creazione/modifica
- Imposta vincoli appropriati (NOT NULL, UNIQUE)

config.php - Connessione Database

```
1 <?php
2 define('DB_HOST', 'localhost');
3 define('DB_NAME', 'library_db');
4 define('DB_USER', 'root');
5 define('DB_PASS', '');
6
7 try {
8     $pdo = new PDO(
9         "mysql:host=" . DB_HOST . ";dbname=" . DB_NAME ,
10        DB_USER ,
11        DB_PASS ,
12        [
13            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION ,
14            PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC ,
15            PDO::ATTR_EMULATE_PREPARES => false
16        ]
17    );
18 } catch (PDOException $e) {
19     http_response_code(500);
20     echo json_encode([
21         'error' => 'Database connection failed'
22     ]);
23     exit;
24 }
```

PDO vs MySQLi

PDO (PHP Data Objects)

- Supporta 12 diversi database (MySQL, PostgreSQL, SQLite, ecc.)
- Sintassi orientata agli oggetti
- Prepared statements nativi
- Più portabile e moderno
- **Raccomandato per nuovi progetti**

MySQLi

- Specifico per MySQL/MariaDB
- Supporta sia OOP che procedurale
- Leggermente più veloce in alcuni scenari
- Meno portabile

Prepared Statements - Sicurezza

Problema: SQL Injection

```
1 // PERICOLOSO - MAI fare così!
2 $sql = "SELECT * FROM BOOK WHERE BookId = " . $_GET['id'];
3 $result = $pdo->query($sql);
4
```

Un attaccante potrebbe inserire: id=1 OR 1=1 ed estrarre tutti i dati.

Soluzione: Prepared Statements

```
1 // SICURO
2 $sql = "SELECT * FROM BOOK WHERE BookId = :id";
3 $stmt = $pdo->prepare($sql);
4 $stmt->bindParam(':id', $_GET['id'], PDO::PARAM_INT);
5 $stmt->execute();
6 $book = $stmt->fetch();
7
```

I prepared statements separano la logica SQL dai dati, prevenendo l'iniezione di codice malevolo.

Perché le prepared statements prevengono SQL Injection?

- **Separazione logica:** la query SQL e i dati viaggiano *separatamente* al server SQL.
- Il database *compila* la query prima di ricevere i valori, qualsiasi carattere speciale (', ", ;, -) viene trattato come *dato*, non come sintassi SQL.

Esempio sicuro con PDO

```
1      $id = $_GET['id'];
2      $stmt = $pdo->prepare(
3          "SELECT * FROM users WHERE id = ?"
4      );
5      $stmt->execute([$id]);    // il DB non reinterpreta
6      il contenuto
```

Confronto: query “concatenata” vs prepared statement

Codice vulnerabile

```
1 $query = "SELECT * FROM users  
2 WHERE id = $id";  
3 // id = 1 OR 1=1 --  
4 // restituisce TUTTI gli  
5 utenti
```

Prepared statement

```
1 $stmt = $pdo->prepare(  
2 "SELECT * FROM users  
3 WHERE id = ?"  
4 );  
5 $stmt->execute([$id]);  
6 // id = 1 OR 1=1 --  
7 // viene interpretato come  
8 // valore letterale  
9
```

Mai concatenare input utente nella stringa SQL!

index.php - Routing Base

```
1  <?php
2      require_once 'config.php';
3
4      // Imposta header per JSON
5      header('Content-Type: application/json');
6      header('Access-Control-Allow-Origin: *');
7      header('Access-Control-Allow-Methods: GET, POST, PUT, DELETE',
8 );
9      header('Access-Control-Allow-Headers: Content-Type');
10
11     // Gestisci preflight request
12     if ($_SERVER['REQUEST_METHOD'] === 'OPTIONS') {
13         http_response_code(200);
14         exit;
15     }
16
17     $method = $_SERVER['REQUEST_METHOD'];
18     $request = explode('/', trim($_SERVER['PATH_INFO'] ?? '', '/'));
19     $resource = $request[0] ?? null;
20     $id = $request[1] ?? null;
21
22     // Router
23     if ($resource === 'books') {
```

Cosa fa index.php in un'API REST PHP?

- **Entry-point unico**: ogni richiesta HTTP (GET, POST, PUT, DELETE) entra da questo file.
- **Bootstrap**: carica la configurazione (connessione DB, autoloader, costanti).
- **Normalizza la comunicazione**: imposta header CORS e Content-Type JSON.
- **Routing semplice**: analizza PATH_INFO per capire la risorsa (books) e l'eventuale ID.

Esempio di chiamata

GET /index.php/books/123

```
1      $resource = 'books';
2      $id       = '123';
3      //      carica il controller books e restituisce il
4      libro 123 in JSON
```

CORS: Cross-Origin Resource Sharing

- **Problema di base:** per motivi di sicurezza, un browser blocca le richieste AJAX verso domini diversi da quello di origine (Same-Origin Policy).
- **CORS:** meccanismo che consente a un server di dichiarare quali origini esterne sono autorizzate ad accedere alle sue risorse.
- **Chi decide:** non il client, ma il **server**, tramite specifici header HTTP.

Header CORS più comuni

```
1 Access-Control-Allow-Origin: *
2 Access-Control-Allow-Methods: GET, POST, PUT, DELETE
3 Access-Control-Allow-Headers: Content-Type,
4 Authorization
```

- **Preflight:** per alcune richieste il browser invia prima una richiesta OPTIONS.
- **Nota pratica:** senza CORS correttamente configurato, un'API REST funziona via curl ma non dal browser.

BookController.php - Struttura

```
1  <?php
2  class BookController {
3      private $pdo;
4
5      public function __construct($pdo) {
6          $this->pdo = $pdo;
7      }
8
9      public function handleRequest($method, $id) {
10         switch($method) {
11             case 'GET':
12                 $id ? $this->getOne($id) : $this->getAll();
13                 break;
14             case 'POST':
15                 $this->create();
16                 break;
17             case 'PUT':
18                 $this->update($id);
19                 break;
20             case 'DELETE':
21                 $this->delete($id);
22                 break;
23             default:
24                 http_response_code(405);
```

BookController.php: a cosa serve

- **Controller REST:** gestisce tutte le richieste HTTP relative alla risorsa books.
- **Punto di smistamento:** decide quale operazione eseguire in base al metodo HTTP ricevuto.
- **Separazione dei ruoli:** isola la logica applicativa dal routing e dall'accesso diretto al database.
- **Idea chiave:** ogni metodo HTTP corrisponde a un'operazione CRUD.
- **Vantaggio:** codice più leggibile, estendibile e coerente con lo stile REST.

GET - Recuperare tutti i libri

```
1  private function getAll() {
2      try {
3          $sql = "SELECT * FROM BOOK ORDER BY CreatedAt DESC";
4          $stmt = $this->pdo->query($sql);
5          $books = $stmt->fetchAll();
6
7          http_response_code(200);
8          echo json_encode([
9              'success' => true,
10             'count' => count($books),
11             'data' => $books
12         ]);
13     } catch (PDOException $e) {
14         http_response_code(500);
15         echo json_encode([
16             'error' => 'Database error'
17         ]);
18     }
19 }
20 }
```

GET - Recuperare un singolo libro

```
1  private function getOne($id) {
2      if (! $id || ! is_numeric($id)) {
3          http_response_code(400);
4          echo json_encode(['error' => 'Invalid ID']);
5          return;
6      }
7
8      try {
9          $sql = "SELECT * FROM BOOK WHERE BookId = :id";
10         $stmt = $this->pdo->prepare($sql);
11         $stmt->execute(['id' => $id]);
12         $book = $stmt->fetch();
13
14         if ($book) {
15             http_response_code(200);
16             echo json_encode(['success' => true, 'data' => $book]);
17         } else {
18             http_response_code(404);
19             echo json_encode(['error' => 'Book not found']);
20         }
21     } catch (PDOException $e) {
22         http_response_code(500);
23         echo json_encode(['error' => 'Database error']);
24     }
}
```

POST - Creare un nuovo libro

```
1  private function create() {
2      $data = json_decode(file_get_contents('php://input'), true)
3      ;
4
5      if (!$data || !isset($data['Title'], $data['Author'])) {
6          http_response_code(400);
7          echo json_encode(['error' => 'Missing required fields']);
8          return;
9      }
10
11     try {
12         $sql = "INSERT INTO BOOK (Title, Author, Topic)
13             VALUES (:title, :author, :topic)";
14         $stmt = $this->pdo->prepare($sql);
15         $stmt->execute([
16             'title' => $data['Title'],
17             'author' => $data['Author'],
18             'topic' => $data['Topic'] ?? null
19         ]);
20
21         http_response_code(201);
22         echo json_encode([
23             'success' => true,
24             'id' => $this->pdo->lastInsertId()
25         ]);
26     } catch (\Exception $e) {
27         http_response_code(500);
28         echo json_encode(['error' => $e->getMessage()]);
29     }
30
31 }
```

PUT - Aggiornare un libro

```
1  private function update($id) {
2      if (! $id || ! is_numeric($id)) {
3          http_response_code(400);
4          echo json_encode(['error' => 'Invalid ID']);
5          return;
6      }
7
8      $data = json_decode(file_get_contents('php://input'), true)
9      ;
10
11     try {
12         $sql = "UPDATE BOOK SET
13             Title = :title,
14             Author = :author,
15             Topic = :topic
16             WHERE BookId = :id";
17         $stmt = $this->pdo->prepare($sql);
18         $stmt->execute([
19             'title' => $data['Title'],
20             'author' => $data['Author'],
21             'topic' => $data['Topic'],
22             'id' => $id
23         ]);
24     }
25 }
```

DELETE - Eliminare un libro

```
1  private function delete($id) {
2      if (!$id || !is_numeric($id)) {
3          http_response_code(400);
4          echo json_encode(['error' => 'Invalid ID']);
5          return;
6      }
7
8      try {
9          $sql = "DELETE FROM BOOK WHERE BookId = :id";
10         $stmt = $this->pdo->prepare($sql);
11         $stmt->execute(['id' => $id]);
12
13         if ($stmt->rowCount() > 0) {
14             http_response_code(200);
15             echo json_encode(['success' => true]);
16         } else {
17             http_response_code(404);
18             echo json_encode(['error' => 'Book not found']);
19         }
20     } catch (PDOException $e) {
21         http_response_code(500);
22         echo json_encode(['error' => 'Deletion failed']);
23     }
24 }
```

Principio fondamentale

Never trust user input!

Tutti i dati provenienti dal client devono essere validati e sanitizzati.

Tipologie di validazione

- **Tipo:** Verificare che un campo sia numerico, stringa, email, ecc.
- **Formato:** Controllare pattern (es. regex per email, telefono)
- **Lunghezza:** Limiti minimi e massimi per stringhe
- **Range:** Valori numerici entro intervalli accettabili
- **Obbligatorietà:** Verificare presenza di campi richiesti
- **Business logic:** Regole specifiche dell'applicazione

Esempio di validazione

```
1  <?php
2      function validateBook($data) {
3          $errors = [];
4
5          // Validazione Title
6          if (empty($data['Title'])) {
7              $errors[] = 'Title is required';
8          } elseif (strlen($data['Title']) > 100) {
9              $errors[] = 'Title too long (max 100 chars)';
10         }
11
12         // Validazione Author
13         if (empty($data['Author'])) {
14             $errors[] = 'Author is required';
15         }
16
17         // Validazione Year (opzionale)
18         if (isset($data['PublishedYear'])) {
19             $year = intval($data['PublishedYear']);
20             if ($year < 1000 || $year > date('Y')) {
21                 $errors[] = 'Invalid publication year';
22             }
23         }
24     }
```

Funzioni PHP per sanitizzare

- `filter_var()`: Filtra variabili con filtri specifici
- `htmlspecialchars()`: Converte caratteri speciali HTML
- `strip_tags()`: Rimuove tag HTML/PHP
- `trim()`: Rimuove spazi bianchi iniziali/finali

Attenzione

La sanitizzazione NON sostituisce i prepared statements per le query SQL. Usa sempre prepared statements per prevenire SQL injection.

Sanitizzazione - Esempio

```
1 <?php
2     function sanitizeBook($data) {
3         return [
4             'Title' => trim(strip_tags($data['Title'] ?? '')),
5             'Author' => trim(strip_tags($data['Author'] ?? '')),
6             'Topic' => trim(strip_tags($data['Topic'] ?? '')),
7             'PublishedYear' => filter_var(
8                 $data['PublishedYear'] ?? null,
9                 FILTER_VALIDATE_INT
10            ),
11            'ISBN' => preg_replace(
12                '/[^0-9-]/',
13                '',
14                $data['ISBN'] ?? ''
15            )
16        ];
17    }
18 ?>
```

Autenticazione e Autorizzazione

Autenticazione

Verifica l'identità dell'utente: "Chi sei?"

- Basic Authentication (username/password in header)
- Token-based (JWT - JSON Web Token)
- OAuth 2.0 (delegazione autenticazione a terze parti)
- API Keys

Autorizzazione

Verifica i permessi dell'utente: "Cosa puoi fare?"

- Role-based (RBAC): admin, user, guest
- Permission-based: fine-grained permissions
- Resource-based: accesso a specifiche risorse

JWT - JSON Web Token

- Standard per creare token di accesso sicuri
- Struttura: Header.Payload.Signature
- **Header:** Tipo di token e algoritmo di firma
- **Payload:** Dati dell'utente (claims)
- **Signature:** Firma crittografica per verificare autenticità

Vantaggi JWT

- Stateless: nessuna sessione server-side
- Self-contained: contiene tutte le info necessarie
- Scalabile: funziona su sistemi distribuiti
- Cross-domain: può essere usato su domini diversi

Sicurezza

Usa sempre HTTPS per trasmettere JWT. Imposta una scadenza ragionevole (es. 1 ora) e implementa il refresh token per sessioni lunghe.

- **Usa sostantivi per le risorse:** /books, /users
- **Usa plurali per collezioni:** /books non /book
- **Usa kebab-case per URI:** /book-reviews
- **Usa camelCase per JSON:** {"firstName": "John"}
- **Evita verbi negli URI:** Il metodo HTTP indica l'azione
- **Usa gerarchie logiche:** /books/5/reviews
- **Versiona l'API:** /api/v1/books

Coerenza

La coerenza è più importante della convenzione specifica. Scegli uno stile e mantienilo in tutta l'API.

- ① Usa sempre HTTPS in produzione**
- ② Valida tutti gli input lato server**
- ③ Usa prepared statements per query SQL**
- ④ Implementa rate limiting contro attacchi DDoS**
- ⑤ Non esporre informazioni sensibili nei messaggi di errore**
- ⑥ Implementa autenticazione robusta (JWT, OAuth)**
- ⑦ Usa CORS correttamente**
- ⑧ Mantieni aggiornate le dipendenze PHP**
- ⑨ Logga le attività sospette**
- ⑩ Implementa HSTS (HTTP Strict Transport Security)**

Riepilogo

- REST è lo standard per Web API moderne
- PHP con PDO offre strumenti robusti per implementare API
- La sicurezza deve essere prioritaria
- Documentazione e testing sono essenziali
- Segui le best practices per API scalabili e manutenibili
- Monitoring e logging sono cruciali in produzione

Approfondimenti consigliati

- Studiare OAuth 2.0 per autenticazione avanzata
- Implementare GraphQL come alternativa a REST
- Esplorare framework PHP moderni (Laravel, Symfony)
- Containerizzare l'API con Docker
- Implementare CI/CD pipeline
- Studiare microservices architecture

- <https://www.w3schools.com/php/>
- <https://www.w3schools.com/mysql/>
- <https://www.php.net/manual/en/book pdo.php>
- <https://restfulapi.net/>
- <https://swagger.io/docs/>
- <https://jwt.io/>
- <https://www.postman.com/api-platform/>

Fine

Domande?

Grazie per l'attenzione!

Prof. Fedeli Massimo
IIS Fermi Sacconi Cpi di Ascoli Piceno