# Programmazione Web con PHP e SQL

## Prepared statements

prof. Leonardo Essam Dei Rossi

ITT "M. Buonarroti" - Trento (TN)

Anno scolastico 2025/2026

# Indice

# Indice

# Definizione

**Definizione 13.1: Prepared statement**

Una **prepared statement** (istruzione preparata) in PHP è una query SQL *precompilata* che permette di separare la struttura della query dai dati che verranno inseriti successivamente, migliorando **sicurezza**, **prestazioni** e **manutenibilità** del codice.

Una prepared statement è una query SQL con parametri segnaposto (*placeholder*) che viene:

1. **Preparata** dal database (analizzata e compilata);
2. **Eseguita** successivamente una o più volte passando valori concreti ai parametri.

In PHP si utilizza principalmente tramite:

1. PDO (PHP Data Objects);
2. MySQLi.

# Definizione

**Definizione 13.1: Prepared statement**

Una **prepared statement** (istruzione preparata) in PHP è una query SQL *precompilata* che permette di separare la struttura della query dai dati che verranno inseriti successivamente, migliorando **sicurezza**, **prestazioni** e **manutenibilità** del codice.

Una prepared statement è una query SQL con parametri segnaposto (*placeholder*) che viene:

1. **Preparata** dal database (analizzata e compilata);
2. **Eseguita** successivamente una o più volte passando valori concreti ai parametri.

In PHP si utilizza principalmente tramite:

1. PDO (PHP Data Objects);
2. MySQLi.

# Definizione

## Definizione 13.1: Prepared statement

Una **prepared statement** (istruzione preparata) in PHP è una query SQL *precompilata* che permette di separare la struttura della query dai dati che verranno inseriti successivamente, migliorando **sicurezza**, **prestazioni** e **manutenibilità** del codice.

Una prepared statement è una query SQL con parametri segnaposto (*placeholder*) che viene:

1. **Preparata** dal database (analizzata e compilata);
2. **Eseguita** successivamente una o più volte passando valori concreti ai parametri.

In PHP si utilizza principalmente tramite:

1. PDO (PHP Data Objects);
2. MySQLi.

# Perché si usano? (1)

Il vantaggio principale è la protezione contro **SQL Injection**.
Separando i dati dalla query:

- I valori vengono trattati come dati puri;
- Non possono modificare la struttura della query.

**Esempio 13.1: Prepared statement con PDO**

```php
$pdo = new PDO("mysql:host=localhost;dbname=test", "user", "password");

$email = "mario@example.com";

$stmt = $pdo->prepare("SELECT * FROM utenti WHERE email = :email");
$stmt->bindParam(':email', $email);

$stmt->execute();

$result = $stmt->fetchAll();
```

# Perché si usano? (1)

Il vantaggio principale è la protezione contro **SQL Injection**.
Separando i dati dalla query:

- I valori vengono trattati come dati puri;
- Non possono modificare la struttura della query.

## Esempio 13.1: Prepared statement con PDO

```php
$pdo = new PDO("mysql:host=localhost;dbname=test", "user", "password");

$email = "mario@example.com";

$stmt = $pdo->prepare("SELECT * FROM utenti WHERE email = :email");
$stmt->bindParam(':email', $email);

$stmt->execute();

$result = $stmt->fetchAll();
```

# Perché si usano? (2)

In questo esempio (13.1):

- `:email` è un **placeholder nominativo**;
- Il database prepara la query;
- Il valore viene associato separatamente.

# Perché si usano? (3)

## Esempio 13.2: Prepared statement con MySQLi

```php
$conn = new mysqli("localhost", "user", "password", "test");

$email = "mario@example.com";

$stmt = $conn->prepare("SELECT * FROM utenti WHERE email = ?");
$stmt->bind_param("s", $email);

$stmt->execute();

$result = $stmt->get_result();
```

In questo caso:

- ? è un **placeholder posizionale**;
- "s" indica che il parametro è una stringa;

# Perché si usano? (3)

## Esempio 13.2: Prepared statement con MySQLi

```php
$conn = new mysqli("localhost", "user", "password", "test");

$email = "mario@example.com";

$stmt = $conn->prepare("SELECT * FROM utenti WHERE email = ?");
$stmt->bind_param("s", $email);

$stmt->execute();

$result = $stmt->get_result();
```

In questo caso:

- ? è un **placeholder posizionale**;
- `"s"` indica che il parametro è una stringa;

# Indice

# Connessione al database

Sia definita la seguente connessione:

## Esempio 13.3: Connessione al database

```php
$conn = new mysqli("localhost", "user", "password", "database");

if ($conn->connect_errno) {
    die("Connessione fallita: " . $conn->connect_error);
}
```

# Operazione SELECT (1)

Si consideri la seguente entità:

$\text{UTENTE}(\text{id} : \text{INT}, \text{nome} : \text{VARCHAR}(128), \text{email} : \text{VARCHAR}(128))$

## Esempio 13.4: Operazione di SELECT

```php
$email = "mario@example.com";

$stmt = $conn->prepare("SELECT id, nome, email FROM UTENTE WHERE email = ?");
$stmt->bind_param("s", $email);

$stmt->execute();
$result = $stmt->get_result();

while ($row = $result->fetch_assoc()) {
    echo($row["nome"]);
}
```

# Operazione SELECT (1)

Si consideri la seguente entità:

$\text{UTENTE}(\text{id} : \text{INT}, \text{nome} : \text{VARCHAR}(128), \text{email} : \text{VARCHAR}(128))$

## Esempio 13.4: Operazione di SELECT

```php
$email = "mario@example.com";

$stmt = $conn->prepare("SELECT id, nome, email FROM UTENTE WHERE email = ?");
$stmt->bind_param("s", $email);

$stmt->execute();
$result = $stmt->get_result();

while ($row = $result->fetch_assoc()) {
    echo($row["nome"]);
}
```

# Operazione SELECT (2)

Alcune osservazioni:

- `"s"` → tipo del parametro (stringa);
- `get_result()` → ottiene il *result set*;
- `fetch_assoc()` → array associativo.

**Definizione 13.2: Result set di una query**

Un result set (insieme di risultati) è l'insieme delle tuple restituite dal DBMS in seguito all'esecuzione di una query SQL (tipicamente una SELECT).

# Operazione SELECT (2)

Alcune osservazioni:

- `"s"` → tipo del parametro (stringa);
- `get_result()` → ottiene il *result set*;
- `fetch_assoc()` → array associativo.

**Definizione 13.2: Result set di una query**

Un result set (insieme di risultati) è l'insieme delle tuple restituite dal DBMS in seguito all'esecuzione di una query SQL (tipicamente una SELECT).

# Operazione INSERT (1)

## Esempio 13.5: Operazione di INSERT

```php
$nome = "Mario";
$email = "mario@example.com";

$stmt = $conn->prepare("INSERT INTO UTENTE (nome, email) VALUES (?, ?)");
$stmt->bind_param("ss", $nome, $email);

$stmt->execute();

echo "Righe inserite: " . $stmt->affected_rows;
echo "ID generato:    " . $stmt->insert_id;
```

# Operazione INSERT (2)

Alcune osservazioni:

- `"ss"` sono due stringhe (due parametri);
- `affected_rows` rappresenta il numero di righe modificate[1];
- `insert_id` rappresenta l'ultimo ID (con flag `AUTO_INCREMENT`) inserito dalla query.

[1] Vale per: INSERT, UPDATE e DELETE.

# Operazione UPDATE

## Esempio 13.6: Operazione di UPDATE

```php
$email = "nuova@email.com";
$id = 3;

$stmt = $conn->prepare("UPDATE UTENTE SET email = ? WHERE id = ?");
$stmt->bind_param("si", $email, $id);

$stmt->execute();

echo "Righe aggiornate: " . $stmt->affected_rows;
```

**Nota:** `"si"` → stringa + intero.

# Operazione DELETE

## Esempio 13.7: Operazione di DELETE

```php
$id = 3;

$stmt = $conn->prepare("DELETE FROM UTENTE WHERE id = ?");
$stmt->bind_param("i", $id);

$stmt->execute();

echo "Righe eliminate: " . $stmt->affected_rows;
```

**Nota:** `"i"` → intero.

# Buone pratiche

Sono buone pratiche quando si usano i *prepared statement*:

1. Controllare sempre la non-nullità: `($stmt === false);`
2. Chiudere lo *statement*: `$stmt->close();`
3. Chiudere la connessione: `$conn->close();`

# Buone pratiche

Sono buone pratiche quando si usano i *prepared statement*:

1. Controllare sempre la non-nullità: `($stmt === false);`
2. Chiudere lo *statement*: `$stmt->close();`
3. Chiudere la connessione: `$conn->close();`

# Buone pratiche

Sono buone pratiche quando si usano i *prepared statement*:

1. Controllare sempre la non-nullità: `($stmt === false)`;
2. Chiudere lo *statement*: `$stmt->close();`
3. Chiudere la connessione: `$conn->close();`

# Indice

# Tipi per bind_param()

Di seguito i tipi ammessi per il metodo `bind_param()`:

| **Lettera** | **Tipo** |
|---|---|
| s | Stringa (`string`) |
| i | Numero intero (`integer`) |
| d | Numero decimale (`double`) |
| b | Oggetto binario (`BLOB`) |

**Approfondimento:** [1]

# Tipi per bind_param()

Di seguito i tipi ammessi per il metodo `bind_param()`:

| **Lettera** | **Tipo** |
|---|---|
| s | Stringa (`string`) |
| i | Numero intero (`integer`) |
| d | Numero decimale (`double`) |
| b | Oggetto binario (`BLOB`) |

**Approfondimento:** [1]

# Indice

# Esercizio #1

*Si vuole realizzare un sistema per la gestione dei soci di un club di golf.*
Creare una tabella che contenga i seguenti dati: numero di tessera, cognome, nome, data di nascita, indirizzo di residenza e data di iscrizione.

Si realizzi la relativa base di dati e si implementino le seguenti pagine:

1. **Pagina "Home" (`index.php`):** dove viene mostrato l'elenco di tutti i soci;
2. **Pagina "Nuovo socio" (`nuovo_socio.php`):** dove è presente una form per inserire un nuovo socio nel database.

Utilizzare i *prepared statements* per gestire in sicurezza l'inserimento di un nuovo socio nel database.

# Esercizio #2

*Si vuole realizzare un'estensione dell'esercizio "Il gestionale dell'Accademia - Pagina di login".* Partendo dalle query SQL fornite su Classroom per la creazione della base di dati, modificare l'esercizio svolto sostituendo la lettura delle credenziali da file con la lettura delle credenziali dal database.

Utilizzare i *prepared statements* per gestire in sicurezza la fase di autenticazione dell'utente.

# Riferimenti e approfondimenti

[1] Oracle, *The BLOB and TEXT Types*, [Online; accessed 25-February-2026], 2026. indirizzo: https://docs.oracle.com/cd/E17952_01/mysql-8.0-en/blob.html.