



**UNIVERSITÀ DEGLI STUDI
DI CAGLIARI**

CORSO DI LAUREA IN INFORMATICA

Silvia Maria Massa – silviam.massa@unica.it

Fondamenti di Programmazione Web

Memorizzare e consultare i dati (2)





Accesso al DB tramite pg-promise

Database integration

- L'integrazione dei database nelle applicazioni Express può essere facilmente realizzata installando dei moduli Node.js
- In questa pagina, <https://expressjs.com/en/guide/database-integration.html>, potete trovare una guida su come aggiungere e utilizzare, nelle applicazioni Express, alcuni dei moduli Node.js più comuni per i sistemi di database:
 - Cassandra
 - Couchbase
 - CouchDB
 - LevelDB
 - MySQL
 - MongoDB
 - Neo4j
 - Oracle
 - **PostgreSQL**
 - Redis
 - SQL Server
 - SQLite
 - Elasticsearch

pg-promise

- **pg-promise** è l'Interfaccia PostgreSQL per Node.js
- Installazione: **npm install pg-promise**
- Inizializzazione (senza le opzioni di inizializzazione):
const pgp = require('pg-promise')();
- Creazione dell'oggetto Database dalla connessione (senza l'inserimento dei parametri opzionali): **const db = pgp(connection);**
 - Il parametro **connection** è un oggetto di configurazione ({**host**: 'indirizzo IP o nome host del database', **port**: porta del database (di solito 5432 per PostgreSQL), **database**: 'nome del database', **user**: 'nome utente', **password**: 'password per accedere al database'}) o una stringa di connessione ('postgres://username:password@host:port/database').
 - L'oggetto **db** rappresenta il protocollo di comunicazione con il database, caratterizzato da una connessione di tipo lazy.

connessione stabilita solo quando è necessario eseguire effettivamente un'interrogazione o un'operazione sul database quando è necessario e viene rilasciata automaticamente quando l'operazione è completata.


pg-promise - Result-specific methods

- Il metodo più semplice per eseguire una query è: **query**
await db.query ('SELECT * FROM product WHERE id = 3');
- Esistono tuttavia dei metodi per eseguire le query in base al numero di righe che si prevede di ottenere come risultato.
- Per ogni query si dovrebbe scegliere il metodo giusto:
 - **none**: query senza alcun risultato
 - **one**: query con esattamente una riga di risultato
 - **oneOrNone**: query con zero o una riga di risultato
 - **many**: query con più di una riga di risultato
 - **any = manyOrNone**: query con zero o più di una riga di risultato
- ~~await db.one~~('SELECT * FROM product WHERE id = 3');
- ~~await db.none~~('SELECT * FROM product WHERE id = 3');
- await db.oneOrNone('SELECT * FROM product WHERE id = 3');
- Se si ottengono un numero di righe diverse da quelle aspettate viene lanciato un errore. Migliora la robustezza del codice.

pg-promise - Result-specific methods

```
■ async function getUsers() {  
  try {  
    return await db.one('SELECT * FROM users');  
  } catch (error) {  
    console.log(error);  
    return ('Si è verificato un errore')  
  }  
}
```

```
app.get('/', async (req, res) => {  
  let result = await getUsers();  
  res.send(result);  
});
```



```
QueryResultError {  
  code: queryResultErrorCode.multiple  
  message: "Multiple rows were not expected."  
  received: 4  
  query: "SELECT * FROM users"  
}
```

pg-promise - Result-specific methods

```
■ async function getUsers() {  
  try {  
    return await db.any('SELECT * FROM users');  
  } catch (error) {  
    console.log(error);  
    return ('Si è verificato un errore')  
  }  
}
```

```
app.get('/', async (req, res) => {  
  let result = await getUsers();  
  res.send(result);  
});
```



```
[  
  {  
    "id": 1,  
    "email": "john.doe@example.com",  
    "username": "johndoe",  
    "password": "$2b$12$jsQ5c8EPFQ2LX",  
    "name": "John",  
    "surname": "Doe",  
    "gender": "MAN",  
    "birthday": "1990-05-14T22:00:00.000Z",  
    "created_at": "2024-02-24T11:47:08.899Z"  
  },  
  .....  
]
```

pg-promise - Query formatting

- pg-promise è dotata di un query-formatting engine integrato.
- Un **query-formatting engine** è un meccanismo che formatta automaticamente le query SQL, è molto utile ad esempio per evitare SQL injection.
- La sintassi di formattazione per le variabili è decisa dal tipo di valore passato:
 - **Index Variables** quando il valore è un array o un singolo tipo di base;
 - **Named Parameters** quando il valore è un oggetto (diverso da array o null).

Query formatting - Index variables

- La formattazione più semplice (classica) utilizza la sintassi `$1`, `$2`, ... per iniettare valori nella stringa di query, in base al loro indice (da `$1` a `$100.000`) nell'array di valori

```
await db.any( 'SELECT * FROM product
```

```
    WHERE price BETWEEN $1 AND $2', [1, 10] );
```

```
// SELECT * FROM product WHERE price BETWEEN 1 AND 10
```

- Il query formatting engine supporta anche la parametrizzazione a valore singolo per le query che utilizzano solo la variabile `$1`

```
await db.any('SELECT * FROM users WHERE name = $1', 'John' );
```

```
// 'SELECT * FROM users WHERE name = 'John'
```

- Questo però funziona solo per i tipi **number**, **bigint**, **string**, **boolean**, **Date** e **null**, perché tipi come **Object** cambiano il modo in cui i parametri vengono interpretati.

Query formatting - Named parameters

- Quando alla query vengono passati oggetti come valore, il formatting engine si aspetta che la query utilizzi la sintassi **Named Parameter** `$*propName*`, con `*` equivalente a una qualsiasi delle seguenti coppie di apertura-chiusura: `{}`, `()`, `<>`, `[]`, `//`.
- Se necessario, possiamo utilizzare contemporaneamente tutte le sintassi delle variabili supportate

```
await db.none( 'INSERT INTO users (first_name, last_name, age)
                VALUES( ${name.first}, $<name.last>, $/age/ )',
                { name: { first: 'John', last: 'Dow'}, age: 30} );
// INSERT INTO users (first_name, last_name, age) VALUES( 'John', 'Dow', 30)
```

- La proprietà **this** si riferisce all'oggetto stesso, da inserire come stringa formattata in JSON

```
await db.none( 'INSERT INTO documents(id, doc)
                VALUES( ${id}, ${this} )',
                { id: 123, body: 'qualche testo' } );
// INSERT INTO documents(id, doc) VALUES(123, '{"id":123, "body": "some text"}')
```

Query formatting - Formatting filters

- Per impostazione predefinita, tutti i valori sono formattati in base al loro tipo JavaScript.
- I **filtri di formattazione** permettono di formattare diversamente il valore (ne vedremo alcuni).
- I filtri come **:name** o **:raw** seguono immediatamente il nome della variabile (senza spazi) all'interno della stringa della query.

- **Index variables**

```
await db.any( 'SELECT $1:name FROM $2:name',  
              ['price', 'products'] );  
// SELECT "price" FROM "products"
```

- **Named parameters**

```
await db.any( 'SELECT ${column:name} FROM ${table:name}',  
              { column: 'price', table: 'products' } );  
// SELECT "price" FROM "products"
```

Formatting filters - SQL names

- Quando il nome di una variabile termina con **:name**, o con la sintassi più breve **~** (tilde), rappresenta **un nome** o **un identificatore SQL** (es. parti dello schema SQL come tabelle o colonne)

- Utilizzo filtro **:name**

```
await db.none( 'INSERT INTO $1:name($2:name)VALUES(...)',  
               ['Table Name', 'Column Name'] );  
// INSERT INTO "Table Name"("Column Name")VALUES(...)
```

- Utilizzo filtro **~**

```
await db.none( 'INSERT INTO $1~($2~)VALUES(...)',  
               ['Table Name', 'Column Name'] );  
// INSERT INTO "Table Name"("Column Name")VALUES(...)
```

Formatting filters - SQL names

- Gli **SQL names** possono essere forniti in diversi modi:
 - Una stringa che contiene solo * viene automaticamente riconosciuta come tutte le colonne

```
await db.any( 'SELECT $1:name FROM $2:name', ['*', 'table'] );  
// SELECT * FROM "table"
```
 - Un array di stringhe per rappresentare i nomi delle colonne:

```
await db.any('SELECT ${columns:name} FROM ${table:name}',  
             { columns: ['column1', 'column2'], table: 'table' } );  
// SELECT "column1","column2" FROM "table"
```
 - Qualsiasi oggetto che non sia un array in cui i nomi delle colonne sono rappresentati dalle sue proprietà:

```
const obj = { one: 1, two: 2 };  
await db.query( 'SELECT $1:name FROM $2:name', [obj, 'table'] );  
// SELECT "one","two" FROM "table"
```

Formatting filters – Raw text

- Quando il nome di una variabile termina con **:raw**, o con la sintassi più breve **^**, il valore deve essere iniettato come **testo grezzo** (non viene trattato).
- Tali variabili non possono essere **null** o **undefined**, e tali valori lanceranno l'errore **'Values null/undefined cannot be used as raw text'**.
- **const where = pgp.as.format('WHERE price BETWEEN \$1 AND \$2', [5, 10]);**
// pre-formattazione manuale della condizione WHERE
await db.any('SELECT * FROM products \$1:raw', where) ;
// SELECT * FROM products WHERE price BETWEEN 5 AND 10
- Questo filtro non è sicuro e non dovrebbe essere usato per i valori provenienti dal lato client, in quanto potrebbe causare una **SQL injection**.

Formatting filters – Open values

- Quando il nome di una variabile termina con **:value**, o con la sintassi più breve **#**, si tratta il valore della variabile come consueto a seconda del tipo, ma **quando il suo tipo è una stringa, le virgolette non vengono aggiunte**.
- Tali variabili non possono essere **null** o **undefined**, e tali valori lanceranno l'errore **'Open values cannot be null or undefined'**.
- **const name = 'John';**
db.any('SELECT * FROM users WHERE name LIKE \'%\$1:value%\', name);
// SELECT * FROM users WHERE name LIKE '%John%'
- Questo filtro non è sicuro e non dovrebbe essere usato per i valori provenienti dal lato client, in quanto potrebbe causare una **SQL injection**.

Formatting filters – CSV filter

- Quando il nome di una variabile termina con **:csv** o **:list**, viene formattata come **un elenco di valori separati da virgole**, con ciascun valore formattato secondo il suo tipo JavaScript (in genere si usa quando la variabile è un array).

- Utilizzo filtro **:csv**

```
const ids = [1, 2, 3];
```

```
await db.any( 'SELECT * FROM table WHERE id IN ($1:csv)', [ids] );
```

```
// SELECT * FROM table WHERE id IN (1,2,3)
```

- Utilizzo filtro **:list**

```
const ids = [1, 2, 3];
```

```
await db.any( 'SELECT * FROM table WHERE id IN ($1:list)', [ids] );
```

```
// SELECT * FROM table WHERE id IN (1,2,3)
```


Formatting filters – CSV filter

- Enumerazione con filtro **:csv**

```
const obj = {first: 123, second: 'text'};
```

```
await db.none( 'INSERT INTO table($1:name) VALUES($1:csv)', [obj] );  
// INSERT INTO table("first","second") VALUES(123,'text')
```

```
await db.none( 'INSERT INTO table(${this:name}) VALUES(${this:csv})', obj );  
// INSERT INTO table("first","second") VALUES(123,'text')
```

- Enumerazione con filtro **:list**

```
const obj = {first: 123, second: 'text'};
```

```
await db.none( 'INSERT INTO table( $1:name ) VALUES( $1:list )', [ obj ] );  
// INSERT INTO table("first","second") VALUES(123,'text')
```

```
await db.none( 'INSERT INTO table(${this:name}) VALUES(${this:list})', obj );  
// INSERT INTO table("first","second") VALUES(123,'text')
```

pg-promise - Query files

- QueryFile è un oggetto speciale di pg-promise che permette di caricare una query SQL da un file esterno.
- L'uso di file SQL esterni (tramite QueryFile) offre molti vantaggi come:
 - Codice JavaScript molto più pulito, con tutto l'SQL conservato in file esterni
 - Molto più facile scrivere SQL di grandi dimensioni e ben formattato, con molti commenti
 - Le query possono essere riutilizzate ovunque.
- Ogni query method (db.one, db.any, db.query, ecc.) della libreria accetta QueryFile come parametro.
- Esempio

File **findUser.sql** contenuto nella cartella **sql**

```
/*  
    multi-line comments are supported  
*/  
SELECT name, surname -- single-line comments are supported  
FROM Users  
WHERE id = ${id}
```

pg-promise - Query files

```
■ const path = require('path');
function sql (file) {
    const fullPath = path.join (__dirname, file);
    return new pgp.QueryFile (fullPath);
}

const sqlFindUser = sql ('./sql/findUser.sql');

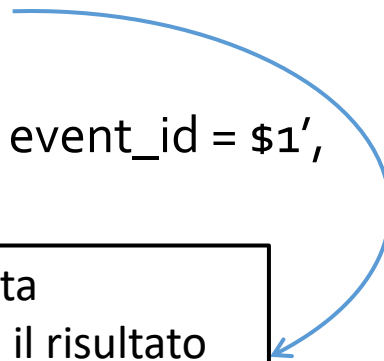
.....
try {
    const user = await db.one (sqlFindUser, { id: 123 });
    console.log(user);
} catch (error) {
    if (error instanceof pgp.errors.QueryFileError) {
        // L'errore è legato al nostro QueryFile
    } else {
        // Gestire altri tipi di errori
    }
}
};
....
```

Esempio di codice da eseguire dentro una richiesta HTTP

pg-promise - Tasks

- Un **task** rappresenta una connessione condivisa per l'esecuzione di più query.
- Dovrebbero essere utilizzati ogni volta che si esegue più di una query alla volta.

```
try {  
  const data = await db.task( async t => {  
    const count = await t.one( 'SELECT count(*) FROM events WHERE id = $1',  
                               123, a => Number( a.count ) );  
  
    if (count > 0) {  
      const logs = await t.any( 'SELECT * FROM log WHERE event_id = $1',  
                               123 );  
  
      return { count, logs };  
    }  
    return { count };  
  });  
  // successo, data = {count} o {count, logs}  
} catch( error ) {  
  // fallito  
};
```



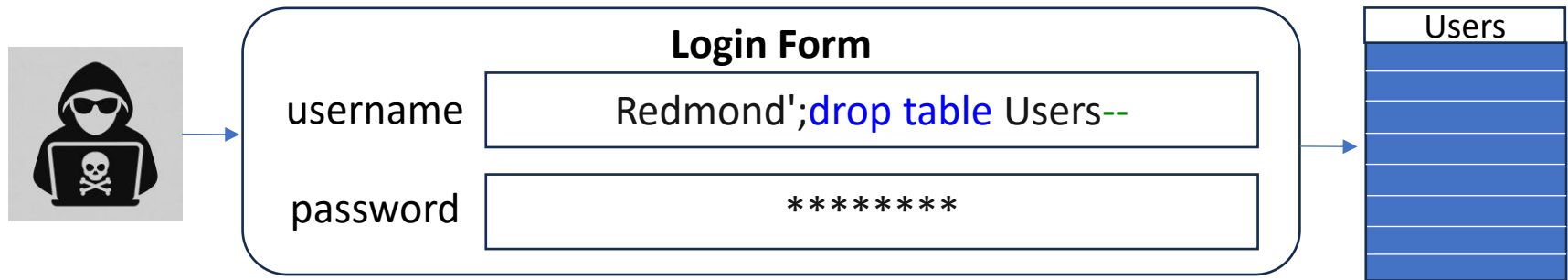
Senza questa operazione il risultato della query sarebbe un oggetto come questo {count: x}, dove x è il numero di eventi con codice 123

pg-promise - Transactions

- Il **metodo di transazione tx** è come un task, che esegue anche BEGIN + COMMIT/ROLLBACK.
- "BEGIN" è un'istruzione SQL che segna l'inizio di una transazione. L'inizio di una transazione indica l'inizio di una serie di operazioni di database che vengono trattate come un'unica unità di lavoro.
- Se la funzione di callback restituisce una rejected promise o lancia un errore, il metodo eseguirà automaticamente ROLLBACK alla fine. In tutti gli altri casi, la transazione verrà chiusa automaticamente con COMMIT.

```
try {  
  const data = await db.tx( async t => {  
    await t.none( 'UPDATE users SET active = $1 WHERE id = $2',  
                  [true, 123] );  
    await t.one( 'INSERT INTO audit(entity, id) VALUES ($1, $2)',  
                 ['users', 123] );  
  })  
  // successo, viene eseguito il COMMIT  
} catch( error ) {  
  // fallimento, viene eseguito il ROLLBACK  
};
```

pg-promise - Sicurezza



- Alcune tecniche per proteggersi da **SQL injection** sono:
 - Assicurarsi che le query SQL utilizzino parametri piuttosto che valori dei parametri inseriti direttamente nell'istruzione SQL.
 - Assicurarsi di convalidare e filtrare l'input dell'utente prima di utilizzarlo in una query SQL lato front-end e lato back-end. Ad esempio limitando il tipo e il formato dei dati accettati per ridurre il rischio di inserire input dannosi.
 - Utilizzare account del database con privilegi minimi (CRUD) per eseguire le operazioni richieste. Se un'istruzione SQL viene compromessa, l'SQL injector avrà accesso solo a risorse limitate.
 - Mantenere un registro delle attività del database e monitora costantemente le query SQL eseguite. Ti permetterà di individuare tempestivamente eventuali anomalie o attività sospette.
- Lo strumento open-source **sqlmap** (<https://sqlmap.org/>) permette di rilevare le vulnerabilità di SQL injection nell'applicazione.