Query Builder

Query Builder

Anonymous · 09/09/2026

Query Builder

`derafu/query` provides two complementary ways to build queries:

1. **Fluent API** (`SqlQueryBuilder`) — chainable method calls, useful in application code.
2. **Declarative Config** (`QueryConfig`) — array, YAML, or JSON definitions, useful for reusable templates and API-driven queries.

Both produce the same output: a `SqlQuery` with an SQL string and named-parameter array.

Setting Up SqlQueryBuilder

`SqlQueryBuilder` requires an engine and an expression parser:

```
use Derafu\Query\Builder\SqlQueryBuilder;
use Derafu\Query\Engine\PdoEngine;
use Derafu\Query\Filter\CompositeExpressionParser;
use Derafu\Query\Filter\ExpressionParser;
use Derafu\Query\Filter\FilterParser;
use Derafu\Query\Filter\PathParser;
use Derafu\Query\Operator\OperatorLoader;
use Derafu\Query\Operator\OperatorManager;

// 1. Set up the operator registry.
$loader  = new OperatorLoader();
$manager = new OperatorManager(
    $loader->loadFromFile('vendor/derafu/query/resources/operators.yaml')
);

// 2. Build the expression parser.
$parser = new CompositeExpressionParser(
    new ExpressionParser(new PathParser(), new FilterParser($manager))
);

// 3. Create a database engine.
$pdo    = new PDO('pgsql:host=localhost;dbname=mydb', $user, $pass);
```

```
$engine = new PdoEngine($pdo);

// 4. Instantiate the builder.
$qb = new SqlQueryBuilder($engine, $parser);
```

For Doctrine DBAL use `DoctrineEngine` instead of `PdoEngine`. For framework-integrated setups, consider the bridges instead of `SqlQueryBuilder`.

Fluent API Reference

All mutating methods return `$this` (or a clone via `new()`) so they can be chained.

`table(string $table, ?string $alias = null): self`

Sets the `FROM` clause and resets the builder state via an internal `new()` clone. Call this first when starting a new query from a builder instance that may have prior state.

```
$qb->table('products')->where('price?>1000')->execute();
$qb->table('invoices', 'i')->select('i.id, i.total')->execute();
```

`from(string $table, ?string $alias = null): self`

Sets the `FROM` clause without resetting. Use `table()` for new queries; use `from()` internally or when you already have a fresh builder.

`select(string|array $columns, bool $sanitize = true): self`

Sets the `SELECT` columns. Accepts a comma-separated string or an array.

```
$qb->select('id, name, price');
$qb->select(['id', 'name', 'price']);
$qb->select('c.name AS customer_name, i.total');
$qb->select('COUNT(*) AS total');
$qb->select('DISTINCT type');  // Use distinct() instead for the DISTINCT keyword
```

Pass `$sanitize = false` to skip identifier sanitization (use only for trusted, pre-validated expressions).

`distinct(bool $distinct = true): self`

Adds `DISTINCT` to the `SELECT` clause.

```
$qb->table('customers')->select('type')->distinct()->execute();
```

```
// → SELECT DISTINCT type FROM customers
```

`where(string|array|Condition|CompositeCondition $condition): self`

Sets the `WHERE` clause. Resets any previous `WHERE` conditions. All elements are AND-combined.

```
$qb->where('status?=active');
$qb->where(['status?=active', 'total?>1000']);
$qb->where('status?=active&&total?>1000');  // composite string
```

`andWhere(string|array|Condition|CompositeCondition $condition): self`

Appends conditions to the existing `WHERE` with AND. If no `WHERE` exists yet, behaves like `where()`.

```
$qb->where('status?=active')->andWhere('total?>1000');
```

`orWhere(string|array|Condition|CompositeCondition $condition): self`

Wraps the existing `WHERE` and the new condition in an OR composite.

```
// WHERE (status = 'electronics' OR category = 'hardware')
$qb->where('category?=electronics')->orWhere('category?=hardware');

// WHERE (category = 'software' OR (category = 'hardware' AND price > 200))
$qb->where('category?=software')->orWhere(['category?=hardware', 'price?>200']);

// Multiple OR groups:
// WHERE (status = 'cancelled' OR status = 'draft' OR (status = 'issued' AND total >
1000))
$qb->where('status?=cancelled')
   ->orWhere(['status?=draft', ['status?=issued', 'total?>1000']]);
```

`andWhereOr(array|Condition|CompositeCondition $conditions): self`

Appends an OR composite to the existing AND WHERE. Each element of the array is one OR branch; arrays within the array become AND groups.

```
// WHERE status = 'active' AND (type = 'person' OR tax_id LIKE '78%')
$qb->where('status?=active')
   ->andWhereOr(['type?=person', 'tax_id?^78']);

// WHERE status = 'active' AND ((status = 'paid' AND total > 1000) OR (status =
'issued' AND date >= '2024-03-01'))
$qb->where('customer_id?in:1,2')
   ->andWhereOr([
       ['status?=paid', 'total?>1000'],
```

```
        ['status?=issued', 'date?>=2024-03-01'],
    ]);
```

`join(string $table, string $condition, string $type = 'INNER', ?string $alias = null): self`

Adds an explicit JOIN clause. Valid types: `INNER`, `LEFT`, `RIGHT`, `CROSS`. Duplicates (same table+alias combination) are ignored.

```
$qb->join('customers', 'i.customer_id = c.id', 'INNER', 'c');
```

Convenience wrappers:

```
$qb->innerJoin('customers', 'i.customer_id = c.id', 'c');
$qb->leftJoin('payments', 'i.id = p.invoice_id', 'p');
$qb->rightJoin('invoices', 'c.id = i.customer_id', 'i');
$qb->crossJoin('currencies');
```

Note: When using path expressions in `where()`, joins are generated automatically from the path segments. Manual `join()` calls are only needed when you want explicit control, or when building the query without path expressions.

`groupBy(string|array $columns): self`

Adds `GROUP BY` columns. Multiple calls accumulate.

```
$qb->groupBy('category');
$qb->groupBy(['c.id', 'c.name']);
```

`having(string|array|Condition|CompositeCondition $condition): self`

Sets the `HAVING` clause. Accepts the same expression formats as `where()`.

```
$qb->groupBy('category')->having('AVG(price)?>500');
$qb->groupBy('status')->having('COUNT(*)?>1');
```

`orderBy(string|array $columns, string $direction = 'ASC'): self`

Sets `ORDER BY`. Accepts a column string, or an associative array of `column => direction`.

```
$qb->orderBy('price', 'DESC');
$qb->orderBy(['category' => 'ASC', 'price' => 'DESC']);
$qb->orderBy(['created_at', 'id']);  // defaults to ASC
```

`limit(int $limit): self` / `offset(int $offset): self`

Set pagination. `OFFSET` is only appended when both `LIMIT` and `OFFSET` are set.

```
$qb->limit(20)->offset(40);
// → LIMIT 20 OFFSET 40
```

`getQuery(): QueryInterface`

Builds and returns the SQL without executing. Returns a `SqlQuery` that implements both `QueryInterface` and `ArrayAccess`:

```
$query = $qb->table('products')->where('price?>1000')->getQuery();

echo $query['sql'];         // SELECT * FROM products WHERE (price > :param_price_...)
echo $query['parameters'];  // ['param_price_...' => '1000']
```

`execute(): array`

Builds and executes the query, returning rows as an array of associative arrays.

```
$rows = $qb->table('products')->where('price?>1000')->execute();
```

Complete Query Examples

Simple Filters

```
// Active customers.
$qb->table('customers')->where('status?=active')->execute();

// Products in a price range.
$qb->table('products')->where('price?between:100,1000')->execute();

// Soft-deleted records.
$qb->table('users')->where('deleted_at?isnot:null')->execute();

// March 2024 invoices (SQLite).
$qb->table('invoices')->where('date?period:202403')->execute();
```

Composite Conditions

```
// AND: software products over $200.
```

```
$qb->table('products')
   ->where(['category?=software', 'price?>200'])
   ->execute();

// OR: electronics or hardware.
$qb->table('products')
   ->where('category?=electronics')
   ->orWhere('category?=hardware')
   ->execute();

// AND + OR: active customers who are persons OR whose tax_id starts with "78".
$qb->table('customers')
   ->where('status?=active')
   ->andWhereOr(['type?=person', 'tax_id?^78'])
   ->execute();
```

Path-Based Joins

```
// Auto-generated JOIN from path.
$qb->select('c.name AS customer_name, i.number, i.total')
   ->where('customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000')
   ->execute();
// → SELECT … FROM customers AS c INNER JOIN invoices AS i ON c.id = i.customer_id
WHERE i.total > :p

// Multi-level join.
$qb->select('p.name, i.number, c.name AS customer')
   ->where([
       'products[alias:p]__category?=electronics',
'products[alias:p]__invoice_details[on:id=product_id,alias:id]__invoices[on:invoice_id
=id,alias:i]__status?=paid',
'products[alias:p]__invoice_details[on:id=product_id,alias:id]__invoices[on:invoice_id
=id,alias:i]__customers[on:customer_id=id,alias:c]__type?=company',
   ])
   ->execute();
```

Subquery / EXISTS

```
// Invoices with no payments.
$qb->table('invoices')
   ->where('___payments[on:id=invoice_id]?is:empty')
   ->execute();
// → SELECT * FROM invoices WHERE NOT EXISTS (SELECT 1 FROM payments WHERE invoices.id
= payments.invoice_id)

// Invoices with at least one pending payment.
$qb->table('invoices')
   ->where('___payments[on:id=invoice_id]__status?=pending')
   ->execute();
```

```
// Invoices where total payments >= 1200.
$qb->table('invoices')
   ->where('___payments[on:id=invoice_id]__SUM(amount)?>=1200')
   ->execute();
```

Grouping, Having, Ordering, Pagination

```
// Category stats.
$qb->table('products')
   ->select('category, AVG(price) AS avg_price')
   ->groupBy('category')
   ->having('AVG(price)?>500')
   ->orderBy('avg_price', 'DESC')
   ->limit(10)
   ->execute();

// Paginated customer list.
$qb->table('customers')
   ->orderBy(['name' => 'ASC'])
   ->limit(25)
   ->offset(50)
   ->execute();
```

Explicit Joins

```
// Manual JOIN for a complex ON clause.
$qb->table('invoices', 'i')
   ->select('i.id, i.number, i.total, c.name AS customer_name')
   ->innerJoin('customers', 'i.customer_id = c.id', 'c')
   ->where('i.status?=paid')
   ->execute();

// LEFT JOIN with GROUP BY.
$qb->table('customers', 'c')
   ->select('c.name, COUNT(i.id) AS invoice_count')
   ->leftJoin('invoices', 'c.id = i.customer_id', 'i')
   ->groupBy(['c.id', 'c.name'])
   ->execute();
```

Declarative Query Configuration

`QueryConfig` provides an array-based way to describe a query. It is particularly useful for storing query templates in files and for API-driven queries.

Basic Usage

```
use Derafu\Query\Config\QueryConfig;

$config = new QueryConfig([
    'table'   => 'products',
    'select'  => 'id, name, price',
    'where'   => 'category?=electronics',
    'orderBy' => ['price' => 'DESC'],
    'limit'   => 10,
]);

$result = $config->applyTo($qb)->execute();
```

`applyTo()` calls the corresponding builder methods in order and returns the modified builder, so you can chain additional calls after:

```
$builder = $config->applyTo($qb);
if ($extraFilter) {
    $builder->andWhere('status?=active');
}
$rows = $builder->execute();
```

Loading from Files

```
// YAML file.
$config = QueryConfig::fromYamlFile('/path/to/queries/product_report.yaml');

// JSON file.
$config = QueryConfig::fromJsonFile('/path/to/queries/sales.json');

// Auto-detect by extension (.yml, .yaml, .json).
$config = QueryConfig::fromFile('/path/to/query.yaml');

// From strings.
$config = QueryConfig::fromYamlString($yamlString);
$config = QueryConfig::fromJsonString($jsonString);
```

YAML Template Example

```
# recent_products.yaml
table: products
select: id, name, price, created_at
where: deleted_at?is:null
orderBy:
    created_at: DESC
limit: 20
```

```
$config  = QueryConfig::fromYamlFile('queries/recent_products.yaml');
$builder = $config->applyTo($qb);

// Add extra filter at runtime.
if ($category) {
    $builder->andWhere('category?=' . $category);
}

$rows = $builder->execute();
```

Complete Configuration Reference

All `QueryConfig` keys and their equivalent builder calls:

Key	Builder Method	Value Type
`table`	`table()`	string
`alias`	`table($t, $alias)`	string
`select`	`select()`	string or array
`distinct`	`distinct(true)`	boolean
`where`	`where()`	string, array, or nested
`andWhere`	`andWhere()`	string or array
`orWhere`	`orWhere()`	string or array
`andWhereOr`	`andWhereOr()`	array of arrays
`innerJoin`	`innerJoin()`	`{table, condition, alias?}`
`leftJoin`	`leftJoin()`	`{table, condition, alias?}`
`rightJoin`	`rightJoin()`	`{table, condition, alias?}`
`crossJoin`	`crossJoin()`	`{table, alias?}`
`groupBy`	`groupBy()`	string or array
`having`	`having()`	string or array
`orderBy`	`orderBy()`	`{column: direction}`
`limit`	`limit()`	integer
`offset`	`offset()`	integer

```
$config = new QueryConfig([
    'table'    => 'invoices',
    'alias'    => 'i',
    'select'   => 'i.id, i.number, c.name AS customer_name',
```

```
    'distinct' => true,

    'where'      => 'i.status?=paid',
    'andWhere'   => 'i.total?>1000',
    'orWhere'    => 'i.date?period:202403',
    'andWhereOr' => [
        ['i.category?=service', 'i.total?>500'],
        ['i.category?=product', 'i.total?>1000'],
    ],

    'innerJoin' => ['table' => 'customers', 'alias' => 'c', 'condition' =>
'i.customer_id = c.id'],

    'groupBy' => ['i.status'],
    'having'  => 'COUNT(*)?>1',
    'orderBy' => ['i.created_at' => 'DESC'],
    'limit'   => 20,
    'offset'  => 40,
]);
```

API-Driven Queries

`QueryConfig` is well-suited for accepting query parameters from an API:

```
$requestData = $request->getJsonBody();

$config = new QueryConfig([
    'table'   => 'products',
    'where'   => $requestData['filters'] ?? [],
    'orderBy' => $requestData['sort'] ?? ['id' => 'ASC'],
    'limit'   => min($requestData['limit'] ?? 20, 100),
    'offset'  => max($requestData['offset'] ?? 0, 0),
]);

$rows = $config->applyTo($qb)->execute();
```

Ensure the `where` filters are either pre-validated strings that only allow known column names, or use the full path syntax so that only correctly-structured expressions reach the parser.

Last updated on 09/09/2026