Path Syntax

Path Syntax

Anonymous · 09/09/2026

Path Syntax

The **path** is the left-hand side of a `path?filter` expression. It identifies which column to apply the filter to, and optionally encodes the JOIN chain required to reach that column.

Simple Paths (Single Segment)

A simple path is just a column name:

```
price
status
created_at
deleted_at
```

When used with `SqlQueryBuilder.where()`, the column is unqualified — it refers to a column in the driving table set via `table()` or `from()`.

When used with the Doctrine ORM bridge, single-segment paths are automatically qualified with the root entity alias (e.g. `status` → `c.status`).

Multi-Segment Paths (Join Chains)

Segments are separated by double underscores `__`. The first segment is the **base table**, intermediate segments are **join targets**, and the last segment is the **column**:

```
customers__name
invoices__customers__name
products__invoice_details__invoices__customers__type
```

```
customers  __  name
^— table      ^— column
```

```
invoices  __  customers  __  name
^— table     ^— join target   ^— column
```

When `SqlQueryBuilder` processes a multi-segment path, it automatically generates INNER JOIN clauses for all intermediate segments that carry `on:` options. The base table is inferred from the first segment when no explicit `table()` call has been made.

Column Qualification

`SqlBuilderWhere` qualifies the column with the alias (or name) of the segment immediately before it:

```
invoices__customers__name          →  customers.name
invoices[alias:i]__customers[alias:c]__name  →  c.name
```

Segment Options

Each segment can carry metadata inside square brackets `[key:value,key2:value2]`:

```
customers[alias:c]
invoices[on:id=customer_id,alias:i,join:left]
```

`alias:value`

Sets an alias for the table in the generated SQL. The alias is used for column qualification, JOIN declarations, and correlation conditions.

```
customers[alias:c]__invoices[alias:i]__total
→  FROM customers AS c INNER JOIN invoices AS i …  →  i.total
```

`on:left_col=right_col`

Defines the JOIN condition between the previous segment and this segment. `left_col` belongs to the previous table; `right_col` belongs to this table.

```
invoices__customers[on:customer_id=id]__name
→  … INNER JOIN customers ON invoices.customer_id = customers.id
```

Multiple `on:` pairs in the same brackets add multiple conditions (AND):

```
orders__items[on:order_id=id,on:branch_id=branch_id]__product
→  … INNER JOIN items ON orders.order_id = items.id AND orders.branch_id =
```

```
items.branch_id
```

`join:type`

Overrides the join type for this segment. Valid values: `inner` (default), `left`, `right`, `cross`.

```
customers__invoices[on:id=customer_id,join:left]__total
→  … LEFT JOIN invoices ON customers.id = invoices.customer_id
```

Combining Options

All options can be combined in any order, separated by commas:

```
customers[alias:c]__invoices[on:id=customer_id,join:left,alias:i]__total
```

The segment above sets `alias=i`, `on.id=customer_id`, and `join=left` simultaneously.

Exists/Subquery Paths (`__`)

A path that starts with **triple underscore** `___` generates a correlated subquery instead of a JOIN. This is the mechanism for filtering by the existence or properties of child records.

Basic Existence Check

```
___payments[on:id=invoice_id]
```

No column segment → pure existence check. Combine with `is:empty` or `isnot:empty`:

```
___payments[on:id=invoice_id]?is:empty       →  NOT EXISTS (SELECT 1 FROM payments
WHERE …)
___payments[on:id=invoice_id]?isnot:empty    →  EXISTS (SELECT 1 FROM payments WHERE
…)
```

Column Filter Inside the Subquery

Adding a column segment after `__` filters on that column inside the `EXISTS`:

```
___payments[on:id=invoice_id]__status?=pending
→  EXISTS (SELECT 1 FROM payments WHERE invoices.id = payments.invoice_id AND
payments.status = :p)
```

Any operator can be used for the column filter inside the subquery:

```
___payments[on:id=invoice_id]__amount?>500
___payments[on:id=invoice_id]__method?in:card,transfer
___payments[on:id=invoice_id]__created_at?date:20240101
```

Aggregate Scalar Subqueries

When the column segment is an aggregate function call, a scalar subquery is generated:

```
___payments[on:id=invoice_id]__SUM(amount)?>=1200
→  (SELECT SUM(p.amount) FROM payments p WHERE p.invoice_id = invoices.id) >= :p
```

Supported aggregate functions: `SUM`, `AVG`, `COUNT`, `MIN`, `MAX`.

```
___payments[on:id=invoice_id]__COUNT(*)?>1
___invoices[on:id=customer_id]__AVG(total)?<1000
___invoices[on:id=customer_id]__MIN(total)?>=100
```

`COUNT(*)` optimizations: when the comparison is a pure existence check, it is automatically rewritten to `EXISTS` / `NOT EXISTS`:

Expression	Rewritten as
`COUNT(*)?=0`	`NOT EXISTS (…)`
`COUNT(*)?>0`	`EXISTS (…)`
`COUNT(*)?!=0`	`EXISTS (…)`
`COUNT(*)?>=1`	`EXISTS (…)`
`COUNT(*)?<1`	`NOT EXISTS (…)`
`COUNT(*)?<=0`	`NOT EXISTS (…)`
`COUNT(*)?>1`	scalar subquery (kept as-is)

Aliases on Exists Segments

You can add `alias:` to a subquery path segment:

```
___payments[alias:p,on:id=invoice_id]__status?=pending
→  EXISTS (SELECT 1 FROM payments AS p WHERE invoices.id = p.invoice_id AND p.status =
:p)
```

Nested Exists (Multi-Level)

Chain multiple `___` separators to traverse deeper relationships:

```
___payments[on:id=invoice_id]___items[on:id=payment_id]__amount?>100
```

This generates an EXISTS subquery with an inner JOIN to `items`, filtering on `items.amount`.

Function Notation in Paths

The **last segment** may be an aggregate or SQL function call. This is used for HAVING conditions and similar:

```
AVG(price)?>500     ← in a having() call
SUM(i.total)?>1000  ← qualified with a prior alias
```

When a parent segment exists, `SqlBuilderWhere` qualifies the function arguments automatically:

```
products[alias:p]__AVG(price)?>500   →  AVG(p.price) > :p
```

Path Validation Rules

`PathParser` enforces these rules:

- A path cannot be empty.
- Segment names cannot be empty — `author__` (trailing `__`) and `__books` (leading `__`) are invalid.
- Segment names must start and end with `[a-zA-Z0-9_]` (closing `)` also allowed at the end, for function calls).
- Option keys and values must both be non-empty.
- An `on:` option value must contain `=` with non-empty parts on both sides.
- Exists paths (`___`) must have a non-empty body after the triple underscore.

Path Examples Reference

Path	Result
`status`	`status` (no qualification)
`invoices__status`	`invoices.status`

Path	Result
`invoices[alias:i]__status`	`i.status`
`customers[alias:c]__invoices[on:id=customer_id,alias:i]__total`	`i.total` with JOIN
`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]__name`	3-level deep join
`___payments[on:id=invoice_id]`	Subquery path, no column
`___payments[on:id=invoice_id]__status`	Subquery path with column
`___payments[on:id=invoice_id]__SUM(amount)`	Aggregate subquery
`___payments[on:id=invoice_id]___items[on:id=payment_id]__amount`	Nested subquery

Paths in Doctrine ORM

The Doctrine ORM bridge handles paths differently from SQL bridges:

- **Single-segment paths** are automatically qualified with the root entity alias (e.g. `status` → `c.status`).
- **Multi-segment paths** use the association name from Doctrine entity mappings, not SQL column names. The `on:` option is **ignored** — Doctrine resolves join conditions from the mapping.
- **EXISTS paths** use DQL `SIZE(alias.assoc) = 0` for simple emptiness checks, and correlated `EXISTS(SELECT sub.id FROM EntityClass sub WHERE …)` for column filters.

Last updated on 09/09/2026