Framework Bridges

Framework Bridges

Anonymous · 09/09/2026

Framework Bridges

Bridges allow you to apply `derafu/query` expressions to existing third-party query builder instances — without replacing them. You keep your existing query builder and add Derafu filters on top.

All bridges accept a parsed `ConditionInterface | CompositeConditionInterface` and apply it to the query builder. You parse expressions with `CompositeExpressionParser` and pass the result to the bridge.

Shared Setup: The Expression Parser

All bridges share the same expression parser setup:

```
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;

$loader  = new OperatorLoader();
$manager = new OperatorManager(
    $loader->loadFromFile('vendor/derafu/query/resources/operators.yaml')
);
$parser = new CompositeExpressionParser(
    new ExpressionParser(new PathParser(), new FilterParser($manager))
);
```

Doctrine DBAL Bridge

Class: `Derafu\Query\Bridge\DoctrineDBALQueryBuilderConditionApplier`

Applies conditions to a Doctrine DBAL `QueryBuilder`. Supports all operators including date, regex, and bitwise operators.

Methods

```
public function apply(object $queryBuilder,
ConditionInterface|CompositeConditionInterface $condition): void
public function applyHaving(object $queryBuilder,
ConditionInterface|CompositeConditionInterface $condition): void
```

Usage

```
use Derafu\Query\Bridge\DoctrineDBALQueryBuilderConditionApplier;

$applier = new DoctrineDBALQueryBuilderConditionApplier();

// Parse the expression.
$condition = $parser->parse('status?=active&&total?>1000');

// Apply to an existing DBAL QueryBuilder.
$applier->apply($dbalQb, $condition);

// The QB now has: WHERE (status = :param_status_... AND total > :param_total_...)
$rows = $dbalQb->executeQuery()->fetchAllAssociative();
```

FROM and JOIN Inference

When a condition uses multi-segment paths, the bridge:

1. **Infers `FROM`** from the first segment of the first multi-segment path, if no `FROM` has been set yet.
2. **Generates JOINs** from intermediate segments, with deduplication (same alias → skip).

```
// No from() call needed — it's inferred from the path.
$condition = $parser->parse(
    'customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000'
);
$applier->apply($dbalQb, $condition);
// Sets FROM customers AS c, adds INNER JOIN invoices AS i ON c.id = i.customer_id
// WHERE i.total > :p
```

Driver Detection

The bridge reads the database platform from the DBAL connection (via reflection on the internal `connection` property) and maps it to the driver name used by `SqlBuilderWhere`:

Platform	Driver
`PostgreSQLPlatform`	`pgsql`
`MySQLPlatform`	`mysql`
`SQLitePlatform`	`sqlite`
`SQLServerPlatform`	`sqlsrv`
`OraclePlatform`	`oci`
Other	`pgsql` (fallback)

HAVING

```
$dbalQb->groupBy('status');
$condition = $parser->parse('COUNT(*)?>1');
$applier->applyHaving($dbalQb, $condition);
```

Doctrine ORM Bridge

Class: `Derafu\Query\Bridge\DoctrineORMQueryBuilderConditionApplier`

Applies conditions to a Doctrine ORM `QueryBuilder` as DQL. This bridge generates DQL-compatible fragments — it does not produce raw SQL.

Methods

```
public function apply(object $queryBuilder,
ConditionInterface|CompositeConditionInterface $condition): void
public function applyHaving(object $queryBuilder,
ConditionInterface|CompositeConditionInterface $condition): void
```

Usage

```
use Derafu\Query\Bridge\DoctrineORMQueryBuilderConditionApplier;

$applier = new DoctrineORMQueryBuilderConditionApplier();

$condition = $parser->parse('status?=active&&total?>1000');

$ormQb = $em->createQueryBuilder()
    ->select('i')
    ->from(Invoice::class, 'i');

$applier->apply($ormQb, $condition);
```

```
$invoices = $ormQb->getQuery()->getResult();
```

Single-Segment Path Qualification

Single-segment paths like `status` are automatically qualified with the root entity alias. The root alias is read from the first `FROM` clause:

```
status?=active   →   i.status = :param_status_...   (when root alias is 'i')
```

Multi-segment paths are left as-is, allowing explicit alias control.

JOIN Generation

The ORM bridge adds JOINs to the QueryBuilder using Doctrine's association names — the `on:` option in path segments is **ignored**. Doctrine derives join conditions from entity mappings.

```
invoices[alias:i]__payments[alias:p]__status
```

If `Invoice` has a `payments` association, the bridge calls:

```
$ormQb->innerJoin('i.payments', 'p');
```

EXISTS Paths in DQL

Subquery paths (`___`) are translated to DQL:

Expression	DQL
`___payments?is:empty`	`SIZE(i.payments) = 0`
`___payments?isnot:empty`	`SIZE(i.payments) > 0`
`___payments__status?=pending`	`EXISTS(SELECT _payments0.id FROM App\Entity\Payment _payments0 WHERE _payments0.invoice = i AND _payments0.status = :p)`
`___invoices__AVG(total)?<1000`	`(SELECT AVG(_i0.total) FROM App\Entity\Invoice _i0 WHERE _i0.customer = i) < :p`
`___invoices__COUNT(*)?=0`	`SIZE(i.invoices) = 0` (optimized)

DQL-Incompatible Operators

The following operator types throw `UnsupportedOperatorException` when used with this bridge:

- **`date` type**: `date:`, `month:`, `year:`, `period:` — require SQL functions not available in DQL.
- **`binary` type**: `b&`, `b|`, `b^`, etc. — bitwise SQL not supported in DQL.

- **`regexp` type**: `~`, `~*`, `similarto:`, etc. — database-specific regex not in DQL.
- **`ilike:` and `notilike:`** — ILIKE is not DQL.

In the API Platform `SmartFilter`, these exceptions are caught and the filter is silently skipped.

Illuminate (Laravel) Bridge

Class: `Derafu\Query\Bridge\IlluminateQueryBuilderConditionApplier`

Applies conditions to Illuminate's `Query\Builder` or Eloquent's `Builder`. Both are accepted — Eloquent builders are resolved to their underlying `Query\Builder` via `toBase()`.

Methods

```
public function apply(object $queryBuilder,
ConditionInterface|CompositeConditionInterface $condition): void
public function applyHaving(object $queryBuilder,
ConditionInterface|CompositeConditionInterface $condition): void
```

Usage

```
use Derafu\Query\Bridge\IlluminateQueryBuilderConditionApplier;

$applier = new IlluminateQueryBuilderConditionApplier();

$condition = $parser->parse('status?=active&&total?>1000');

// With a plain Query\Builder.
$qb = DB::table('invoices');
$applier->apply($qb, $condition);
$rows = $qb->get();

// With an Eloquent Builder.
$builder = Invoice::query();
$applier->apply($builder, $condition);
$invoices = $builder->get();
```

FROM and JOIN Inference

Same behavior as the DBAL bridge: infers the `FROM` table from multi-segment paths and adds `join()`, `leftJoin()`, or `rightJoin()` calls as needed. Duplicate joins (same table reference) are skipped.

Driver Detection

The bridge reads the driver name directly from `Connection::getDriverName()`:

Driver name	SQL generated
`pgsql`	PostgreSQL SQL
`mysql`	MySQL SQL
`sqlite`	SQLite SQL

HAVING

```
$condition = $parser->parse('AVG(price)?>500');
$applier->applyHaving($qb, $condition);
// Calls: $qb->havingRaw($sql, $params)
```

API Platform Bridge

Class: `Derafu\Query\Bridge\ApiPlatform\SmartFilter`

Integrates `derafu/query` as an API Platform filter via the `QueryParameter` attribute. When a request comes in, the filter builds a Derafu expression from the parameter's property name and value, parses it, and applies it to the ORM `QueryBuilder`.

Registration

```
use ApiPlatform\Metadata\ApiResource;
use ApiPlatform\Metadata\QueryParameter;
use Derafu\Query\Bridge\ApiPlatform\SmartFilter;

#[ApiResource]
#[QueryParameter(key: 'price',  property: 'price',  filter: SmartFilter::class)]
#[QueryParameter(key: 'status', property: 'status', filter: SmartFilter::class)]
class Product
{
    // …
}
```

URL Filter Syntax

The URL parameter value is the filter part (operator + value); the property name provides the path segment:

```
GET /api/products?price=>1000&status=in:paid,issued
```

The `SmartFilter` builds expressions internally:

- `price` property + `>1000` value → expression `price?>1000`
- `status` property + `in:paid,issued` value → expression `status?in:paid,issued`

Generic SmartFilter Property

For a single parameter that accepts a full composite expression, use the special `__derafu_smart_filter` property name:

```
#[QueryParameter(key: 'filter', property: '__derafu_smart_filter', filter:
SmartFilter::class)]
```

```
GET /api/products?filter=status?=active&&price?>1000
```

Dependency Injection (Symfony)

Register `CompositeExpressionParser` and `DoctrineORMQueryBuilderConditionApplier` as services and inject them into `SmartFilter`:

```
# services.yaml
services:
    Derafu\Query\Bridge\ApiPlatform\SmartFilter:
        arguments:
            $compositeParser:
'@Derafu\Query\Filter\Contract\CompositeExpressionParserInterface'
            $applier:
'@Derafu\Query\Bridge\DoctrineORMQueryBuilderConditionApplier'
```

Error Handling

`SmartFilter` silently catches two exceptions:

- **`UnsupportedOperatorException`**: DQL-incompatible operators (`date:`, `b&`, `ilike:`, regex) — the filter is skipped.
- **Any `Throwable`**: Malformed expressions — the filter is skipped.

This prevents a bad filter parameter from crashing the entire collection endpoint.

Bridge Comparison

Feature	Doctrine DBAL	Doctrine ORM	Illuminate
SQL operators (all)	✅	⚠️†	✅
Date operators	✅	✅	✅
Regex operators	✅	✅	✅
Bitwise operators	✅	✅	✅
`ilike:` / `notilike:`	✅	✅	✅
FROM inference from paths	✅	⚠️‡	✅
JOIN from path segments	✅	✅ (ORM assoc)	✅
EXISTS paths	✅	✅ (SIZE/EXISTS DQL)	✅
Aggregate subqueries	✅	✅	✅

† Compatible operators only — DQL-incompatible operators throw `UnsupportedOperatorException`.
‡ ORM bridge requires an explicit `FROM` / `from()` call before applying conditions.

Last updated on 09/09/2026