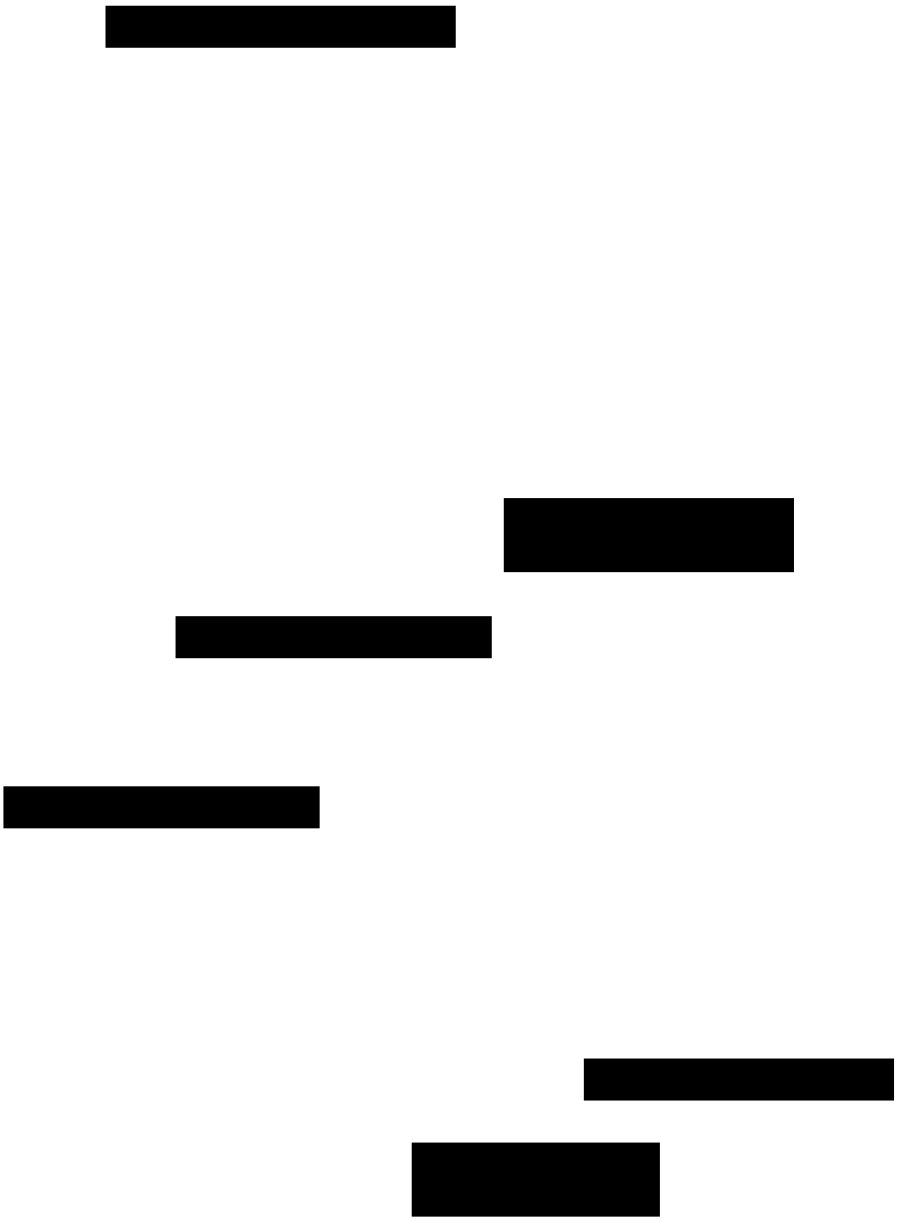
Architecture

Architecture

Anonymous · 09/09/2026

Architecture

`derafu/query` is organized into six cooperating layers. Understanding them helps you choose which classes to instantiate, which bridges to use, and where to add custom behavior.



Layers at a Glance

Namespace	Responsibility
`Derafu\Query\Filter`	Parse string expressions into structured condition objects
`Derafu\Query\Operator`	Load, validate, and manage operator definitions from YAML
`Derafu\Query\Builder`	Build SQL strings and named-parameter arrays from conditions
`Derafu\Query\Engine`	Execute SQL against a real database connection
`Derafu\Query\Bridge`	Apply conditions to third-party query builders (Doctrine, Illuminate, etc.)
`Derafu\Query\Config`	Load declarative query definitions from arrays, YAML, or JSON

Filter Layer

The filter layer turns a raw string expression like `customers[alias:c]__invoices[on:id=customer_id,alias:i]__total?>1000` into an object tree.

Key Classes

Class	Interface	Role
`CompositeExpressionParser`	`CompositeExpressionParserInterface`	Entry point. Parses composite expressions with `&&`, `||`, `()` into a tree of conditions.
`ExpressionParser`	`ExpressionParserInterface`	Parses a single `path?filter` string into a `Condition`.
`PathParser`	`PathParserInterface`	Splits the path part (`table__column`) into `Segment` objects.
`FilterParser`	`FilterParserInterface`	Matches the filter part (`>1000`) against known operators.
`Path`	`PathInterface`	Immutable value object holding an ordered list of `Segment` objects.
`Segment`	`SegmentInterface`	Immutable value object for one path segment: name + options.
`Filter`	`FilterInterface`	Immutable value object: operator + raw value string.

Class	Interface	Role
`Condition`	`ConditionInterface`	Ties a `Path` to a `Filter`. Carries a `literal` flag.
`CompositeCondition`	`CompositeConditionInterface`	AND or OR container of `Condition`/`CompositeCondition` objects.

Parsing Pipeline

```
"status?=active&&total?>1000"
       │
       ▼ CompositeExpressionParser
  CompositeCondition (AND)
  ├── Condition
  │   ├── Path [Segment("status")]
  │   └── Filter [Operator("="), value="active"]
  └── Condition
      ├── Path [Segment("total")]
      └── Filter [Operator(">"), value="1000"]
```

The `literal` flag on `Condition` distinguishes normal filter values from expression references (used with the `?E` marker for self-referencing conditions like `id?E!=other_alias.id`).

Operator Layer

Operators are defined in `resources/operators.yaml` and loaded at startup. Each operator carries:

- A **symbol** (e.g. `>=`, `like:`, `b&`)
- A **type** classifying its behavior
- Engine-specific **SQL templates** with `{{column}}` / `{{value}}` / `{{values}}` / `{{value_1}}` / `{{value_2}}` placeholders
- An optional **validation pattern** (regex) for the value
- Optional **casting rules** that transform the value before binding
- An optional **alias** pointing to another operator's SQL template

Key Classes

Class	Interface	Role
`Operator`	`OperatorInterface`	Immutable value object for one operator definition
`OperatorLoader`	`OperatorLoaderInterface`	Reads and validates `operators.yaml`
`OperatorManager`	`OperatorManagerInterface`	Registry; provides operators sorted by symbol length for greedy matching
`OperatorManagerFactory`	`OperatorManagerFactoryInterface`	Convenience factory: loader + manager in one call

Builder Layer

The builder layer converts condition objects into SQL strings with named parameters.

Key Classes

Class	Interface	Role
`SqlBuilderWhere`	`QueryBuilderWhereInterface`	Converts a single `Condition` or `CompositeCondition` into a WHERE fragment + parameters
`SqlQueryBuilder`	`QueryBuilderInterface`	Full query builder: SELECT, FROM, JOIN, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET
`SqlQuery`	`QueryInterface`	Immutable value object holding SQL string and parameters. Implements `ArrayAccess` for `$q['sql']` and `$q['parameters']`.
`SqlSanitizerTrait`	—	Sanitizes identifiers and expressions; supports a custom quoting callback

SqlBuilderWhere Internals

`SqlBuilderWhere` is constructed with a database driver name (`pgsql`, `mysql`, `sqlite`). For each condition it:

1. Resolves the effective SQL template (operator's own `sql` key, or its base operator's).
2. Validates the raw value against the operator's pattern (if defined).
3. Normalizes the value: booleans → `0`/`1`, arrays → delimiter-joined string.
4. Applies casting rules (e.g. `like_start` appends `%`, `date` normalizes `YYYYMMDD` → `YYYY-MM-DD`).
5. Creates named parameters (`param_{column}_{uniqid}`).
6. Replaces `{{column}}`, `{{value}}`, `{{values}}`, `{{value_1}}`, `{{value_2}}` in the template.

For composite conditions it recursively builds each child and joins them with `AND` / `OR` inside parentheses.

For EXISTS paths (starting with `___`) it generates correlated `EXISTS (SELECT 1 FROM …)` or `NOT EXISTS (…)` subqueries. Aggregate column segments like `SUM(amount)` produce scalar subqueries: `(SELECT SUM(p.amount) FROM payments p WHERE …) >= :param`.

Column Resolution from Paths

When a path has more than one segment, `SqlBuilderWhere` qualifies the column with the previous segment's alias (or name):

```
authors__books__title  →  books.title
```

```
authors[alias:a]__books[alias:b]__title  →  b.title
```

For `SqlQueryBuilder`, intermediate segments automatically become JOIN clauses (INNER by default; override with `join:left` option).

Engine Layer

The engine layer executes the SQL produced by the builder against a real connection.

Class	Interface	Role
`PdoEngine`	`SqlEngineInterface`	Wraps a `PDO` instance; prepares, executes, and fetches rows
`DoctrineEngine`	`SqlEngineInterface`	Wraps a Doctrine DBAL `Connection`
`AbstractSqlEngine`	—	Shared `getDriver()` detection logic

Bridge Layer

Bridges apply parsed conditions to existing third-party query builder instances, without requiring the full `SqlQueryBuilder`.

Class	Target QB	Notes
`DoctrineDBALQueryBuilderConditionApplier`	Doctrine DBAL `QueryBuilder`	Resolves driver via reflection; handles FROM inference and JOIN deduplication
`DoctrineORMQueryBuilderConditionApplier`	Doctrine ORM `QueryBuilder`	Generates DQL; rewrites single-segment paths to qualify with root alias; throws `UnsupportedOperatorException` for DQL-incompatible operators
`IlluminateQueryBuilderConditionApplier`	Illuminate `Query\Builder` / `Eloquent\Builder`	Accepts both; resolves Eloquent to base via `toBase()`
`SmartFilter` (API Platform)	Doctrine ORM `QueryBuilder`	Wraps `DoctrineORMQueryBuilderConditionApplier` for use as an API Platform `#[QueryParameter]` filter

All bridge appliers share `ConditionApplierTrait`, which provides:

- `buildConditionSql()` — compiles to SQL via `SqlBuilderWhere`
- `extractPaths()` — collects all paths from a condition tree
- `inferFromTable()` — detects the base table from multi-segment paths

- `buildJoinSpecsFromPaths()` — builds JOIN specs for SQL-level bridges
- `buildOrmJoinSpecsFromPaths()` — builds JOIN specs for Doctrine ORM DQL

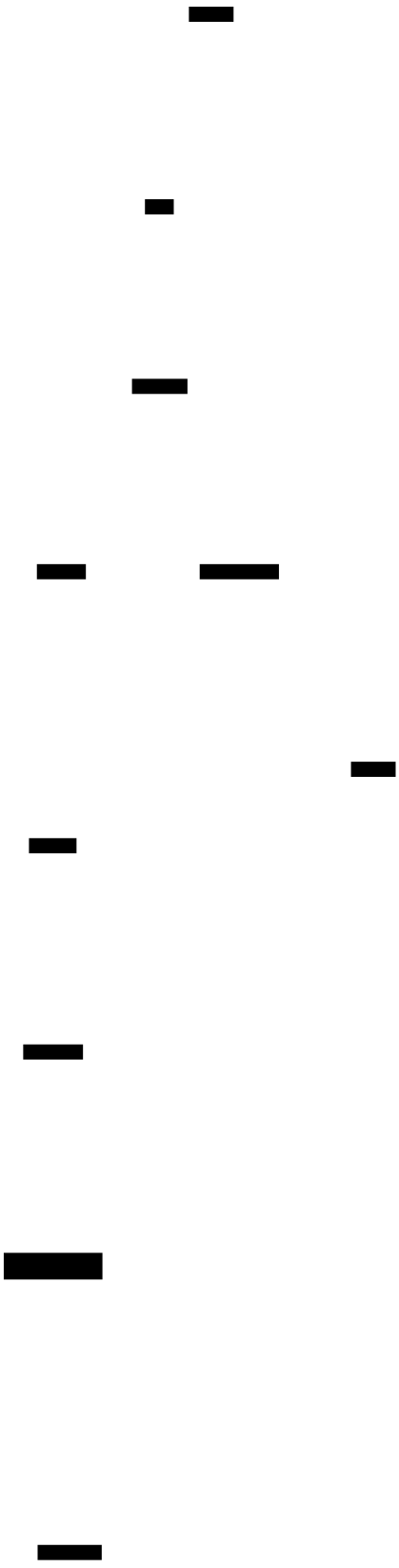
Config Layer

The config layer provides a declarative alternative to the fluent builder API.

Class	Role
`QueryConfig`	Wraps an array; calls builder methods when `applyTo($builder)` is invoked
`YamlConfigLoader`	Loads YAML files or strings into arrays
`JsonConfigLoader`	Loads JSON files or strings into arrays

`QueryConfig::fromFile()` auto-detects YAML vs JSON by file extension.

Data Flow Summary



```
User provides string expression
        │
        ▼
CompositeExpressionParser
  ├── ExpressionParser
  │     ├── PathParser  → Path (Segments)
  │     └── FilterParser → Filter (Operator + value)
  └── returns Condition | CompositeCondition
        │
        ▼
SqlQueryBuilder  (or a Bridge applier)
  └── SqlBuilderWhere
        ├── Resolves SQL template (operator + engine)
        ├── Normalizes & casts value
        ├── Creates named parameters
        └── Returns SqlQuery { sql, parameters }
              │
              ▼
        Engine::execute(sql, parameters)
              │
              ▼
        array of result rows
```

Last updated on 09/09/2026