Docs

Selector Project

Derafu Selector

Anonymous · 24/08/2026 · #php

Table of Contents

Selector Project

Derafu Selector

Anonymous · 24/08/2026 · #php

Derafu Selector

Last updated on 24/08/2026

Introduction

Elegant Data Structure Navigation for PHP

Anonymous · 24/08/2026

Elegant Data Structure Navigation for PHP

A powerful, flexible, and efficient PHP library for querying, extracting, and modifying complex data structures using a **declarative selector syntax**.

Why Derafu\Selector?

A Smarter Way to Work with Data Structures

Traditional PHP methods for accessing and modifying nested data structures (`$array['key']['subkey']`) are:

- **Error-prone**: Risk of `Undefined index` or `Trying to access array offset on null` errors.
- **Verbose & Hard to Read**: Requires multiple `isset()` checks and deep nesting.
- **Limited**: No built-in support for **filters, conditional selectors, or transformations**.

What Makes Derafu\Selector Unique?

Feature	Derafu\Selector	PHP Native (`isset()` & loops)	Other Libraries
Dot-Notation Access	Yes	No	⚠ Limited
Nested Selection with Conditions	Yes	No	⚠ Limited
Dynamic Path Resolution	Yes	No	⚠ Varies
Auto-Handles Undefined Keys	Yes	No	⚠ Partial
Filters & Expressions	Yes	No	⚠ Partial

Overview

Derafu Selector gives you a clean, robust way to work with nested arrays and objects in PHP. It eliminates the need for repetitive null checks, nested loops, and error-prone array access, replacing them with a powerful, declarative syntax.

```
// Instead of this:
$userName = isset($data['user']) && isset($data['user']['profile']) &&
isset($data['user']['profile']['name'])
    ? $data['user']['profile']['name']
    : null;

// Simply write this:
$userName = Selector::get($data, 'user.profile.name');

// Or even filter arrays with conditions:
$adminEmail = Selector::get($data, 'users[role=admin:email]');
```

Features

- **Powerful Dot Notation** - Access deeply nested data with simple paths (`user.profile.name`).
- **Array Access** - Use numeric indices to access array elements (`items[0].name`).
- **Filtering** - Find array elements by key/value conditions (`users[id=123:email]`).
- **Conditional Logic** - Use ternary-like expressions (`((type) = "admin" ? (admin_role) : (user_role))`).
- **Multiple Query Languages** - Support for native syntax, JSONPath, and JMESPath.
- **Error Handling** - No more “undefined index” or “trying to access array offset on null” errors.
- **Write Support** - Modify data structures with the same powerful syntax.
- **Zero Dependencies** - Lightweight core with optional support for JSONPath and JMESPath.

Installation

```
composer require derafu/selector
```

Quick Start

Reading Data

```
use Derafu\Selector\Selector;
```

```
$data = [
    'user' => [
        'name' => 'John Doe',
        'email' => 'john@example.com',
        'profile' => [
            'avatar' => 'john.jpg',
            'settings' => ['theme' => 'dark', 'notifications' => true],
        ],
        'roles' => ['editor', 'contributor'],
    ],
    'products' => [
        ['id' => 101, 'name' => 'Laptop', 'price' => 999],
        ['id' => 102, 'name' => 'Phone', 'price' => 699],
        ['id' => 103, 'name' => 'Headphones', 'price' => 149],
    ],
];

// Simple property access.
$name = Selector::get($data, 'user.name');  // "John Doe"

// Nested properties.
$theme = Selector::get($data, 'user.profile.settings.theme');  // "dark"

// Array elements by index.
$firstRole = Selector::get($data, 'user.roles[0]');  // "editor"

// Array filtering.
$laptop = Selector::get($data, 'products[id=101:name]');  // "Laptop"

// With default values.
$address = Selector::get($data, 'user.address', 'Not specified');  // "Not specified"

// Check if a path exists.
$hasAvatar = Selector::has($data, 'user.profile.avatar');  // true
```

Writing Data

```
// Set a simple value.
Selector::set($data, 'user.address', '123 Main St');

// Create nested structures automatically.
Selector::set($data, 'user.profile.settings.language', 'en');

// Update array elements.
Selector::set($data, 'products[id=101:price]', 899);

// Remove a value.
Selector::clear($data, 'user.profile.settings.notifications');
```

Advanced Selector Syntax

```
// Conditional selectors.
$role = Selector::get($data, '((user.type) = "admin" ? (admin_role) : (user_role))');

// String concatenation.
$greeting = Selector::get($data, '"Hello, "(user.name)"!"');  // "Hello, John Doe!"

// OR operator for fallbacks.
$contactInfo = Selector::get($data, 'user.phone||user.email');  // Uses email if phone
is null

// JSONPath integration.
$expensiveProducts = Selector::get($data, '$.products[?(@.price > 500)].name');  //
["Laptop", "Phone"]

// JMESPath integration.
$productNames = Selector::get($data, 'jmespath:products[*].name');  // ["Laptop",
"Phone", "Headphones"]
```

Last updated on 24/08/2026

Basic Usage

Basic Usage

Anonymous · 24/08/2026

Basic Usage

This guide covers the fundamentals of working with Derafu Selector for navigating and manipulating data structures.

Reading Data

The most common operation with Selector is to read values from complex data structures. The `get()` method provides a clean, safe way to extract values without worry about undefined indices or null values.

Basic Dot Notation

```
use Derafu\Selector\Selector;

$data = [
    'user' => [
        'name' => 'John Doe',
        'email' => 'john@example.com',
        'profile' => [
            'avatar' => 'john.jpg',
        ],
    ],
];

// Simple property access.
$name = Selector::get($data, 'user.name');  // "John Doe"

// Nested properties.
$avatar = Selector::get($data, 'user.profile.avatar');  // "john.jpg"

// Non-existent path returns null.
$age = Selector::get($data, 'user.age');  // null
```

Default Values

You can provide a default value that will be returned if the selector path doesn’t exist:

```
// With default value
$address = Selector::get($data, 'user.address', 'No address provided');  // "No
address provided"
```

Array Access

Access array elements by index:

```
$data = [
    'items' => ['apple', 'banana', 'cherry'],
];

$firstItem = Selector::get($data, 'items[0]');  // "apple"
$lastItem = Selector::get($data, 'items[2]');   // "cherry"
```

Combining Dot Notation and Array Access

```
$data = [
    'categories' => [
        'fruits' => ['apple', 'banana', 'cherry'],
        'vegetables' => ['carrot', 'broccoli', 'spinach'],
    ],
];

$firstFruit = Selector::get($data, 'categories.fruits[0]');  // "apple"
$secondVegetable = Selector::get($data, 'categories.vegetables[1]');  // "broccoli"
```

Writing Data

Writing data is just as easy as reading it. The `set()` method handles creating intermediate structures as needed.

Setting Simple Values

```
$data = ['user' => ['name' => 'John']];

// Update an existing value.
Selector::set($data, 'user.name', 'Jane');
// $data is now ['user' => ['name' => 'Jane']]

// Create a new property.
Selector::set($data, 'user.email', 'jane@example.com');
// $data is now ['user' => ['name' => 'Jane', 'email' => 'jane@example.com']]
```

Creating Nested Structures

Selector will automatically create intermediate arrays:

```
$data = [];

// Create a deep structure.
Selector::set($data, 'settings.theme.colors.primary', '#3366FF');
// $data is now ['settings' => ['theme' => ['colors' => ['primary' => '#3366FF']]]]
```

Working with Arrays

```
$data = ['items' => ['apple', 'banana']];

// Add or update an array element.
Selector::set($data, 'items[2]', 'cherry');
// $data is now ['items' => ['apple', 'banana', 'cherry']]

// Update a specific element.
Selector::set($data, 'items[0]', 'red apple');
// $data is now ['items' => ['red apple', 'banana', 'cherry']]
```

Merging Arrays

When setting an array value to an existing array path, the arrays are merged:

```
$data = [
    'settings' => [
        'display' => ['theme' => 'light'],
    ],
];

Selector::set($data, 'settings.display', ['fontSize' => 'large']);
// $data is now ['settings' => ['display' => ['theme' => 'light', 'fontSize' =>
'large']]]
```

Checking if Paths Exist

The `has()` method checks if a value exists at the specified path:

```
$data = [
    'user' => [
        'name' => 'John',
        'email' => null,
    ],
```

```
];

Selector::has($data, 'user.name');      // true
Selector::has($data, 'user.email');     // false (because the value is null)
Selector::has($data, 'user.address');   // false (path doesn't exist)
```

Clearing Values

Remove values from the data structure with the `clear()` method:

```
$data = [
    'user' => [
        'name' => 'John',
        'email' => 'john@example.com',
        'settings' => ['theme' => 'dark'],
    ],
];

// Remove a simple property.
Selector::clear($data, 'user.email');
// $data is now ['user' => ['name' => 'John', 'settings' => ['theme' => 'dark']]]

// Remove a nested property.
Selector::clear($data, 'user.settings.theme');
// $data is now ['user' => ['name' => 'John', 'settings' => []]]

// Parent arrays are cleaned up if they become empty.
Selector::clear($data, 'user.name');
// $data is now ['user' => ['settings' => []]]
```

Working with Different Data Types

Selector handles various data types properly:

```
$data = [
    'values' => [
        'string' => 'text',
        'number' => 42,
        'boolean' => true,
        'null_value' => null,
        'array' => [1, 2, 3],
        'object' => ['key' => 'value'],
    ],
];

Selector::get($data, 'values.string');     // "text"
```

```
Selector::get($data, 'values.number');     // 42
Selector::get($data, 'values.boolean');    // true
Selector::get($data, 'values.null_value'); // null
Selector::get($data, 'values.array');      // [1, 2, 3]
Selector::get($data, 'values.object');     // ['key' => 'value']
```

Error Handling

Selector is designed to be forgiving and avoid common PHP errors:

```
// No "undefined index" errors.
Selector::get($data, 'non.existent.path');  // returns null

// No "trying to access array offset on null" errors.
$data = ['user' => null];
Selector::get($data, 'user.name');  // returns null

// However, if you have syntax errors in your selector, you'll get a
SelectorException.
try {
    Selector::get($data, 'user..name');  // Invalid selector with double dot
} catch (\Derafu\Selector\Exception\SelectorException $e) {
    echo $e->getMessage();  // "Failed to read using selector 'user..name'..."
}
```

For more advanced usage, check out the other guides in the documentation.

Last updated on 24/08/2026

Working with Arrays and Filters

Working with Arrays and Filters

Anonymous · 24/08/2026

Working with Arrays and Filters

This guide focuses on techniques for working with arrays and using filters in Derafu Selector.

Basic Array Access

Arrays can be accessed using both dot notation and square bracket syntax:

```
use Derafu\Selector\Selector;

$data = [
    'items' => ['apple', 'banana', 'cherry'],
    'counts' => [5, 10, 15],
];

// Access entire array.
$allItems = Selector::get($data, 'items');
// ['apple', 'banana', 'cherry']

// First element.
$firstItem = Selector::get($data, 'items[0]');
// 'apple'

// Last element.
$lastItem = Selector::get($data, 'items[2]');
// 'cherry'

// Non-existent index.
$nonExistent = Selector::get($data, 'items[5]');
// null
```

Numeric Indexing

```
$data = [
    'matrix' => [
        [1, 2, 3],
        [4, 5, 6],
```

```
        [7, 8, 9],
    ],
];

// Accessing multi-dimensional arrays.
$center = Selector::get($data, 'matrix[1][1]');
// 5

// Alternative dot notation.
$center = Selector::get($data, 'matrix.1.1');
// 5

// Mixed notation.
$firstRow = Selector::get($data, 'matrix[0]');
// [1, 2, 3]
$firstRowThirdElement = Selector::get($data, 'matrix[0][2]');
// 3
```

Filtering Arrays by Condition

The most powerful feature for array handling is filtering by key/value conditions:

```
$data = [
    'users' => [
        ['id' => 1, 'name' => 'John', 'role' => 'admin'],
        ['id' => 2, 'name' => 'Jane', 'role' => 'user'],
        ['id' => 3, 'name' => 'Bob', 'role' => 'user'],
    ]
];

// Find user by ID.
$userName = Selector::get($data, 'users[id=2:name]');
// 'Jane'

// Find user by role.
$adminName = Selector::get($data, 'users[role=admin:name]');
// 'John'

// Non-matching filter.
$nonExistent = Selector::get($data, 'users[id=99:name]');
// null
```

Syntax for Dependent Selectors

The syntax for array filtering with dependent selectors is:

```
arrayKey[requiredKey=requiredValue:dependentKey]
```

Where:

- `arrayKey` is the key containing the array to search in.
- `requiredKey` is the key to match in each array element.
- `requiredValue` is the value that `requiredKey` should equal.
- `dependentKey` is the key to extract from the matching element.

Complex Filtering

You can combine filters with further navigation:

```
$data = [
    'orders' => [
        [
            'id' => 1001,
            'customer' => 'John',
            'items' => [
                ['product' => 'Laptop', 'price' => 999],
                ['product' => 'Mouse', 'price' => 25],
            ],
        ],
        [
            'id' => 1002,
            'customer' => 'Jane',
            'items' => [
                ['product' => 'Phone', 'price' => 699],
                ['product' => 'Headphones', 'price' => 149],
            ],
        ],
    ],
];

// Get the first item from order 1002.
$product = Selector::get($data, 'orders[id=1002:items][0].product');
// 'Phone'

// Get the price of the second item from order 1001.
$price = Selector::get($data, 'orders[id=1001:items][1].price');
// 25

// Find an item by product name within an order.
$data = [
    'orders' => [
        [
            'id' => 1001,
            'items' => [
                ['product' => 'Laptop', 'price' => 999],
```

```
                ['product' => 'Mouse', 'price' => 25],
            ],
        ],
    ],
];
$mousePrice = Selector::get($data, 'orders[id=1001:items][product=Mouse:price]');
// 25
```

Nested Arrays

For deeply nested structures:

```
$data = [
    'departments' => [
        [
            'name' => 'Engineering',
            'teams' => [
                [
                    'name' => 'Frontend',
                    'members' => [
                        ['id' => 101, 'name' => 'Alice'],
                        ['id' => 102, 'name' => 'Bob'],
                    ]
                ],
                [
                    'name' => 'Backend',
                    'members' => [
                        ['id' => 201, 'name' => 'Charlie'],
                        ['id' => 202, 'name' => 'Diana'],
                    ],
                ],
            ],
        ],
    ],
];

// Find Charlie's ID through the department and team.
$charlieId = Selector::get($data,
    'departments[name=Engineering:teams][name=Backend:members][name=Charlie:id]'
); // 201
```

Array Operations

Writing to Arrays

```
$data = [
```

```
    'users' => [
        ['id' => 1, 'name' => 'John'],
        ['id' => 2, 'name' => 'Jane'],
    ],
];

// Update a value.
Selector::set($data, 'users[id=1:name]', 'Johnny');
// $data['users'][0]['name'] is now 'Johnny'

// Add a new property.
Selector::set($data, 'users[id=2:email]', 'jane@example.com');
// $data['users'][1]['email'] is now 'jane@example.com'

// Add a new element.
Selector::set($data, 'users[3]', ['id' => 3, 'name' => 'Bob']);
// $data['users'][3] is now ['id' => 3, 'name' => 'Bob']

// If no matching element exists, a new one is created.
Selector::set($data, 'users[id=4:name]', 'Alice');
// Adds a new element to $data['users'] with id=4 and name='Alice'
```

Clearing Array Elements

```
$data = [
    'users' => [
        ['id' => 1, 'name' => 'John', 'email' => 'john@example.com'],
        ['id' => 2, 'name' => 'Jane', 'email' => 'jane@example.com'],
    ],
];

// Remove a property.
Selector::clear($data, 'users[id=1:email]');
// John's email is now removed.

// Remove an entire array element.
Selector::clear($data, 'users[id=2]');
// Jane's record is now completely removed.

// Clean up empty array elements.
$data = [
    'teams' => [
        ['id' => 1, 'members' => ['Alice', 'Bob']],
        ['id' => 2, 'members' => ['Charlie']],
    ],
];

// Remove the last member from team 2.
Selector::clear($data, 'teams[id=2:members][0]');
// Now teams[id=2:members] is an empty array.
```

```
// Derafu Selector automatically cleans up empty arrays:
// $data['teams'][1]['members'] is now an empty array.
// If we clear team 1's members too, it would clean up further.
Selector::clear($data, 'teams[id=1:members]');
// Now both teams have empty members arrays.
```

Working with Mixed Array Types

Selector handles both associative and indexed arrays seamlessly:

```
$data = [
    'products' => [
        'electronics' => [
            ['id' => 'e1', 'name' => 'Laptop', 'price' => 999],
            ['id' => 'e2', 'name' => 'Phone', 'price' => 699],
        ],
        'books' => [
            ['id' => 'b1', 'name' => 'Novel', 'price' => 15],
            ['id' => 'b2', 'name' => 'Textbook', 'price' => 50],
        ],
    ],
];

// Access by category and index.
$firstElectronic = Selector::get($data, 'products.electronics[0].name');
// 'Laptop'

// Access by category and ID.
$textbookPrice = Selector::get($data, 'products.books[id=b2:price]');
// 50

// Numeric indices still work with filtering.
$secondBookName = Selector::get($data, 'products.books[1].name');
// 'Textbook'
```

Handling Empty or Non-existent Arrays

Selector provides safe handling for empty or non-existent arrays:

```
$data = [
    'categories' => [
        'active' => [
            ['id' => 1, 'name' => 'Electronics'],
            ['id' => 2, 'name' => 'Books'],
        ],
        'inactive' => [],
    ],
];
```

```
// Access on empty array.
$inactiveCategory = Selector::get($data, 'categories.inactive[0]');
// null

// Filter on empty array.
$result = Selector::get($data, 'categories.inactive[id=1:name]');
// null

// Non-existent array path.
$result = Selector::get($data, 'categories.archived[0].name');
// null

// Providing defaults.
$result = Selector::get($data, 'categories.archived[0].name', 'Not found');
// 'Not found'
```

Modifying Array Elements in Place

You can modify array elements directly:

```
$data = [
    'cart' => [
        ['id' => 'p1', 'name' => 'Laptop', 'quantity' => 1],
        ['id' => 'p2', 'name' => 'Mouse', 'quantity' => 2],
    ],
];

// Increase quantity.
$currentQuantity = Selector::get($data, 'cart[id=p1:quantity]');
Selector::set($data, 'cart[id=p1:quantity]', $currentQuantity + 1);
// Now p1's quantity is 2

// Add a new property to an array element.
Selector::set($data, 'cart[id=p2:price]', 25);
// Mouse now has a price property

// Modify a nested property.
$data = [
    'orders' => [
        [
            'id' => 1,
            'items' => [
                ['product' => 'Laptop', 'price' => 999],
            ],
        ],
    ],
];

// Apply discount.
```

```
Selector::set($data, 'orders[id=1:items][product=Laptop:price]', 899);
// Price is now 899
```

By leveraging these array handling capabilities, you can work with complex nested data structures in a clean, readable way without worrying about undefined indices or complex nesting logic.

Last updated on 24/08/2026

Advanced Syntax

Advanced Selector Syntax

Anonymous · 24/08/2026

Advanced Selector Syntax

This guide covers the advanced selector syntax features that make Derafu Selector powerful for complex data manipulation.

String Literals and Concatenation

You can include literal strings in your selectors and concatenate them with values from your data:

```
$data = ['user' => ['name' => 'John']];

// Simple string concatenation.
$greeting = Selector::get($data, '"Hello, "(user.name)"!"');
// "Hello, John!"

// Multiple concatenations.
$message = Selector::get($data, '"User "(user.name)" logged in at "(timestamp)');
// "User John logged in at 2023-04-15 10:30:45"
```

OR Operator for Fallbacks

The `||` operator provides fallback values when a selector path doesn't exist or returns null:

```
$data = [
    'user' => [
        'name' => 'John',
        'email' => 'john@example.com',
        // phone is not set
    ],
];

// Use email if phone doesn't exist.
$contact = Selector::get($data, 'user.phone||user.email');
// "john@example.com"

// Chains of fallbacks.
$identifier = Selector::get($data, 'user.id||user.username||user.email');
```

```
// "john@example.com"

// Fallback to a literal string.
$phone = Selector::get($data, 'user.phone||"Not provided"');
// "Not provided"

// Combine with string concatenation.
$phoneDisplay = Selector::get($data, '"Phone: "(user.phone||"N/A")');
// "Phone: N/A"
```

Conditional (Ternary) Selectors

Conditionals let you choose between different selector paths based on conditions:

```
$data = [
    'user' => [
        'type' => 'admin',
        'admin_role' => 'super_admin',
        'user_role' => 'regular',
    ],
];

// Simple condition using equality.
$role = Selector::get($data,
    '((user.type) = "admin" ? (user.admin_role) : (user.user_role))'
); // "super_admin"

// Using not equal.
$accessLevel = Selector::get($data,
    '((user.type) != "guest" ? ("authenticated") : ("anonymous"))'
); // "authenticated"

// Numeric comparisons.
$data = ['score' => 85];
$grade = Selector::get($data,
    '((score) >= "90" ? ("A") : ((score) >= "80" ? ("B") : ("C")))'
); // "B"
```

Dependent Selectors

Find and access specific elements in arrays based on key/value matching:

```
$data = [
    'users' => [
        ['id' => 101, 'name' => 'John', 'email' => 'john@example.com'],
        ['id' => 102, 'name' => 'Jane', 'email' => 'jane@example.com'],
```

```
        ['id' => 103, 'name' => 'Bob', 'email' => 'bob@example.com'],
    ],
];

// Get the name of user with id 102.
$name = Selector::get($data, 'users[id=102:name]');
// "Jane"

// Get the email of user with id 101.
$email = Selector::get($data, 'users[id=101:email]');
// "john@example.com"

// Nested properties.
$data = [
    'orders' => [
        [
            'id' => 1001,
            'customer' => ['id' => 101, 'name' => 'John'],
            'items' => [['product' => 'Laptop', 'price' => 999]],
        ],
        [
            'id' => 1002,
            'customer' => ['id' => 102, 'name' => 'Jane'],
            'items' => [['product' => 'Phone', 'price' => 699]],
        ],
    ],
];

// Get Jane's first order item.
$janeProduct = Selector::get($data, 'orders[id=1002:items][0].product');
// "Phone"

// Access by string values with special characters.
$data = [
    'categories' => [
        ['id' => 'cat-1', 'name' => 'Electronics'],
        ['id' => 'cat-2', 'name' => 'Books']
    ]
];

$catName = Selector::get($data, 'categories[id=cat-1:name]');
// "Electronics"
```

Comparison Operators

Conditional selectors support several comparison operators:

```
$data = [
```

```
    'product' => [
        'price' => 50,
        'stock' => 10,
        'name' => 'Gadget',
        'tags' => ['electronics', 'new'],
    ],
];

// Equality.
$isGadget = Selector::get($data,
    '((product.name) = "Gadget" ? ("Yes") : ("No"))'
); // "Yes"

// Not equal.
$isExpensive = Selector::get($data,
    '((product.price) != "100" ? ("Affordable") : ("Expensive"))'
); // "Affordable"

// Greater than.
$priceTier = Selector::get($data,
    '((product.price) > "75" ? ("Premium") : ("Standard"))'
); // "Standard"

// Less than or equal.
$stockStatus = Selector::get($data,
    '((product.stock) <= "5" ? ("Low Stock") : ("In Stock"))'
); // "In Stock"

// Contains (for arrays and strings).
$isNew = Selector::get($data,
    '((product.tags) contains "new" ? ("New Arrival") : ("Regular Item"))'
); // "New Arrival"

// Length check.
$nameLength = Selector::get($data,
    '((product.name) length "6" ? ("Six Letters") : ("Other Length"))'
); // "Six Letters"

// Null check.
$data['product']['description'] = null;
$hasDescription = Selector::get($data,
    '((product.description) is "null" ? ("No Description") : ("Has Description"))'
); // "No Description"
```

Special Functions

The selector syntax includes several special functions:

```
$data = [
    'product' => [
        'name' => 'Gadget',
        'price' => 49.99,
        'tags' => ['electronics', 'gadget', 'new'],
    ],
];

// contains - check if an array or string contains a value.
$hasTag = Selector::get($data,
    '((product.tags) contains "gadget" ? ("Tagged as gadget") : ("Not tagged"))'
); // "Tagged as gadget"

// length - check the length of a string or array.
$tagCount = Selector::get($data,
    '((product.tags) length "3" ? ("Has 3 tags") : ("Has another tag count"))'
); // "Has 3 tags"

$nameLength = Selector::get($data,
    '((product.name) length "6" ? ("Name has 6 chars") : ("Name has different
length"))'
); // "Name has 6 chars"

// is - check the type of a value (currently supports null checks).
$hasDescription = Selector::get($data,
    '((product.description) is "null" ? ("No description") : ("Has description"))'
); // "No description"
```

Parentheses and Escaping

The selector syntax uses parentheses and quotes extensively:

```
// (selector) extracts a value from the data
// "string" is a literal string

$data = ['message' => 'Hello World'];

// Simple extraction.
$msg = Selector::get($data, '(message)'); // "Hello World"

// Literal string.
$literal = Selector::get($data, '"Static text"'); // "Static text"

// Concatenating extractions and literals.
$full = Selector::get($data, '(message)" - welcome!"'); // "Hello World - welcome!"

// Escaping quotes and parentheses.
$escaped = Selector::get($data, '"This is a \\"quoted\\" string"'); // 'This is a
```

```
"quoted" string'
$parentheses = Selector::get($data, '"Formula: (x + y)"'); // "Formula: (x + y)"

// Nested parentheses for complex expressions.
$nested = Selector::get($data, '((message) = "Hello World" ? ("Greeting") : ("Other
message"))');
// "Greeting"
```

For even more advanced capabilities, explore the integration with JSONPath and JMESPath in their respective guides.

Last updated on 24/08/2026

JSONPath Integration

JSONPath Integration

Anonymous · 24/08/2026

JSONPath Integration

This guide explains how to use JSONPath with Derafu Selector for powerful data structure querying.

Introduction to JSONPath

JSONPath is a query language for JSON, similar to XPath for XML. It provides a powerful way to extract data from complex JSON structures. Derafu Selector integrates JSONPath to give you even more flexibility in navigating your data.

All JSONPath expressions in Derafu Selector start with `$.` to distinguish them from regular selectors.

Basic JSONPath Queries

Here are some basic JSONPath examples:

```
use Derafu\Selector\Selector;

$data = [
    'store' => [
        'books' => [
            ['title' => 'Book 1', 'price' => 10],
            ['title' => 'Book 2', 'price' => 20],
            ['title' => 'Book 3', 'price' => 30],
        ],
        'bicycles' => [
            ['model' => 'Model A', 'price' => 100],
            ['model' => 'Model B', 'price' => 200],
        ],
    ],
    'user' => [
        'name' => 'John',
        'email' => 'john@example.com',
    ],
];

// Access a simple property.
$userName = Selector::get($data, '$.user.name');
```

```
// "John"

// Access an array element by index.
$firstBook = Selector::get($data, '$.store.books[0].title');
// "Book 1"

// Get all book titles.
$allTitles = Selector::get($data, '$.store.books[*].title');
// ["Book 1", "Book 2", "Book 3"]

// Get the entire books array.
$allBooks = Selector::get($data, '$.store.books');
// The complete books array
```

Advanced JSONPath Features

JSONPath offers powerful filtering capabilities:

```
// Filter books by price (more than 15).
$expensiveBooks = Selector::get($data, '$.store.books[?(@.price > 15)].title');
// ["Book 2", "Book 3"]

// Filter books by title (exact match).
$specificBook = Selector::get($data, '$.store.books[?(@.title == "Book 2")]');
// Returns the complete book object with title "Book 2"

// Multiple conditions with AND.
$filtered = Selector::get($data,
    '$.store.books[?(@.price > 15 && @.price < 25)].title'
);
// ["Book 2"]

// Regular expression matching (if supported by the JSONPath implementation).
$booksWithPattern = Selector::get($data, '$.store.books[?(@.title =~ /Book
[1-2]/)].title');
// ["Book 1", "Book 2"]

// Array slicing.
$firstTwoBooks = Selector::get($data, '$.store.books[0:2].title');
// ["Book 1", "Book 2"]
```

Common JSONPath Syntax Elements

Symbol	Description
`$`	The root object/element
`@`	The current object/element

Symbol	Description
`.`	Child operator
`..`	Recursive descent (find all matches at any level)
`*`	Wildcard (all objects/elements)
`[n]`	Array index (0-based)
`[n:m]`	Array slice from n to m
`[?()]`	Filter expression
`==` `!=`	Equality operators
`>` `<` `>=` `<=`	Comparison operators
`&&` `||`	Logical AND and OR

Combining JSONPath with Derafu Selectors

You can mix JSONPath with regular Derafu selectors for maximum flexibility:

```
// Use JSONPath to get a value and then concatenate with text.
$message = Selector::get($data, '"User: "($.user.name)');
// "User: John"

// Use JSONPath as part of an OR selector.
$contact = Selector::get($data, '$.user.phone||$.user.email');
// Falls back to "john@example.com" if phone doesn't exist

// Use JSONPath result in a conditional expression.
$userType = Selector::get($data,
    '(($.user.email) contains "example.com" ? ("Standard") : ("Premium"))'
);
// "Standard"

// Format array results.
$bookList = Selector::get($data,
    '"Available books: "($.store.books[*].title)'
);
// "Available books: [\"Book 1\", \"Book 2\", \"Book 3\"]"

// Mix JSONPath with regular selectors.
$data = [
    'config' => ['theme' => 'dark'],
    'preferences' => [
        'admin' => ['theme' => 'light'],
        'user' => ['theme' => 'system']
    ]
];
```

```
$theme = Selector::get($data,
    '(($.user.role) = "admin" ? (preferences.admin.theme) : (config.theme))'
);
// Falls back to "dark" if user.role doesn't exist
```

Performance Considerations

While JSONPath provides powerful querying capabilities, it may have different performance characteristics compared to native Derafu selectors:

1. **Initialization Cost**: JSONPath queries require initializing the JSONPath processor, which adds some overhead.

2. **Complex Queries**: For very complex queries with multiple conditions, JSONPath may be more efficient than chaining multiple native selectors.

3. **Large Data Sets**: For very large data structures, JSONPath's optimized filtering can be more efficient than iterating through the data manually.

4. **Writing Operations**: Note that JSONPath in Derafu Selector supports read operations only. For writing operations, you need to use native selectors.

Best Practices:

- Use native Derafu selectors for simple path access (`user.profile.name`).
- Use JSONPath for complex filtering and array operations.
- Consider caching results of expensive JSONPath queries if they're used repeatedly.
- Avoid using `..` (recursive descent) on very large data structures if performance is critical.

```
// Less efficient for simple access.
$name = Selector::get($data, '$.user.name');

// More efficient native syntax for simple access.
$name = Selector::get($data, 'user.name');

// JSONPath is more efficient for complex filtering.
$expensiveItems = Selector::get($data,
    '$.items[?(@.price > 100 && @.category == "electronics")].name'
);
```

By understanding when to use JSONPath versus native selectors, you can optimize your code for both readability and performance.

Last updated on 24/08/2026

JMESPath Integration

JMESPath Integration

Anonymous · 24/08/2026

JMESPath Integration

This guide explains how to use JMESPath with Derafu Selector for powerful data querying.

Introduction to JMESPath

JMESPath is a query language for JSON that allows you to extract and transform elements from complex data structures. It provides a more structured and feature-rich alternative to JSONPath with a clear specification.

All JMESPath expressions in Derafu Selector start with `jmespath:` to distinguish them from regular selectors.

Basic JMESPath Queries

Here are some basic JMESPath examples:

```
use Derafu\Selector\Selector;

$data = [
    'people' => [
        [
            'name' => 'John',
            'age' => 30,
            'phones' => ['home' => '555-1234', 'mobile' => '555-5678'],
        ],
        [
            'name' => 'Jane',
            'age' => 25,
            'phones' => ['home' => '555-4321', 'mobile' => '555-8765'],
        ],
    ],
    'locations' => [
        'Seattle' => ['state' => 'WA', 'population' => 724305],
        'New York' => ['state' => 'NY', 'population' => 8804190],
    ],
];
```

```
// Access a simple path.
$firstPerson = Selector::get($data, 'jmespath:people[0]');
// Complete first person object.

// Access a specific property.
$firstName = Selector::get($data, 'jmespath:people[0].name');
// "John"

// Access all names.
$allNames = Selector::get($data, 'jmespath:people[*].name');
// ["John", "Jane"]

// Access a nested property.
$firstMobile = Selector::get($data, 'jmespath:people[0].phones.mobile');
// "555-5678"

// Use bracket notation for keys with special characters.
$nyState = Selector::get($data, 'jmespath:locations."New York".state');
// "NY"
```

Advanced JMESPath Features

JMESPath offers powerful filtering, projection, and transformation capabilities:

Filtering

```
// Filter people over age 25.
$over25 = Selector::get($data, 'jmespath:people[?age > `25`]');
// Returns the complete person object for John.

// Filter with multiple conditions.
$filtered = Selector::get($data,
    'jmespath:people[?age > `25` && contains(name, `Jo`)]'
);
// Returns John's complete object.

// Get just the names of filtered results.
$names = Selector::get($data, 'jmespath:people[?age > `25`].name');
// ["John"]

// Filter on nested properties.
$withMobile = Selector::get($data,
    'jmespath:people[?phones.mobile != null].name'
);
// ["John", "Jane"]
```

Multi-select and Projections

```
// Select specific fields (projection).
$simplified = Selector::get($data, 'jmespath:people[*].{n: name, a: age}');
// [{"n":"John","a":30}, {"n":"Jane","a":25}]

// Multi-select on a single object.
$details = Selector::get($data, 'jmespath:people[0].{name: name, mobile:
phones.mobile}');
// {"name":"John","mobile":"555-5678"}

// Flatten nested structures.
$phones = Selector::get($data, 'jmespath:people[*].phones.*');
// ["555-1234", "555-5678", "555-4321", "555-8765"]
```

Functions

JMESPath supports a variety of built-in functions:

```
// String functions.
$upperNames = Selector::get($data, 'jmespath:people[*].name | [*].to_upper(@)');
// ["JOHN", "JANE"]

// Length/count.
$peopleCount = Selector::get($data, 'jmespath:length(people)');
// 2

// Min/max of values.
$data = ['values' => [5, 3, 8, 1, 7]];
$maxValue = Selector::get($data, 'jmespath:max(values)');
// 8

// Sort function.
$sortedNames = Selector::get($data, 'jmespath:sort(people[*].name)');
// ["Jane", "John"]

// Map function for transformations.
$ages = Selector::get($data, 'jmespath:people[*].age | map(&to_string(@), @)');
// ["30", "25"]
```

Slicing and Indexing

```
// Array slicing.
$firstPerson = Selector::get($data, 'jmespath:people[:1]');
// [Complete first person object]

// Negative indices.
$lastPerson = Selector::get($data, 'jmespath:people[-1]');
```

```
// Complete last person object (Jane).

// Step values.
$everyOtherPerson = Selector::get($data, 'jmespath:people[::2]');
// Every other person in the array.
```

Combining JMESPath with Derafu Selectors

You can mix JMESPath with regular Derafu selectors:

```
// Use JMESPath to get a value and then concatenate with text.
$greeting = Selector::get($data, '"Hello, "(jmespath:people[0].name)"!"');
// "Hello, John!"

// Use JMESPath as part of an OR selector.
$location = Selector::get($data,
    'jmespath:current_location||jmespath:locations."Seattle".state'
);
// Falls back to "WA" if current_location doesn't exist.

// Use JMESPath within a conditional.
$message = Selector::get($data,
    '((jmespath:people | length(@)) > "1" ? ("Multiple people") : ("One person"))'
);
// "Multiple people"

// Format array results.
$nameList = Selector::get($data,
    '"People: "(jmespath:people[*].name)'
);
// "People: [\"John\", \"Jane\"]"
```

Choosing Between JMESPath and JSONPath

Derafu Selector supports both JMESPath and JSONPath. Here are some considerations for choosing between them:

JMESPath Advantages

- **Formal Specification**: JMESPath has a formal, well-defined specification.
- **Functions**: Built-in functions for transformation and manipulation.
- **Multi-select Projection**: Create new structures with the data you extract.
- **Expressions**: More powerful expression language.
- **Pipe Operator**: Chain operations together.

JSONPath Advantages

- **Simplicity**: More straightforward syntax for basic queries.
- **Familiarity**: If you're already using JSONPath elsewhere.
- **Recursive Descent**: The `..` operator for finding elements at any level.

Examples of the Same Query in Both

```
$data = [
    'products' => [
        ['name' => 'Laptop', 'price' => 999, 'category' => 'electronics'],
        ['name' => 'Phone', 'price' => 699, 'category' => 'electronics'],
        ['name' => 'Book', 'price' => 15, 'category' => 'media'],
    ],
];

// Get electronics products with JSONPath.
$electronics = Selector::get($data,
    '$.products[?(@.category == "electronics")].name'
);
// ["Laptop", "Phone"]

// Same query with JMESPath.
$electronics = Selector::get($data,
    'jmespath:products[?category == `electronics`].name'
);
// ["Laptop", "Phone"]

// Filter by price and sort with JSONPath (sorting may not be available in all
implementations).
// May require multiple selector calls.

// With JMESPath, this is built-in.
$sortedProducts = Selector::get($data,
    'jmespath:products[?price > `500`].name | sort(@)'
);
// ["Laptop", "Phone"] (sorted)
```

By understanding the strengths of each query language, you can choose the right tool for your specific data navigation needs.

Last updated on 24/08/2026