Custom Scalars

- What are Custom Scalars?
- When to Use Custom Scalars vs Strings
  - Use Custom Scalars When:
  - Use Strings When:
  - Example: DateTime Scalar with Formatting
  - Example: When String Makes Sense
- Available Scalars
- Creating Custom Scalars
  - Registration with Inline Functions
  - Best Practices
  - Error Handling
  - Testing Your Scalar
- Integration with WordPress
- Future Considerations
  - Field Definition Directives
  - Decision Framework
- Related Documentation

WPGraphQL provides custom scalar types that enhance validation, type safety, and developer experience beyond the standard GraphQL scalars (String, Int, Float, Boolean, ID).

What are Custom Scalars?

Custom scalars are specialized data types that provide:

- **Enhanced Validation** - Built-in format checking using WordPress standards
- **Type Safety** - Semantic meaning in your GraphQL schema
- **Better Tooling** - IDEs and tools understand the data type
- **Consistent Behavior** - Standardized parsing and serialization across your API

When to Use Custom Scalars vs Strings

Understanding when to create a custom scalar versus using a String type with field-specific logic is crucial for good API design.

Use Custom Scalars When:

- **Core Data Type is Universal** - The underlying data structure and validation rules are consistent
- **Type Safety Matters** - Tools should understand the semantic meaning
- **Cross-Field Consistency** - Multiple fields share the same validation and parsing logic
- **Client-Side Benefits** - Frontend tools can provide better UX (validation, autocomplete, etc.)
- **Presentation Can Vary** - Different formatting options are just different views of the same data type

Use Strings When:

- **Field-Specific Validation** - Different fields have completely different validation rules
- **Context-Dependent Logic** - The data structure itself changes based on context
- **No Semantic Meaning** - The data is truly just arbitrary text
- **Legacy Compatibility** - Existing APIs that can't be easily changed

Example: DateTime Scalar with Formatting

A `DateTime` scalar would be perfect because the core data type is universal, but presentation can vary:

```
# □ Good: Universal DateTime validation, flexible presentation
type Post {
  date: DateTime                          # Default ISO format
  dateCreated(format: "Y-m-d"): DateTime   # Custom format
  publishedAt(format: "F j, Y"): DateTime  # Human readable
}
```

```
type User {
  registered: DateTime                    # Same validation everywhere
  lastLogin(format: "c"): DateTime        # ISO 8601 format
}
```

The `DateTime` scalar would:

- Always validate the same way (valid date/time)
- Always parse the same way (from ISO strings, timestamps, etc.)
- Support consistent formatting arguments across all DateTime fields

Example: When String Makes Sense

```
# □ Good use of String - truly arbitrary text with no universal format:
type Post {
  title: String # Free-form text, no universal validation rules
  excerpt: String # Free-form text, context-dependent length
  customFieldValue: String # Could be anything - text, numbers, JSON, etc.
}

type User {
  bio: String # Free-form biographical text
  notes: String # Admin notes - completely arbitrary
}
```

In contrast, these SHOULD be custom scalars because they have universal validation:

```
# □ These deserve custom scalars:
type Post {
  publishedAt: DateTime # Always a valid date/time
}
```

Available Scalars

WPGraphQL core provides standard GraphQL scalars (String, Int, Float, Boolean, ID). Additional custom scalars may be available through experiments or extensions.

Creating Custom Scalars

When you need to create your own custom scalar, follow these patterns established by WPGraphQL.

Registration with Inline Functions

```php
<?php

use GraphQL\Error\Error;
use GraphQL\Language\AST\StringValueNode;
use GraphQL\Utils\Utils;

// Register the scalar type with inline callback functions
add_action('graphql_register_types', function() {
    register_graphql_scalar('MyCustomScalar', [
        'description' => 'A custom scalar type for specific data validation',

        // Serializes an internal value to include in a response
        'serialize' => function($value) {
            if (!is_string($value)) {
                throw new Error(
                    'Value must be a string: ' . Utils::printSafe($value)
                );
            }

            // Apply any output formatting/sanitization
            return sanitize_text_field($value);
        },

        // Parses a value from a client input (variables)
        'parseValue' => function($value) {
            if (!is_string($value)) {
                throw new Error(
                    'Value must be a string: ' . Utils::printSafe($value)
                );
            }
```

```
            // Validate the input - implement your validation logic here
            if (!is_valid_custom_format($value)) {
                throw new Error('Invalid format for MyCustomScalar');
            }

            return $value;
        },

        // Parses a literal value from a GraphQL query
        'parseLiteral' => function($valueNode, ?array $variables = null) {
            if (!$valueNode instanceof StringValueNode) {
                throw new Error(
                    'Can only parse string values: ' . Utils::printSafe($valueNode)
                );
            }

            // Validate the literal value
            if (!is_valid_custom_format($valueNode->value)) {
                throw new Error('Invalid format for MyCustomScalar');
            }

            return $valueNode->value;
        },
    ]);
});

// Helper function for validation
function is_valid_custom_format($value): bool {
    // Implement your validation logic here
    return true;
}
```

Best Practices

1. **Validation First** - Always validate input in `parseValue` and `parseLiteral`
2. **WordPress Integration** - Leverage WordPress functions when possible (`is_email()`, `sanitize_text_field()`, etc.)
3. **Consistent Errors** - Use clear, consistent error messages that help developers
4. **Type Safety** - Check input types before processing
5. **Documentation** - Provide clear examples and use cases in your scalar description

6. **Naming Convention** - Use PascalCase for scalar names to match GraphQL conventions

Error Handling

```
use GraphQL\Error\Error;
use GraphQL\Utils\Utils;

// Good error messages are specific and helpful
throw new Error(
    sprintf(
        'Expected valid email address, received: %s',
        Utils::printSafe($value)
    )
);
```

Testing Your Scalar

```
// Test your scalar with GraphQL queries
add_action('init', function() {
    // Register a test field that uses your scalar
    register_graphql_field('RootQuery', 'testMyScalar', [
        'type' => 'MyCustomScalar',
        'resolve' => function() {
            return 'test-value';
        }
    ]);
});

// Test with a GraphQL query
$query = '{ testMyScalar }';
$result = graphql(['query' => $query]);

// Test input validation in mutations
register_graphql_mutation('testScalarInput', [
    'inputFields' => [
        'value' => ['type' => 'MyCustomScalar']
    ],
```

```
    'outputFields' => [
        'result' => ['type' => 'String']
    ],
    'mutateAndGetPayload' => function($input) {
        // If we get here, the scalar validation passed
        return ['result' => 'Valid: ' . $input['value']];
    }
]);

// Test the mutation
$mutation = 'mutation($input: TestScalarInputInput!) {
    testScalarInput(input: $input) { result }
}';
$variables = ['input' => ['value' => 'test-value']];
$result = graphql(['query' => $mutation, 'variables' => $variables]);
```

Integration with WordPress

WPGraphQL's custom scalars integrate seamlessly with WordPress:

- **Validation** - Use WordPress validation functions (`is_email()`, `wp_http_validate_url()`, etc.)
- **Sanitization** - Apply WordPress sanitization (`sanitize_email()`, `sanitize_text_field()`, etc.)
- **Capabilities** - Respect WordPress user capabilities for sensitive data
- **Filters** - Allow customization through WordPress filter hooks

Future Considerations

As discussed in WPGraphQL PR #3390, some data types might benefit from field-specific arguments rather than custom scalars:

Field Definition Directives

For complex data types that need formatting options, Field Definition Directives could provide consistent arguments:

```
# Instead of field directives in queries:
{
  posts {
    nodes {
      content @formatHTML(format: RAW)
    }
  }
}

# Field Definition Directives would enable arguments:
{
  posts {
    nodes {
      content(format: RAW)
    }
  }
}
```

This approach could work well for:

- **HTML Content** - `format: RAW | RENDERED` arguments
- **Dates** - `formatDate: "Y-m-d"` arguments
- **Colors** - `formatColor: HEX | RGB | RGBA` options

Decision Framework

When considering a new data type, ask:

1. **Does it need field-specific arguments?** → Consider Field Definition Directives
2. **Is validation universal and simple?** → Custom Scalar
3. **Is it just a string with loose validation?** → Keep as String

Related Documentation

- Default Types and Fields
- Building Extensions
- Settings
- Users

□□□ #1

□ □□□ □□□ 2026-08-29 23:36:26 UTC

□ □□□ □□□ 2026-08-29 23:36:26 UTC