Okyline

Getting Started Guide

Learn how to describe and validate JSON data with examples.

Your first steps into example-driven schema design.

Version 1.0.0 | November 2025

COPYRIGHT & LICENSE

Document Information

Getting Started with Okyline - Your First Steps

Version 1.0.0 | November 2025

Based on Okyline Language Specification v1.0.0

TABLE OF CONTENTS

Welcome to Okyline 1

Why Choose Okyline? 1

Your First Okyline Schema 2

Constraint Symbols 4

Practical Examples 5

Combining Constraints 6

Nested Objects 7

Advanced Features 8

Collections: Arrays and Maps 8

Special Directives 10

Conditional Validation 12

Best Practices 14

Start Simple, Add Constraints Progressively 14

Use Descriptive Field Labels 15

Leverage Built-in Formats 15

Common Pitfalls to Avoid 15

Quick Reference Card 16

Special Directives 16

Conditional Directives 17

Common Patterns 18

Compute Functions 19

String Functions 19
Numeric Functions 20
Date Functions 20
Aggregation Functions 20

Schema Metadata 21

Complete Schema with Metadata 21
$additionalProperties Behavior 22

Real-World Example 23

Complete API Schema: Create Order 23

Next Steps 25

Try It Online 25

Welcome to Okyline

Okyline is a revolutionary approach to JSON schema design. Instead of wrestling with complex schema languages, you simply write a JSON example and add inline constraints. It's that simple.

💡 The Core Syntax

```
field name | constraints | label
```

This simple pattern is all you need: the field name, followed by optional validation constraints and label, then the example value. Everything inline, right where you need it.

Progressive approach: Your example JSON is already a valid Okyline schema that validates structure and data types. Add constraints progressively as your validation needs grow.

Why Choose Okyline?

📝 Example-Driven

Your schema is a real JSON document with actual example values. What you see is what you validate.

🎯 Type Inference

No need to declare types explicitly. Okyline infers them from your example values automatically.

⚡ Progressive

Start simple, add constraints as needed. You don't have to define everything upfront.

🔄 JSON Schema Compatible

Convert your Okyline schemas to standard JSON Schema whenever you need to.

Your First Okyline Schema

Let's jump right in with a complete example:

Example: User Profile

A simple user profile with validation rules and documentation labels

```
{
  "$oky": {
    "username|@ {3,20}|User login name": "alice",
    "email|@ ~$Email~|Primary email address": "alice.vannier@example.com",
    "age|@ (18..120)|Age in years": 25,
    "role|('USER','ADMIN')|User role": "USER",
    "verified|?|Email verification status": true
  }
}
```

Breaking It Down

Field Name	Constraints	Label	Meaning
`username`	`@ {3,20}`	User login name	Required string between 3 and 20 characters
`email`	`@ ~$Email~`	Primary email address	Required string matching email format
`age`	`@ (18..120)`	Age in years	Required number between 18 and 120
`role`	`('USER','ADMIN')`	User role	Optional enumeration with two possible values
`verified`	`?`	Email verification status	Optional field, value can be null if present

✨ **Pro Tip**

All Okyline schemas must start with a `$oky` root object. This identifies the document as an Okyline schema.

Constraint Symbols

Okyline uses a simple symbol-based syntax for validation rules. Master these 9 core symbols and you're ready to design:

@

Required

Field must be present

?

Nullable

Value can be null

{,}

String Length

Minimum and maximum characters

(...)

Value Constraint

Number ranges or enumerations

~ ~

Format

Regex pattern or built-in format

[,]

Array Size

Minimum and maximum elements

#

Key Field

Marks field as unique identifier

!

Uniqueness

No duplicate objects in array

→

Item Constraint

Validation rules for array elements

Practical Examples

Constraint	Example	Validation Rule
`@` Required	`"name|@": "Alice"`	Field must be present
`?` Nullable	`"middleName|?": "Marie""`	Optional field, value can be null if present
`{min,max}` Length	`"username|{3,20}": "Alice"`	String must be 3-20 characters
`(min..max)` Range	`"age|(18..120)": 25`	Number must be between 18 and 120
`(1,5,9,12)` Enum	`"status|(1,9)"`	Must be one of the listed values
`('A','B')` Enum	`"status|('ACTIVE','INACTIVE')"`	Must be one of the listed values
`~format~` Pattern	`"email|~$Email~"`	Must match the format or regex
`[min,max]` Array	`"tags|[1,10]": ["tech"]`	Array must have 1-10 elements
`#` Key	`"id|#": 12345`	Marks field as unique identifier
`!` Unique	`"emails|[*]!": ["alice@example.com"]`	All array elements must be unique
`->` Items	`"codes|[*]->{2,5}"`	Each array element: 2-5 characters

Combining Constraints

Constraints can be combined for powerful validation rules:

Example: Product Catalog

```
{
  "$oky": {
    "productId|@ #": 12345,
    "name|@ {1,100}": "Laptop",
    "price|@ (≥0)": 999.99,
    "stock|@ (10..1000)": 42,
    "category|@ ('ELECTRONICS','BOOKS','CLOTHING')": "ELECTRONICS",
    "tags|[1,10]→{2,20}!": ["tech", "computers"]
  }
}
```

- **productId:** required (@) and unique identifier (#)
- **name:** required (@) string, 1-100 characters
- **price:** required (@), minimum value 0
- **stock:** required (@), value between 10 and 1000 inclusive
- **category:** one of the listed values
- **tags:** 1-10 items, each 2-20 chars, all unique (!)

Nested Objects

Okyline naturally supports nested structures:

Example: User with Address

```
{
  "$oky": {
    "user|@": {
      "name|@ {2,100}": "Alice Smith",
      "email|@ ~$Email~": "alice@example.com",
      "address|@": {
        "street|@ {5,200}": "123 Main St",
        "city|@ {2,100}": "Springfield",
        "country|@ {2,2}": "US",
        "postalCode|~^[0-9]{5}$~": "12345"
      }
    }
  }
}
```

Advanced Features

Collections: Arrays and Maps

Okyline treats collections homogeneously, whether they are **Arrays** (lists with numeric indices) or **Maps** (objects with dynamic string keys). Both use similar constraint syntax with the `->` operator.

Arrays (Lists)

Arrays use `[size_constraint]` followed by element constraints:

Example: Array with Element Constraints

```
"tags|@ [1,10] → {2,20}!": ["eco", "bio", "local"]
```

- `[1,10]` = array size between 1 and 10
- `→ {2,20}` = each element is a string of 2-20 characters
- `!` = all elements must be unique

Maps (Dynamic Keys)

Maps use `[key_pattern:size_constraint]` followed by value constraints:

Example: Map with Any Keys

```
"metadata|[*:20] → {1,100}": {
  "author": "Alice",
  "version": "1.2.3",
  "env": "production"
}
```

- `[*:20]` = any string keys, maximum 20 entries
- `→ {1,100}` = each value is a string of 1-100 characters

Example: Map with Pattern-Constrained Keys

```
"translations|@ [~^[a-z]{2}$~:10] → {1,100}": {
  "en": "Hello",
  "fr": "Bonjour",
  "es": "Hola"
}
```

- `[~^[a-z]{2}$~:10]` = keys must match regex (2 lowercase letters), max 10 entries
- `→ {1,100}` = each value is a string of 1-100 characters

Example: Map with Object Values

```
"products|[~^SKU-\\d{5}$~:*]": {
  "SKU-12345": {
    "name|@": "Product A",
    "price|@ (0..1000)": 29.99
  }
}
```

- `[~^SKU-\\d{5}$~:*]` = keys match "SKU-" + 5 digits, unlimited entries
- Each value is an object with name and price

✨ **Unified Collection Syntax**

Arrays: `[size]` controls the number of elements
Maps: `[key_pattern:size]` controls key format and number of entries
Both: Use `→` to apply constraints to elements/values

Special Directives

Okyline provides three powerful directives that start with `$` to add advanced capabilities to your schemas:

$nomenclature - Reusable Value Registries

Define centralized lists of allowed values (enumerations) that can be referenced throughout your schema:

Example: Centralized Enumerations

```
{
  "$nomenclature": {
    "STATUS": "DRAFT,VALIDATED,REJECTED,ACTIVE,INACTIVE,ARCHIVED",
    "COUNTRIES": "FRA,DEU,ESP,USA,GBR,ITA",
    "CURRENCIES": "USD,EUR,GBP,JPY,CHF"
  },
  "$oky": {
    "orderId|@ #": 12345,
    "status|@ ($STATUS)": "ACTIVE",
    "shippingCountry|@ ($COUNTRIES)": "FRA",
    "billingCountry|($COUNTRIES)": "DEU",
    "currency|@ ($CURRENCIES)": "EUR"
  }
}
```

- Define value lists once in `$nomenclature`
- Reference them with `|($NAME)` in field constraints
- Values are comma-separated (no quotes needed)
- Improves maintainability and ensures consistency

✨ **Combining Directives**

You can use `$format`, `$nomenclature`, and `$compute` together in the same schema for maximum power and maintainability!

$format - Custom String Formats

Define reusable format validators that can be referenced throughout your schema:

Example: Custom Formats

```
{
  "$oky": {
    "$format": {
      "ProductCode": "^[A-Z]{2}-[0-9]{4}$",
      "SKU": "^SKU-[0-9]{8}$"
    },
    "productCode|@ ~$ProductCode~": "AB-1234",
    "sku|@ ~$SKU~": "SKU-00012345",
    "relatedProducts|[*]->~$ProductCode~": ["CD-5678", "EF-9012"]
  }
}
```

- Define formats once in `$format` block
- Reference them with `~$FormatName~` anywhere in the schema
- Formats are regex patterns for string validation

$compute - Calculated Values

Define expressions that compute values from other fields. Use these computed values in validation constraints:

Example: Invoice with Calculations

```
{
  "$oky": {
    "name": "Lea",
    "total|@ (%CheckTotal)": 120,
    "items|@ [1,100]": [
      {
        "quantity|@ (1..100)": 2,
        "unitPrice|@ (>0)": 50,
        "tax|@ (0.05,0.1,0.5,0.2)": 0.2,
        "amount|@ (%CheckLineAmount)": 120
      }
    ]
  },
  "$compute": {
    "LineAmount": "quantity * round((unitPrice * (1 + tax)),2)",
    "CheckLineAmount": "amount == %LineAmount",
    "CheckTotal": "total == sum(items, amount)"
  }
}
```

- Fields can validate against computed values: `|@ (%CheckTotal)`
- Compute expressions can reference each other using `%ComputeName`
- Supports string, arithmetic, aggregations (sum, average, min, max), and more

🎯 **How Compute Expressions Work**

Evaluation Context: When a compute expression is evaluated, it has access to all fields in the **parent object** where the validation is triggered.
For example, in the invoice above, the `CheckLineAmount` expression can access `amount`, `quantity`, `unitPrice` and `tax` because they are in the same object (an item).
When one expression uses another expression, it passes its context to it.
In this example, %LineAmount, referenced by CheckLineAmount, therefore also has access to `quantity`, `unitPrice` and `tax`

Two Usage Modes:

- **Standalone compute:** Just calculate a value `"LineAmount": "quantity * round((unitPrice * (1 + tax)),2)"` → computes the total line with tax
- **Validation constraint:** Check that a condition is true `"amount|@ (%CheckTotal)"` → validates that CheckTotal returns true

How Validation Works: When you write `"amount|@ (%CheckAmount)"`, Okyline evaluates the `CheckAmount` expression and checks that it returns `true`. The expression must be a **boolean condition**. For example:

- `CheckTotal` should be: `"total == sum(items, amount)"`

Key Point: The compute expression must include the comparison (`==`, `>`, `<`, etc.) to return true or false.

Conditional Validation

Okyline provides powerful conditional validation mechanisms. Conditions can be based on **field values** or **field existence**. There are three main types of conditional constraints:

1. $appliedIf - Conditional Structure

Add fields dynamically based on conditions. Supports `$else` for alternative structures.

Directive	Condition Type	Meaning
`$appliedIf` / `$appliedIfNot`	Field value	Add structure when field value matches / doesn't match condition
`$appliedIfExist` / `$appliedIfNotExist`	Field existence	Add structure when field exists / doesn't exist

Example: Value-based $appliedIf

```
{
  "$oky": {
    "userType|@ ('INDIVIDUAL','COMPANY')": "COMPANY",
    "$appliedIf userType('INDIVIDUAL')": {
      "companyName|@ {2,100}": "Acme Corp",
      "taxId|@ {9,20}": "123456789",
      "$else": {
        "firstName|@ {2,50}": "John",
        "lastName|@ {2,50}": "Doe"
      }
    }
  }
}
```

If userType is COMPANY → company fields required, else → individual fields required

Example: Existence-based $appliedIfExist

```
{
  "$oky": {
    "tracking": "ABC123",
    "$appliedIfExist tracking": {
      "carrier|@": "DHL",
      "estimatedDelivery|@~$Date~": "2025-12-25"
    },
    "$appliedIfNotExist email": {
      "phone|@": "+33612345678",
      "phoneVerified|@": true
    }
  }
}
```

- If tracking exists → carrier and estimatedDelivery are required
- If email doesn't exist → phone and phoneVerified are required

Example: Switch-Case Mode (We love this one ! 🚀)

```
{
  "$oky": {
    "paymentMethod|@ ('CARD','BANK','CASH')": "CARD",
    "$appliedIf paymentMethod": {
      "('CARD')": {
        "cardNumber|@ {16}": "1234567812345678",
        "cvv|@ {3}": "123",
        "expiryDate|@ ~$Date~": "2026-12-31"
      },
      "('BANK')": {
        "iban|@ {15,34}": "FR7612345678901234567890123",
        "bic|@ {8,11}": "BNPAFRPP"
      },
      "('CASH')": {
        "receiptNumber|@": "RCP-2025-001"
      },
      "$else": {
        "note|@": "Unknown payment method"
      }
    }
  }
}
```

Switch-case syntax: Multiple `('value')` branches based on a single field. Each value gets its own structure. Much cleaner than nested if/else.

2. $requiredIf - Conditional Required Fields

Make specific fields required based on conditions.

Directive	Condition Type	Meaning
`$requiredIf` / `$requiredIfNot`	Field value	Fields required when value matches / doesn't match condition
`$requiredIfExist` / `$requiredIfNotExist`	Field existence	Fields required when another field exists / doesn't exist

Example: Value-based Required

```
{
  "$oky": {
    "accountType|@ ('PERSONAL','BUSINESS')": "BUSINESS",
    "email": "aurelien.barrot@example.com",
    "businessLicense": "BL-12345",
    "$requiredIf accountType('BUSINESS')": ["businessLicense"],
    "$requiredIfNot accountType('BUSINESS')": ["email"]
  }
}
```

- If accountType is BUSINESS → businessLicense required
- If accountType is NOT BUSINESS → email required

Example: Existence-based Required

```
{
  "$oky": {
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com",
    "phone": "+33612345678",
    "$requiredIfExist firstName": ["lastName"],
    "$requiredIfNotExist email": ["phone"]
  }
}
```

- If firstName exists → lastName is required
- If email doesn't exist → phone is required

3. $forbiddenIf - Conditional Forbidden Fields

Prevent specific fields from being present based on conditions.

Directive	Condition Type	Meaning
`$forbiddenIf` / `$forbiddenIfNot`	Field value	Fields forbidden when value matches / doesn't match condition

`$forbiddenIfExist` / `$forbiddenIfNotExist`	Field existence	Fields forbidden when another field exists / doesn't exist

Example: Value-based Forbidden

```
{
  "$oky": {
    "accountStatus|@ ('ACTIVE','SUSPENDED','CLOSED')": "CLOSED",
    "lastLogin|~$DateTime~": "2025-01-10T15:30:00Z",
    "closureReason|{10,500}": "User requested deletion",
    "$forbiddenIf accountStatus('CLOSED')": ["lastLogin"],
    "$forbiddenIfNot accountStatus('CLOSED')": ["closureReason"]
  }
}
```

- If status is CLOSED → lastLogin must NOT be present
- If status is NOT CLOSED → closureReason must NOT be present

Example: Existence-based Forbidden

```
{
  "$oky": {
    "archived": true,
    "sku": "SKU-12345",
    "internalCode": "INT-999",
    "$forbiddenIfExist archived": ["active"],
    "$forbiddenIfNotExist sku": ["internalCode"]
  }
}
```

- If archived exists → active must NOT be present
- If sku doesn't exist → internalCode must NOT be present

Combining Conditional Constraints

You can combine multiple conditional directives in the same schema:

Example: Complex Conditional Logic

```
{
  "$oky": {
    "employeeStatus|@ ('ACTIVE','ON_LEAVE','TERMINATED')": "TERMINATED",
    "workDays|(1..22)": 20,
    "leaveReason|{10,200}": "Parental leave",
    "terminationDate|~$Date~": "2025-12-31",
    "email": "[email protected]",
    "$requiredIf employeeStatus('ACTIVE')": ["workDays"],
    "$requiredIfExist leaveReason": ["employeeStatus"],
    "$forbiddenIfNot employeeStatus('TERMINATED')": ["terminationDate"],
    "$forbiddenIfNotExist email": ["phone"]
  }
}
```

Multiple conditional rules (value-based and existence-based) working together

✨ **Conditional Validation Summary**

Value-based conditions: Test field values with If/IfNot (e.g., `status('TERMINATED')`)
Existence-based conditions: Test field presence with IfExist/IfNotExist
Three constraint types: $appliedIf (structure), $requiredIf (mandatory), $forbiddenIf (prohibited)

Best Practices

Start Simple, Add Constraints Progressively

Don't try to write the perfect schema on the first try. Start with a basic example and add constraints as you discover validation needs:

✨ **Progressive Enhancement**

1. **Step 1:** Write a valid JSON example
2. **Step 2:** Wrap it in `{"$oky": { … }}`
3. **Step 3:** Add `@` to required fields
4. **Step 4:** Add specific constraints (length, range, format, nomenclature)
5. **Step 5:** Add conditional logic if needed
6. **Step 6:** Add computed constraints if needed

Use Descriptive Field Labels

Okyline supports inline documentation after the constraints:

```
{
  "$oky": {
    "email|@ ~$Email~|User's primary email address": "lea.bocase@example.com",
    "age|@ (18..120)|Age in years": 25
  }
}
```

Leverage Built-in Formats

Don't reinvent the wheel. Okyline provides some built-in format validators:

- `~$Date~` for ISO 8601 dates (YYYY-MM-DD)
- `~$DateTime~` for ISO 8601 timestamps
- `~$Time~` for ISO 8601 times
- `~$Email~` for email addresses
- `~$Uuid~`, `~$Uri~`, `~$IPv4~`, `~$IPv6~`, `$Hostname~`

Common Pitfalls to Avoid

⚠️ **Watch Out!**

- **Don't forget the $oky wrapper:** All schemas must start with `{"$oky": { … }}`
- **Array size vs element constraints:** `[1,10]` is size, `[*]→{2,5}` is element constraint
- **Enum values must match type:** If example is a string, enum values must be strings

Quick Reference Card

Special Directives

Directive	Purpose	Example
`$oky`	Root wrapper (required)	`{"$oky": { … }}`
`$nomenclature`	Reusable value lists (enums)	`"$nomenclature": {"STATUS": "ACTIVE,INACTIVE"}`
`$format`	Define reusable formats	`"$format": {"SKU": "^[A-Z]{3}-[0-9]{4}$"}`
`$compute`	Define calculated values	`"$compute": {"Total": "price * qty"}`

Conditional Directives

Directive	Purpose	Example
`$appliedIf` / `$appliedIfNot`	Conditional structure (value-based, supports $else). Switch-Case Mode also supported with $appliedIf	`"$appliedIf status('ACTIVE')": { … }`
`$appliedIfExist` / `$appliedIfNotExist`	Conditional structure (existence-based)	`"$appliedIfExist tracking": { … }`
`$requiredIf` / `$requiredIfNot`	Required fields (value-based)	`"$requiredIf type('BUSINESS')": ["taxId"]`
`$requiredIfExist` / `$requiredIfNotExist`	Required fields (existence-based)	`"$requiredIfExist firstName": ["lastName"]`

`$forbiddenIf` / `$forbiddenIfNot`	Forbidden fields (value-based)	`"$forbiddenIf status('CLOSED')": ["login"]`
`$forbiddenIfExist` / `$forbiddenIfNotExist`	Forbidden fields (existence-based)	`"$forbiddenIfExist archived": ["active"]`

Common Patterns

Pattern: Required with String Length

```
"name|@ {2,50}": "Alice Smith"
```

Pattern: String Enumeration

```
"status|('DRAFT','PUBLISHED','ARCHIVED')": "DRAFT"
```

Pattern: numeric Enumeration

```
"status|(1,2,5,10)":5
```

Pattern: numeric range

```
"amount|@ (20..50)": 7500.60
```

Pattern: Required Email

```
"email|@ ~$Email~": "sophie.riberro@example.com"
```

Pattern: Required Date

```
"createdAt|@ ~$Date~": "2025-01-15"
```

Pattern: Required Array of Unique IDs

```
"productIds|@ [1,*] -> (1..1000) ! ": [101, 102, 103]
```

Pattern: Required but Nullable

```
"middleName|@ ?{1,50}": "Marie"
```

Field must be present, but value can be null (example shows a valid value for type inference)

Pattern: Required Array of Unique Objects

```
"items|@ [1,*] → !": [
  {
    "sku|@ #": "SKU-001",
    "name|@": "Product A"
  },
  {
    "sku|@ #": "SKU-002",
    "name|@": "Product B"
  }
]
```

Objects are unique based on the **key field marked with #**

Pattern: Required Map with Dynamic Keys

```
"translations|@ [~^[a-z]{2}$~:10] → {1,100}": {
  "en": "Hello",
  "fr": "Bonjour",
  "es": "Hola"
}
```

Map with keys matching pattern (2 lowercase letters), max 10 entries, each value 1-100 characters

Pattern: Custom Format from $format

```
"code|~$Code~": "ABC-1234"
```

```
"$format": {
  "Code": "^[A-Z]{3}-[0-9]{4}$"
}
```

Pattern: Enum from $nomenclature

```
"country|($COUNTRY)": "FRA"
```

```
"$nomenclature": {
  "COUNTRY": "FRA,DEU,ESP,USA,GBR,ITA"
}
```

Pattern: Computed Validation

```
"unitPrice|@ (≥0)": 100.50,
"quantity|@ (1..500)": 3,
"amount|@ (%CheckAmount)": 301.50
```

```
"$compute": {
  "CheckAmount": "unitPrice * quantity"
}
```

The amount field must match the computed value (unitPrice × quantity)

Pattern: Simple Conditional

```
{
  "userType|@ ('INDIVIDUAL','COMPANY')": "COMPANY",
  "$appliedIf userType('COMPANY')": {
    "companyName|@ {2,100}": "Acme Corp"
  }
}
```

Compute Expression Language Functions

Okyline's compute expressions support a rich set of functions for data manipulation and validation. All functions are **null-safe** and **deterministic** (no side effects).

Null Safety

All functions handle null values gracefully without throwing exceptions. Arithmetic operations propagate null (except string concatenation which treats null as ""). Use the `??` operator for null coalescing: `price ?? 0` (price will be equal to 0 if null)

String Functions

- `isNullOrEmpty(s)`
- `isEmpty(s)`
- `substring(s,start,len)`
- `substringBefore(s,delim)`
- `substringAfter(s,delim)`
- `replace(s,target,repl)`
- `trim(s)`
- `length(s)`
- `startsWith(s,prefix)`
- `endsWith(s,suffix)`
- `contains(s,search)`
- `toUpperCase(s)`
- `toLowerCase(s)`
- `capitalize(s)`
- `decapitalize(s)`
- `padStart(s,len,ch)`
- `padEnd(s,len,ch)`
- `repeat(times,ch)`
- `indexOf(s,sub)`
- `indexOfLast(s,sub)`

Numeric Functions

- `abs(x)`
- `sqrt(x)`
- `floor(x,scale?)`
- `ceil(x,scale?)`
- `round(x,scale?,mode?)`
- `mod(a,b)`
- `pow(base,exp)`
- `log(x)`
- `log10(x)`
- `random(min?,max?)`
- `toInt(v)`
- `toNum(v)`

Date Functions

- `date(dateString,pattern?)`
- `formatDate(date,pattern?)`
- `today()`
- `daysBetween(start,end)`
- `plusDays(date,days)`
- `minusDays(date,days)`
- `plusMonths(date,months)`
- `minusMonths(date,months)`

`plusYears(date,years)`	`year(date)`
`minusYears(date,years)`	`month(date)`
`isWeekend(date)`	`day(date)`
`isLeapYear(date)`	

Aggregation Functions

`sum(collection,expr)`	`count(collection)`
`average(collection,expr)`	`countAll(collection)`
`min(collection,expr)`	`countIf(collection,expr)`
`max(collection,expr)`	

📖 **Full Documentation**

For detailed documentation on each function including parameters, examples, and behavior, refer to the **Okyline Language Specification v1.0.0 - Annex C**.

Schema Metadata

Okyline schemas can include optional metadata fields at the root level to document and version your schemas:

Metadata Field	Purpose	Example
`$okylineVersion`	Version of your schema	`"1.0.0"`
`$id`	Unique identifier for your Okyline scheme within your organization	`"E-ORDER-001"`
`$version`	Version of your schema	`"1.0.4"`
`$title`	Human-readable schema title	`"User Profile Schema"`
`$description`	Description of what the schema validates	`"Schema for user profiles"`
`$additionalProperties`	Allow unknown fields in validated data	`false` (default)

Complete Schema with Metadata

Example: Fully Documented Schema

```
{
  "$okylineVersion": "1.0.0",
  "$version": "1.0.4",
  "$id":"E-ORDER-001",
  "$title": "E-commerce Order Schema",
  "$description": "Schema for validating customer orders",
  "$additionalProperties": false,
  "$oky": {
    "orderId|@ ~$OrderId~": "ORD-12345678",
    "status|@ ($STATUS)": "PENDING",
    "customerEmail|@ ~$Email~": "[email protected]",
    "total|@ (0..1000)": 99.99
  },
  "$nomenclature": {
    "STATUS": "PENDING,CONFIRMED,SHIPPED,DELIVERED,CANCELLED"
  },
  "$format": {
    "OrderId": "^ORD-[0-9]{8}$"
  }
}
```

A production-ready schema with complete metadata and documentation

$additionalProperties Behavior

The `$additionalProperties` field controls whether extra fields not defined in the schema are allowed:

- **false (default):** Only fields defined in the schema are allowed. Unknown fields cause validation errors.
- **true:** Additional fields not defined in the schema are allowed and ignored during validation.

🔧 **Scope and Inheritance**

`$additionalProperties` can be defined at the **root level** (applies globally) or **within specific objects** (applies only to that object). Child objects inherit the root setting unless they override it locally.

Example: Local $additionalProperties Override

```
{
  "$additionalProperties": false,
  "$oky": {
    "user|@": {
      "name|@": "Alice",
      "email|@ ~$Email~": "[email protected]"
    },
    "origin": {
      "$additionalProperties": true,
      "source": "web",
      "timestamp": "2025-01-15T10:30:00Z"
    }
  }
}
```

- Root level: strict mode (no extra fields)
- origin object: allows additional properties

✨ **Best Practice**

Always include `$version`, `$title`, and `$id` in production schemas to improve maintainability and documentation. Use semantic versioning for `$version` (e.g., "1.2.3").

Real-World Example

Complete API Schema: Create Order

Let's put it all together with a realistic API request/response schema:

Example: E-commerce Order API

```
{
  "$version": "2.1.0",
  "$id": "E-ORDER-001",
  "$title": "Order Schema",
  "$description": "Schema for e-commerce orders",
  "$oky": {
    "order": {
      "orderId|@ #~$OrderId~|Unique order identifier": "ORD-12345678",
      "customerId|@ (>0)|Customer ID": 42,
      "orderDate|@ ~$DateTime~|Order timestamp": "2025-01-15T10:30:00Z",
      "status|@ ($ORDER_STATUS)|Order status": "PENDING",
      "items|@ [1,100] → !|Order items": [
        {
          "sku|@ #~$Sku~|Product SKU": "SKU-ABC12345",
          "name|@ {2,200}|Product name": "Wireless Mouse",
          "quantity|@ (1..1000)|Quantity": 2,
          "vat|(0.05,0.1,0.15,0.2)":0.2,
          "unitPrice|@ (>0)|Unit price": 50.0,
          "netAmount|(%CheckNetAmount)":100.0,
          "grossAmount|(%CheckGrossAmount)":120.00
        }
      ],
      "paymentMethod|@ ($PAYMENT_METHOD)|Payment method": "CARD",
      "subTotal|(%CheckSubtotal)|Total net amount":100.0,
      "total|@ (%CheckTotal)|Total gross amount": 120.00,
      "$requiredIf status('SHIPPED','DELIVERED')": ["trackingNumber"]
    }
  },
  "$nomenclature": {
    "ORDER_STATUS": "PENDING,CONFIRMED,SHIPPED,DELIVERED,CANCELLED",
    "PAYMENT_METHOD": "CARD,PAYPAL,BANK_TRANSFER"
  },
  "$format": {
    "OrderId": "^ORD-[0-9]{8}$",
    "Sku": "^SKU-[A-Z]{3}[0-9]{5}$"
  },
  "$compute": {
    "CheckNetAmount": "netAmount == round(unitPrice * quantity,2)",
    "CheckGrossAmount": "grossAmount == round(netAmount * (1 + vat),2)",
    "CheckSubtotal": "subTotal == sum(items,netAmount)",
    "CheckTotal": "total == sum(items,grossAmount)"
  }
}
```

Key Features Demonstrated

- **Metadata fields:** `$version`, `$id`, `$title`, `$description` for schema documentation
- **Custom formats:** `orderId|@ #~$OrderId~` and `sku|@ #~$Sku~` with regex patterns defined in `$format`
- **Nomenclature enums:** `status|@ ($ORDER_STATUS)` and `paymentMethod|@ ($PAYMENT_METHOD)` from `$nomenclature`
- **Key fields:** `#` marker on orderId and sku for unique identifiers
- **Array constraints:** `items|@ [1,100] -> !` requires 1-100 unique items
- **Computed validation:** `netAmount|(%CheckNetAmount)`, `grossAmount|(%CheckGrossAmount)`, `subTotal|(%CheckSubtotal)` validate calculations
- **Inline labels:** Each field has a human-readable description after the constraints
- **Conditional required:** `$requiredIf status('SHIPPED','DELIVERED')` makes trackingNumber required for certain statuses
- **Range and comparisons:** `customerId|@ (>0)`, `quantity|@ (1..1000)`, `total|@ (>0)`

Next Steps

Try It Online

The best way to learn Okyline is to experiment with it. Visit the free online editor:

> 🚀 Okyline Free Studio
>
> **https://community.studio.okyline.io**
>
> Features live validation, JSON Schema export, and interactive documentation.

Additional Resources

- **Full Specification:** https://public-docs.okyline.io/Okyline-Language-Specification-v1.0.0.pdf
- **Online Editor:** https://community.studio.okyline.io

💡 **Ready to Design?**

You now have everything you need to start designing Okyline schemas. Remember: start with an example, add constraints progressively, and validate as you go. Happy schema designing!