



GraphQL

Working Draft – October 2016

Introduction

This is a Draft RFC Specification for GraphQL, a query language created by Facebook in 2012 for describing the capabilities and requirements of data models for client-server applications. The development of this standard started in 2015. GraphQL is a new and evolving language and is not complete. Significant enhancement will continue in future editions of this specification.

Copyright notice

Copyright (c) 2015–2016, Facebook, Inc. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name Facebook nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Contents



- 1 Overview
- 2 Language
 - 2.1 Source Text
 - 2.1.1 Unicode
 - 2.1.2 White Space
 - 2.1.3 Line Terminators
 - 2.1.4 Comments
 - 2.1.5 Insignificant Commas
 - 2.1.6 Lexical Tokens
 - 2.1.7 Ignored Tokens
 - 2.1.8 Punctuators
 - 2.1.9 Names
 - 2.2 Query Document
 - 2.3 Operations
 - 2.4 Selection Sets
 - 2.5 Fields
 - 2.6 Arguments
 - 2.7 Field Alias
 - 2.8 Fragments
 - 2.8.1 Type Conditions
 - 2.8.2 Inline Fragments
 - 2.9 Input Values
 - 2.9.1 Int Value
 - 2.9.2 Float Value
 - 2.9.3 Boolean Value
 - 2.9.4 String Value
 - 2.9.5 Null Value
 - 2.9.6 Enum Value
 - 2.9.7 List Value
 - 2.9.8 Input Object Values
 - 2.10 Variables
 - 2.11 Input Types
 - 2.12 Directives
- 3 Type System
 - 3.1 Types
 - 3.1.1 Scalars
 - 3.1.1.1 Int
 - 3.1.1.2 Float
 - 3.1.1.3 String
 - 3.1.1.4 Boolean
 - 3.1.1.5 ID
 - 3.1.2 Objects



- 3.1.2.1 Object Field Arguments
 - 3.1.2.2 Object Field deprecation
 - 3.1.2.3 Object type validation
 - 3.1.3 Interfaces
 - 3.1.3.1 Interface type validation
 - 3.1.4 Unions
 - 3.1.4.1 Union type validation
 - 3.1.5 Enums
 - 3.1.6 Input Objects
 - 3.1.6.1 Input Object type validation
 - 3.1.7 Lists
 - 3.1.8 Non-Null
 - 3.2 Directives
 - 3.2.1 @skip
 - 3.2.2 @include
 - 3.3 Initial types
- ## 4 Introspection
- 4.1 General Principles
 - 4.1.1 Naming conventions
 - 4.1.2 Documentation
 - 4.1.3 Deprecation
 - 4.1.4 Type Name Introspection
 - 4.2 Schema Introspection
 - 4.2.1 The __Type Type
 - 4.2.2 Type Kinds
 - 4.2.2.1 Scalar
 - 4.2.2.2 Object
 - 4.2.2.3 Union
 - 4.2.2.4 Interface
 - 4.2.2.5 Enum
 - 4.2.2.6 Input Object
 - 4.2.2.7 List
 - 4.2.2.8 Non-Null
 - 4.2.2.9 Combining List and Non-Null
 - 4.2.3 The __Field Type
 - 4.2.4 The __InputValue Type
 - 4.2.5 The __Directive Type
- ## 5 Validation
- 5.1 Operations
 - 5.1.1 Named Operation Definitions
 - 5.1.1.1 Operation Name Uniqueness
 - 5.1.2 Anonymous Operation Definitions
 - 5.1.2.1 Lone Anonymous Operation
 - 5.2 Fields



- 5.2.1 Field Selections on Objects, Interfaces, and Unions Types
- 5.2.2 Field Selection Merging
- 5.2.3 Leaf Field Selections
- 5.3 Arguments
 - 5.3.1 Argument Names
 - 5.3.2 Argument Uniqueness
 - 5.3.3 Argument Values Type Correctness
 - 5.3.3.1 Compatible Values
 - 5.3.3.2 Required Non-Null Arguments
- 5.4 Fragments
 - 5.4.1 Fragment Declarations
 - 5.4.1.1 Fragment Name Uniqueness
 - 5.4.1.2 Fragment Spread Type Existence
 - 5.4.1.3 Fragments On Composite Types
 - 5.4.1.4 Fragments Must Be Used
 - 5.4.2 Fragment Spreads
 - 5.4.2.1 Fragment spread target defined
 - 5.4.2.2 Fragment spreads must not form cycles
 - 5.4.2.3 Fragment spread is possible
 - 5.4.2.3.1 Object Spreads In Object Scope
 - 5.4.2.3.2 Abstract Spreads in Object Scope
 - 5.4.2.3.3 Object Spreads In Abstract Scope
 - 5.4.2.3.4 Abstract Spreads in Abstract Scope
- 5.5 Values
 - 5.5.1 Input Object Field Uniqueness
- 5.6 Directives
 - 5.6.1 Directives Are Defined
 - 5.6.2 Directives Are In Valid Locations
 - 5.6.3 Directives Are Unique Per Location
- 5.7 Variables
 - 5.7.1 Variable Uniqueness
 - 5.7.2 Variable Default Values Are Correctly Typed
 - 5.7.3 Variables Are Input Types
 - 5.7.4 All Variable Uses Defined
 - 5.7.5 All Variables Used
 - 5.7.6 All Variable Usages are Allowed
- 6 Execution
 - 6.1 Executing Requests
 - 6.1.1 Validating Requests
 - 6.1.2 Coercing Variable Values
 - 6.2 Executing Operations
 - 6.3 Executing Selection Sets
 - 6.3.1 Normal and Serial Execution
 - 6.3.2 Field Collection



- 6.4 Executing Fields
 - 6.4.1 Coercing Field Arguments
 - 6.4.2 Value Resolution
 - 6.4.3 Value Completion
 - 6.4.4 Errors and Non-Nullability
- 7 Response
 - 7.1 Serialization Format
 - 7.1.1 JSON Serialization
 - 7.2 Response Format
 - 7.2.1 Data
 - 7.2.2 Errors
- A Appendix: Notation Conventions
 - A.1 Context-Free Grammar
 - A.2 Lexical and Syntactical Grammar
 - A.3 Grammar Notation
 - A.4 Grammar Semantics
 - A.5 Algorithms
- B Appendix: Grammar Summary
 - B.1 Ignored Tokens
 - B.2 Lexical Tokens
 - B.3 Query Document
- § Index

Overview

GraphQL is a query language designed to build client applications by providing an intuitive and flexible syntax and system for describing their data requirements and interactions.

For example, this GraphQL request will receive the name of the user with id 4 from the Facebook implementation of GraphQL.

```
{  
  user(id: 4) {  
    name  
  }  
}
```

Which produces the resulting data (in JSON):



```
{  
  "user": {  
    "name": "Mark Zuckerberg"  
  }  
}
```

GraphQL is not a programming language capable of arbitrary computation, but is instead a language used to query application servers that have capabilities defined in this specification. GraphQL does not mandate a particular programming language or storage system for application servers that implement it. Instead, application servers take their capabilities and map them to a uniform language, type system, and philosophy that GraphQL encodes. This provides a unified interface friendly to product development and a powerful platform for tool-building.

GraphQL has a number of design principles:

- **Hierarchical:** Most product development today involves the creation and manipulation of view hierarchies. To achieve congruence with the structure of these applications, a GraphQL query itself is structured hierarchically. The query is shaped just like the data it returns. It is a natural way for clients to describe data requirements.
- **Product-centric:** GraphQL is unapologetically driven by the requirements of views and the front-end engineers that write them. GraphQL starts with their way of thinking and requirements and build the language and runtime necessary to enable that.
- **Strong-typing:** Every GraphQL server defines an application-specific type system. Queries are executed within the context of that type system. Given a query, tools can ensure that the query is both syntactically correct and valid within the GraphQL type system before execution, i.e. at development time, and the server can make certain guarantees about the shape and nature of the response.
- **Client-specified queries:** Through its type system, a GraphQL server publishes the capabilities that its clients are allowed to consume. It is the client that is responsible for specifying exactly how it will consume those published capabilities. These queries are specified at field-level granularity. In the majority of client-server applications written without GraphQL, the server determines the data returned in its various scripted endpoints. A GraphQL query, on the other hand, returns exactly what a client asks for and no more.
- **Introspective:** GraphQL is introspective. A GraphQL server's type system must be queryable by the GraphQL language itself, as will be described in this specification. GraphQL introspection serves as a powerful platform for building common tools and client software libraries.

Because of these principles, GraphQL is a powerful and productive environment for building client applications. Product developers and designers building applications against working GraphQL servers -- supported with quality tools -- can quickly become productive without

reading extensive documentation and with little or no formal training. To enable that experience, there must be those that build those servers and tools.



The following formal specification serves as a reference for those builders. It describes the language and its grammar, the type system and the introspection system used to query it, and the execution and validation engines with the algorithms to power them. The goal of this specification is to provide a foundation and framework for an ecosystem of GraphQL tools, client libraries, and server implementations -- spanning both organizations and platforms -- that has yet to be built. We look forward to working with the community in order to do that.

Language

Clients use the GraphQL query language to make requests to a GraphQL service. We refer to these request sources as documents. A document may contain operations (queries and mutations are both operations) as well as fragments, a common unit of composition allowing for query reuse.

A GraphQL document is defined as a syntactic grammar where terminal symbols are tokens (indivisible lexical units). These tokens are defined in a lexical grammar which matches patterns of source characters (defined by a double-colon `::`).

Source Text

SourceCharacter ::

`/[\u0009\u000A\u000D\u0020-\uFFFF]/`

GraphQL documents are expressed as a sequence of [Unicode](#) characters. However, with few exceptions, most of GraphQL is expressed only in the original non-control ASCII range so as to be as widely compatible with as many existing tools, languages, and serialization formats as possible and avoid display issues in text editors and source control.

Unicode

UnicodeBOM ::

Byte Order Mark (U+FEFF)

Non-ASCII Unicode characters may freely appear within *StringValue* and *Comment* portions of GraphQL.

The “Byte Order Mark” is a special Unicode character which may appear at the beginning of a file containing Unicode which programs may use to determine the fact that the text stream is Unicode, what endianness the text stream is in, and which of several Unicode encodings to interpret.

White Space

WhiteSpace ::

Horizontal Tab (U+0009)

Space (U+0020)

White space is used to improve legibility of source text and act as separation between tokens, and any amount of white space may appear before or after any token. White space between tokens is not significant to the semantic meaning of a GraphQL query document, however white space characters may appear within a *String* or *Comment* token.

Note

GraphQL intentionally does not consider Unicode “Zs” category characters as white-space, avoiding misinterpretation by text editors and source control tools.

Line Terminators

LineTerminator ::

New Line (U+000A)

Carriage Return (U+000D) [lookahead ≠ New Line (U+000A)]

Carriage Return (U+000D) New Line (U+000A)

Like white space, line terminators are used to improve the legibility of source text, any amount may appear before or after any other token and have no significance to the semantic meaning of a GraphQL query document. Line terminators are not found within any other token.

Note

Any error reporting which provide the line number in the source of the offending syntax should use the preceding amount of *LineTerminator* to produce the line number.

Comments

Comment ::

*CommentChar*_{list, opt}

CommentChar ::

SourceCharacter but not *LineTerminator*



GraphQL source documents may contain single-line comments, starting with the # marker.

A comment can contain any Unicode code point except *LineTerminator* so a comment always consists of all code points starting with the # character up to but not including the line terminator.

Comments behave like white space and may appear after any token, or before a line terminator, and have no significance to the semantic meaning of a GraphQL query document.

Insignificant Commas

Comma ::

,

Similar to white space and line terminators, commas (,) are used to improve the legibility of source text and separate lexical tokens but are otherwise syntactically and semantically insignificant within GraphQL query documents.

Non-significant comma characters ensure that the absence or presence of a comma does not meaningfully alter the interpreted syntax of the document, as this can be a common user-error in other languages. It also allows for the stylistic use of either trailing commas or line-terminators as list delimiters which are both often desired for legibility and maintainability of source code.

Lexical Tokens

Token ::

Punctuator

Name

IntValue

FloatValue

StringValue

A GraphQL document is comprised of several kinds of indivisible lexical tokens defined here in a lexical grammar by patterns of source Unicode characters.

Tokens are later used as terminal symbols in a GraphQL query document syntactic grammars.

Ignored Tokens

Ignored ::

UnicodeBOM
WhiteSpace
LineTerminator
Comment
Comma



Before and after every lexical token may be any amount of ignored tokens including *WhiteSpace* and *Comment*. No ignored regions of a source document are significant, however ignored source characters may appear within a lexical token in a significant way, for example a *String* may contain white space characters.

No characters are ignored while parsing a given token, as an example no white space characters are permitted between the characters defining a *FloatValue*.

Punctuators

Punctuator :: one of

! \$ () ... : = @ [] { | }

GraphQL documents include punctuation in order to describe structure. GraphQL is a data description language and not a programming language, therefore GraphQL lacks the punctuation often used to describe mathematical expressions.

Names

Name ::

/[_A-Za-z][_0-9A-Za-z]*/

GraphQL query documents are full of named things: operations, fields, arguments, directives, fragments, and variables. All names must follow the same grammatical form.

Names in GraphQL are case-sensitive. That is to say *name*, *Name*, and *NAME* all refer to different names. Underscores are significant, which means *other_name* and *othername* are two different names.

Names in GraphQL are limited to this ASCII subset of possible characters to support interoperability with as many other systems as possible.

Query Document

Document :



Definition :

OperationDefinition

FragmentDefinition

A GraphQL query document describes a complete file or request string received by a GraphQL service. A document contains multiple definitions of Operations and Fragments. GraphQL query documents are only executable by a server if they contain an operation. However documents which do not contain operations may still be parsed and validated to allow client to represent a single request across many documents.

If a document contains only one operation, that operation may be unnamed or represented in the shorthand form, which omits both the query keyword and operation name. Otherwise, if a GraphQL query document contains multiple operations, each operation must be named. When submitting a query document with multiple operations to a GraphQL service, the name of the desired operation to be executed must also be provided.

Operations

OperationDefinition :

OperationType *Name*_{opt} *VariableDefinitions*_{opt} *Directives*_{opt} *SelectionSet*
SelectionSet

OperationType : **one of**
query mutation

There are two types of operations that GraphQL models:

- query – a read-only fetch.
- mutation – a write followed by a fetch.

Each operation is represented by an optional operation name and a selection set.

For example, this mutation operation might “like” a story and then retrieve the new number of likes:

```
mutation {
  likeStory(storyID: 12345) {
    story {
      likeCount
    }
  }
}
```

```
}  
}
```



Query shorthand

If a document contains only one query operation, and that query defines no variables and contains no directives, that operation may be represented in a short-hand form which omits the query keyword and query name.

For example, this unnamed query operation is written via query shorthand.

```
{  
  field  
}
```

Note

many examples below will use the query short-hand syntax.

Selection Sets

SelectionSet :

```
{ Selectionlist }
```

Selection :

- Field*
- FragmentSpread*
- InlineFragment*

An operation selects the set of information it needs, and will receive exactly that information and nothing more, avoiding over-fetching and under-fetching data.

```
{  
  id  
  firstName  
  lastName  
}
```

In this query, the `id`, `firstName`, and `lastName` fields form a selection set. Selection sets may also contain fragment references.

Fields



Field :

*Alias*_{opt} *Name* *Arguments*_{opt} *Directives*_{opt} *SelectionSet*_{opt}

A selection set is primarily composed of fields. A field describes one discrete piece of information available to request within a selection set.

Some fields describe complex data or relationships to other data. In order to further explore this data, a field may itself contain a selection set, allowing for deeply nested requests. All GraphQL operations must specify their selections down to fields which return scalar values to ensure an unambiguously shaped response.

For example, this operation selects fields of complex data and relationships down to scalar values.

```
{
  me {
    id
    firstName
    lastName
    birthday {
      month
      day
    }
    friends {
      name
    }
  }
}
```

Fields in the top-level selection set of an operation often represent some information that is globally accessible to your application and its current viewer. Some typical examples of these top fields include references to a current logged-in viewer, or accessing certain types of data referenced by a unique identifier.

`me` could represent the currently logged in viewer.

```
{
  me {
    name
  }
}
```

`user` represents one of many users in a graph of data, referred to by a



```
# unique identifier.
{
  user(id: 4) {
    name
  }
}
```

Arguments

Arguments :
(Argument_{list})

Argument :
Name : Value

Fields are conceptually functions which return values, and occasionally accept arguments which alter their behavior. These arguments often map directly to function arguments within a GraphQL server's implementation.

In this example, we want to query a specific user (requested via the `id` argument) and their profile picture of a specific `size`:

```
{
  user(id: 4) {
    id
    name
    profilePic(size: 100)
  }
}
```

Many arguments can exist for a given field:

```
{
  user(id: 4) {
    id
    name
    profilePic(width: 100, height: 50)
  }
}
```

Arguments are unordered

Arguments may be provided in any syntactic order and maintain identical semantic meaning.

These two queries are semantically identical:

```
{
  picture(width: 200, height: 100)
}
```

```
{
  picture(height: 100, width: 200)
}
```

Field Alias

Alias :

Name :

By default, the key in the response object will use the field name queried. However, you can define a different name by specifying an alias.

In this example, we can fetch two profile pictures of different sizes and ensure the resulting object will not have duplicate keys:

```
{
  user(id: 4) {
    id
    name
    smallPic: profilePic(size: 64)
    bigPic: profilePic(size: 1024)
  }
}
```

Which returns the result:

```
{
  "user": {
    "id": 4,
    "name": "Mark Zuckerberg",
    "smallPic": "https://cdn.site.io/pic-4-64.jpg",
    "bigPic": "https://cdn.site.io/pic-4-1024.jpg"
  }
}
```



```
}
}
```

Since the top level of a query is a field, it also can be given an alias:

```
{
  zuck: user(id: 4) {
    id
    name
  }
}
```

Returns the result:

```
{
  "zuck": {
    "id": 4,
    "name": "Mark Zuckerberg"
  }
}
```

A field's response key is its alias if an alias is provided, and it is otherwise the field's name.

Fragments

FragmentSpread :

... *FragmentName* *Directives*_{opt}

FragmentDefinition :

fragment *FragmentName* *TypeCondition* *Directives*_{opt} *SelectionSet*

FragmentName :

Name but not **on**

Fragments are the primary unit of composition in GraphQL.

Fragments allow for the reuse of common repeated selections of fields, reducing duplicated text in the document. Inline Fragments can be used directly within a selection to condition upon a type condition when querying against an interface or union.

For example, if we wanted to fetch some common information about mutual friends as well as friends of some user:



```
query noFragments {  
  user(id: 4) {  
    friends(first: 10) {  
      id  
      name  
      profilePic(size: 50)  
    }  
    mutualFriends(first: 10) {  
      id  
      name  
      profilePic(size: 50)  
    }  
  }  
}
```

The repeated fields could be extracted into a fragment and composed by a parent fragment or query.

```
query withFragments {  
  user(id: 4) {  
    friends(first: 10) {  
      ...friendFields  
    }  
    mutualFriends(first: 10) {  
      ...friendFields  
    }  
  }  
}
```

```
fragment friendFields on User {  
  id  
  name  
  profilePic(size: 50)  
}
```

Fragments are consumed by using the spread operator (...). All fields selected by the fragment will be added to the query field selection at the same level as the fragment invocation. This happens through multiple levels of fragment spreads.

For example:

```
query withNestedFragments {  
  user(id: 4) {
```



```

    friends(first: 10) {
      ...friendFields
    }
    mutualFriends(first: 10) {
      ...friendFields
    }
  }
}

```

```

fragment friendFields on User {
  id
  name
  ...standardProfilePic
}

```

```

fragment standardProfilePic on User {
  profilePic(size: 50)
}

```

The queries `noFragments`, `withFragments`, and `withNestedFragments` all produce the same response object.

Type Conditions

TypeCondition :

on *NamedType*

Fragments must specify the type they apply to. In this example, `friendFields` can be used in the context of querying a `User`.

Fragments cannot be specified on any input value (scalar, enumeration, or input object).

Fragments can be specified on object types, interfaces, and unions.

Selections within fragments only return values when concrete type of the object it is operating on matches the type of the fragment.

For example in this query on the Facebook data model:

```

query FragmentTyping {
  profiles(handles: ["zuck", "cocacola"]) {
    handle
    ...userFragment
  }
}

```



```

    ...pageFragment
  }
}

```

```

fragment userFragment on User {
  friends {
    count
  }
}

```

```

fragment pageFragment on Page {
  likers {
    count
  }
}

```

The profiles root field returns a list where each element could be a Page or a User. When the object in the profiles result is a User, friends will be present and likers will not. Conversely when the result is a Page, likers will be present and friends will not.

```

{
  "profiles": [
    {
      "handle": "zuck",
      "friends": { "count" : 1234 }
    },
    {
      "handle": "cocacola",
      "likers": { "count" : 90234512 }
    }
  ]
}

```

Inline Fragments

InlineFragment :

... TypeCondition_{opt} Directives_{opt} SelectionSet

Fragments can be defined inline within a selection set. This is done to conditionally include fields based on their runtime type. This feature of standard fragment inclusion was demonstrated in the query FragmentTyping example. We could accomplish the same thing using inline fragments.



```

query inlineFragmentTyping {
  profiles(handles: ["zuck", "cocacola"]) {
    handle
    ... on User {
      friends {
        count
      }
    }
    ... on Page {
      likers {
        count
      }
    }
  }
}

```

Inline fragments may also be used to apply a directive to a group of fields. If the TypeCondition is omitted, an inline fragment is considered to be of the same type as the enclosing context.

```

query inlineFragmentNoType($expandedInfo: Boolean) {
  user(handle: "zuck") {
    id
    name
    ... @include(if: $expandedInfo) {
      firstName
      lastName
      birthday
    }
  }
}

```

Input Values

```

Value[Const] :
  [~Const] Variable
  IntValue
  FloatValue
  StringValue
  BooleanValue
  NullValue

```

EnumValue
*ListValue*_[?Const]
*ObjectValue*_[?Const]



Field and directive arguments accept input values of various literal primitives; input values can be scalars, enumeration values, lists, or input objects.

If not defined as constant (for example, in *DefaultValue*), input values can be specified as a variable. List and inputs objects may also contain variables (unless defined to be constant).

Int Value

IntValue ::
IntegerPart

IntegerPart ::
*NegativeSign*_{opt} **0**
*NegativeSign*_{opt} *NonZeroDigit* *Digit*_{list, opt}

NegativeSign ::
 -

Digit :: **one of**
0 1 2 3 4 5 6 7 8 9

NonZeroDigit ::
Digit but not **0**

An Int number is specified without a decimal point or exponent (ex. 1).

Float Value

FloatValue ::
IntegerPart *FractionalPart*
IntegerPart *ExponentPart*
IntegerPart *FractionalPart* *ExponentPart*

FractionalPart ::
 . *Digit*_{list}

ExponentPart ::
ExponentIndicator *Sign*_{opt} *Digit*_{list}

ExponentIndicator :: **one of**

e E

Sign :: **one of**
+ -

A Float number includes either a decimal point (ex. 1.0) or an exponent (ex. 1e50) or both (ex. 6.0221413e23).

Boolean Value

BooleanValue : **one of**
true false

The two keywords `true` and `false` represent the two boolean values.

String Value

StringValue ::
 " "
 " *StringCharacter*_{list} "

StringCharacter ::
SourceCharacter but not " or \ or *LineTerminator*
 \u *EscapedUnicode*
 \ *EscapedCharacter*

EscapedUnicode ::
 /[0-9A-Fa-f]{4}/

EscapedCharacter :: **one of**
 " \ / **b f n r t**

Strings are sequences of characters wrapped in double-quotes ("). (ex. "Hello World"). White space and other otherwise-ignored characters are significant within a string value.

Note

Unicode characters are allowed within String value literals, however GraphQL source must not contain some ASCII control characters so escape sequences must be used to represent these characters.

Semantics

StringValue :: " "



1. Return an empty Unicode character sequence.

StringValue :: " *StringCharacter*_{list} "

1. Return the Unicode character sequence of all *StringCharacter* Unicode character values.

StringCharacter :: *SourceCharacter* but not " or \ or *LineTerminator*

1. Return the character value of *SourceCharacter*.

StringCharacter :: \u *EscapedUnicode*

1. Return the character whose code unit value in the Unicode Basic Multilingual Plane is the 16-bit hexadecimal value *EscapedUnicode*.

StringCharacter :: \ *EscapedCharacter*

1. Return the character value of *EscapedCharacter* according to the table below.

Escaped Character	Code Unit Value	Character Name
"	U+0022	double quote
\	U+005C	reverse solidus (back slash)
/	U+002F	solidus (forward slash)
b	U+0008	backspace
f	U+000C	form feed
n	U+000A	line feed (new line)
r	U+000D	carriage return
t	U+0009	horizontal tab

Null Value

NullValue :

null

Null values are represented as the keyword **null**.

GraphQL has two semantically different ways to represent the lack of a value:

- Explicitly providing the literal value: **null**.
- Implicitly not providing a value at all.

For example, these two field calls are similar, but are not identical:



```
{
  field(arg: null)
  field
}
```

The first has explicitly provided **null** to the argument “arg”, while the second has implicitly not provided a value to the argument “arg”. These two forms may be interpreted differently. For example, a mutation representing deleting a field vs not altering a field, respectively. Neither form may be used for an input expecting a Non-Null type.

Note

The same two methods of representing the lack of a value are possible via variables by either providing the a variable value as **null** and not providing a variable value at all.

Enum Value

EnumValue :

Name but not **true** or **false** or **null**

Enum values are represented as unquoted names (ex. MOBILE_WEB). It is recommended that Enum values be “all caps”. Enum values are only used in contexts where the precise enumeration type is known. Therefore it’s not necessary to supply an enumeration type name in the literal.

List Value

*ListValue*_[Const] :

[]

[*Value*_{[?Const]list}]

Lists are ordered sequences of values wrapped in square-brackets []. The values of a List literal may be any value literal or variable (ex. [1, 2, 3]).

Commas are optional throughout GraphQL so trailing commas are allowed and repeated commas do not represent missing values.

Semantics

ListValue : []

1. Return a new empty list value.

ListValue : [*Value*_{list}]



1. Let *inputList* be a new empty list value.
2. For each *Value*_{list}
 - a. Let *value* be the result of evaluating *Value*.
 - b. Append *value* to *inputList*.
3. Return *inputList*

Input Object Values

*ObjectValue*_[Const] :

{ }

{ *ObjectField*_{[?Const]list} }

*ObjectField*_[Const] :

Name : *Value*_[?Const]

Input object literal values are unordered lists of keyed input values wrapped in curly-braces { }. The values of an object literal may be any input value literal or variable (ex. { name: "Hello world", score: 1.0 }). We refer to literal representation of input objects as “object literals.”

Input object fields are unordered

Input object fields may be provided in any syntactic order and maintain identical semantic meaning.

These two queries are semantically identical:

```
{
  nearestThing(location: { lon: 12.43, lat: -53.211 })
}
```

```
{
  nearestThing(location: { lat: -53.211, lon: 12.43 })
}
```

Semantics

ObjectValue : { }

1. Return a new input object value with no fields.



ObjectValue : { *ObjectField*_{list} }

1. Let *inputObject* be a new input object value with no fields.
2. For each *field* in *ObjectField*_{list}
 - a. Let *name* be *Name* in *field*.
 - b. Let *value* be the result of evaluating *Value* in *field*.
 - c. Add a field to *inputObject* of name *name* containing value *value*.
3. Return *inputObject*

Variables

Variable :

\$ *Name*

VariableDefinitions :

(*VariableDefinition*_{list})

VariableDefinition :

Variable : *Type* *DefaultValue*_{opt}

DefaultValue :

= *Value*_[Const]

A GraphQL query can be parameterized with variables, maximizing query reuse, and avoiding costly string building in clients at runtime.

If not defined as constant (for example, in *DefaultValue*), a *Variable* can be supplied for an input value.

Variables must be defined at the top of an operation and are in scope throughout the execution of that operation.

In this example, we want to fetch a profile picture size based on the size of a particular device:

```
query getZuckProfile($devicePicSize: Int) {
  user(id: 4) {
    id
    name
    profilePic(size: $devicePicSize)
  }
}
```

Values for those variables are provided to a GraphQL service along with a request so they may be substituted during execution. If providing JSON for the variables' values, we could run this query and request profilePic of size 60 width:

```
{
  "devicePicSize": 60
}
```

Variable use within Fragments

Query variables can be used within fragments. Query variables have global scope with a given operation, so a variable used within a fragment must be declared in any top-level operation that transitively consumes that fragment. If a variable is referenced in a fragment and is included by an operation that does not define that variable, the operation cannot be executed.

Input Types

Type :

NamedType

ListType

NonNullType

NamedType :

Name

ListType :

[*Type*]

NonNullType :

NamedType !

ListType !

GraphQL describes the types of data expected by query variables. Input types may be lists of another input type, or a non-null variant of any other input type.

Semantics

Type : *Name*

1. Let *name* be the string value of *Name*
2. Let *type* be the type defined in the Schema named *name*
3. *type* must not be **null**
4. Return *type*

Type : [*Type*]



1. Let *itemType* be the result of evaluating *Type*
2. Let *type* be a List type where *itemType* is the contained type.
3. Return *type*

Type : *Type* !

1. Let *nullableType* be the result of evaluating *Type*
2. Let *type* be a Non-Null type where *nullableType* is the contained type.
3. Return *type*

Directives

Directives :

*Directive*_{list}

Directive :

@ *Name* *Arguments*_{opt}

Directives provide a way to describe alternate runtime execution and type validation behavior in a GraphQL document.

In some cases, you need to provide options to alter GraphQL's execution behavior in ways field arguments will not suffice, such as conditionally including or skipping a field. Directives provide this by describing additional information to the executor.

Directives have a name along with a list of arguments which may accept values of any input type.

Directives can be used to describe additional information for fields, fragments, and operations.

As future versions of GraphQL adopt new configurable execution capabilities, they may be exposed via directives.

Type System

The GraphQL Type system describes the capabilities of a GraphQL server and is used to determine if a query is valid. The type system also describes the input types of query variables

to determine if values provided at runtime are valid.



A GraphQL server's capabilities are referred to as that server's "schema". A schema is defined in terms of the types and directives it supports.

A given GraphQL schema must itself be internally valid. This section describes the rules for this validation process where relevant.

A GraphQL schema is represented by a root type for each kind of operation: query and mutation; this determines the place in the type system where those operations begin.

All types within a GraphQL schema must have unique names. No two provided types may have the same name. No provided type may have a name which conflicts with any built in types (including Scalar and Introspection types).

All directives within a GraphQL schema must have unique names. A directive and a type may share the same name, since there is no ambiguity between them.

Types

The fundamental unit of any GraphQL Schema is the type. There are eight kinds of types in GraphQL.

The most basic type is a `Scalar`. A scalar represents a primitive value, like a string or an integer. Oftentimes, the possible responses for a scalar field are enumerable. GraphQL offers an `Enum` type in those cases, where the type specifies the space of valid responses.

Scalars and Enums form the leaves in response trees; the intermediate levels are `Object` types, which define a set of fields, where each field is another type in the system, allowing the definition of arbitrary type hierarchies.

GraphQL supports two abstract types: interfaces and unions.

An `Interface` defines a list of fields; `Object` types that implement that interface are guaranteed to implement those fields. Whenever the type system claims it will return an interface, it will return a valid implementing type.

A `Union` defines a list of possible types; similar to interfaces, whenever the type system claims a union will be returned, one of the possible types will be returned.

All of the types so far are assumed to be both nullable and singular: e.g. a scalar string returns either null or a singular string. The type system might want to define that it returns a list of other types; the `List` type is provided for this reason, and wraps another type. Similarly, the `Non-Null` type wraps another type, and denotes that the result will never be null. These two types are referred to as "wrapping types"; non-wrapping types are referred to as "base types".

A wrapping type has an underlying “base type”, found by continually unwrapping the type until a base type is found.

Finally, oftentimes it is useful to provide complex structs as inputs to GraphQL queries; the `Input Object` type allows the schema to define exactly what data is expected from the client in these queries.

Scalars

As expected by the name, a scalar represents a primitive value in GraphQL. GraphQL responses take the form of a hierarchical tree; the leaves on these trees are GraphQL scalars.

All GraphQL scalars are representable as strings, though depending on the response format being used, there may be a more appropriate primitive for the given scalar type, and server should use those types when appropriate.

GraphQL provides a number of built-in scalars, but type systems can add additional scalars with semantic meaning. For example, a GraphQL system could define a scalar called `Time` which, while serialized as a string, promises to conform to ISO-8601. When querying a field of type `Time`, you can then rely on the ability to parse the result with an ISO-8601 parser and use a client-specific primitive for time. Another example of a potentially useful custom scalar is `Url`, which serializes as a string, but is guaranteed by the server to be a valid URL.

A server may omit any of the built-in scalars from its schema, for example if a schema does not refer to a floating-point number, then it will not include the `Float` type. However, if a schema includes a type with the name of one of the types described here, it must adhere to the behavior described. As an example, a server must not include a type called `Int` and use it to represent 128-bit numbers, or internationalization information.

Result Coercion

A GraphQL server, when preparing a field of a given scalar type, must uphold the contract the scalar type describes, either by coercing the value or producing an error.

For example, a GraphQL server could be preparing a field with the scalar type `Int` and encounter a floating-point number. Since the server must not break the contract by yielding a non-integer, the server should truncate the fractional value and only yield the integer value. If the server encountered a boolean `true` value, it should return `1`. If the server encountered a string, it may attempt to parse the string for a base-10 integer value. If the server encounters some value that cannot be reasonably coerced to an `Int`, then it must raise a field error.

Since this coercion behavior is not observable to clients of the GraphQL server, the precise rules of coercion are left to the implementation. The only requirement is that the server must yield values which adhere to the expected `Scalar` type.

Input Coercion



If a GraphQL server expects a scalar type as input to an argument, coercion is observable and the rules must be well defined. If an input value does not match a coercion rule, a query error must be raised.

GraphQL has different constant literals to represent integer and floating-point input values, and coercion rules may apply differently depending on which type of input value is encountered. GraphQL may be parameterized by query variables, the values of which are often serialized when sent over a transport like HTTP. Since some common serializations (ex. JSON) do not discriminate between integer and floating-point values, they are interpreted as an integer input value if they have an empty fractional part (ex. `1.0`) and otherwise as floating-point input value.

For all types below, with the exception of Non-Null, if the explicit value **null** is provided, then the result of input coercion is **null**.

Built-in Scalars

GraphQL provides a basic set of well-defined Scalar types. A GraphQL server should support all of these types, and a GraphQL server which provide a type by these names must adhere to the behavior described below.

Int

The Int scalar type represents a signed 32-bit numeric non-fractional value. Response formats that support a 32-bit integer or a number type should use that type to represent this scalar.

Result Coercion

GraphQL servers should coerce non-int raw values to Int when possible otherwise they must raise a field error. Examples of this may include returning 1 for the floating-point number `1.0`, or 2 for the string `"2"`.

Input Coercion

When expected as an input type, only integer input values are accepted. All other input values, including strings with numeric content, must raise a query error indicating an incorrect type. If the integer input value represents a value less than -2^{31} or greater than or equal to 2^{31} , a query error should be raised.

Note

Numeric integer values larger than 32-bit should either use String or a custom-defined Scalar type, as not all platforms and transports support encoding integer numbers larger than 32-bit.

Float



The Float scalar type represents signed double-precision fractional values as specified by [IEEE 754](#). Response formats that support an appropriate double-precision number type should use that type to represent this scalar.

Result Coercion

GraphQL servers should coerce non-floating-point raw values to Float when possible otherwise they must raise a field error. Examples of this may include returning `1.0` for the integer number 1, or `2.0` for the string "2".

Input Coercion

When expected as an input type, both integer and float input values are accepted. Integer input values are coerced to Float by adding an empty fractional part, for example `1.0` for the integer input value 1. All other input values, including strings with numeric content, must raise a query error indicating an incorrect type. If the integer input value represents a value not representable by IEEE 754, a query error should be raised.

String

The String scalar type represents textual data, represented as UTF-8 character sequences. The String type is most often used by GraphQL to represent free-form human-readable text. All response formats must support string representations, and that representation must be used here.

Result Coercion

GraphQL servers should coerce non-string raw values to String when possible otherwise they must raise a field error. Examples of this may include returning the string `"true"` for a boolean true value, or the string `"1"` for the integer 1.

Input Coercion

When expected as an input type, only valid UTF-8 string input values are accepted. All other input values must raise a query error indicating an incorrect type.

Boolean

The Boolean scalar type represents `true` or `false`. Response formats should use a built-in boolean type if supported; otherwise, they should use their representation of the integers 1 and 0.

Result Coercion

GraphQL servers should coerce non-boolean raw values to Boolean when possible otherwise they must raise a field error. Examples of this may include returning `true` for any non-zero number.

Input Coercion

When expected as an input type, only boolean input values are accepted. All other input values must raise a query error indicating an incorrect type.

ID

The ID scalar type represents a unique identifier, often used to refetch an object or as the key for a cache. The ID type is serialized in the same way as a `String`; however, it is not intended to be human-readable. While it is often numeric, it should always serialize as a `String`.

Result Coercion

GraphQL is agnostic to ID format, and serializes to string to ensure consistency across many formats ID could represent, from small auto-increment numbers, to large 128-bit random numbers, to base64 encoded values, or string values of a format like [GUID](#).

GraphQL servers should coerce as appropriate given the ID formats they expect. When coercion is not possible they must raise a field error.

Input Coercion

When expected as an input type, any string (such as `"4"`) or integer (such as `4`) input value should be coerced to ID as appropriate for the ID formats a given GraphQL server expects. Any other input value, including float input values (such as `4.0`), must raise a query error indicating an incorrect type.

Objects

GraphQL queries are hierarchical and composed, describing a tree of information. While Scalar types describe the leaf values of these hierarchical queries, Objects describe the intermediate levels.

GraphQL Objects represent a list of named fields, each of which yield a value of a specific type. Object values should be serialized as ordered maps, where the queried field names (or aliases) are the keys and the result of evaluating the field is the value, ordered by the order in which they appear in the query.

For example, a type `Person` could be described as:



```
type Person {  
  name: String  
  age: Int  
  picture: Url  
}
```

Where `name` is a field that will yield a `String` value, and `age` is a field that will yield an `Int` value, and `picture` is a field that will yield a `Url` value.

A query of an object value must select at least one field. This selection of fields will yield an ordered map containing exactly the subset of the object queried, which should be represented in the order in which they were queried. Only fields that are declared on the object type may validly be queried on that object.

For example, selecting all the fields of `Person`:

```
{  
  name  
  age  
  picture  
}
```

Would yield the object:

```
{  
  "name": "Mark Zuckerberg",  
  "age": 30,  
  "picture": "http://some.cdn/picture.jpg"  
}
```

While selecting a subset of fields:

```
{  
  age  
  name  
}
```

Must only yield exactly that subset:

```
{  
  "age": 30,  
  "name": "Mark Zuckerberg"  
}
```

A field of an Object type may be a Scalar, Enum, another Object type, an Interface, or a Union. Additionally, it may be any wrapping type whose underlying base type is one of those five.

For example, the Person type might include a relationship:

```
type Person {
  name: String
  age: Int
  picture: Url
  relationship: Person
}
```

Valid queries must supply a nested field set for a field that returns an object, so this query is not valid:

```
{
  name
  relationship
}
```

However, this example is valid:

```
{
  name
  relationship {
    name
  }
}
```

And will yield the subset of each object type queried:

```
{
  "name": "Mark Zuckerberg",
  "relationship": {
    "name": "Priscilla Chan"
  }
}
```

Field Ordering

When querying an Object, the resulting mapping of fields are conceptually ordered in the same order in which they were encountered during query execution, excluding fragments for which

the type does not apply and fields or fragments that are skipped via `@skip` or `@include` directives. This ordering is correctly produced when using the `CollectFields()` algorithm.



Response serialization formats capable of representing ordered maps should maintain this ordering. Serialization formats which can only represent unordered maps should retain this order grammatically (such as JSON).

Producing a response where fields are represented in the same order in which they appear in the request improves human readability during debugging and enables more efficient parsing of responses if the order of properties can be anticipated.

If a fragment is spread before other fields, the fields that fragment specifies occur in the response before the following fields.

```
{  
  foo  
  ...Frag  
  qux  
}
```

```
fragment Frag on Query {  
  bar  
  baz  
}
```

Produces the ordered result:

```
{  
  "foo": 1,  
  "bar": 2,  
  "baz": 3,  
  "qux": 4  
}
```

If a field is queried multiple times in a selection, it is ordered by the first time it is encountered. However fragments for which the type does not apply does not affect ordering.

```
{  
  foo  
  ...Ignored  
  ...Matching  
  bar  
}
```



```
fragment Ignored on UnknownType {  
  qux  
  baz  
}
```

```
fragment Matching on Query {  
  bar  
  qux  
  foo  
}
```

Produces the ordered result:

```
{  
  "foo": 1,  
  "bar": 2,  
  "qux": 3  
}
```

Also, if directives result in fields being excluded, they are not considered in the ordering of fields.

```
{  
  foo @skip(if: true)  
  bar  
  foo  
}
```

Produces the ordered result:

```
{  
  "bar": 1,  
  "foo": 2  
}
```

Result Coercion

Determining the result of coercing an object is the heart of the GraphQL executor, so this is covered in that section of the spec.

Input Coercion

Objects are never valid inputs.

Object Field Arguments



Object fields are conceptually functions which yield values. Occasionally object fields can accept arguments to further specify the return value. Object field arguments are defined as a list of all possible argument names and their expected input types.

For example, a `Person` type with a `picture` field could accept an argument to determine what size of an image to return.

```
type Person {  
  name: String  
  picture(size: Int): Url  
}
```

GraphQL queries can optionally specify arguments to their fields to provide these arguments.

This example query:

```
{  
  name  
  picture(size: 600)  
}
```

May yield the result:

```
{  
  "name": "Mark Zuckerberg",  
  "picture": "http://some.cdn/picture_600.jpg"  
}
```

The type of an object field argument can be any Input type.

Object Field deprecation

Fields in an object may be marked as deprecated as deemed necessary by the application. It is still legal to query for these fields (to ensure existing clients are not broken by the change), but the fields should be appropriately treated in documentation and tooling.

Object type validation

Object types have the potential to be invalid if incorrectly defined. This set of rules must be adhered to by every Object type in a GraphQL schema.

1. An Object type must define one or more fields.
2. The fields of an Object type must have unique names within that Object type; no two fields may share the same name.
3. An object type must be a super-set of all interfaces it implements:
 1. The object type must include a field of the same name for every field defined in an interface.
 1. The object field must be of a type which is equal to or a sub-type of the interface field (covariant).
 1. An object field type is a valid sub-type if it is equal to (the same type as) the interface field type.
 2. An object field type is a valid sub-type if it is an Object type and the interface field type is either an Interface type or a Union type and the object field type is a possible type of the interface field type.
 3. An object field type is a valid sub-type if it is a List type and the interface field type is also a List type and the list-item type of the object field type is a valid sub-type of the list-item type of the interface field type.
 4. An object field type is a valid sub-type if it is a Non-Null variant of a valid sub-type of the interface field type.
 2. The object field must include an argument of the same name for every argument defined in the interface field.
 1. The object field argument must accept the same type (invariant) as the interface field argument.
 3. The object field may include additional arguments not defined in the interface field, but any additional argument must not be required.

Interfaces

GraphQL Interfaces represent a list of named fields and their arguments. GraphQL objects can then implement an interface, which guarantees that they will contain the specified fields.

Fields on a GraphQL interface have the same rules as fields on a GraphQL object; their type can be Scalar, Object, Enum, Interface, or Union, or any wrapping type whose base type is one of those five.

For example, an interface may describe a required field and types such as Person or Business may then implement this interface.

```
interface NamedEntity {
  name: String
}

type Person implements NamedEntity {
  name: String
```



```
    age: Int
  }
```

```
type Business implements NamedEntity {
  name: String
  employeeCount: Int
}
```

Fields which yield an interface are useful when one of many Object types are expected, but some fields should be guaranteed.

To continue the example, a `Contact` might refer to `NamedEntity`.

```
type Contact {
  entity: NamedEntity
  phoneNumber: String
  address: String
}
```

This allows us to write a query for a `Contact` that can select the common fields.

```
{
  entity {
    name
  }
  phoneNumber
}
```

When querying for fields on an interface type, only those fields declared on the interface may be queried. In the above example, `entity` returns a `NamedEntity`, and `name` is defined on `NamedEntity`, so it is valid. However, the following would not be a valid query:

```
{
  entity {
    name
    age
  }
  phoneNumber
}
```

because `entity` refers to a `NamedEntity`, and `age` is not defined on that interface. Querying for `age` is only valid when the result of `entity` is a `Person`; the query can express this using a fragment or an inline fragment:



```
{  
  entity {  
    name  
    ... on Person {  
      age  
    }  
  },  
  phoneNumber  
}
```

Result Coercion

The interface type should have some way of determining which object a given result corresponds to. Once it has done so, the result coercion of the interface is the same as the result coercion of the object.

Input Coercion

Interfaces are never valid inputs.

Interface type validation

Interface types have the potential to be invalid if incorrectly defined.

1. An Interface type must define one or more fields.
2. The fields of an Interface type must have unique names within that Interface type; no two fields may share the same name.

Unions

GraphQL Unions represent an object that could be one of a list of GraphQL Object types, but provides for no guaranteed fields between those types. They also differ from interfaces in that Object types declare what interfaces they implement, but are not aware of what unions contain them.

With interfaces and objects, only those fields defined on the type can be queried directly; to query other fields on an interface, typed fragments must be used. This is the same as for unions, but unions do not define any fields, so **no** fields may be queried on this type without the use of typed fragments.

For example, we might have the following type system:



```
union SearchResult = Photo | Person
```

```
type Person {  
  name: String  
  age: Int  
}
```

```
type Photo {  
  height: Int  
  width: Int  
}
```

```
type SearchQuery {  
  firstSearchResult: SearchResult  
}
```

When querying the `firstSearchResult` field of type `SearchQuery`, the query would ask for all fields inside of a fragment indicating the appropriate type. If the query wanted the name if the result was a `Person`, and the height if it was a photo, the following query is invalid, because the union itself defines no fields:

```
{  
  firstSearchResult {  
    name  
    height  
  }  
}
```

Instead, the query would be:

```
{  
  firstSearchResult {  
    ... on Person {  
      name  
    }  
    ... on Photo {  
      height  
    }  
  }  
}
```

Result Coercion

The union type should have some way of determining which object a given result corresponds to. Once it has done so, the result coercion of the union is the same as the result coercion of the object.

Input Coercion

Unions are never valid inputs.

Union type validation

Union types have the potential to be invalid if incorrectly defined.

1. The member types of a Union type must all be Object base types; Scalar, Interface and Union types may not be member types of a Union. Similarly, wrapping types may not be member types of a Union.
2. A Union type must define one or more member types.

Enums

GraphQL Enums are a variant on the Scalar type, which represents one of a finite set of possible values.

GraphQL Enums are not references for a numeric value, but are unique values in their own right. They serialize as a string: the name of the represented value.

Result Coercion

GraphQL servers must return one of the defined set of possible values. If a reasonable coercion is not possible they must raise a field error.

Input Coercion

GraphQL has a constant literal to represent enum input values. GraphQL string literals must not be accepted as an enum input and instead raise a query error.

Query variable transport serializations which have a different representation for non-string symbolic values (for example, [EDN](#)) should only allow such values as enum input values. Otherwise, for most transport serializations that do not, strings may be interpreted as the enum input value with the same name.

Input Objects

Fields can define arguments that the client passes up with the query, to configure their behavior. These inputs can be Strings or Enums, but they sometimes need to be more complex than this.

The `Object` type defined above is inappropriate for re-use here, because `Objects` can contain fields that express circular references or references to interfaces and unions, neither of which is appropriate for use as an input argument. For this reason, input objects have a separate type in the system.

An `Input Object` defines a set of input fields; the input fields are either scalars, enums, or other input objects. This allows arguments to accept arbitrarily complex structs.

Result Coercion

An input object is never a valid result.

Input Coercion

The value for an input object should be an input object literal or an unordered map, otherwise an error should be thrown. This unordered map should not contain any entries with names not defined by a field of this input object type, otherwise an error should be thrown.

If any non-nullable fields defined by the input object do not have corresponding entries in the original value, were provided a variable for which a value was not provided, or for which the value **null** was provided, an error should be thrown.

The result of coercion is an environment-specific unordered map defining slots for each field both defined by the input object type and provided by the original value.

For each field of the input object type, if the original value has an entry with the same name, and the value at that entry is a literal value or a variable which was provided a runtime value, an entry is added to the result with the name of the field.

The value of that entry in the result is the outcome of input coercing the original entry value according to the input coercion rules of the type declared by the input field.

Following are examples of Input Object coercion for the type:

```
input ExampleInputObject {
  a: String
  b: Int!
}
```

Original Value	Variables	Coerced Value
{ a: "abc", b: 123 }	null	{ a: "abc", b: 123 }
{ a: 123, b: "123" }	null	{ a: "123", b: 123 }
{ a: "abc" }	null	Error: Missing required field <i>b</i>
{ a: "abc", b: null }	null	Error: <i>b</i> must be non-null.



Original Value	Variables	Coerced Value
{ a: null, b: 1 }	null	{ a: null, b: 1 }
{ b: \$var }	{ var: 123 }	{ b: 123 }
{ b: \$var }	{ }	Error: Missing required field <i>b</i> .
{ b: \$var }	{ var: null }	Error: <i>b</i> must be non-null.
{ a: \$var, b: 1 }	{ var: null }	{ a: null, b: 1 }
{ a: \$var, b: 1 }	{ }	{ b: 1 }

Note

there is a semantic difference between the input value explicitly declaring an input field's value as the value **null** vs having not declared the input field at all.

Input Object type validation

1. An Input Object type must define one or more fields.
2. The fields of an Input Object type must have unique names within that Input Object type; no two fields may share the same name.
3. The return types of each defined field must be an Input type.

Lists

A GraphQL list is a special collection type which declares the type of each item in the List (referred to as the *item type* of the list). List values are serialized as ordered lists, where each item in the list is serialized as per the item type. To denote that a field uses a List type the item type is wrapped in square brackets like this: `pets: [Pet]`.

Result Coercion

GraphQL servers must return an ordered list as the result of a list type. Each item in the list must be the result of a result coercion of the item type. If a reasonable coercion is not possible they must raise a field error. In particular, if a non-list is returned, the coercion should fail, as this indicates a mismatch in expectations between the type system and the implementation.

Input Coercion

When expected as an input, list values are accepted only when each item in the list can be accepted by the list's item type.

If the value passed as an input to a list type is *not* a list and not the **null** value, it should be coerced as though the input was a list of size one, where the value passed is the only item in the list. This is to allow inputs that accept a “var args” to declare their input type as a list; if only one argument is passed (a common case), the client can just pass that value rather than constructing the list.

Note

that when a **null** value is provided via a runtime variable value for a list type that it is interpreted as no list being provided, and not a list of size one with the value **null**.

Non-Null

By default, all types in GraphQL are nullable; the **null** value is a valid response for all of the above types. To declare a type that disallows null, the GraphQL Non-Null type can be used. This type wraps an underlying type, and this type acts identically to that wrapped type, with the exception that **null** is not a valid response for the wrapping type. A trailing exclamation mark is used to denote a field that uses a Non-Null type like this: `name: String!`.

Nullable vs. Optional

Fields are *always* optional within the context of a query, a field may be omitted and the query is still valid. However fields that return Non-Null types will never return the value **null** if queried.

Inputs (such as field arguments), are always optional by default. However a non-null input type is required. In addition to not accepting the value **null**, it also does not accept omission. For the sake of simplicity nullable types are always optional and non-null types are always required.

Result Coercion

In all of the above result coercions, **null** was considered a valid value. To coerce the result of a Non-Null type, the coercion of the wrapped type should be performed. If that result was not **null**, then the result of coercing the Non-Null type is that result. If that result was **null**, then a field error must be raised.

Input Coercion

If an argument or input-object field of a Non-Null type is not provided, is provided with the literal value **null**, or is provided with a variable that was either not provided a value at runtime, or was provided the value **null**, then a query error must be raised.

If the value provided to the Non-Null type is provided with a literal value other than **null**, or a Non-Null variable value, it is coerced using the input coercion for the wrapped type.



Example :

A non-null argument cannot be omitted.

```
{  
  fieldWithNonNullArg  
}
```

Example :

The value {null } cannot be provided to a non-null argument.

```
{  
  fieldWithNonNullArg(nonNullArg: null)  
}
```

Example :

A variable of a nullable type cannot be provided to a non-null argument.

```
query withNullableVariable($var: String) {  
  fieldWithNonNullArg(nonNullArg: $var)  
}
```

Note

The Validation section defines providing a nullable variable type to a non-null input type as invalid.

Non-Null type validation

1. A Non-Null type must not wrap another Non-Null type.

Directives

A GraphQL schema includes a list of the directives the execution engine supports.

GraphQL implementations should provide the @skip and @include directives.

@skip

The @skip directive may be provided for fields, fragment spreads, and inline fragments, and allows for conditional exclusion during execution as described by the if argument.

In this example `experimentalField` will be queried only if the `$someTest` is provided a ~~false~~ value.

```
query myQuery($someTest: Boolean) {
  experimentalField @skip(if: $someTest)
}
```

@include

The `@include` directive may be provided for fields, fragment spreads, and inline fragments, and allows for conditional inclusion during execution as described by the `if` argument.

In this example `experimentalField` will be queried only if the `$someTest` is provided a `true` value.

```
query myQuery($someTest: Boolean) {
  experimentalField @include(if: $someTest)
}
```

Note

Neither `@skip` nor `@include` has precedence over the other. In the case that both the `@skip` and `@include` directives are provided in on the same the field or fragment, it *must* be queried only if the `@skip` condition is false *and* the `@include` condition is true. Stated conversely, the field or fragment must *not* be queried if either the `@skip` condition is true *or* the `@include` condition is false.

Initial types

A GraphQL schema includes types, indicating where query and mutation operations start. This provides the initial entry points into the type system. The query type must always be provided, and is an Object base type. The mutation type is optional; if it is null, that means the system does not support mutations. If it is provided, it must be an object base type.

The fields on the query type indicate what fields are available at the top level of a GraphQL query. For example, a basic GraphQL query like this one:

```
query getMe {
  me
}
```


Is valid when the type provided for the query starting type has a field named “me”. Similarly

```
mutation setName {
  setName(name: "Zuck") {
    newName
  }
}
```

Is valid when the type provided for the mutation starting type is not null, and has a field named “setName” with a string argument named “name”.

Introspection

A GraphQL server supports introspection over its schema. This schema is queried using GraphQL itself, creating a powerful platform for tool-building.

Take an example query for a trivial app. In this case there is a User type with three fields: id, name, and birthday.

For example, given a server with the following type definition:

```
type User {
  id: String
  name: String
  birthday: Date
}
```

The query

```
{
  __type(name: "User") {
    name
    fields {
      name
      type {
        name
      }
    }
  }
}
```



```
}
}
```

would return

```
{
  "__type": {
    "name": "User",
    "fields": [
      {
        "name": "id",
        "type": { "name": "String" }
      },
      {
        "name": "name",
        "type": { "name": "String" }
      },
      {
        "name": "birthday",
        "type": { "name": "Date" }
      }
    ]
  }
}
```

General Principles

Naming conventions

Types and fields required by the GraphQL introspection system that are used in the same context as user-defined types and fields are prefixed with two underscores. This in order to avoid naming collisions with user-defined GraphQL types. Conversely, GraphQL type system authors must not define any types, fields, arguments, or any other type system artifact with two leading underscores.

Documentation

All types in the introspection system provide a `description` field of type `String` to allow type designers to publish documentation in addition to capabilities. A GraphQL server may return

the `description` field using Markdown syntax. Therefore it is recommended that any tool that displays description use a Markdown renderer.

Deprecation

To support the management of backwards compatibility, GraphQL fields and enum values can indicate whether or not they are deprecated (`isDeprecated: Boolean`) and a description of why it is deprecated (`deprecationReason: String`).

Tools built using GraphQL introspection should respect deprecation by discouraging deprecated use through information hiding or developer-facing warnings.

Type Name Introspection

GraphQL supports type name introspection at any point within a query by the meta field `__typename: String!` when querying against any Object, Interface, or Union. It returns the name of the object type currently being queried.

This is most often used when querying against Interface or Union types to identify which actual type of the possible types has been returned.

This field is implicit and does not appear in the fields list in any defined type.

Schema Introspection

The schema introspection system is accessible from the meta-fields `__schema` and `__type` which are accessible from the type of the root of a query operation.

```
__schema: __Schema!
__type(name: String!): __Type
```

These fields are implicit and do not appear in the fields list in the root type of the query operation.

The schema of the GraphQL schema introspection system:

```
type __Schema {
  types: [__Type!]!
  queryType: __Type!
  mutationType: __Type
  directives: [__Directive!]!
```



```
}
```

```
type __Type {  
  kind: __TypeKind!  
  name: String  
  description: String  
  
  # OBJECT and INTERFACE only  
  fields(includeDeprecated: Boolean = false): [__Field!]  
  
  # OBJECT only  
  interfaces: [__Type!]  
  
  # INTERFACE and UNION only  
  possibleTypes: [__Type!]  
  
  # ENUM only  
  enumValues(includeDeprecated: Boolean = false): [__EnumValue!]  
  
  # INPUT_OBJECT only  
  inputFields: [__InputValue!]  
  
  # NON_NULL and LIST only  
  ofType: __Type  
}
```

```
type __Field {  
  name: String!  
  description: String  
  args: [__InputValue!]!  
  type: __Type!  
  isDeprecated: Boolean!  
  deprecationReason: String  
}
```

```
type __InputValue {  
  name: String!  
  description: String  
  type: __Type!  
  defaultValue: String  
}
```



```
type __EnumValue {
  name: String!
  description: String
  isDeprecated: Boolean!
  deprecationReason: String
}

enum __TypeKind {
  SCALAR
  OBJECT
  INTERFACE
  UNION
  ENUM
  INPUT_OBJECT
  LIST
  NON_NULL
}

type __Directive {
  name: String!
  description: String
  locations: [__DirectiveLocation!]!
  args: [__InputValue!]!
}

enum __DirectiveLocation {
  QUERY
  MUTATION
  FIELD
  FRAGMENT_DEFINITION
  FRAGMENT_SPREAD
  INLINE_FRAGMENT
}
```

The __Type Type

`__Type` is at the core of the type introspection system. It represents scalars, interfaces, object types, unions, enums in the system.

`__Type` also represents type modifiers, which are used to modify a type that it refers to (`ofType: __Type`). This is how we represent lists, non-nullable types, and the combinations thereof.

Type Kinds



There are several different kinds of type. In each kind, different fields are actually valid. These kinds are listed in the `__TypeKind` enumeration.

Scalar

Represents scalar types such as `Int`, `String`, and `Boolean`. Scalars cannot have fields.

A GraphQL type designer should describe the data format and scalar coercion rules in the description field of any scalar.

Fields

- `kind` must return `__TypeKind.SCALAR`.
- `name` must return a `String`.
- `description` may return a `String` or **`null`**.
- All other fields must return **`null`**.

Object

Object types represent concrete instantiations of sets of fields. The introspection types (e.g. `__Type`, `__Field`, etc) are examples of objects.

Fields

- `kind` must return `__TypeKind.OBJECT`.
- `name` must return a `String`.
- `description` may return a `String` or **`null`**.
- `fields`: The set of fields query-able on this type.
 - Accepts the argument `includeDeprecated` which defaults to **`false`**. If **`true`**, deprecated fields are also returned.
- `interfaces`: The set of interfaces that an object implements.
- All other fields must return **`null`**.

Union

Unions are an abstract type where no common fields are declared. The possible types of a union are explicitly listed out in `possibleTypes`. Types can be made parts of unions without modification of that type.

Fields



- kind must return `__TypeKind.UNION`.
- name must return a String.
- description may return a String or **null**.
- possibleTypes returns the list of types that can be represented within this union. They must be object types.
- All other fields must return **null**.

Interface

Interfaces are an abstract type where there are common fields declared. Any type that implements an interface must define all the fields with names and types exactly matching. The implementations of this interface are explicitly listed out in `possibleTypes`.

Fields

- kind must return `__TypeKind.INTERFACE`.
- name must return a String.
- description may return a String or **null**.
- fields: The set of fields required by this interface.
 - Accepts the argument `includeDeprecated` which defaults to **false**. If **true**, deprecated fields are also returned.
- possibleTypes returns the list of types that implement this interface. They must be object types.
- All other fields must return **null**.

Enum

Enums are special scalars that can only have a defined set of values.

Fields

- kind must return `__TypeKind.ENUM`.
- name must return a String.
- description may return a String or **null**.
- enumValues: The list of `EnumValue`. There must be at least one and they must have unique names.
 - Accepts the argument `includeDeprecated` which defaults to **false**. If **true**, deprecated enum values are also returned.
- All other fields must return **null**.

Input Object

Input objects are composite types used as inputs into queries defined as a list of named input values.

For example the input object `Point` could be defined as:

```
type Point {  
  x: Int  
  y: Int  
}
```

Fields

- `kind` must return `__TypeKind.INPUT_OBJECT`.
- `name` must return a `String`.
- `description` may return a `String` or **null**.
- `inputFields`: a list of `InputValue`.
- All other fields must return **null**.

List

Lists represent sequences of values in GraphQL. A List type is a type modifier: it wraps another type instance in the `ofType` field, which defines the type of each item in the list.

Fields

- `kind` must return `__TypeKind.LIST`.
- `ofType`: Any type.
- All other fields must return **null**.

Non-Null

GraphQL types are nullable. The value **null** is a valid response for field type.

A Non-null type is a type modifier: it wraps another type instance in the `ofType` field. Non-null types do not allow **null** as a response, and indicate required inputs for arguments and input object fields.

- `kind` must return `__TypeKind.NON_NULL`.
- `ofType`: Any type except Non-null.
- All other fields must return **null**.

Combining List and Non-Null

List and Non-Null can compose, representing more complex types.



If the modified type of a List is Non-Null, then that List may not contain any **null** items.

If the modified type of a Non-Null is List, then **null** is not accepted, however an empty list is accepted.

If the modified type of a List is a List, then each item in the first List is another List of the second List's type.

A Non-Null type cannot modify another Non-Null type.

The __Field Type

The __Field type represents each field in an Object or Interface type.

Fields

- name must return a String
- description may return a String or **null**
- args returns a List of __InputValue representing the arguments this field accepts.
- type must return a __Type that represents the type of value returned by this field.
- isDeprecated returns **true** if this field should no longer be used, otherwise **false**.
- deprecationReason optionally provides a reason why this field is deprecated.

The __InputValue Type

The __InputValue type represents field and directive arguments as well as the inputFields of an input object.

Fields

- name must return a String
- description may return a String or **null**
- type must return a __Type that represents the type this input value expects.
- defaultValue may return a String encoding (using the GraphQL language) of the default value used by this input value in the condition a value is not provided at runtime. If this input value has no default value, returns **null**.

The __Directive Type

The __Directive type represents a Directive that a server supports.

Fields



- `name` must return a `String`
- `description` may return a `String` or `null`
- `locations` returns a List of `__DirectiveLocation` representing the valid locations this directive may be placed.
- `args` returns a List of `__InputValue` representing the arguments this directive accepts.

Validation

GraphQL does not just verify if a request is syntactically correct, but also ensures that it is unambiguous and mistake-free in the context of a given GraphQL schema.

An invalid request is still technically executable, and will always produce a stable result as defined by the procedures in the Execution section, however that result may be ambiguous, surprising, or unexpected relative to the request containing validation errors, so execution should not occur for invalid requests.

Typically validation is performed in the context of a request immediately before execution, however a GraphQL service may execute a request without explicitly validating it if that exact same request is known to have been validated before. For example: the request may be validated during development, provided it does not later change, or a service may validate a request once and memoize the result to avoid validating the same request again in the future. Any client-side or development-time tool should report validation errors and not allow the formulation or execution of requests known to be invalid at that given point in time.

Type system evolution

As GraphQL type system schema evolve over time by adding new types and new fields, it is possible that a request which was previously valid could later become invalid. Any change that can cause a previously valid request to become invalid is considered a *breaking change*. GraphQL services and schema maintainers are encouraged to avoid breaking changes, however in order to be more resilient to these breaking changes, sophisticated GraphQL systems may still allow for the execution of requests which *at some point* were known to be free of any validation errors, and have not changed since.

Examples

For this section of this schema, we will assume the following type system in order to demonstrate examples:

```
enum DogCommand { SIT, DOWN, HEEL }
```



```
type Dog implements Pet {
  name: String!
  nickname: String
  barkVolume: Int
  doesKnowCommand(dogCommand: DogCommand!): Boolean!
  isHousetrained(atOtherHomes: Boolean): Boolean!
  owner: Human
}

interface Sentient {
  name: String!
}

interface Pet {
  name: String!
}

type Alien implements Sentient {
  name: String!
  homePlanet: String
}

type Human implements Sentient {
  name: String!
}

enum CatCommand { JUMP }

type Cat implements Pet {
  name: String!
  nickname: String
  doesKnowCommand(catCommand: CatCommand!): Boolean!
  meowVolume: Int
}

union CatOrDog = Cat | Dog
union DogOrHuman = Dog | Human
union HumanOrAlien = Human | Alien

type QueryRoot {
  dog: Dog
}
```



Named Operation Definitions

Operation Name Uniqueness

Formal Specification

- For each operation definition *operation* in the document
- Let *operationName* be the name of *operation*.
- If *operationName* exists
 - Let *operations* be all operation definitions in the document named *operationName*.
 - *operations* must be a set of one.

Explanatory Text

Each named operation definition must be unique within a document when referred to by its name.

For example the following document is valid:

```
query getDogName {  
  dog {  
    name  
  }  
}
```

```
query getOwnerName {  
  dog {  
    owner {  
      name  
    }  
  }  
}
```

While this document is invalid:

```
query getName {  
  dog {  
    name  
  }  
}
```



```
query getName {  
  dog {  
    owner {  
      name  
    }  
  }  
}
```

It is invalid even if the type of each operation is different:

```
query dogOperation {  
  dog {  
    name  
  }  
}
```

```
mutation dogOperation {  
  mutateDog {  
    id  
  }  
}
```

Anonymous Operation Definitions

Lone Anonymous Operation

Formal Specification

- Let *operations* be all operation definitions in the document.
- Let *anonymous* be all anonymous operation definitions in the document.
- If *operations* is a set of more than 1:
 - *anonymous* must be empty.

Explanatory Text

GraphQL allows a short-hand form for defining query operations when only that one operation exists in the document.

For example the following document is valid:

```
{  
  dog {
```



```
    name
  }
}
```

While this document is invalid:

```
{
  dog {
    name
  }
}
```

```
query getName {
  dog {
    owner {
      name
    }
  }
}
```

Fields

Field Selections on Objects, Interfaces, and Unions Types

Formal Specification

- For each *selection* in the document.
- Let *fieldName* be the target field of *selection*
- *fieldName* must be defined on type in scope

Explanatory Text

The target field of a field selection must be defined on the scoped type of the selection set. There are no limitations on alias names.

For example the following fragment would not pass validation:

```
fragment fieldNotDefined on Dog {
  meowVolume
}
```



```
fragment aliasedLyingFieldTargetNotDefined on Dog {  
  barkVolume: kawVolume  
}
```

For interfaces, direct field selection can only be done on fields. Fields of concrete implementors are not relevant to the validity of the given interface-typed selection set.

For example, the following is valid:

```
fragment interfaceFieldSelection on Pet {  
  name  
}
```

and the following is invalid:

```
fragment definedOnImplementorsButNotInterface on Pet {  
  nickname  
}
```

Because unions do not define fields, fields may not be directly selected from a union-typed selection set, with the exception of the meta-field `__typename`. Fields from a union-typed selection set must only be queried indirectly via a fragment.

For example the following is valid:

```
fragment inDirectFieldSelectionOnUnion on CatOrDog {  
  __typename  
  ... on Pet {  
    name  
  }  
  ... on Dog {  
    barkVolume  
  }  
}
```

But the following is invalid:

```
fragment directFieldSelectionOnUnion on CatOrDog {  
  name  
  barkVolume  
}
```

Field Selection Merging



Formal Specification

- Let *set* be any selection set defined in the GraphQL document.
- `FieldsInSetCanMerge(set)` must be true.

`FieldsInSetCanMerge(set)` :

1. Let *fieldsForName* be the set of selections with a given response name in *set* including visiting fragments and inline fragments.
2. Given each pair of members *fieldA* and *fieldB* in *fieldsForName*:
 - a. `SameResponseShape(fieldA, fieldB)` must be true.
 - b. If the parent types of *fieldA* and *fieldB* are equal or if either is not an Object Type:
 - i. *fieldA* and *fieldB* must have identical field names.
 - ii. *fieldA* and *fieldB* must have identical sets of arguments.
 - iii. Let *mergedSet* be the result of adding the selection set of *fieldA* and the selection set of *fieldB*.
 - iv. `FieldsInSetCanMerge(mergedSet)` must be true.

`SameResponseShape(fieldA, fieldB)` :

1. Let *typeA* be the return type of *fieldA*.
2. Let *typeB* be the return type of *fieldB*.
3. If *typeA* or *typeB* is Non-Null.
 - a. *typeA* and *typeB* must both be Non-Null.
 - b. Let *typeA* be the nullable type of *typeA*
 - c. Let *typeB* be the nullable type of *typeB*
4. If *typeA* or *typeB* is List.
 - a. *typeA* and *typeB* must both be List.
 - b. Let *typeA* be the item type of *typeA*
 - c. Let *typeB* be the item type of *typeB*
 - d. Repeat from step 3.
5. If *typeA* or *typeB* is Scalar or Enum.
 - a. *typeA* and *typeB* must be the same type.
6. Assert: *typeA* and *typeB* are both composite types.
7. Let *mergedSet* be the result of adding the selection set of *fieldA* and the selection set of *fieldB*.
8. Let *fieldsForName* be the set of selections with a given response name in *mergedSet* including visiting fragments and inline fragments.
9. Given each pair of members *subfieldA* and *subfieldB* in *fieldsForName*:
 - a. `SameResponseShape(subfieldA, subfieldB)` must be true.

Explanatory Text

If multiple field selections with the same response names are encountered during execution, the field and arguments to execute and the resulting value should be unambiguous. Therefore any two field selections which might both be encountered for the same object are only valid if they are equivalent.

For simple hand-written GraphQL, this rule is obviously a clear developer error, however nested fragments can make this difficult to detect manually.

The following selections correctly merge:

```
fragment mergeIdenticalFields on Dog {
  name
  name
}
```

```
fragment mergeIdenticalAliasesAndFields on Dog {
  otherName: name
  otherName: name
}
```

The following is not able to merge:

```
fragment conflictingBecauseAlias on Dog {
  name: nickname
  name
}
```

Identical arguments are also merged if they have identical arguments. Both values and variables can be correctly merged.

For example the following correctly merge:

```
fragment mergeIdenticalFieldsWithIdenticalArgs on Dog {
  doesKnowCommand(dogCommand: SIT)
  doesKnowCommand(dogCommand: SIT)
}
```

```
fragment mergeIdenticalFieldsWithIdenticalValues on Dog {
  doesKnowCommand(dogCommand: $dogCommand)
  doesKnowCommand(dogCommand: $dogCommand)
}
```

The following do not correctly merge:



```
fragment conflictingArgsOnValues on Dog {  
  doesKnowCommand(dogCommand: SIT)  
  doesKnowCommand(dogCommand: HEEL)  
}
```

```
fragment conflictingArgsValueAndVar on Dog {  
  doesKnowCommand(dogCommand: SIT)  
  doesKnowCommand(dogCommand: $dogCommand)  
}
```

```
fragment conflictingArgsWithVars on Dog {  
  doesKnowCommand(dogCommand: $varOne)  
  doesKnowCommand(dogCommand: $varTwo)  
}
```

```
fragment differingArgs on Dog {  
  doesKnowCommand(dogCommand: SIT)  
  doesKnowCommand  
}
```

The following fields would not merge together, however both cannot be encountered against the same object, so they are safe:

```
fragment safeDifferingFields on Pet {  
  ... on Dog {  
    volume: barkVolume  
  }  
  ... on Cat {  
    volume: meowVolume  
  }  
}
```

```
fragment safeDifferingArgs on Pet {  
  ... on Dog {  
    doesKnowCommand(dogCommand: SIT)  
  }  
  ... on Cat {  
    doesKnowCommand(catCommand: JUMP)  
  }  
}
```

However, the field responses must be shapes which can be merged. For example, scalar values must not differ. In this example, `someValue` might be a `String` or an `Int`:

```
fragment conflictingDifferingResponses on Pet {
  ... on Dog {
    someValue: nickname
  }
  ... on Cat {
    someValue: meowVolume
  }
}
```

Leaf Field Selections

Formal Specification

- For each *selection* in the document
- Let *selectionType* be the result type of *selection*
- If *selectionType* is a scalar:
 - The subselection set of that selection must be empty
- If *selectionType* is an interface, union, or object
 - The subselection set of that selection must NOT BE empty

Explanatory Text

Field selections on scalars are never allowed: scalars are the leaf nodes of any GraphQL query.

The following is valid.

```
fragment scalarSelection on Dog {
  barkVolume
}
```

The following is invalid.

```
fragment scalarSelectionsNotAllowedOnBoolean on Dog {
  barkVolume {
    sinceWhen
  }
}
```

Conversely the leaf field selections of GraphQL queries must be scalars. Leaf selections on objects, interfaces, and unions without subfields are disallowed.

Let's assume the following additions to the query root type of the schema:



```
extend type QueryRoot {  
  human: Human  
  pet: Pet  
  catOrDog: CatOrDog  
}
```

The following examples are invalid

```
query directQueryOnObjectWithoutSubFields {  
  human  
}
```

```
query directQueryOnInterfaceWithoutSubFields {  
  pet  
}
```

```
query directQueryOnUnionWithoutSubFields {  
  catOrDog  
}
```

Arguments

Arguments are provided to both fields and directives. The following validation rules apply in both cases.

Argument Names

Formal Specification

- For each *argument* in the document
- Let *argumentName* be the Name of *argument*.
- Let *argumentDefinition* be the argument definition provided by the parent field or definition named *argumentName*.
- *argumentDefinition* must exist.

Explanatory Text

Every argument provided to a field or directive must be defined in the set of possible arguments of that field or directive.



For example the following are valid:

```
fragment argOnRequiredArg on Dog {  
  doesKnowCommand(dogCommand: SIT)  
}
```

```
fragment argOnOptional on Dog {  
  isHousetrained(atOtherHomes: true) @include(if: true)  
}
```

the following is invalid since `command` is not defined on `DogCommand`.

```
fragment invalidArgName on Dog {  
  doesKnowCommand(command: CLEAN_UP_HOUSE)  
}
```

and this is also invalid as `unless` is not defined on `@include`.

```
fragment invalidArgName on Dog {  
  isHousetrained(atOtherHomes: true) @include(unless: false)  
}
```

In order to explore more complicated argument examples, let's add the following to our type system:

```
type Arguments {  
  multipleReqs(x: Int!, y: Int!): Int!  
  booleanArgField(booleanArg: Boolean): Boolean  
  floatArgField(floatArg: Float): Float  
  intArgField(intArg: Int): Int  
  nonNullBooleanArgField(nonNullBooleanArg: Boolean!): Boolean!  
  booleanListArgField(booleanListArg: [Boolean]!): [Boolean]  
}  
  
extend type QueryRoot {  
  arguments: Arguments  
}
```

Order does not matter in arguments. Therefore both the following example are valid.

```
fragment multipleArgs on Arguments {  
  multipleReqs(x: 1, y: 2)
```

```
}
```



```
fragment multipleArgsReverseOrder on Arguments {  
  multipleReqs(y: 1, x: 2)  
}
```

Argument Uniqueness

Fields and directives treat arguments as a mapping of argument name to value. More than one argument with the same name in an argument set is ambiguous and invalid.

Formal Specification

- For each *argument* in the Document.
- Let *argumentName* be the Name of *argument*.
- Let *arguments* be all Arguments named *argumentName* in the Argument Set which contains *argument*.
- *arguments* must be the set containing only *argument*.

Argument Values Type Correctness

Compatible Values

Formal Specification

- For each *argument* in the document
- Let *value* be the Value of *argument*
- If *value* is not a Variable
 - Let *argumentName* be the Name of *argument*.
 - Let *argumentDefinition* be the argument definition provided by the parent field or definition named *argumentName*.
 - Let *type* be the type expected by *argumentDefinition*.
 - The type of *literalArgument* must be coercible to *type*.

Explanatory Text

Literal values must be compatible with the type defined by the argument they are being provided to, as per the coercion rules defined in the Type System chapter.

For example, an Int can be coerced into a Float.



```
fragment goodBooleanArg on Arguments {
  booleanArgField(booleanArg: true)
}
```

```
fragment coercedIntIntoFloatArg on Arguments {
  floatArgField(floatArg: 1)
}
```

An incoercible conversion, is string to int. Therefore, the following example is invalid.

```
fragment stringIntoInt on Arguments {
  intArgField(intArg: "3")
}
```

Required Non-Null Arguments

- For each Field or Directive in the document.
- Let *arguments* be the arguments provided by the Field or Directive.
- Let *argumentDefinitions* be the set of argument definitions of that Field or Directive.
- For each *definition* in *argumentDefinitions*:
 - Let *type* be the expected type of *definition*.
 - If *type* is Non-Null:
 - Let *argumentName* be the name of *definition*.
 - Let *argument* be the argument in *arguments* named *argumentName*
 - *argument* must exist.
 - Let *value* be the value of *argument*.
 - *value* must not be the **null** literal.

Explanatory Text

Arguments can be required. Arguments are required if the type of the argument is non-null. If it is not non-null, the argument is optional. When an argument type is non-null, and is required, the explicit value **null** may also not be provided.

For example the following are valid:

```
fragment goodBooleanArg on Arguments {
  booleanArgField(booleanArg: true)
}
```

```
fragment goodNonNullArg on Arguments {
  nonNullBooleanArgField(nonNullBooleanArg: true)
}
```



The argument can be omitted from a field with a nullable argument.

Therefore the following query is valid:

```
fragment goodBooleanArgDefault on Arguments {  
  booleanArgField  
}
```

but this is not valid on a non-null argument.

```
fragment missingRequiredArg on Arguments {  
  nonNullBooleanArgField  
}
```

Providing the explicit value **null** is also not valid.

```
fragment missingRequiredArg on Arguments {  
  notNullBooleanArgField(nonNullBooleanArg: null)  
}
```

Fragments

Fragment Declarations

Fragment Name Uniqueness

Formal Specification

- For each fragment definition *fragment* in the document
- Let *fragmentName* be the name of *fragment*.
- Let *fragments* be all fragment definitions in the document named *fragmentName*.
- *fragments* must be a set of one.

Explanatory Text

Fragment definitions are referenced in fragment spreads by name. To avoid ambiguity, each fragment's name must be unique within a document.

Inline fragments are not considered fragment definitions, and are unaffected by this validation rule.



For example the following document is valid:

```
{
  dog {
    ...fragmentOne
    ...fragmentTwo
  }
}

fragment fragmentOne on Dog {
  name
}

fragment fragmentTwo on Dog {
  owner {
    name
  }
}
```

While this document is invalid:

```
{
  dog {
    ...fragmentOne
  }
}

fragment fragmentOne on Dog {
  name
}

fragment fragmentOne on Dog {
  owner {
    name
  }
}
```

Fragment Spread Type Existence

Formal Specification

- For each named spread *namedSpread* in the document



- Let *fragment* be the target of *namedSpread*
- The target type of *fragment* must be defined in the schema

Explanatory Text

Fragments must be specified on types that exist in the schema. This applies for both named and inline fragments. If they are not defined in the schema, the query does not validate.

For example the following fragments are valid:

```
fragment correctType on Dog {  
  name  
}
```

```
fragment inlineFragment on Dog {  
  ... on Dog {  
    name  
  }  
}
```

```
fragment inlineFragment2 on Dog {  
  ... @include(if: true) {  
    name  
  }  
}
```

and the following do not validate:

```
fragment notOnExistingType on NotInSchema {  
  name  
}
```

```
fragment inlineNotExistingType on Dog {  
  ... on NotInSchema {  
    name  
  }  
}
```

Fragments On Composite Types

Formal Specification

- For each *fragment* defined in the document.

- The target type of fragment must have kind *UNION*, *INTERFACE*, or *OBJECT*.



Explanatory Text

Fragments can only be declared on unions, interfaces, and objects. They are invalid on scalars. They can only be applied on non-leaf fields. This rule applies to both inline and named fragments.

The following fragment declarations are valid:

```
fragment fragOnObject on Dog {  
  name  
}
```

```
fragment fragOnInterface on Pet {  
  name  
}
```

```
fragment fragOnUnion on CatOrDog {  
  ... on Dog {  
    name  
  }  
}
```

and the following are invalid:

```
fragment fragOnScalar on Int {  
  something  
}
```

```
fragment inlineFragOnScalar on Dog {  
  ... on Boolean {  
    somethingElse  
  }  
}
```

Fragments Must Be Used

Formal Specification

- For each *fragment* defined in the document.
- *fragment* must be the target of at least one spread in the document

Explanatory Text



Defined fragments must be used within a query document.

For example the following is an invalid query document:

```
fragment nameFragment on Dog { # unused
  name
}

{
  dog {
    name
  }
}
```

Fragment Spreads

Field selection is also determined by spreading fragments into one another. The selection set of the target fragment is unioned with the selection set at the level at which the target fragment is referenced.

Fragment spread target defined

Formal Specification

- For every *namedSpread* in the document.
- Let *fragment* be the target of *namedSpread*
- *fragment* must be defined in the document

Explanatory Text

Named fragment spreads must refer to fragments defined within the document. If the target of a spread is not defined, this is an error:

```
{
  dog {
    ...undefinedFragment
  }
}
```

Fragment spreads must not form cycles

Formal Specification



- For each *fragmentDefinition* in the document
- Let *visited* be the empty set.
- DetectCycles(*fragmentDefinition*, *visited*)

DetectCycles(*fragmentDefinition*, *visited*) :

- Let *spreads* be all fragment spread descendants of *fragmentDefinition*
- For each *spread* in *spreads*
 - *visited* must not contain *spread*
 - Let *nextVisited* be the set including *spread* and members of *visited*
 - Let *nextFragmentDefinition* be the target of *spread*
 - DetectCycles(*nextFragmentDefinition*, *nextVisited*)

Explanatory Text

The graph of fragment spreads must not form any cycles including spreading itself. Otherwise an operation could infinitely spread or infinitely execute on cycles in the underlying data.

This invalidates fragments that would result in an infinite spread:

```
{
  dog {
    ...nameFragment
  }
}
```

```
fragment nameFragment on Dog {
  name
  ...barkVolumeFragment
}
```

```
fragment barkVolumeFragment on Dog {
  barkVolume
  ...nameFragment
}
```

If the above fragments were inlined, this would result in the infinitely large:

```
{
  dog {
    name
    barkVolume
    name
  }
}
```



```

    barkVolume
    name
    barkVolume
    name
    # forever...
  }
}

```

This also invalidates fragments that would result in an infinite recursion when executed against cyclic data:

```

{
  dog {
    ...dogFragment
  }
}

```

```

fragment dogFragment on Dog {
  name
  owner {
    ...ownerFragment
  }
}

```

```

fragment ownerFragment on Dog {
  name
  pets {
    ...dogFragment
  }
}

```

Fragment spread is possible

Formal Specification

- For each *spread* (named or inline) defined in the document.
- Let *fragment* be the target of *spread*
- Let *fragmentType* be the type condition of *fragment*
- Let *parentType* be the type of the selection set containing *spread*
- Let *applicableTypes* be the intersection of `GetPossibleTypes(fragmentType)` and `GetPossibleTypes(parentType)`
- *applicableTypes* must not be empty.

GetPossibleTypes(*type*) :



1. If *type* is an object type, return a set containing *type*
2. If *type* is an interface type, return the set of types implementing *type*
3. If *type* is a union type, return the set of possible types of *type*

Explanatory Text

Fragments are declared on a type and will only apply when the runtime object type matches the type condition. They also are spread within the context of a parent type. A fragment spread is only valid if its type condition could ever apply within the parent type.

Object Spreads In Object Scope

In the scope of an object type, the only valid object type fragment spread is one that applies to the same type that is in scope.

For example

```
fragment dogFragment on Dog {  
  ... on Dog {  
    barkVolume  
  }  
}
```

and the following is invalid

```
fragment catInDogFragmentInvalid on Dog {  
  ... on Cat {  
    meowVolume  
  }  
}
```

Abstract Spreads in Object Scope

In scope of an object type, unions or interface spreads can be used if the object type implements the interface or is a member of the union.

For example

```
fragment petNameFragment on Pet {  
  name  
}
```



```
fragment interfaceWithinObjectFragment on Dog {  
  ...petNameFragment  
}
```

is valid because *Dog* implements *Pet*.

Likewise

```
fragment catOrDogNameFragment on CatOrDog {  
  ... on Cat {  
    meowVolume  
  }  
}
```

```
fragment unionWithObjectFragment on Dog {  
  ...catOrDogNameFragment  
}
```

is valid because *Dog* is a member of the *CatOrDog* union. It is worth noting that if one inspected the contents of the *CatOrDogNameFragment* you could note that no valid results would ever be returned. However we do not specify this as invalid because we only consider the fragment declaration, not its body.

Object Spreads In Abstract Scope

Union or interface spreads can be used within the context of an object type fragment, but only if the object type is one of the possible types of that interface or union.

For example, the following fragments are valid:

```
fragment petFragment on Pet {  
  name  
  ... on Dog {  
    barkVolume  
  }  
}
```

```
fragment catOrDogFragment on CatOrDog {  
  ... on Cat {  
    meowVolume  
  }  
}
```




petFragment is valid because *Dog* implements the interface *Pet*. *catOrDogFragment* is valid because *Cat* is a member of the *CatOrDog* union.

By contrast the following fragments are invalid:

```
fragment sentientFragment on Sentient {
  ... on Dog {
    barkVolume
  }
}
```

```
fragment humanOrAlienFragment on HumanOrAlien {
  ... on Cat {
    meowVolume
  }
}
```

Dog does not implement the interface *Sentient* and therefore *sentientFragment* can never return meaningful results. Therefore the fragment is invalid. Likewise *Cat* is not a member of the union *HumanOrAlien*, and it can also never return meaningful results, making it invalid.

Abstract Spreads in Abstract Scope

Union or interfaces fragments can be used within each other. As long as there exists at least *one* object type that exists in the intersection of the possible types of the scope and the spread, the spread is considered valid.

So for example

```
fragment unionWithInterface on Pet {
  ...dogOrHumanFragment
}
```

```
fragment dogOrHumanFragment on DogOrHuman {
  ... on Dog {
    barkVolume
  }
}
```

is consider valid because *Dog* implements interface *Pet* and is a member of *DogOrHuman*.

However



```
fragment nonIntersectingInterfaces on Pet {  
  ...sentientFragment  
}
```

```
fragment sentientFragment on Sentient {  
  name  
}
```

is not valid because there exists no type that implements both *Pet* and *Sentient*.

Values

Input Object Field Uniqueness

Formal Specification

- For each input object value *inputObject* in the document.
- For every *inputField* in *inputObject*
 - Let *name* be the Name of *inputField*.
 - Let *fields* be all Input Object Fields named *name* in *inputObject*.
 - *fields* must be the set containing only *inputField*.

Explanatory Text

Input objects must not contain more than one field of the same name, otherwise an ambiguity would exist which includes an ignored portion of syntax.

For example the following query will not pass validation.

```
{  
  field(arg: { field: true, field: false })  
}
```

Directives

Directives Are Defined

Formal Specification



- For every *directive* in a document.
- Let *directiveName* be the name of *directive*.
- Let *directiveDefinition* be the directive named *directiveName*.
- *directiveDefinition* must exist.

Explanatory Text

GraphQL servers define what directives they support. For each usage of a directive, the directive must be available on that server.

Directives Are In Valid Locations

Formal Specification

- For every *directive* in a document.
- Let *directiveName* be the name of *directive*.
- Let *directiveDefinition* be the directive named *directiveName*.
- Let *locations* be the valid locations for *directiveDefinition*.
- Let *adjacent* be the AST node the directive affects.
- *adjacent* must be represented by an item within *locations*.

Explanatory Text

GraphQL servers define what directives they support and where they support them. For each usage of a directive, the directive must be used in a location that the server has declared support for.

For example the following query will not pass validation because `@skip` does not provide `QUERY` as a valid location.

```
query @skip(if: $foo) {  
  field  
}
```

Directives Are Unique Per Location

Formal Specification

- For every *location* in the document for which Directives can apply:
 - Let *directives* be the set of Directives which apply to *location*.
 - For each *directive* in *directives*:
 - Let *directiveName* be the name of *directive*.
 - Let *namedDirectives* be the set of all Directives named *directiveName* in *directives*.
 - *namedDirectives* must be a set of one.

Explanatory Text



Directives are used to describe some metadata or behavioral change on the definition they apply to. When more than one directive of the same name is used, the expected metadata or behavior becomes ambiguous, therefore only one of each directive is allowed per location.

For example, the following query will not pass validation because `@skip` has been used twice for the same field:

```
query ($foo: Boolean = true, $bar: Boolean = false) {  
  field @skip(if: $foo) @skip(if: $bar)  
}
```

However the following example is valid because `@skip` has been used only once per location, despite being used twice in the query and on the same named field:

```
query ($foo: Boolean = true, $bar: Boolean = false) {  
  field @skip(if: $foo) {  
    subfieldA  
  }  
  field @skip(if: $bar) {  
    subfieldB  
  }  
}
```

Variables

Variable Uniqueness

Formal Specification

- For every *operation* in the document
 - For every *variable* defined on *operation*
 - Let *variableName* be the name of *variable*
 - Let *variables* be the set of all variables named *variableName* on *operation*
 - *variables* must be a set of one

Explanatory Text

If any operation defines more than one variable with the same name, it is ambiguous and invalid. It is invalid even if the type of the duplicate variable is the same.

```

query houseTrainedQuery($atOtherHomes: Boolean, $atOtherHomes: Boolean) {
  dog {
    isHousetrained(atOtherHomes: $atOtherHomes)
  }
}

```



It is valid for multiple operations to define a variable with the same name. If two operations reference the same fragment, it might actually be necessary:

```

query A($atOtherHomes: Boolean) {
  ...HouseTrainedFragment
}

```

```

query B($atOtherHomes: Boolean) {
  ...HouseTrainedFragment
}

```

```

fragment HouseTrainedFragment {
  dog {
    isHousetrained(atOtherHomes: $atOtherHomes)
  }
}

```

Variable Default Values Are Correctly Typed

Formal Specification

- For every *operation* in a document
- For every *variable* on each *operation*
 - Let *variableType* be the type of *variable*
 - If *variableType* is non-null it cannot have a default value
 - If *variable* has a default value it must be of the same type or able to be coerced to *variableType*

Explanatory Text

Variables defined by operations are allowed to define default values if the type of that variable is not non-null.

For example the following query will pass validation.

```

query houseTrainedQuery($atOtherHomes: Boolean = true) {
  dog {

```



```

    isHousetrained(atOtherHomes: $atOtherHomes)
  }
}

```

However if the variable is defined as non-null, default values are unreachable. Therefore queries such as the following fail validation

```

query houseTrainedQuery($atOtherHomes: Boolean! = true) {
  dog {
    isHousetrained(atOtherHomes: $atOtherHomes)
  }
}

```

Default values must be compatible with the types of variables. Types must match or they must be coercible to the type.

Non-matching types fail, such as in the following example:

```

query houseTrainedQuery($atOtherHomes: Boolean = "true") {
  dog {
    isHousetrained(atOtherHomes: $atOtherHomes)
  }
}

```

However if a type is coercible the query will pass validation.

For example:

```

query intToFloatQuery($floatVar: Float = 1) {
  arguments {
    floatArgField(floatArg: $floatVar)
  }
}

```

Variables Are Input Types

Formal Specification

- For every *operation* in a *document*
- For every *variable* on each *operation*
 - Let *variableType* be the type of *variable*
 - While *variableType* is *LIST* or *NON_NULL*
 - Let *variableType* be the referenced type of *variableType*

- *variableType* must be of kind *SCALAR*, *ENUM* or *INPUT_OBJECT*



Explanatory Text

Variables can only be scalars, enums, input objects, or lists and non-null variants of those types. These are known as input types. Objects, unions, and interfaces cannot be used as inputs.

For these examples, consider the following typesystem additions:

```
input ComplexInput { name: String, owner: String }
```

```
extend type QueryRoot {  
  findDog(complex: ComplexInput): Dog  
  booleanList(booleanListArg: [Boolean!]): Boolean  
}
```

The following queries are valid:

```
query takesBoolean($atOtherHomes: Boolean) {  
  dog {  
    isHousetrained(atOtherHomes: $atOtherHomes)  
  }  
}
```

```
query takesComplexInput($complexInput: ComplexInput) {  
  findDog(complex: $complexInput) {  
    name  
  }  
}
```

```
query TakesListOfBooleanBang($booleans: [Boolean!]) {  
  booleanList(booleanListArg: $booleans)  
}
```

The following queries are invalid:

```
query takesCat($cat: Cat) {  
  # ...  
}
```

```
query takesDogBang($dog: Dog!) {  
  # ...  
}
```



```
query takesListOfPet($pets: [Pet]) {
  # ...
}
```

```
query takesCatOrDog($catOrDog: CatOrDog) {
  # ...
}
```

All Variable Uses Defined

Formal Specification

- For each *operation* in a document
 - For each *variableUsage* in scope, variable must be in *operation*'s variable list.
 - Let *fragments* be every fragment referenced by that *operation* transitively
 - For each *fragment* in *fragments*
 - For each *variableUsage* in scope of *fragment*, variable must be in *operation*'s variable list.

Explanatory Text

Variables are scoped on a per-operation basis. That means that any variable used within the context of an operation must be defined at the top level of that operation

For example:

```
query variableIsDefined($atOtherHomes: Boolean) {
  dog {
    isHousetrained(atOtherHomes: $atOtherHomes)
  }
}
```

is valid. *\$atOtherHomes* is defined by the operation.

By contrast the following query is invalid:

```
query variableIsNotDefined {
  dog {
    isHousetrained(atOtherHomes: $atOtherHomes)
  }
}
```

\$atOtherHomes is not defined by the operation.

Fragments complicate this rule. Any fragment transitively included by an operation has ^{access} to the variables defined by that operation. Fragments can appear within multiple operations and therefore variable usages must correspond to variable definitions in all of those operations.

For example the following is valid:

```
query variableIsDefinedUsedInSingleFragment($atOtherHomes: Boolean) {
  dog {
    ...isHousetrainedFragment
  }
}
```

```
fragment isHousetrainedFragment on Dog {
  isHousetrained(atOtherHomes: $atOtherHomes)
}
```

since *isHousetrainedFragment* is used within the context of the operation *variableIsDefinedUsedInSingleFragment* and the variable is defined by that operation.

On the other hand, if a fragment is included within an operation that does not define a referenced variable, the query is invalid.

```
query variableIsNotDefinedUsedInSingleFragment {
  dog {
    ...isHousetrainedFragment
  }
}
```

```
fragment isHousetrainedFragment on Dog {
  isHousetrained(atOtherHomes: $atOtherHomes)
}
```

This applies transitively as well, so the following also fails:

```
query variableIsNotDefinedUsedInNestedFragment {
  dog {
    ...outerHousetrainedFragment
  }
}
```

```
fragment outerHousetrainedFragment on Dog {
  ...isHousetrainedFragment
}
```



```
fragment isHousetrainedFragment on Dog {  
  isHousetrained(atOtherHomes: $atOtherHomes)  
}
```

Variables must be defined in all operations in which a fragment is used.

```
query housetrainedQueryOne($atOtherHomes: Boolean) {  
  dog {  
    ...isHousetrainedFragment  
  }  
}
```

```
query housetrainedQueryTwo($atOtherHomes: Boolean) {  
  dog {  
    ...isHousetrainedFragment  
  }  
}
```

```
fragment isHousetrainedFragment on Dog {  
  isHousetrained(atOtherHomes: $atOtherHomes)  
}
```

However the following does not validate:

```
query housetrainedQueryOne($atOtherHomes: Boolean) {  
  dog {  
    ...isHousetrainedFragment  
  }  
}
```

```
query housetrainedQueryTwoNotDefined {  
  dog {  
    ...isHousetrainedFragment  
  }  
}
```

```
fragment isHousetrainedFragment on Dog {  
  isHousetrained(atOtherHomes: $atOtherHomes)  
}
```

This is because *housetrainedQueryTwoNotDefined* does not define a variable *\$atOtherHomes* but that variable is used by *isHousetrainedFragment* which is included in that operation.

All Variables Used



Formal Specification

- For every *operation* in the document.
- Let *variables* be the variables defined by that *operation*
- Each *variable* in *variables* must be used at least once in either the operation scope itself or any fragment transitively referenced by that operation.

Explanatory Text

All variables defined by an operation must be used in that operation or a fragment transitively included by that operation. Unused variables cause a validation error.

For example the following is invalid:

```
query variableUnused($atOtherHomes: Boolean) {  
  dog {  
    isHousetrained  
  }  
}
```

because *\$atOtherHomes* is not referenced.

These rules apply to transitive fragment spreads as well:

```
query variableUsedInFragment($atOtherHomes: Boolean) {  
  dog {  
    ...isHousetrainedFragment  
  }  
}
```

```
fragment isHousetrainedFragment on Dog {  
  isHousetrained(atOtherHomes: $atOtherHomes)  
}
```

The above is valid since *\$atOtherHomes* is used in *isHousetrainedFragment* which is included by *variableUsedInFragment*.

If that fragment did not have a reference to *\$atOtherHomes* it would be not valid:

```
query variableNotUsedWithinFragment($atOtherHomes: Boolean) {  
  ...isHousetrainedWithoutVariableFragment  
}
```



```
fragment isHousetrainedWithoutVariableFragment on Dog {
  isHousetrained
}
```

All operations in a document must use all of their variables.

As a result, the following document does not validate.

```
query queryWithUsedVar($atOtherHomes: Boolean) {
  dog {
    ...isHousetrainedFragment
  }
}
```

```
query queryWithExtraVar($atOtherHomes: Boolean, $extra: Int) {
  dog {
    ...isHousetrainedFragment
  }
}
```

```
fragment isHousetrainedFragment on Dog {
  isHousetrained(atOtherHomes: $atOtherHomes)
}
```

This document is not valid because *queryWithExtraVar* defines an extraneous variable.

All Variable Usages are Allowed

Formal Specification

- For each *operation* in *document*
- Let *variableUsages* be all usages transitively included in the *operation*
- For each *variableUsage* in *variableUsages*
 - Let *variableType* be the type of variable definition in the *operation*
 - Let *argumentType* be the type of the argument the variable is passed to.
 - Let *hasDefault* be true if the variable definition defines a default.
 - *AreTypesCompatible(argumentType, variableType, hasDefault)* must be true
- *AreTypesCompatible(argumentType, variableType, hasDefault)*:
 - If *hasDefault* is true, treat the *variableType* as non-null.
 - If inner type of *argumentType* and *variableType* are different, return false
 - If *argumentType* and *variableType* have different list dimensions, return false
 - If any list level of *variableType* is not non-null, and the corresponding level in *argument* is non-null, the types are not compatible.

Explanatory Text



Variable usages must be compatible with the arguments they are passed to.

Validation failures occur when variables are used in the context of types that are complete mismatches, or if a nullable type in a variable is passed to a non-null argument type.

Types must match:

```
query intCannotGoIntoBoolean($intArg: Int) {  
  arguments {  
    booleanArgField(booleanArg: $intArg)  
  }  
}
```

\$intArg typed as *Int* cannot be used as a argument to *booleanArg*, typed as *Boolean*.

List cardinality must also be the same. For example, lists cannot be passed into singular values.

```
query booleanListCannotGoIntoBoolean($booleanListArg: [Boolean]) {  
  arguments {  
    booleanArgField(booleanArg: $booleanListArg)  
  }  
}
```

Nullability must also be respected. In general a nullable variable cannot be passed to a non-null argument.

```
query booleanArgQuery($booleanArg: Boolean) {  
  arguments {  
    nonNullBooleanArgField(nonNullBooleanArg: $booleanArg)  
  }  
}
```

A notable exception is when default arguments are provided. They are, in effect, treated as non-nulls.

```
query booleanArgQueryWithDefault($booleanArg: Boolean = true) {  
  arguments {  
    nonNullBooleanArgField(nonNullBooleanArg: $booleanArg)  
  }  
}
```

For list types, the same rules around nullability apply to both outer types and inner types. A nullable list cannot be passed to a non-null list, and a list of nullable values cannot be passed to a list of non-null values. The following is valid:

```
query nonNullListToList($nonNullBooleanList: [Boolean]!) {
  arguments {
    booleanListArgField(booleanListArg: $nonNullBooleanList)
  }
}
```

However, a nullable list cannot be passed to a non-null list:

```
query listToNonNullList($booleanList: [Boolean]) {
  arguments {
    nonNullBooleanListField(nonNullBooleanListArg: $booleanList)
  }
}
```

This would fail validation because a `[T]` cannot be passed to a `[T]!`.

Similarly a `[T]` cannot be passed to a `[T!]`.

Execution

GraphQL generates a response from a request via execution.

A request for execution consists of a few pieces of information:

- The schema to use, typically solely provided by the GraphQL service.
- A Document containing GraphQL Operations and Fragments to execute.
- Optionally: The name of the Operation in the Document to execute.
- Optionally: Values for any Variables defined by the Operation.
- An initial value corresponding to the root type being executed. Conceptually, an initial value represents the “universe” of data available via a GraphQL Service. It is common for a GraphQL Service to always use the same initial value for every request.

Given this information, the result of `ExecuteRequest()` produces the response, to be formatted according to the Response section below.

Executing Requests



To execute a request, the executor must have a parsed Document (as defined in the “Query Language” part of this spec) and a selected operation name to run if the document defines multiple operations, otherwise the document is expected to only contain a single operation. The result of the request is determined by the result of executing this operation according to the “Executing Operations” section below.

`ExecuteRequest(schema, document, operationName, variableValues, initialValue) :`

1. Let *operation* be the result of `GetOperation(document, operationName)`.
2. Let *coercedVariableValues* be the result of `CoerceVariableValues(schema, operation, variableValues)`.
3. If *operation* is a query operation:
 - a. Return `ExecuteQuery(operation, schema, coercedVariableValues, initialValue)`.
4. Otherwise if *operation* is a mutation operation:
 - a. Return `ExecuteMutation(operation, schema, coercedVariableValues, initialValue)`.

`GetOperation(document, operationName) :`

1. If *operationName* is **null**:
 - a. If *document* contains exactly one operation.
 - i. Return the Operation contained in the *document*.
 - b. Otherwise produce a query error requiring *operationName*.
2. Otherwise:
 - a. Let *operation* be the Operation named *operationName* in *document*.
 - b. If *operation* was not found, produce a query error.
 - c. Return *operation*.

Validating Requests

As explained in the Validation section, only requests which pass all validation rules should be executed. If validation errors are known, they should be reported in the list of “errors” in the response and the request must fail without execution.

Typically validation is performed in the context of a request immediately before execution, however a GraphQL service may execute a request without immediately validating it if that exact same request is known to have been validated before. A GraphQL service should only execute requests which *at some point* were known to be free of any validation errors, and have since not changed.

For example: the request may be validated during development, provided it does not later change, or a service may validate a request once and memoize the result to avoid validating the same request again in the future.

Coercing Variable Values



If the operation has defined any variables, then the values for those variables need to be coerced using the input coercion rules of variable's declared type. If a query error is encountered during input coercion of variable values, then the operation fails without execution.

CoerceVariableValues(*schema*, *operation*, *variableValues*) :

1. Let *coercedValues* be an empty unordered Map.
2. Let *variableDefinitions* be the variables defined by *operation*.
3. For each *variableDefinition* in *variableDefinitions*:
 - a. Let *variableName* be the name of *variableDefinition*.
 - b. Let *variableType* be the expected type of *variableDefinition*.
 - c. Let *defaultValue* be the default value for *variableDefinition*.
 - d. Let *value* be the value provided in *variableValues* for the name *variableName*.
 - e. If *value* does not exist (was not provided in *variableValues*):
 - i. If *defaultValue* exists (including **null**):
 1. Add an entry to *coercedValues* named *variableName* with the value *defaultValue*.
 - ii. Otherwise if *variableType* is a Non-Nullable type, throw a query error.
 - iii. Otherwise, continue to the next variable definition.
 - f. Otherwise, if *value* cannot be coerced according to the input coercion rules of *variableType*, throw a query error.
 - g. Let *coercedValue* be the result of coercing *value* according to the input coercion rules of *variableType*.
 - h. Add an entry to *coercedValues* named *variableName* with the value *coercedValue*.
4. Return *coercedValues*.

Note

This algorithm is very similar to **CoerceArgumentValues**().

Executing Operations

The type system, as described in the “Type System” section of the spec, must provide a query root object type. If mutations are supported, it must also provide a mutation root object type.

If the operation is a query, the result of the operation is the result of executing the query's top level selection set with the query root object type.

An initial value may be provided when executing a query.

ExecuteQuery(*query*, *schema*, *variableValues*, *initialValue*) :



1. Let *queryType* be the root Query type in *schema*.
2. Assert: *queryType* is an Object type.
3. Let *selectionSet* be the top level Selection Set in *query*.
4. Let *data* be the result of running `ExecuteSelectionSet(selectionSet, queryType, initialValue, variableValues)` normally (allowing parallelization).
5. Let *errors* be any *field errors* produced while executing the selection set.
6. Return an unordered map containing *data* and *errors*.

If the operation is a mutation, the result of the operation is the result of executing the mutation's top level selection set on the mutation root object type. This selection set should be executed serially.

It is expected that the top level fields in a mutation operation perform side-effects on the underlying data system. Serial execution of the provided mutations ensures against race conditions during these side-effects.

`ExecuteMutation(mutation, schema, variableValues, initialValue) :`

1. Let *mutationType* be the root Mutation type in *schema*.
2. Assert: *mutationType* is an Object type.
3. Let *selectionSet* be the top level Selection Set in *mutation*.
4. Let *data* be the result of running `ExecuteSelectionSet(selectionSet, mutationType, initialValue, variableValues)` serially.
5. Let *errors* be any *field errors* produced while executing the selection set.
6. Return an unordered map containing *data* and *errors*.

Executing Selection Sets

To execute a selection set, the object value being evaluated and the object type need to be known, as well as whether it must be executed serially, or may be executed in parallel.

First, the selection set is turned into a grouped field set; then, each represented field in the grouped field set produces an entry into a response map.

`ExecuteSelectionSet(selectionSet, objectType, objectValue, variableValues) :`

1. Let *groupedFieldSet* be the result of `CollectFields(objectType, selectionSet, variableValues)`.
2. Initialize *resultMap* to an empty ordered map.
3. For each *groupedFieldSet* as *responseKey* and *fields*:
 - a. Let *fieldName* be the name of the first entry in *fields*. Note: This value is unaffected if an alias is used.
 - b. Let *fieldType* be the return type defined for the field *fieldName* of *objectType*.
 - c. If *fieldType* is **null**:
 - i. Continue to the next iteration of *groupedFieldSet*.



d. Let *responseValue* be `ExecuteField(objectType, objectValue, fields, fieldType, variableValues)`.

e. Set *responseValue* as the value for *responseKey* in *resultMap*.

4. Return *resultMap*.

Note

responseMap is ordered by which fields appear first in the query. This is explained in greater detail in the Field Collection section below.

Normal and Serial Execution

Normally the executor can execute the entries in a grouped field set in whatever order it chooses (often in parallel). Because the resolution of fields other than top-level mutation fields must always be side effect-free and idempotent, the execution order must not affect the result, and hence the server has the freedom to execute the field entries in whatever order it deems optimal.

For example, given the following grouped field set to be executed normally:

```
{
  birthday {
    month
  }
  address {
    street
  }
}
```

A valid GraphQL executor can resolve the four fields in whatever order it chose (however of course *birthday* must be resolved before *month*, and *address* before *street*).

When executing a mutation, the selections in the top most selection set will be executed in serial order.

When executing a grouped field set serially, the executor must consider each entry from the grouped field set in the order provided in the grouped field set. It must determine the corresponding entry in the result map for each item to completion before it continues on to the next item in the grouped field set:

For example, given the following selection set to be executed serially:

```
{
  changeBirthday(birthday: $newBirthday) {
```



```

    month
  }
  changeAddress(address: $newAddress) {
    street
  }
}

```

The executor must, in serial:

- Run `ExecuteField()` for `changeBirthday`, which during `CompleteValue()` will execute the `{ month }` sub-selection set normally.
- Run `ExecuteField()` for `changeAddress`, which during `CompleteValue()` will execute the `{ street }` sub-selection set normally.

As an illustrative example, let's assume we have a mutation field `changeTheNumber` that returns an object containing one field, `theNumber`. If we execute the following selection set serially:

```

{
  first: changeTheNumber(newNumber: 1) {
    theNumber
  }
  second: changeTheNumber(newNumber: 3) {
    theNumber
  }
  third: changeTheNumber(newNumber: 2) {
    theNumber
  }
}

```

The executor will execute the following serially:

- Resolve the `changeTheNumber(newNumber: 1)` field
- Execute the `{ theNumber }` sub-selection set of `first` normally
- Resolve the `changeTheNumber(newNumber: 3)` field
- Execute the `{ theNumber }` sub-selection set of `second` normally
- Resolve the `changeTheNumber(newNumber: 2)` field
- Execute the `{ theNumber }` sub-selection set of `third` normally

A correct executor must generate the following result for that selection set:

```

{
  "first": {
    "theNumber": 1
  },

```



```
"second": {
  "theNumber": 3
},
"third": {
  "theNumber": 2
}
}
```

Field Collection

Before execution, the selection set is converted to a grouped field set by calling `CollectFields()`. Each entry in the grouped field set is a list of fields that share a response key. This ensures all fields with the same response key (alias or field name) included via referenced fragments are executed at the same time.

As an example, collecting the fields of this selection set would collect two instances of the field `a` and one of field `b`:

```
{
  a {
    subfield1
  }
  ...ExampleFragment
}
```

```
fragment ExampleFragment on Query {
  a {
    subfield2
  }
  b
}
```

The depth-first-search order of the field groups produced by `CollectFields()` is maintained through execution, ensuring that fields appear in the executed response in a stable and predictable order.

`CollectFields(objectType, selectionSet, variableValues, visitedFragments) :`

1. If *visitedFragments* if not provided, initialize it to the empty set.
2. Initialize *groupedFields* to an empty ordered map of lists.
3. For each *selection* in *selectionSet*:
 - a. If *selection* provides the directive `@skip`, let *skipDirective* be that directive.

- i. If *skipDirective*'s *if* argument is **true** or is a variable in *variableValues* with the value **true**, continue with the next *selection* in *selectionSet*.
- b. If *selection* provides the directive `@include`, let *includeDirective* be that directive.
 - i. If *includeDirective*'s *if* argument is not **true** and is not a variable in *variableValues* with the value **true**, continue with the next *selection* in *selectionSet*.
- c. If *selection* is a *Field*:
 - i. Let *responseKey* be the response key of *selection*.
 - ii. Let *groupForResponseKey* be the list in *groupedFields* for *responseKey*; if no such list exists, create it as an empty list.
 - iii. Append *selection* to the *groupForResponseKey*.
- d. If *selection* is a *FragmentSpread*:
 - i. Let *fragmentSpreadName* be the name of *selection*.
 - ii. If *fragmentSpreadName* is in *visitedFragments*, continue with the next *selection* in *selectionSet*.
 - iii. Add *fragmentSpreadName* to *visitedFragments*.
 - iv. Let *fragment* be the *Fragment* in the current *Document* whose name is *fragmentSpreadName*.
 - v. If no such *fragment* exists, continue with the next *selection* in *selectionSet*.
 - vi. Let *fragmentType* be the type condition on *fragment*.
 - vii. If `DoesFragmentTypeApply(objectType, fragmentType)` is false, continue with the next *selection* in *selectionSet*.
 - viii. Let *fragmentSelectionSet* be the top-level selection set of *fragment*.
 - ix. Let *fragmentGroupedFieldSet* be the result of calling `CollectFields(objectType, fragmentSelectionSet, visitedFragments)`.
 - x. For each *fragmentGroup* in *fragmentGroupedFieldSet*:
 1. Let *responseKey* be the response key shared by all fields in *fragmentGroup*
 2. Let *groupForResponseKey* be the list in *groupedFields* for *responseKey*; if no such list exists, create it as an empty list.
 3. Append all items in *fragmentGroup* to *groupForResponseKey*.
- e. If *selection* is an *InlineFragment*:
 - i. Let *fragmentType* be the type condition on *selection*.
 - ii. If *fragmentType* is not **null** and `DoesFragmentTypeApply(objectType, fragmentType)` is false, continue with the next *selection* in *selectionSet*.
 - iii. Let *fragmentSelectionSet* be the top-level selection set of *selection*.
 - iv. Let *fragmentGroupedFieldSet* be the result of calling `CollectFields(objectType, fragmentSelectionSet, variableValues, visitedFragments)`.
 - v. For each *fragmentGroup* in *fragmentGroupedFieldSet*:
 1. Let *responseKey* be the response key shared by all fields in *fragmentGroup*
 2. Let *groupForResponseKey* be the list in *groupedFields* for *responseKey*; if no such list exists, create it as an empty list.
 3. Append all items in *fragmentGroup* to *groupForResponseKey*.

4. Return *groupedFields*.



DoesFragmentTypeApply(*objectType*, *fragmentType*) :

1. If *fragmentType* is an Object Type:
 - a. if *objectType* and *fragmentType* are the same type, return **true**, otherwise return **false**.
2. If *fragmentType* is an Interface Type:
 - a. if *objectType* is an implementation of *fragmentType*, return **true** otherwise return **false**.
3. If *fragmentType* is a Union:
 - a. if *objectType* is a possible type of *fragmentType*, return **true** otherwise return **false**.

Executing Fields

Each field requested in the grouped field set that is defined on the selected *objectType* will result in an entry in the response map. Field execution first coerces any provided argument values, then resolves a value for the field, and finally completes that value either by recursively executing another selection set or coercing a scalar value.

ExecuteField(*objectType*, *objectValue*, *fieldType*, *fields*, *variableValues*) :

1. Let *field* be the first entry in *fields*.
2. Let *argumentValues* be the result of CoerceArgumentValues(*objectType*, *field*, *variableValues*)
3. Let *resolvedValue* be ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentValues*).
4. Return the result of CompleteValue(*fieldType*, *fields*, *resolvedValue*, *variableValues*).

Coercing Field Arguments

Fields may include arguments which are provided to the underlying runtime in order to correctly produce a value. These arguments are defined by the field in the type system to have a specific input type: Scalars, Enum, Input Object, or List or Non-Null wrapped variations of these three.

At each argument position in a query may be a literal value or a variable to be provided at runtime.

CoerceArgumentValues(*objectType*, *field*, *variableValues*) :

1. Let *coercedValues* be an empty unordered Map.
2. Let *argumentValues* be the argument values provided in *field*.




3. Let *fieldName* be the name of *field*.
4. Let *argumentDefinitions* be the arguments defined by *objectType* for the field named *fieldName*.
5. For each *argumentDefinition* in *argumentDefinitions*:
 - a. Let *argumentName* be the name of *argumentDefinition*.
 - b. Let *argumentType* be the expected type of *argumentDefinition*.
 - c. Let *defaultValue* be the default value for *argumentDefinition*.
 - d. Let *value* be the value provided in *argumentValues* for the name *argumentName*.
 - e. If *value* is a Variable:
 - i. Let *variableName* be the name of Variable *value*.
 - ii. Let *variableValue* be the value provided in *variableValues* for the name *variableName*.
 - iii. If *variableValue* exists (including **null**):
 1. Add an entry to *coercedValues* named *argName* with the value *variableValue*.
 - iv. Otherwise, if *defaultValue* exists (including **null**):
 1. Add an entry to *coercedValues* named *argName* with the value *defaultValue*.
 - v. Otherwise, if *argumentType* is a Non-Nullable type, throw a field error.
 - vi. Otherwise, continue to the next argument definition.
 - f. Otherwise, if *value* does not exist (was not provided in *argumentValues*):
 - i. If *defaultValue* exists (including **null**):
 1. Add an entry to *coercedValues* named *argName* with the value *defaultValue*.
 - ii. Otherwise, if *argumentType* is a Non-Nullable type, throw a field error.
 - iii. Otherwise, continue to the next argument definition.
 - g. Otherwise, if *value* cannot be coerced according to the input coercion rules of *argType*, throw a field error.
 - h. Let *coercedValue* be the result of coercing *value* according to the input coercion rules of *argType*.
 - i. Add an entry to *coercedValues* named *argName* with the value *coercedValue*.
6. Return *coercedValues*.

Note

Variable values are not coerced because they are expected to be coerced before executing the operation in `CoerceVariableValues()`, and valid queries must only allow usage of variables of appropriate types.

Value Resolution

While nearly all of GraphQL execution can be described generically, ultimately the internal system exposing the GraphQL interface must provide values. This is exposed via `ResolveFieldValue`, which produces a value for a given field on a type for a real value.

As an example, this might accept the *objectType* `Person`, the *field* `"soulMate"`, and the *objectValue* representing John Lennon. It would be expected to yield the value representing  Yoko Ono.

`ResolveFieldValue(objectType, objectValue, fieldName, argumentValues) :`

1. Let *resolver* be the internal function provided by *objectType* for determining the resolved value of a field named *fieldName*.
2. Return the result of calling *resolver*, providing *objectValue* and *argumentValues*.

Note

It is common for *resolver* to be asynchronous due to relying on reading an underlying database or networked service to produce a value. This necessitates the rest of a GraphQL executor to handle an asynchronous execution flow.

Value Completion

After resolving the value for a field, it is completed by ensuring it adheres to the expected return type. If the return type is another Object type, then the field execution process continues recursively.

`CompleteValue(fieldType, fields, result, variableValues) :`

1. If the *fieldType* is a Non-Null type:
 - a. Let *innerType* be the inner type of *fieldType*.
 - b. Let *completedResult* be the result of calling `CompleteValue(innerType, fields, result, variableValues)`.
 - c. If *completedResult* is **null**, throw a field error.
 - d. Return *completedResult*.
2. If *result* is **null** (or another internal value similar to **null** such as **undefined** or *NaN*), return **null**.
3. If *fieldType* is a List type:
 - a. If *result* is not a collection of values, throw a field error.
 - b. Let *innerType* be the inner type of *fieldType*.
 - c. Return a list where each list item is the result of calling `CompleteValue(innerType, fields, resultItem, variableValues)`, where *resultItem* is each item in *result*.
4. If *fieldType* is a Scalar or Enum type:
 - a. Return the result of “coercing” *result*, ensuring it is a legal value of *fieldType*, otherwise **null**.
5. If *fieldType* is an Object, Interface, or Union type:
 - a. If *fieldType* is an Object type.
 - i. Let *objectType* be *fieldType*.
 - b. Otherwise if *fieldType* is an Interface or Union type.
 - i. Let *objectType* be `ResolveAbstractType(fieldType, result)`.

- c. Let *subSelectionSet* be the result of calling `MergeSelectionSets(fields)`.
- d. Return the result of evaluating `ExecuteSelectionSet(subSelectionSet, objectType, result, variableValues)` *normally* (allowing for parallelization).

Resolving Abstract Types

When completing a field with an abstract return type, that is an Interface or Union return type, first the abstract type must be resolved to a relevant Object type. This determination is made by the internal system using whatever means appropriate.

Note

A common method of determining the Object type for an *objectValue* in object-oriented environments, such as Java or C#, is to use the class name of the *objectValue*.

`ResolveAbstractType(abstractType, objectValue) :`

1. Return the result of calling the internal method provided by the type system for determining the Object type of *abstractType* given the value *objectValue*.

Merging Selection Sets

When more than one fields of the same name are executed in parallel, their selection sets are merged together when completing the value in order to continue execution of the sub-selection sets.

An example query illustrating parallel fields with the same name with sub-selections.

```
{
  me {
    firstName
  }
  me {
    lastName
  }
}
```

After resolving the value for *me*, the selection sets are merged together so *firstName* and *lastName* can be resolved for one value.

`MergeSelectionSets(fields) :`

1. Let *selectionSet* be an empty list.
2. For each *field* in *fields*:
 - a. Let *fieldSelectionSet* be the selection set of *field*.
 - b. If *fieldSelectionSet* is null or empty, continue to the next field.
 - c. Append all selections in *fieldSelectionSet* to *selectionSet*.

3. Return *selectionSet*.



Errors and Non-Nullability

If an error is thrown while resolving a field, it should be treated as though the field returned **null**, and an error must be added to the **"errors"** list in the response.

If the result of resolving a field is **null** (either because the function to resolve the field returned **null** or because an error occurred), and that field is of a Non-Null type, then a field error is thrown. The error must be added to the **"errors"** list in the response.

If the field returns **null** because of an error which has already been added to the **"errors"** list in the response, the **"errors"** list must not be further affected. That is, only one error should be added to the errors list per field.

Since Non-Null type fields cannot be **null**, field errors are propagated to be handled by the parent field. If the parent field may be **null** then it resolves to **null**, otherwise if it is a Non-Null type, the field error is further propagated to its parent field.

If all fields from the root of the request to the source of the error return Non-Null types, then the **"data"** entry in the response should be **null**.

Response

When a GraphQL server receives a request, it must return a well-formed response. The server's response describes the result of executing the requested operation if successful, and describes any errors encountered during the request.

A response may contain both a partial response as well as encountered errors in the case that an error occurred on a field which was replaced with null.

Serialization Format

GraphQL does not require a specific serialization format. However, clients should use a serialization format that supports the major primitives in the GraphQL response. In particular, the serialization format must support representations of the following four primitives:

- Map

- List
- String
- Null



Serialization formats which can represent an ordered map should preserve the order of requested fields as defined by `CollectFields()` in the Execution section. Serialization formats which can only represent unordered maps should retain this order grammatically (such as JSON).

Producing a response where fields are represented in the same order in which they appear in the request improves human readability during debugging and enables more efficient parsing of responses if the order of properties can be anticipated.

A serialization format may support the following primitives, however, strings may be used as a substitute for those primitives.

- Boolean
- Int
- Float
- Enum Value

JSON Serialization

JSON is the preferred serialization format for GraphQL, though as noted above, GraphQL does not require a specific serialization format. For consistency and ease of notation, examples of the response are given in JSON throughout the spec. In particular, in our JSON examples, we will represent primitives using the following JSON concepts:

GraphQL Value	JSON Value
Map	Object
List	Array
Null	null
String	String
Boolean	true or false
Int	Number
Float	Number
Enum Value	String

Object Property Ordering

While JSON Objects are specified as an **unordered collection of key-value pairs** the pairs are represented in an ordered manner. In other words, while the JSON strings `{ "name": "Mark", "age": 30 }` and `{ "age": 30, "name": "Mark" }` encode the same value, they also have observably different property orderings.

Since the result of evaluating a selection set is ordered, the JSON object serialized should preserve this order by writing the object properties in the same order as those fields were requested as defined by query execution.

For example, if the query was `{ name, age }`, a GraphQL server responding in JSON should respond with `{ "name": "Mark", "age": 30 }` and should not respond with `{ "age": 30, "name": "Mark" }`.

Note

This does not violate the JSON spec, as clients may still interpret objects in the response as unordered Maps and arrive at a valid value.

Response Format

A response to a GraphQL operation must be a map.

If the operation included execution, the response map must contain a first entry with key `data`. The value of this entry is described in the “Data” section. If the operation failed before execution, due to a syntax error, missing information, or validation error, this entry must not be present.

If the operation encountered any errors, the response map must contain a next entry with key `errors`. The value of this entry is described in the “Errors” section. If the operation completed without encountering any errors, this entry must not be present.

The response map may also contain an entry with key `extensions`. This entry, if set, must have a map as its value. This entry is reserved for implementors to extend the protocol however they see fit, and hence there are no additional restrictions on its contents.

To ensure future changes to the protocol do not break existing servers and clients, the top level response map must not contain any entries other than the three described above.

Data

The `data` entry in the response will be the result of the execution of the requested operation. If the operation was a query, this output will be an object of the schema’s query root type; if the operation was a mutation, this output will be an object of the schema’s mutation root type.

If an error was encountered before execution begins, the `data` entry should not be present in the result.

If an error was encountered during the execution that prevented a valid response, the `data` entry in the response should be `null`.

Errors

The `errors` entry in the response is a non-empty list of errors, where each error is a map.

If no errors were encountered during the requested operation, the `errors` entry should not be present in the result.

Every error must contain an entry with the key `message` with a string description of the error intended for the developer as a guide to understand and correct the error.

If an error can be associated to a particular point in the requested GraphQL document, it should contain an entry with the key `locations` with a list of locations, where each location is a map with the keys `line` and `column`, both positive numbers starting from 1 which describe the beginning of an associated syntax element.

GraphQL servers may provide additional entries to error as they choose to produce more helpful or machine-readable errors, however future versions of the spec may describe additional entries to errors.

If the `data` entry in the response is `null` or not present, the `errors` entry in the response must not be empty. It must contain at least one error. The errors it contains should indicate why no data was able to be returned.

If the `data` entry in the response is not `null`, the `errors` entry in the response may contain any errors that occurred during execution. If errors occurred during execution, it should contain those errors.

Appendix: Notation Conventions

This specification document contains a number of notation conventions used to describe technical concepts such as language grammar and semantics as well as runtime algorithms.

This appendix seeks to explain these notations in greater detail to avoid ambiguity.

Context-Free Grammar



A context-free grammar consists of a number of productions. Each production has an abstract symbol called a “non-terminal” as its left-hand side, and zero or more possible sequences of non-terminal symbols and or terminal characters as its right-hand side.

Starting from a single goal non-terminal symbol, a context-free grammar describes a language: the set of possible sequences of characters that can be described by repeatedly replacing any non-terminal in the goal sequence with one of the sequences it is defined by, until all non-terminal symbols have been replaced by terminal characters.

Terminals are represented in this document in a monospace font in two forms: a specific Unicode character or sequence of Unicode characters (ex. `=` or **terminal**), and a pattern of Unicode characters defined by a regular expression (ex `/[0-9]+/`).

Non-terminal production rules are represented in this document using the following notation for a non-terminal with a single definition:

NonTerminalWithSingleDefinition :
NonTerminal **terminal**

While using the following notation for a production with a list of definitions:

NonTerminalWithManyDefinitions :
OtherNonTerminal **terminal**
terminal

A definition may refer to itself, which describes repetitive sequences, for example:

ListOfLetterA :
a
ListOfLetterA **a**

Lexical and Syntactical Grammar

The GraphQL language is defined in a syntactic grammar where terminal symbols are tokens. Tokens are defined in a lexical grammar which matches patterns of source characters. The result of parsing a sequence of source Unicode characters produces a GraphQL AST.

A Lexical grammar production describes non-terminal “tokens” by patterns of terminal Unicode characters. No “whitespace” or other ignored characters may appear between any terminal Unicode characters in the lexical grammar production. A lexical grammar production is distinguished by a two colon `::` definition.

Word ::

/ [A-Za-z] + /



A Syntactical grammar production describes non-terminal “rules” by patterns of terminal Tokens. Whitespace and other ignored characters may appear before or after any terminal Token. A syntactical grammar production is distinguished by a one colon : definition.

Sentence :

Noun Verb

Grammar Notation

This specification uses some additional notation to describe common patterns, such as optional or repeated patterns, or parameterized alterations of the definition of a non-terminal. This section explains these short-hand notations and their expanded definitions in the context-free grammar.

Constraints

A grammar production may specify that certain expansions are not permitted by using the phrase “but not” and then indicating the expansions to be excluded.

For example, the production:

SafeName :

Name but not *SevenCarlinWords*

means that the nonterminal *SafeName* may be replaced by any sequence of characters that could replace *Name* provided that the same sequence of characters could not replace *SevenCarlinWords*.

A grammar may also list a number of restrictions after “but not” separated by “or”.

For example:

NonBooleanName :

Name but not **true** or **false**

Optionality and Lists

A subscript suffix “*Symbol*_{opt}” is shorthand for two possible sequences, one including that symbol and one excluding it.

As an example:

Sentence :

Noun Verb Adverb_{opt}



is shorthand for

Sentence :

Noun Verb

Noun Verb Adverb

A subscript suffix “*Symbol_{list}*” is shorthand for a list of one or more of that symbol.

As an example:

Book :

Cover Page_{list} Cover

is shorthand for

Book :

Cover Page_list Cover

Page_list :

Page

Page_list Page

Parameterized Grammar Productions

A symbol definition subscript suffix parameter in braces “*Symbol_[Param]*” is shorthand for two symbol definitions, one appended with that parameter name, the other without. The same subscript suffix on a symbol is shorthand for that variant of the definition. If the parameter starts with “?”, that form of the symbol is used if in a symbol definition with the same parameter. Some possible sequences can be included or excluded conditionally when respectively prefixed with “[+Param]” and “[~Param]”.

As an example:

Example_[Param] :

A

B_[Param]

C_[?Param]

[+Param] D

[~Param] E

is shorthand for

Example :

A

B_param

C
E



Example_param :

A

B_param

C_param

D

Grammar Semantics

This specification describes the semantic value of many grammar productions in the form of a list of algorithmic steps.

For example, this describes how a parser should interpret a string literal:

StringValue :: ""

1. Return an empty Unicode character sequence.

StringValue :: " *StringCharacter*_{list} "

1. Return the Unicode character sequence of all *StringCharacter* Unicode character values.

Algorithms

This specification describes some algorithms used by the static and runtime semantics, they're defined in the form of a function-like syntax along with a list of algorithmic steps to take.

For example, this describes if a fragment should be spread into place given a runtime *objectType* and the fragment's *fragmentType*:

doesFragmentTypeApply(objectType, fragmentType) :

1. If *fragmentType* is an Object Type:
 - a. if *objectType* and *fragmentType* are the same type, return **true**, otherwise return **false**.
2. If *fragmentType* is an Interface Type:
 - a. if *objectType* is an implementation of *fragmentType*, return **true** otherwise return **false**.
3. If *fragmentType* is a Union:
 - a. if *objectType* is a possible type of *fragmentType*, return **true** otherwise return **false**.

Appendix: Grammar Summary



SourceCharacter ::

/ [\u0009\u000A\u000D\u0020-\uFFFF] /

Ignored Tokens

Ignored ::

UnicodeBOM

WhiteSpace

LineTerminator

Comment

Comma

UnicodeBOM ::

Byte Order Mark (U+FEFF)

WhiteSpace ::

Horizontal Tab (U+0009)

Space (U+0020)

LineTerminator ::

New Line (U+000A)

Carriage Return (U+000D) [lookahead ≠ New Line (U+000A)]

Carriage Return (U+000D) New Line (U+000A)

Comment ::

*CommentChar*_{list, opt}

CommentChar ::

SourceCharacter but not *LineTerminator*

Comma ::

,

Lexical Tokens

Token ::

Punctuator

Name

IntValue



FloatValue
StringValue

Punctuator :: **one of**

! \$ () ... : = @ [] { | }

Name ::

/[_A-Za-z][_0-9A-Za-z]* /

IntValue ::

IntegerPart

IntegerPart ::

*NegativeSign*_{opt} **0**

*NegativeSign*_{opt} *NonZeroDigit* *Digit*_{list, opt}

NegativeSign ::

-

Digit :: **one of**

0 1 2 3 4 5 6 7 8 9

NonZeroDigit ::

Digit but not **0**

FloatValue ::

IntegerPart *FractionalPart*

IntegerPart *ExponentPart*

IntegerPart *FractionalPart* *ExponentPart*

FractionalPart ::

. *Digit*_{list}

ExponentPart ::

ExponentIndicator *Sign*_{opt} *Digit*_{list}

ExponentIndicator :: **one of**

e E

Sign :: **one of**

+ -

StringValue ::

" "

" *StringCharacter*_{list} **"**

StringCharacter ::



SourceCharacter but not " or \ or *LineTerminator*

\u EscapedUnicode

\ EscapedCharacter

EscapedUnicode ::

/ [0-9A-Fa-f] { 4 } /

EscapedCharacter :: **one of**

" \ / b f n r t

Query Document

Document :

*Definition*_{list}

Definition :

OperationDefinition

FragmentDefinition

OperationDefinition :

SelectionSet

OperationType *Name*_{opt} *VariableDefinitions*_{opt} *Directives*_{opt} *SelectionSet*

OperationType : **one of**

query mutation

SelectionSet :

{ *Selection*_{list} **}**

Selection :

Field

FragmentSpread

InlineFragment

Field :

*Alias*_{opt} *Name* *Arguments*_{opt} *Directives*_{opt} *SelectionSet*_{opt}

Alias :

Name :

Arguments :

(*Argument*_{list} **)**

Argument :



Name : *Value*

FragmentSpread :

... *FragmentName* *Directives*_{opt}

InlineFragment :

... *TypeCondition*_{opt} *Directives*_{opt} *SelectionSet*

FragmentDefinition :

fragment *FragmentName* *TypeCondition* *Directives*_{opt} *SelectionSet*

FragmentName :

Name but not **on**

TypeCondition :

on *NamedType*

*Value*_[Const] :

_[~Const] *Variable*

IntValue

FloatValue

StringValue

BooleanValue

NullValue

EnumValue

*ListValue*_[?Const]

*ObjectValue*_[?Const]

BooleanValue : **one of**

true false

NullValue :

null

EnumValue :

Name but not **true** or **false** or **null**

*ListValue*_[Const] :

[]

[*Value*_[?Const]_{list}]

*ObjectValue*_[Const] :

{ }

{ *ObjectField*_[?Const]_{list} }



*ObjectField*_[Const] :

Name : *Value*_[?Const]

VariableDefinitions :

(*VariableDefinition*_{list})

VariableDefinition :

Variable : *Type* *DefaultValue*_{opt}

Variable :

\$ *Name*

DefaultValue :

= *Value*_[Const]

Type :

NamedType

ListType

NonNullType

NamedType :

Name

ListType :

[*Type*]

NonNullType :

NamedType !

ListType !

Directives :

*Directive*_{list}

Directive :

@ *Name* *Arguments*_{opt}

Index

[Alias](#)
[Argument](#)

[Arguments](#)
[BooleanValue](#)

[CoerceArgumentValues](#)
[CoerceVariableValues](#)

CollectFields
Comma
Comment
CommentChar
CompleteValue
DefaultValue
Definition
Digit
Directive
Directives
Document
DoesFragmentTypeApply
EnumValue
EscapedCharacter
EscapedUnicode
Example
ExecuteField
ExecuteMutation
ExecuteQuery
ExecuteRequest
ExecuteSelectionSet
ExponentIndicator
ExponentPart

Field
FieldsInSetCanMerge
FloatValue
FractionalPart
FragmentDefinition
FragmentName
FragmentSpread
GetOperation
GetPossibleTypes
Ignored
InlineFragment
IntValue
IntegerPart
LineTerminator
ListType
ListValue
MergeSelectionSets
Name
NamedType
NegativeSign
NonNullType
NonZeroDigit
NullValue

ObjectField
ObjectValue
OperationDefinition
OperationType
Punctuator
ResolveAbstractType
ResolveFieldValue
SameResponseShape
Selection
SelectionSet
Sign
SourceCharacter
StringCharacter
StringValue
Token
Type
TypeCondition
UnicodeBOM
Value
Variable
VariableDefinition
VariableDefinitions
WhiteSpace



Written in Spec Markdown.