GraphQL

Current Working Draft

Introduction

This is the specification for GraphQL, a query language and execution engine originally created at Facebook in 2012 for describing the capabilities and requirements of data models for client-server applications. The development of this open standard started in 2015. This specification was licensed under OWFa 1.0 in 2017. The GraphQL Foundation was formed in 2019 as a neutral focal point for organizations who support the GraphQL ecosystem, and the GraphQL Specification Project was established also in 2019 as the Joint Development Foundation Projects, LLC, GraphQL Series.

If your organization benefits from GraphQL, please consider becoming a member and helping us to sustain the activities that support the health of our neutral ecosystem.

The GraphQL Specification Project has evolved and may continue to evolve in future editions of this specification. Previous editions of the GraphQL specification can be found at permalinks that match their release tag. The latest working draft release can be found at https://spec.graphql.org/draft.

Copyright Notice

Licensing

The GraphQL Specification Project is made available by the Joint Development Foundation. The current Working Group charter, which includes the IP policy governing all working group deliverables (including specifications, source code, and datasets) may be found at https://technical-charter.graphql.org.

Currently, the licenses governing GraphQL Specification Project deliverables are:

Deliverable	License
Specifications	Open Web Foundation Agreement 1.0 (Patent and Copyright Grants)
Source code	MIT License
Data sets	CC0 1.0

Conformance

A conforming implementation of GraphQL must fulfill all normative requirements. Conformance requirements are described in this document via both descriptive assertions and key words with clearly defined meanings.

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative portions of this document are to be interpreted as described in IETF RFC 2119. These key words may appear in lowercase and still retain their meaning unless explicitly declared as non-normative.

A conforming implementation of GraphQL may provide additional functionality, but must not where explicitly disallowed or would otherwise result in non-conformance.

Conforming Algorithms

Algorithm steps phrased in imperative grammar (e.g. “Return the result of calling resolver”) are to be interpreted with the same level of requirement as the algorithm it is contained within. Any algorithm referenced within an algorithm step (e.g. “Let completedResult be the result of calling CompleteValue()”) is to be interpreted as having at least the same level of requirement as the algorithm containing that step.

Conformance requirements expressed as algorithms can be fulfilled by an implementation of this specification in any way as long as the perceived result is equivalent. Algorithms described in this document are written to be easy to understand. Implementers are encouraged to include equivalent but optimized implementations.

See Appendix A for more details about the definition of algorithms and other notational conventions used in this document.

Non-Normative Portions

All contents of this document are normative except portions explicitly declared as non-normative.

Examples in this document are non-normative, and are presented to aid understanding of introduced concepts and the behavior of normative portions of the specification. Examples are either introduced explicitly in prose (e.g. "for example") or are set apart in example or counter-example blocks, like this:

> *Example № 1*
> ```
> This is an example of a non-normative example.
> ```

> *Counter Example № 2*
> ```
> This is an example of a non-normative counter-example.
> ```

Notes in this document are non-normative, and are presented to clarify intent, draw attention to potential edge-cases and pit-falls, and answer common questions that arise during implementation. Notes are either introduced explicitly in prose (e.g. "Note: ") or are set apart in a note block, like this:

> *Note*
> This is an example of a non-normative note.

Contents

1 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.

Example № 3

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

```
  }
}
```

Which produces the resulting data (in JSON):

Example № 4

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

GraphQL is not a programming language capable of arbitrary computation, but is instead a language used to make requests to application services that have capabilities defined in this specification. GraphQL does not mandate a particular programming language or storage system for application services that implement it. Instead, application services 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:

- **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 builds the language and runtime necessary to enable that.
- **Hierarchical**: Most product development today involves the creation and manipulation of view hierarchies. To achieve congruence with the structure of these applications, a GraphQL request itself is structured hierarchically. The request is shaped just like the data in its response. It is a natural way for clients to describe data requirements.
- **Strong-typing**: Every GraphQL service defines an application-specific type system. Requests are executed within the context of that type system. Given a GraphQL operation, tools can ensure that it is both syntactically correct and valid within that type system before execution, i.e. at development time, and the service can make certain guarantees about the shape and nature of the response.
- **Client-specified response**: Through its type system, a GraphQL service 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 requests are specified at field-level granularity. In the majority of client-server applications written without GraphQL, the service determines the shape of data returned from its various endpoints. A GraphQL response, on the other hand, contains exactly what a client asks for and no more.
- **Introspective**: GraphQL is introspective. A GraphQL service's type system can 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 services—

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 services 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 service 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.

2 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, mutations, and subscriptions) as well as fragments, a common unit of composition allowing for data requirement 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. In this document, syntactic grammar productions are distinguished with a colon `:` while lexical grammar productions are distinguished with a double-colon `::`.

The source text of a GraphQL document must be a sequence of *SourceCharacter*. The character sequence must be described by a sequence of *Token* and *Ignored* lexical grammars. The lexical token sequence, omitting *Ignored*, must be described by a single *Document* syntactic grammar.

> *Note*
> See Appendix A for more information about the lexical and syntactic grammar and other notational conventions used throughout this document.

Lexical Analysis & Syntactic Parse

The source text of a GraphQL document is first converted into a sequence of lexical tokens, *Token*, and ignored tokens, *Ignored*. The source text is scanned from left to right, repeatedly taking the next possible sequence of code-points allowed by the lexical grammar productions as the next token. This sequence of lexical tokens are then scanned from left to right to produce an abstract syntax tree (AST) according to the *Document* syntactical grammar.

Lexical grammar productions in this document use *lookahead restrictions* to remove ambiguity and ensure a single valid lexical analysis. A lexical token is only valid if not followed by a character in its lookahead restriction.

For example, an *IntValue* has the restriction [lookahead ∉ *Digit*], so cannot be followed by a *Digit*. Because of this, the sequence `123` cannot represent the tokens (`12`, `3`) since `12` is followed by the *Digit* `3` and so must only represent a single token. Use *WhiteSpace* or other *Ignored* between characters to represent multiple tokens.

> *Note*
> This typically has the same behavior as a "maximal munch" longest possible match, however some lookahead restrictions include additional constraints.

2.1 Source Text

SourceCharacter **::**
  Any Unicode scalar value

GraphQL documents are interpreted from a source text, which is a sequence of *SourceCharacter*, each *SourceCharacter* being a *Unicode scalar value* which may be any Unicode code point from U+0000 to U+D7FF or U+E000 to U+10FFFF (informally referred to as *"characters"* through most of this specification).

A GraphQL document may be expressed only in the ASCII range 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. Non-ASCII Unicode scalar values may appear within *StringValue* and *Comment*.

> *Note*
> An implementation which uses *UTF-16* to represent GraphQL documents in memory (for example, JavaScript or Java) may encounter a *surrogate pair*. This encodes one *supplementary code point* and is a single valid source character, however an unpaired *surrogate code point* is not a valid source character.

2.1.1 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 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.

2.1.2 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 and separate lexical tokens, any amount may appear before or after any other token and have no significance to the semantic meaning of a GraphQL Document.

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

2.1.3 Comments

Comment **::**
- **#** $CommentChar_{list,\ opt}$ [lookahead ∉ *CommentChar*]

CommentChar **::**
- *SourceCharacter* but not *LineTerminator*

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

A comment may contain any *SourceCharacter* except *LineTerminator* so a comment always consists of all *SourceCharacter* starting with the **#** character up to but not including the *LineTerminator* (or end of the source).

Comments are *Ignored* like white space and may appear after any token, or before a *LineTerminator*, and have no significance to the semantic meaning of a GraphQL Document.

2.1.4 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 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.

2.1.5 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. Lexical tokens may be separated by *Ignored* tokens.

Tokens are later used as terminal symbols in GraphQL syntactic grammar rules.

2.1.6 Ignored Tokens

Ignored ::
- *UnicodeBOM*
- *WhiteSpace*
- *LineTerminator*
- *Comment*
- *Comma*

Ignored tokens are used to improve readability and provide separation between lexical tokens, but are otherwise insignificant and not referenced in syntactical grammar productions.

Any amount of *Ignored* may appear before and after every lexical token. No ignored regions of a source document are significant, however *SourceCharacter* which appear in *Ignored* may also appear within a lexical *Token* in a significant way, for example a *StringValue* may contain white space characters. No *Ignored* may appear *within* a *Token*, for example no white space characters are permitted between the characters defining a *FloatValue*.

Byte Order Mark

UnicodeBOM **::**
Byte Order Mark (U+FEFF)

The *Byte Order Mark* is a special Unicode code point which may appear at the beginning of a file which programs may use to determine the fact that the text stream is Unicode, and what specific encoding has been used. As files are often concatenated, a *Byte Order Mark* may appear before or after any lexical token and is *Ignored*.

2.1.7 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.

2.1.8 Names

Name **::**
NameStart *NameContinue*$_{\text{list, opt}}$ [lookahead ∉ *NameContinue*]

NameStart **::**
Letter
_

NameContinue **::**
Letter
Digit

```
_
```

Letter :: **one of**

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```

Digit :: **one of**

```
0 1 2 3 4 5 6 7 8 9
```

GraphQL Documents are full of named things: operations, fields, arguments, types, 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.

A *Name* must not be followed by a *NameContinue*. In other words, a *Name* token is always the longest possible valid sequence. The source characters `a1` cannot be interpreted as two tokens since `a` is followed by the *NameContinue* `1`.

> *Note*
> Names in GraphQL are limited to the Latin ASCII subset of *SourceCharacter* in order to support interoperation with as many other systems as possible.

Reserved Names

Any *Name* within a GraphQL type system must not start with two underscores "__" unless it is part of the introspection system as defined by this specification.

2.2 Document

Document :
  Definition$_{list}$

Definition :
  ExecutableDefinition
  TypeSystemDefinitionOrExtension

ExecutableDocument :
  ExecutableDefinition$_{list}$

ExecutableDefinition :

OperationDefinition

FragmentDefinition

A GraphQL Document describes a complete file or request string operated on by a GraphQL service or client. A document contains multiple definitions, either executable or representative of a GraphQL type system.

Documents are only executable by a GraphQL service if they are *ExecutableDocument* and contain at least one *OperationDefinition*. A Document which contains *TypeSystemDefinitionOrExtension* must not be executed; GraphQL execution services which receive a Document containing these should return a descriptive error.

GraphQL services which only seek to execute GraphQL requests and not construct a new GraphQL schema may choose to only permit *ExecutableDocument*.

Documents which do not contain *OperationDefinition* or do contain *TypeSystemDefinitionOrExtension* may still be parsed and validated to allow client tools to represent many GraphQL uses which may appear across many individual files.

If a Document contains only one operation, that operation may be unnamed. If that operation is a query without variables or directives then it may also be represented in the shorthand form, omitting both the `query` keyword as well as the operation name. Otherwise, if a GraphQL Document contains multiple operations, each operation must be named. When submitting a Document with multiple operations to a GraphQL service, the name of the desired operation to be executed must also be provided.

2.3 Operations

OperationDefinition **:**

OperationType *Name*$_{opt}$ *VariablesDefinition*$_{opt}$ *Directives*$_{opt}$ *SelectionSet*

SelectionSet

OperationType **: one of**

`query` `mutation` `subscription`

There are three types of operations that GraphQL models:

- query – a read-only fetch.
- mutation – a write followed by a fetch.
- subscription – a long-lived request that fetches data in response to source events.

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:

Example № 5

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

Query Shorthand

If a document contains only one operation and that operation is a query which defines no variables and has no directives applied to it then that operation may be represented in a short-hand form which omits the **query** keyword and operation name.

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

Example № 6

```
{
  field
}
```

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

2.4 Selection Sets

SelectionSet :
  { *Selection*$_{\text{list}}$ }

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.

Example № 7

```
{
  id
```

```
  firstName
  lastName
}
```

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

2.5 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.

Example № 8

```
{
  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.

Example № 9

```
# `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
  }
}
```

2.6 Arguments

Arguments[Const] :
(*Argument*[?Const]list)

Argument[Const] :
Name : *Value*[?Const]

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 service'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`:

Example № 10

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

Many arguments can exist for a given field:

Example № 11

```
{
  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 operations are semantically identical:

Example № 12

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

Example № 13

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

2.7 Field Alias

Alias :
  Name :

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

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

Example № 14

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

```
    bigPic: profilePic(size: 1024)
  }
}
```

which returns the result:

Example № 15

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

The fields at the top level of an operation can also be given an alias:

Example № 16

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

which returns the result:

Example № 17

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

2.8 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:

Example № 18

```
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 operation.

Example № 19

```
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 field selection at the same level as the fragment invocation. This happens through multiple levels of fragment spreads.

For example:

Example № 20

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

2.8.1 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 the concrete type of the object it is operating on matches the type of the fragment.

For example in this operation using the Facebook data model:

Example № 21

```
query FragmentTyping {
  profiles(handles: ["zuck", "coca-cola"]) {
    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.

Example № 22

```
{
  "profiles": [
    {
      "handle": "zuck",
      "friends": { "count": 1234 }
    },
    {
      "handle": "coca-cola",
      "likers": { "count": 90234512 }
    }
  ]
}
```

2.8.2 Inline Fragments

InlineFragment :

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

Fragments can also be defined inline within a selection set. This is useful for conditionally including fields based on a type condition or applying a directive to a selection set.

This feature of standard fragment inclusion was demonstrated in the `query FragmentTyping` example above. We could accomplish the same thing using inline fragments.

Example № 23

```
query inlineFragmentTyping {
  profiles(handles: ["zuck", "coca-cola"]) {
    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.

Example № 24

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

2.9 Input Values

Value$_{[Const]}$:

[if not 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).

2.9.1 Int Value

IntValue ::
IntegerPart [lookahead ∉ {*Digit*, `.`, *NameStart*}]

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

NegativeSign ::
`-`

NonZeroDigit ::
Digit but not `0`

An *IntValue* is specified without a decimal point or exponent but may be negative (ex. `-123`). It must not have any leading `0`.

An *IntValue* must not be followed by a *Digit*. In other words, an *IntValue* token is always the longest possible valid sequence. The source characters `12` cannot be interpreted as two tokens since `1` is followed by the *Digit* `2`. This also means the source `00` is invalid since it can neither be interpreted as a single token nor two `0` tokens.

An *IntValue* must not be followed by a `.` or *NameStart*. If either `.` or *ExponentIndicator* follows then the token must only be interpreted as a possible *FloatValue*. No other *NameStart* character can follow. For example the sequences `0x123` and `123L` have no valid lexical representations.

2.9.2 Float Value

FloatValue **::**

IntegerPart *FractionalPart* *ExponentPart* [lookahead ∉ {*Digit*, `.`, *NameStart*}]

IntegerPart *FractionalPart* [lookahead ∉ {*Digit*, `.`, *NameStart*}]

IntegerPart *ExponentPart* [lookahead ∉ {*Digit*, `.`, *NameStart*}]

FractionalPart **::**

`.` $Digit_{list}$

ExponentPart **::**

ExponentIndicator $Sign_{opt}$ $Digit_{list}$

ExponentIndicator **:: one of**

`e` `E`

Sign **:: one of**

`+` `-`

A *FloatValue* includes either a decimal point (ex. `1.0`) or an exponent (ex. `1e50`) or both (ex. `6.0221413e23`) and may be negative. Like *IntValue*, it also must not have any leading `0`.

A *FloatValue* must not be followed by a *Digit*. In other words, a *FloatValue* token is always the longest possible valid sequence. The source characters `1.23` cannot be interpreted as two tokens since `1.2` is followed by the *Digit* `3`.

A *FloatValue* must not be followed by a `.`. For example, the sequence `1.23.4` cannot be interpreted as two tokens (`1.2`, `3.4`).

A *FloatValue* must not be followed by a *NameStart*. For example the sequence `0x1.2p3` has no valid lexical representation.

> *Note*
> The numeric literals *IntValue* and *FloatValue* both restrict being immediately followed by a letter (or other *NameStart*) to reduce confusion or unexpected behavior since GraphQL only supports decimal numbers.

2.9.3 Boolean Value

BooleanValue : **one of**
`true` `false`

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

2.9.4 String Value

StringValue ::
`""` [lookahead ≠ `"`]
`"` *StringCharacter*$_{list}$ `"`
`"""` *BlockStringCharacter*$_{list,\ opt}$ `"""`

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

EscapedUnicode ::
`{` *HexDigit*$_{list}$ `}`
HexDigit *HexDigit* *HexDigit* *HexDigit*

HexDigit :: **one of**
`0` `1` `2` `3` `4` `5` `6` `7` `8` `9`
`A` `B` `C` `D` `E` `F`
`a` `b` `c` `d` `e` `f`

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

BlockStringCharacter ::
SourceCharacter but not `"""` or `\"""`
`\"""`

A *StringValue* is evaluated to a *Unicode text* value, a sequence of *Unicode scalar value*, by interpreting all escape sequences using the static semantics defined below. White space and other characters ignored between lexical tokens are significant within a string value.

The empty string `""` must not be followed by another `"` otherwise it would be interpreted as the beginning of a block string. As an example, the source `""""""` can only be interpreted as a single empty block string and not three empty strings.

Escape Sequences

In a single-quoted *StringValue*, any *Unicode scalar value* may be expressed using an escape sequence. GraphQL strings allow both C-style escape sequences (for example `\n`) and two forms of Unicode escape sequences: one with a fixed-width of 4 hexadecimal digits (for example `\u000A`) and one with a variable-width most useful for representing a *supplementary character* such as an Emoji (for example `\u{1F4A9}`).

The hexadecimal number encoded by a Unicode escape sequence must describe a *Unicode scalar value*, otherwise must result in a parse error. For example both sources `"\uDEAD"` and `"\u{110000}"` should not be considered valid *StringValue*.

Escape sequences are only meaningful within a single-quoted string. Within a block string, they are simply that sequence of characters (for example `"""\n"""` represents the *Unicode text* [U+005C, U+006E]). Within a comment an escape sequence is not a significant sequence of characters. They may not appear elsewhere in a GraphQL document.

Since *StringCharacter* must not contain some code points directly (for example, a *LineTerminator*), escape sequences must be used to represent them. All other escape sequences are optional and unescaped non-ASCII Unicode characters are allowed within strings. If using GraphQL within a system which only supports ASCII, then escape sequences may be used to represent all Unicode characters outside of the ASCII range.

For legacy reasons, a *supplementary character* may be escaped by two fixed-width unicode escape sequences forming a *surrogate pair*. For example the input `"\uD83D\uDCA9"` is a valid *StringValue* which represents the same *Unicode text* as `"\u{1F4A9}"`. While this legacy form is allowed, it should be avoided as a variable-width unicode escape sequence is a clearer way to encode such code points.

When producing a *StringValue*, implementations should use escape sequences to represent non-printable control characters (U+0000 to U+001F and U+007F to U+009F). Other escape sequences are not necessary, however an implementation may use escape sequences to represent any other range of code points (for example, when producing ASCII-only output). If an implementation chooses to escape a *supplementary character*, it should only use a variable-width unicode escape sequence.

Block Strings

Block strings are sequences of characters wrapped in triple-quotes (`"""`). White space, line terminators, quote, and backslash characters may all be used unescaped to enable verbatim text. Characters must all be valid *SourceCharacter*.

Since block strings represent freeform text often used in indented positions, the string value semantics of a block string excludes uniform indentation and blank initial and trailing lines via BlockStringValue().

For example, the following operation containing a block string:

Example № 25

```
mutation {
  sendEmail(message: """
    Hello,
```

```
      World!

    Yours,
      GraphQL.
  """)
}
```

Is identical to the standard quoted string:

Example № 26

```
mutation {
  sendEmail(message: "Hello,\n  World!\n\nYours,\n  GraphQL.")
}
```

Since block string values strip leading and trailing empty lines, there is no single canonical printed block string for a given value. Because block strings typically represent freeform text, it is considered easier to read if they begin and end with an empty line.

Example № 27

```
"""
This starts with and ends with an empty line,
which makes it easier to read.
"""
```

Counter Example № 28

```
"""This does not start with or end with any empty lines,
which makes it a little harder to read."""
```

Note

If non-printable ASCII characters are needed in a string value, a standard quoted string with appropriate escape sequences must be used instead of a block string.

Static Semantics

A *StringValue* describes a *Unicode text* value, which is a sequence of *Unicode scalar value*.

These semantics describe how to apply the *StringValue* grammar to a source text to evaluate a *Unicode text*. Errors encountered during this evaluation are considered a failure to apply the *StringValue* grammar to a source and must result in a parsing error.

StringValue :: **""**

1. Return an empty sequence.

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

1. Return the *Unicode text* by concatenating the evaluation of all *StringCharacter*.

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

1. Return the *Unicode scalar value SourceCharacter*.

StringCharacter :: \u *EscapedUnicode*

1. Let *value* be the hexadecimal value represented by the sequence of *HexDigit* within *EscapedUnicode*.
2. Assert *value* is a within the *Unicode scalar value* range (≥ 0x0000 and ≤ 0xD7FF or ≥ 0xE000 and ≤ 0x10FFFF).
3. Return the *Unicode scalar value value*.

StringCharacter :: \u *HexDigit HexDigit HexDigit HexDigit* \u *HexDigit HexDigit HexDigit HexDigit*

1. Let *leadingValue* be the hexadecimal value represented by the first sequence of *HexDigit*.
2. Let *trailingValue* be the hexadecimal value represented by the second sequence of *HexDigit*.
3. If *leadingValue* is ≥ 0xD800 and ≤ 0xDBFF (a *Leading Surrogate*):
   a. Assert *trailingValue* is ≥ 0xDC00 and ≤ 0xDFFF (a *Trailing Surrogate*).
   b. Return (*leadingValue* - 0xD800) × 0x400 + (*trailingValue* - 0xDC00) + 0x10000.
4. Otherwise:
   a. Assert *leadingValue* is within the *Unicode scalar value* range.
   b. Assert *trailingValue* is within the *Unicode scalar value* range.
   c. Return the sequence of the *Unicode scalar value leadingValue* followed by the *Unicode scalar value trailingValue*.

> *Note*
> If both escape sequences encode a *Unicode scalar value*, then this semantic is identical to applying the prior semantic on each fixed-width escape sequence. A variable-width escape sequence must only encode a *Unicode scalar value*.

StringCharacter :: \ *EscapedCharacter*

1. Return the *Unicode scalar value* represented by *EscapedCharacter* according to the table below.

Escaped Character	Scalar Value	Character Name
"	U+0022	double quote
\	U+005C	reverse solidus (back slash)
/	U+002F	solidus (forward slash)

Escaped Character	Scalar Value	Character Name
`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

StringValue :: `"""` *BlockStringCharacter*$_{\text{list, opt}}$ `"""`

1. Let *rawValue* be the *Unicode text* by concatenating the evaluation of all *BlockStringCharacter* (which may be an empty sequence).
2. Return the result of BlockStringValue(*rawValue*).

BlockStringCharacter :: *SourceCharacter* but not `"""` or `\"""`

1. Return the *Unicode scalar value SourceCharacter*.

BlockStringCharacter :: `\"""`

1. Return the character sequence `"""`.

BlockStringValue(*rawValue*) :

1. Let *lines* be the result of splitting *rawValue* by *LineTerminator*.
2. Let *commonIndent* be **null**.
3. For each *line* in *lines*:
   a. If *line* is the first item in *lines*, continue to the next line.
   b. Let *length* be the number of characters in *line*.
   c. Let *indent* be the number of leading consecutive *WhiteSpace* characters in *line*.
   d. If *indent* is less than *length*:
      i. If *commonIndent* is **null** or *indent* is less than *commonIndent*:
         1. Let *commonIndent* be *indent*.
4. If *commonIndent* is not **null**:
   a. For each *line* in *lines*:
      i. If *line* is the first item in *lines*, continue to the next line.
      ii. Remove *commonIndent* characters from the beginning of *line*.
5. While the first item *line* in *lines* contains only *WhiteSpace*:
   a. Remove the first item from *lines*.
6. While the last item *line* in *lines* contains only *WhiteSpace*:

a. Remove the last item from *lines*.

7. Let *formatted* be the empty character sequence.
8. For each *line* in *lines*:
   a. If *line* is the first item in *lines*:
      i. Append *formatted* with *line*.
   b. Otherwise:
      i. Append *formatted* with a line feed character (U+000A).
      ii. Append *formatted* with *line*.
9. Return *formatted*.

2.9.5 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:

Example № 29

```
{
  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 variable value as **null** or not providing a variable value at all.

2.9.6 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.

2.9.7 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*

2.9.8 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 operations are semantically identical:

Example № 30

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

Example № 31

```
{
  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*

2.10 Variables

Variable :
  $ *Name*

VariablesDefinition :
  (*VariableDefinition*$_{list}$)

VariableDefinition :
  Variable : *Type* *DefaultValue*$_{opt}$ *Directives*$_{[Const]opt}$

DefaultValue :
  = *Value*$_{[Const]}$

A GraphQL operation can be parameterized with variables, maximizing 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. Values for those variables are provided to a GraphQL service as part of a request so they may be substituted in during execution.

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

Example № 32
```
query getZuckProfile($devicePicSize: Int) {
  user(id: 4) {
    id
    name
    profilePic(size: $devicePicSize)
  }
}
```

If providing JSON for the variables' values, we could request a `profilePic` of size `60`:

Example № 33
```
{
  "devicePicSize": 60
```

```
}
```

Variable Use Within Fragments

Variables can be used within fragments. 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, that operation is invalid (see All Variable Uses Defined).

2.11 Type References

Type :
  NamedType
  ListType
  NonNullType

NamedType :
  Name

ListType :
  [*Type* **]**

NonNullType :
  NamedType **!**
  ListType **!**

GraphQL describes the types of data expected by arguments and 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*

2.12 Directives

Directives$_{[Const]}$:

Directive$_{[?Const]list}$

Directive$_{[Const]}$:

@ *Name* *Arguments*$_{[?Const]opt}$

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 types, fields, fragments and operations.

As future versions of GraphQL adopt new configurable execution capabilities, they may be exposed via directives. GraphQL services and tools may also provide any additional *custom directive* beyond those described here.

Directive Order Is Significant

Directives may be provided in a specific syntactic order which may have semantic interpretation.

These two type definitions may have different semantic meaning:

Example № 34

```
type Person
  @addExternalFields(source: "profiles")
  @excludeField(name: "photo") {
```

```
  name: String
}
```

Example № 35

```
type Person
  @excludeField(name: "photo")
  @addExternalFields(source: "profiles") {
  name: String
}
```

3 Type System

The GraphQL Type system describes the capabilities of a GraphQL service and is used to determine if a requested operation is valid, to guarantee the type of response results, and describes the input types of variables to determine if values provided at request time are valid.

TypeSystemDocument :
  TypeSystemDefinition$_{\text{list}}$

TypeSystemDefinition :
  SchemaDefinition
  TypeDefinition
  DirectiveDefinition

The GraphQL language includes an IDL used to describe a GraphQL service's type system. Tools may use this definition language to provide utilities such as client code generation or service boot-strapping.

GraphQL tools or services which only seek to execute GraphQL requests and not construct a new GraphQL schema may choose not to allow *TypeSystemDefinition*. Tools which only seek to produce schema and not execute requests may choose to only allow *TypeSystemDocument* and not allow *ExecutableDefinition* or *TypeSystemExtension* but should provide a descriptive error if present.

Note
The type system definition language is used throughout the remainder of this specification document when illustrating example type systems.

3.1 Type System Extensions

TypeSystemExtensionDocument :
  TypeSystemDefinitionOrExtension$_{list}$

TypeSystemDefinitionOrExtension :
  TypeSystemDefinition
  TypeSystemExtension

TypeSystemExtension :
  SchemaExtension
  TypeExtension

Type system extensions are used to represent a GraphQL type system which has been extended from some original type system. For example, this might be used by a local service to represent data a GraphQL client only accesses locally, or by a GraphQL service which is itself an extension of another GraphQL service.

Tools which only seek to produce and extend schema and not execute requests may choose to only allow *TypeSystemExtensionDocument* and not allow *ExecutableDefinition* but should provide a descriptive error if present.

3.2 Descriptions

Description :
  StringValue

Documentation is a first-class feature of GraphQL type systems. To ensure the documentation of a GraphQL service remains consistent with its capabilities, descriptions of GraphQL definitions are provided alongside their definitions and made available via introspection.

To allow GraphQL service designers to easily publish documentation alongside the capabilities of a GraphQL service, GraphQL descriptions are defined using the Markdown syntax (as specified by CommonMark). In the type system definition language, these description strings (often *BlockString*) occur immediately before the definition they describe.

GraphQL schema and all other definitions (e.g. types, fields, arguments, etc.) which can be described should provide a *Description* unless they are considered self descriptive.

As an example, this simple GraphQL schema is well described:

Example № 36

```
"""
A simple GraphQL schema which is well described.
"""
schema {
  query: Query
}

"""
Root type for all your query operations
"""
type Query {
  """
  Translates a string from a given language into a different language.
  """
  translate(
    "The original language that `text` is provided in."
    fromLanguage: Language

    "The translated language to be returned."
    toLanguage: Language

    "The text to be translated."
    text: String
  ): String
}

"""
The set of languages supported by `translate`.
"""
enum Language {
  "English"
  EN

  "French"
  FR

  "Chinese"
  CH
}
```

3.3 Schema

SchemaDefinition :
Description$_{opt}$ **schema** *Directives*$_{[Const]opt}$ **{** *RootOperationTypeDefinition*$_{list}$ **}**

RootOperationTypeDefinition :
OperationType : *NamedType*

A GraphQL service's collective type system capabilities are referred to as that service's "schema". A schema is defined in terms of the types and directives it supports as well as the *root operation type* for each kind of operation: query, mutation, and subscription; this determines the place in the type system where those operations begin.

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

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.

All types and directives defined within a schema must not have a name which begins with "__" (two underscores), as this is used exclusively by GraphQL's introspection system.

3.3.1 Root Operation Types

A schema defines the initial *root operation type* for each kind of operation it supports: query, mutation, and subscription; this determines the place in the type system where those operations begin.

The **query** *root operation type* must be provided and must be an Object type.

The **mutation** *root operation type* is optional; if it is not provided, the service does not support mutations. If it is provided, it must be an Object type.

Similarly, the **subscription** *root operation type* is also optional; if it is not provided, the service does not support subscriptions. If it is provided, it must be an Object type.

The **query**, **mutation**, and **subscription** root types must all be different types if provided.

The fields on the **query** *root operation type* indicate what fields are available at the top level of a GraphQL query operation.

For example, this example operation:

Example № 37

```
query {
  myName
}
```

is only valid when the **query** *root operation type* has a field named "myName":

Example № 38

```
type Query {
  myName: String
}
```

Similarly, the following mutation is only valid if the **mutation** *root operation type* has a field named "setName".

Example № 39

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

When using the type system definition language, a document must include at most one **schema** definition.

In this example, a GraphQL schema is defined with both a query and mutation *root operation type*:

Example № 40

```
schema {
  query: MyQueryRootType
  mutation: MyMutationRootType
}

type MyQueryRootType {
  someField: String
}

type MyMutationRootType {
  setSomeField(to: String): String
}
```

Default Root Operation Type Names

The *default root type name* for each **query**, **mutation**, and **subscription** *root operation type* are `"Query"`, `"Mutation"`, and `"Subscription"` respectively.

The type system definition language can omit the schema definition when each *root operation type* uses its respective *default root type name* and no other type uses any *default root type name*.

Likewise, when representing a GraphQL schema using the type system definition language, a schema definition should be omitted if each *root operation type* uses its respective *default root type name* and no other

type uses any *default root type name*.

This example describes a valid complete GraphQL schema, despite not explicitly including a **schema** definition. The "Query" type is presumed to be the **query** *root operation type* of the schema.

Example № 41

```
type Query {
  someField: String
}
```

This example describes a valid GraphQL schema without a **mutation** *root operation type*, even though it contains a type named "Mutation". The schema definition must be included, otherwise the "Mutation" type would be incorrectly presumed to be the **mutation** *root operation type* of the schema.

Example № 42

```
schema {
  query: Query
}

type Query {
  latestVirus: Virus
}

type Virus {
  name: String
  mutations: [Mutation]
}

type Mutation {
  name: String
}
```

3.3.2 Schema Extension

SchemaExtension :

extend schema *Directives*$_{[Const]opt}$ **{** *RootOperationTypeDefinition*$_{list}$ **}**

extend schema *Directives*$_{[Const]}$ [lookahead ≠ **{**]

Schema extensions are used to represent a schema which has been extended from an original schema. For example, this might be used by a GraphQL service which adds additional operation types, or additional directives to an existing schema.

Note

> Schema extensions without additional operation type definitions must not be followed by a `{` (such as a query shorthand) to avoid parsing ambiguity. The same limitation applies to the type definitions and extensions below.

Schema Validation

Schema extensions have the potential to be invalid if incorrectly defined.

1. The Schema must already be defined.
2. Any non-repeatable directives provided must not already apply to the original Schema.

3.4 Types

TypeDefinition :
- *ScalarTypeDefinition*
- *ObjectTypeDefinition*
- *InterfaceTypeDefinition*
- *UnionTypeDefinition*
- *EnumTypeDefinition*
- *InputObjectTypeDefinition*

The fundamental unit of any GraphQL Schema is the type. There are six kinds of named type definitions in GraphQL, and two wrapping types.

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 and other Interface types which implement this Interface are guaranteed to implement those fields. Whenever a field claims it will return an Interface type, it will return a valid implementing Object type during execution.

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.

Finally, oftentimes it is useful to provide complex structs as inputs to GraphQL field arguments or variables; the `Input Object` type allows the schema to define exactly what data is expected.

3.4.1 Wrapping Types

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.

A GraphQL schema may describe that a field represents a list of another type; the `List` type is provided for this reason, and wraps another type.

Similarly, the `Non-Null` type wraps another type, and denotes that the resulting value will never be **null** (and that a *field error* cannot result in a **null** value).

These two types are referred to as "wrapping types"; non-wrapping types are referred to as "named types". A wrapping type has an underlying named type, found by continually unwrapping the type until a named type is found.

3.4.2 Input and Output Types

Types are used throughout GraphQL to describe both the values accepted as input to arguments and variables as well as the values output by fields. These two uses categorize types as *input types* and *output types*. Some kinds of types, like Scalar and Enum types, can be used as both input types and output types; other kinds of types can only be used in one or the other. Input Object types can only be used as input types. Object, Interface, and Union types can only be used as output types. Lists and Non-Null types may be used as input types or output types depending on how the wrapped type may be used.

IsInputType(*type*) **:**

1. If *type* is a List type or Non-Null type:
   a. Let *unwrappedType* be the unwrapped type of *type*.
   b. Return IsInputType(*unwrappedType*)
2. If *type* is a Scalar, Enum, or Input Object type:
   a. Return **true**
3. Return **false**

IsOutputType(*type*) **:**

1. If *type* is a List type or Non-Null type:
   a. Let *unwrappedType* be the unwrapped type of *type*.
   b. Return IsOutputType(*unwrappedType*)

2. If *type* is a Scalar, Object, Interface, Union, or Enum type:
    a. Return **true**
3. Return **false**

3.4.3 Type Extensions

TypeExtension :
  ScalarTypeExtension
  ObjectTypeExtension
  InterfaceTypeExtension
  UnionTypeExtension
  EnumTypeExtension
  InputObjectTypeExtension

Type extensions are used to represent a GraphQL type which has been extended from some original type. For example, this might be used by a local service to represent additional fields a GraphQL client only accesses locally.

3.5 Scalars

ScalarTypeDefinition :
  Description$_{opt}$ `scalar` *Name* *Directives*$_{[Const]opt}$

Scalar types represent primitive leaf values in a GraphQL type system. GraphQL responses take the form of a hierarchical tree; the leaves of this tree are typically GraphQL Scalar types (but may also be Enum types or **null** values).

GraphQL provides a number of built-in scalars which are fully defined in the sections below, however type systems may also add additional custom scalars to introduce additional semantic meaning.

Built-in Scalars

GraphQL specifies a basic set of well-defined Scalar types: *Int*, *Float*, *String*, *Boolean*, and *ID*. A GraphQL framework should support all of these types, and a GraphQL service which provides a type by these names must adhere to the behavior described for them in this document. As an example, a service must not include a type called *Int* and use it to represent 64-bit numbers, internationalization information, or anything other than what is defined in this document.

When returning the set of types from the `__Schema` introspection type, all referenced built-in scalars must be included. If a built-in scalar type is not referenced anywhere in a schema (there is no field, argument, or

input field of that type) then it must not be included.

When representing a GraphQL schema using the type system definition language, all built-in scalars must be omitted for brevity.

Custom Scalars

GraphQL services may use custom scalar types in addition to the built-in scalars. For example, a GraphQL service could define a scalar called `UUID` which, while serialized as a string, conforms to RFC 4122. When querying a field of type `UUID`, you can then rely on the ability to parse the result with a RFC 4122 compliant parser. Another example of a potentially useful custom scalar is `URL`, which serializes as a string, but is guaranteed by the service to be a valid URL.

When defining a custom scalar, GraphQL services should provide a *scalar specification URL* via the `@specifiedBy` directive or the `specifiedByURL` introspection field. This URL must link to a human-readable specification of the data format, serialization, and coercion rules for the scalar.

For example, a GraphQL service providing a `UUID` scalar may link to RFC 4122, or some custom document defining a reasonable subset of that RFC. If a *scalar specification URL* is present, systems and tools that are aware of it should conform to its described rules.

Example № 43

```
scalar UUID @specifiedBy(url: "https://tools.ietf.org/html/rfc4122")
scalar URL @specifiedBy(url: "https://tools.ietf.org/html/rfc3986")
scalar DateTime
  @specifiedBy(url: "https://scalars.graphql.org/andimarek/date-time")
```

Custom *scalar specification URL*s should provide a single, stable format to avoid ambiguity. If the linked specification is in flux, the service should link to a fixed version rather than to a resource which might change.

Note

Some community-maintained custom scalar specifications are hosted at scalars.graphql.org.

Custom *scalar specification URL*s should not be changed once defined. Doing so would likely disrupt tooling or could introduce breaking changes within the linked specification's contents.

Built-in scalar types must not provide a *scalar specification URL* as they are specified by this document.

Note

Custom scalars should also summarize the specified format and provide examples in their description; see the GraphQL scalars implementation guide for more guidance.

Result Coercion and Serialization

A GraphQL service, when preparing a field of a given scalar type, must uphold the contract the scalar type describes, either by coercing the value or producing a *field error* if a value cannot be coerced or if coercion may result in data loss.

A GraphQL service may decide to allow coercing different internal types to the expected return type. For example when coercing a field of type *Int* a boolean **true** value may produce `1` or a string value `"123"` may be parsed as base-10 `123`. However if internal type coercion cannot be reasonably performed without losing information, then it must raise a *field error*.

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

GraphQL scalars are serialized according to the serialization format being used. There may be a most appropriate serialized primitive for each given scalar type, and the service should produce each primitive where appropriate.

See Serialization Format for more detailed information on the serialization of scalars in common JSON and other formats.

Input Coercion

If a GraphQL service 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 *request error* must be raised (input values are validated before execution begins).

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 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**.

3.5.1 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

Fields returning the type *Int* expect to encounter 32-bit integer internal values.

GraphQL services may coerce non-integer internal values to integers when reasonable without losing information, otherwise they must raise a *field error*. Examples of this may include returning `1` for the floating-point number `1.0`, or returning `123` for the string `"123"`. In scenarios where coercion may lose data, raising a field error is more appropriate. For example, a floating-point number `1.2` should raise a field error instead of being truncated to `1`.

If the integer internal value represents a value less than -2^{31} or greater than or equal to 2^{31}, a *field error* should be raised.

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 request 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 *request 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.

3.5.2 Float

The Float scalar type represents signed double-precision finite 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

Fields returning the type *Float* expect to encounter double-precision floating-point internal values.

GraphQL services may coerce non-floating-point internal values to *Float* when reasonable without losing information, otherwise they must raise a *field error*. Examples of this may include returning `1.0` for the integer number `1`, or `123.0` for the string `"123"`.

Non-finite floating-point internal values (*NaN* and *Infinity*) cannot be coerced to *Float* and must raise a *field error*.

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 *request error* indicating an incorrect type. If the input value otherwise represents a value not representable by finite IEEE 754 (e.g. *NaN*, *Infinity*, or a value outside the available precision), a *request error* must be raised.

3.5.3 String

The String scalar type represents textual data, represented as a sequence of Unicode code points. The String type is most often used by GraphQL to represent free-form human-readable text. How the String is encoded internally (for example UTF-8) is left to the service implementation. All response serialization formats must support a string representation (for example, JSON Unicode strings), and that representation must be used to serialize this type.

Result Coercion

Fields returning the type *String* expect to encounter Unicode string values.

GraphQL services may coerce non-string raw values to *String* when reasonable without losing information, 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 Unicode string input values are accepted. All other input values must raise a *request error* indicating an incorrect type.

3.5.4 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

Fields returning the type *Boolean* expect to encounter boolean internal values.

GraphQL services may coerce non-boolean raw values to *Boolean* when reasonable without losing information, otherwise they must raise a *field error*. Examples of this may include returning `true` for non-zero numbers.

Input Coercion

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

3.5.5 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 services 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` or `-4`) input value should be coerced to ID as appropriate for the ID formats a given GraphQL service expects. Any other input value, including float input values (such as `4.0`), must raise a *request error* indicating an incorrect type.

3.5.6 Scalar Extensions

ScalarTypeExtension :
extend scalar *Name* *Directives*[Const]

Scalar type extensions are used to represent a scalar type which has been extended from some original scalar type. For example, this might be used by a GraphQL tool or service which adds directives to an existing scalar.

Type Validation

Scalar type extensions have the potential to be invalid if incorrectly defined.

1. The named type must already be defined and must be a Scalar type.
2. Any non-repeatable directives provided must not already apply to the original Scalar type.

3.6 Objects

ObjectTypeDefinition :
*Description*opt **type** *Name* *ImplementsInterfaces*opt *Directives*[Const]opt *FieldsDefinition*
*Description*opt **type** *Name* *ImplementsInterfaces*opt *Directives*[Const]opt [lookahead ≠ **{**]

ImplementsInterfaces :
ImplementsInterfaces **&** *NamedType*
implements **&**opt *NamedType*

FieldsDefinition :
{ *FieldDefinition*list **}**

FieldDefinition :
*Description*opt *Name* *ArgumentsDefinition*opt **:** *Type* *Directives*[Const]opt

GraphQL operations are hierarchical and composed, describing a tree of information. While Scalar types describe the leaf values of these hierarchical operations, 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 selected 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 selection set.

All fields defined within an Object type must not have a name which begins with `"__"` (two underscores), as this is used exclusively by GraphQL's introspection system.

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

> *Example № 44*

```
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`:

Example № 45

```
{
  name
  age
  picture
}
```

Would yield the object:

Example № 46

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

While selecting a subset of fields:

Example № 47

```
{
  age
  name
}
```

Must only yield exactly that subset:

Example № 48

```
{
  "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`:

Example № 49

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

Valid operations must supply a nested field set for any field that returns an object, so this operation is not valid:

Counter Example № 50

```
{
  name
  relationship
}
```

However, this example is valid:

Example № 51

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

And will yield the subset of each object type queried:

Example № 52

```
{
  "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 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 (such as JSON) should retain this order textually. That is, if two fields `{foo, bar}` were queried in that order, the resulting JSON serialization should contain `{"foo": "...", "bar": "..."}` in the same order.

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.

Example № 53

```
{
  foo
  ...Frag
  qux
}

fragment Frag on Query {
  bar
  baz
}
```

Produces the ordered result:

Example № 54

```
{
  "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 do not affect ordering.

Example № 55

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

fragment Ignored on UnknownType {
  qux
  baz
}

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

Produces the ordered result:

Example № 56

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

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

Example № 57

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

Produces the ordered result:

Example № 58

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

Result Coercion

Determining the result of coercing an object is the heart of the GraphQL executor, see Value Completion.

Input Coercion

Objects are never valid inputs.

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. For each field of an Object type:
    1. The field must have a unique name within that Object type; no two fields may share the same name.
    2. The field must not have a name which begins with the characters "__" (two underscores).
    3. The field must return a type where IsOutputType(*fieldType*) returns **true**.
    4. For each argument of the field:
        1. The argument must not have a name which begins with the characters "__" (two underscores).
        2. The argument must have a unique name within that field; no two arguments may share the same name.
        3. The argument must accept a type where IsInputType(*argumentType*) returns **true**.
        4. If argument type is Non-Null and a default value is not defined:
            - The `@deprecated` directive must not be applied to this argument.
3. An object type may declare that it implements one or more unique interfaces.
4. An object type must be a super-set of all interfaces it implements:
    1. Let this object type be *objectType*.
    2. For each interface declared implemented as *interfaceType*, IsValidImplementation(*objectType*, *interfaceType*) must be **true**.

IsValidImplementation(*type*, *implementedType*) **:**

1. If *implementedType* declares it implements any interfaces, *type* must also declare it implements those interfaces.
2. *type* must include a field of the same name for every field defined in *implementedType*.
    a. Let *field* be that named field on *type*.
    b. Let *implementedField* be that named field on *implementedType*.

c. *field* must include an argument of the same name for every argument defined in *implementedField*.

i. That named argument on *field* must accept the same type (invariant) as that named argument on *implementedField*.

d. *field* may include additional arguments not defined in *implementedField*, but any additional argument must not be required, e.g. must not be of a non-nullable type.

e. *field* must return a type which is equal to or a sub-type of (covariant) the return type of *implementedField* field's return type:

i. Let *fieldType* be the return type of *field*.

ii. Let *implementedFieldType* be the return type of *implementedField*.

iii. IsValidImplementationFieldType(*fieldType*, *implementedFieldType*) must be **true**.

IsValidImplementationFieldType(*fieldType*, *implementedFieldType*) **:**

1. If *fieldType* is a Non-Null type:
   a. Let *nullableType* be the unwrapped nullable type of *fieldType*.
   b. Let *implementedNullableType* be the unwrapped nullable type of *implementedFieldType* if it is a Non-Null type, otherwise let it be *implementedFieldType* directly.
   c. Return IsValidImplementationFieldType(*nullableType*, *implementedNullableType*).
2. If *fieldType* is a List type and *implementedFieldType* is also a List type:
   a. Let *itemType* be the unwrapped item type of *fieldType*.
   b. Let *implementedItemType* be the unwrapped item type of *implementedFieldType*.
   c. Return IsValidImplementationFieldType(*itemType*, *implementedItemType*).
3. Return IsSubType(*fieldType*, *implementedFieldType*).

IsSubType(*possibleSubType*, *superType*) **:**

1. If *possibleSubType* is the same type as *superType* then return **true**.
2. If *possibleSubType* is an Object type and *superType* is a Union type and *possibleSubType* is a possible type of *superType* then return **true**.
3. If *possibleSubType* is an Object or Interface type and *superType* is an Interface type and *possibleSubType* declares it implements *superType* then return **true**.
4. Otherwise return **false**.

3.6.1 Field Arguments

ArgumentsDefinition **:**
(*InputValueDefinition*$_{list}$)

InputValueDefinition **:**
Description$_{opt}$ *Name* **:** *Type* *DefaultValue*$_{opt}$ *Directives*$_{[Const]opt}$

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.

All arguments defined within a field must not have a name which begins with "__" (two underscores), as this is used exclusively by GraphQL's introspection system.

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

Example № 59

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

Operations can optionally specify arguments to their fields to provide these arguments.

This example operation:

Example № 60

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

May return the result:

Example № 61

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

The type of an object field argument must be an input type (any type except an Object, Interface, or Union type).

3.6.2 Field Deprecation

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

When using the type system definition language, `@deprecated` directives are used to indicate that a field is deprecated:

Example № 62

```
type ExampleType {
  oldField: String @deprecated
}
```

3.6.3 Object Extensions

ObjectTypeExtension :

extend type *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ *FieldsDefinition*

extend type *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]}$ [lookahead ≠ **{**]

extend type *Name* *ImplementsInterfaces* [lookahead ≠ **{**]

Object type extensions are used to represent a type which has been extended from some original type. For example, this might be used to represent local data, or by a GraphQL service which is itself an extension of another GraphQL service.

In this example, a local data field is added to a `Story` type:

Example № 63

```
extend type Story {
  isHiddenLocally: Boolean
}
```

Object type extensions may choose not to add additional fields, instead only adding interfaces or directives.

In this example, a directive is added to a `User` type without adding fields:

Example № 64

```
extend type User @addedDirective
```

Type Validation

Object type extensions have the potential to be invalid if incorrectly defined.

1. The named type must already be defined and must be an Object type.
2. The fields of an Object type extension must have unique names; no two fields may share the same name.

3. Any fields of an Object type extension must not be already defined on the original Object type.
4. Any non-repeatable directives provided must not already apply to the original Object type.
5. Any interfaces provided must not be already implemented by the original Object type.
6. The resulting extended object type must be a super-set of all interfaces it implements.

3.7 Interfaces

InterfaceTypeDefinition :

Description$_{opt}$ **interface** *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ *FieldsDefinition*

Description$_{opt}$ **interface** *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ [lookahead ≠ **{**]

GraphQL interfaces represent a list of named fields and their arguments. GraphQL objects and interfaces can then implement these interfaces which requires that the implementing type will define all fields defined by those interfaces.

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 `NamedEntity` may describe a required field and types such as `Person` or `Business` may then implement this interface to guarantee this field will always exist.

Types may also implement multiple interfaces. For example, `Business` implements both the `NamedEntity` and `ValuedEntity` interfaces in the example below.

Example № 65

```
interface NamedEntity {
  name: String
}

interface ValuedEntity {
  value: Int
}

type Person implements NamedEntity {
  name: String
  age: Int
}

type Business implements NamedEntity & ValuedEntity {
  name: String
  value: Int
```

```
  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`.

Example № 66

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

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

Example № 67

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

When selecting 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 selection set against `Contact`:

Counter Example № 68

```
{
  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`; this can be expressed using a fragment or an inline fragment:

Example № 69

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

Interfaces Implementing Interfaces

When defining an interface that implements another interface, the implementing interface must define each field that is specified by the implemented interface. For example, the interface Resource must define the field id to implement the Node interface:

Example № 70

```
interface Node {
  id: ID!
}

interface Resource implements Node {
  id: ID!
  url: String
}
```

Transitively implemented interfaces (interfaces implemented by the interface that is being implemented) must also be defined on an implementing type or interface. For example, `Image` cannot implement `Resource` without also implementing `Node`:

Example № 71

```
interface Node {
  id: ID!
}

interface Resource implements Node {
  id: ID!
  url: String
}

interface Image implements Resource & Node {
  id: ID!
  url: String
```

```
  thumbnail: String
}
```

Interface definitions must not contain cyclic references nor implement themselves. This example is invalid because `Node` and `Named` implement themselves and each other:

Counter Example № 72

```
interface Node implements Named & Node {
  id: ID!
  name: String
}

interface Named implements Node & Named {
  id: ID!
  name: String
}
```

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.

Type Validation

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

1. An Interface type must define one or more fields.
2. For each field of an Interface type:
    1. The field must have a unique name within that Interface type; no two fields may share the same name.
    2. The field must not have a name which begins with the characters "__" (two underscores).
    3. The field must return a type where IsOutputType(*fieldType*) returns **true**.
    4. For each argument of the field:
        1. The argument must not have a name which begins with the characters "__" (two underscores).
        2. The argument must have a unique name within that field; no two arguments may share the same name.

3. The argument must accept a type where IsInputType(*argumentType*) returns **true**.

3. An interface type may declare that it implements one or more unique interfaces, but may not implement itself.
4. An interface type must be a super-set of all interfaces it implements:
   1. Let this interface type be *implementingType*.
   2. For each interface declared implemented as *implementedType*, IsValidImplementation(*implementingType*, *implementedType*) must be **true**.

3.7.1 Interface Extensions

InterfaceTypeExtension :

extend interface *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ *FieldsDefinition*

extend interface *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]}$ [lookahead ≠ **{**]

extend interface *Name* *ImplementsInterfaces* [lookahead ≠ **{**]

Interface type extensions are used to represent an interface which has been extended from some original interface. For example, this might be used to represent common local data on many types, or by a GraphQL service which is itself an extension of another GraphQL service.

In this example, an extended data field is added to a `NamedEntity` type along with the types which implement it:

Example № 73

```
extend interface NamedEntity {
  nickname: String
}

extend type Person {
  nickname: String
}

extend type Business {
  nickname: String
}
```

Interface type extensions may choose not to add additional fields, instead only adding directives.

In this example, a directive is added to a `NamedEntity` type without adding fields:

Example № 74

```
extend interface NamedEntity @addedDirective
```

Type Validation

Interface type extensions have the potential to be invalid if incorrectly defined.

1. The named type must already be defined and must be an Interface type.
2. The fields of an Interface type extension must have unique names; no two fields may share the same name.
3. Any fields of an Interface type extension must not be already defined on the original Interface type.
4. Any Object or Interface type which implemented the original Interface type must also be a super-set of the fields of the Interface type extension (which may be due to Object type extension).
5. Any non-repeatable directives provided must not already apply to the original Interface type.
6. The resulting extended Interface type must be a super-set of all Interfaces it implements.

3.8 Unions

UnionTypeDefinition :
  Description$_{opt}$ **union** *Name* *Directives*$_{[Const]opt}$ *UnionMemberTypes*$_{opt}$

UnionMemberTypes :
  UnionMemberTypes **|** *NamedType*
  = **|**$_{opt}$ *NamedType*

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 type refining fragments or inline fragments (with the exception of the meta-field *__typename*).

For example, we might define the following types:

Example № 75

```
union SearchResult = Photo | Person

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

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

type SearchQuery {
  firstSearchResult: SearchResult
}
```

In this example, a query operation wants the name if the result was a Person, and the height if it was a photo. However because a union itself defines no fields, this could be ambiguous and is invalid.

Counter Example № 76

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

A valid operation includes typed fragments (in this example, inline fragments):

Example № 77

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

Union members may be defined with an optional leading `|` character to aid formatting when representing a longer list of possible types:

Example № 78

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

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.

Type Validation

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

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

3.8.1 Union Extensions

UnionTypeExtension :

extend union *Name* *Directives*$_{[Const]opt}$ *UnionMemberTypes*

extend union *Name* *Directives*$_{[Const]}$

Union type extensions are used to represent a union type which has been extended from some original union type. For example, this might be used to represent additional local data, or by a GraphQL service which is itself an extension of another GraphQL service.

Type Validation

Union type extensions have the potential to be invalid if incorrectly defined.

1. The named type must already be defined and must be a Union type.
2. The member types of a Union type extension must all be Object base types; Scalar, Interface and Union types must not be member types of a Union. Similarly, wrapping types must not be member types of a Union.
3. All member types of a Union type extension must be unique.
4. All member types of a Union type extension must not already be a member of the original Union type.
5. Any non-repeatable directives provided must not already apply to the original Union type.

3.9 Enums

EnumTypeDefinition :

Description$_{opt}$ **enum** *Name* *Directives*$_{[Const]opt}$ *EnumValuesDefinition*

Description$_{opt}$ **enum** *Name* *Directives*$_{[Const]opt}$ [lookahead ≠ **{**]

EnumValuesDefinition :

{ *EnumValueDefinition*$_{list}$ **}**

EnumValueDefinition :

Description$_{opt}$ *EnumValue* *Directives*$_{[Const]opt}$

GraphQL Enum types, like Scalar types, also represent leaf values in a GraphQL type system. However Enum types describe the set of possible values.

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

In this example, an Enum type called `Direction` is defined:

Example № 79

```
enum Direction {
  NORTH
  EAST
  SOUTH
  WEST
}
```

Result Coercion

GraphQL services 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 request error.

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.

Type Validation

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

1. An Enum type must define one or more unique enum values.

3.9.1 Enum Extensions

EnumTypeExtension :
  extend enum *Name* *Directives*$_{[Const]opt}$ *EnumValuesDefinition*
  extend enum *Name* *Directives*$_{[Const]}$ [lookahead ≠ **{**]

Enum type extensions are used to represent an enum type which has been extended from some original enum type. For example, this might be used to represent additional local data, or by a GraphQL service which is itself an extension of another GraphQL service.

Type Validation

Enum type extensions have the potential to be invalid if incorrectly defined.

1. The named type must already be defined and must be an Enum type.
2. All values of an Enum type extension must be unique.
3. All values of an Enum type extension must not already be a value of the original Enum.
4. Any non-repeatable directives provided must not already apply to the original Enum type.

3.10 Input Objects

InputObjectTypeDefinition :
  Description$_{opt}$ **input** *Name* *Directives*$_{[Const]opt}$ *InputFieldsDefinition*
  Description$_{opt}$ **input** *Name* *Directives*$_{[Const]opt}$ [lookahead ≠ **{**]

InputFieldsDefinition :
  { *InputValueDefinition*$_{list}$ **}**

Fields may accept arguments to configure their behavior. These inputs are often scalars or enums, but they sometimes need to represent more complex values.

A GraphQL 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.

In this example, an Input Object called `Point2D` describes `x` and `y` inputs:

> *Example № 80*

```
input Point2D {
  x: Float
  y: Float
}
```

Note

The GraphQL Object type (*ObjectTypeDefinition*) defined above is inappropriate for re-use here, because Object types can contain fields that define arguments or contain 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.

Circular References

Input Objects are allowed to reference other Input Objects as field types. A circular reference occurs when an Input Object references itself either directly or through referenced Input Objects.

Circular references are generally allowed, however they may not be defined as an unbroken chain of Non-Null singular fields. Such Input Objects are invalid because there is no way to provide a legal value for them.

This example of a circularly-referenced input type is valid as the field `self` may be omitted or the value **null**.

Example № 81

```
input Example {
  self: Example
  value: String
}
```

This example is also valid as the field `self` may be an empty List.

Example № 82

```
input Example {
  self: [Example!]!
  value: String
}
```

This example of a circularly-referenced input type is invalid as the field `self` cannot be provided a finite value.

Counter Example № 83

```
input Example {
  value: String
```

```
  self: Example!
}
```

This example is also invalid, as there is a non-null singular circular reference via the `First.second` and `Second.first` fields.

Counter Example № 84

```
input First {
  second: Second!
  value: String
}

input Second {
  first: First!
  value: String
}
```

Result Coercion

An input object is never a valid result. Input Object types cannot be the return type of an Object or Interface field.

Input Coercion

The value for an input object should be an input object literal or an unordered map supplied by a variable, otherwise a *request error* must be raised. In either case, the input object literal or unordered map must not contain any entries with names not defined by a field of this input object type, otherwise a response error must be raised.

The result of coercion is an unordered map with an entry for each field both defined by the input object type and for which a value exists. The resulting map is constructed with the following rules:

- If no value is provided for a defined input object field and that field definition provides a default value, the default value should be used. If no default value is provided and the input object field's type is non-null, an error should be raised. Otherwise, if the field is not required, then no entry is added to the coerced unordered map.
- If the value **null** was provided for an input object field, and the field's type is not a non-null type, an entry in the coerced unordered map is given the value **null**. In other words, there is a semantic difference between the explicitly provided value **null** versus having not provided a value.
- If a literal value is provided for an input object field, an entry in the coerced unordered map is given the result of coercing that value according to the input coercion rules for the type of that field.
- If a variable is provided for an input object field, the runtime value of that variable must be used. If the runtime value is **null** and the field type is non-null, a *field error* must be raised. If no runtime value is

provided, the variable definition's default value should be used. If the variable definition does not provide a default value, the input object field definition's default value should be used.

Following are examples of input coercion for an input object type with a `String` field `a` and a required (non-null) `Int!` field `b`:

Example № 85

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

Literal Value	Variables	Coerced Value
`{ a: "abc", b: 123 }`	`{}`	`{ a: "abc", b: 123 }`
`{ a: null, b: 123 }`	`{}`	`{ a: null, b: 123 }`
`{ b: 123 }`	`{}`	`{ b: 123 }`
`{ a: $var, b: 123 }`	`{ var: null }`	`{ a: null, b: 123 }`
`{ a: $var, b: 123 }`	`{}`	`{ b: 123 }`
`{ b: $var }`	`{ var: 123 }`	`{ b: 123 }`
`$var`	`{ var: { b: 123 } }`	`{ b: 123 }`
`"abc123"`	`{}`	Error: Incorrect value
`$var`	`{ var: "abc123" }`	Error: Incorrect value
`{ a: "abc", b: "123" }`	`{}`	Error: Incorrect value for field *b*
`{ a: "abc" }`	`{}`	Error: Missing required field *b*
`{ b: $var }`	`{}`	Error: Missing required field *b*.
`$var`	`{ var: { a: "abc" } }`	Error: Missing required field *b*
`{ a: "abc", b: null }`	`{}`	Error: *b* must be non-null.
`{ b: $var }`	`{ var: null }`	Error: *b* must be non-null.
`{ b: 123, c: "xyz" }`	`{}`	Error: Unexpected field *c*

Type Validation

1. An Input Object type must define one or more input fields.
2. For each input field of an Input Object type:

1. The input field must have a unique name within that Input Object type; no two input fields may share the same name.
2. The input field must not have a name which begins with the characters "`__`" (two underscores).
3. The input field must accept a type where IsInputType(*inputFieldType*) returns **true**.
4. If input field type is Non-Null and a default value is not defined:
   - The `@deprecated` directive must not be applied to this input field.

3. If an Input Object references itself either directly or through referenced Input Objects, at least one of the fields in the chain of references must be either a nullable or a List type.

3.10.1 Input Object Extensions

InputObjectTypeExtension :
`extend input` *Name* *Directives*$_{[Const]opt}$ *InputFieldsDefinition*
`extend input` *Name* *Directives*$_{[Const]}$ [lookahead ≠ **`{`**]

Input object type extensions are used to represent an input object type which has been extended from some original input object type. For example, this might be used by a GraphQL service which is itself an extension of another GraphQL service.

Type Validation

Input object type extensions have the potential to be invalid if incorrectly defined.

1. The named type must already be defined and must be a Input Object type.
2. All fields of an Input Object type extension must have unique names.
3. All fields of an Input Object type extension must not already be a field of the original Input Object.
4. Any non-repeatable directives provided must not already apply to the original Input Object type.

3.11 List

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]`. Nesting lists is allowed: `matrix: [[Int]]`.

Result Coercion

GraphQL services 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 it 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.

If a list's item type is nullable, then errors occurring during preparation or coercion of an individual item in the list must result in a the value **null** at that position in the list along with a *field error* added to the response. If a list's item type is non-null, a field error occurring at an individual item in the list must result in a field error for the entire list.

> *Note*
> See Handling Field Errors for more about this behavior.

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, then the result of input coercion is a list of size one, where the single item value is the result of input coercion for the list's item type on the provided value (note this may apply recursively for nested lists).

This allows inputs which accept one or many arguments (sometimes referred to as "var args") to declare their input type as a list while for the common case of a single value, a client can just pass that value directly rather than constructing the list.

Following are examples of input coercion with various list types and values:

Expected Type	Provided Value	Coerced Value
`[Int]`	`[1, 2, 3]`	`[1, 2, 3]`
`[Int]`	`[1, "b", true]`	Error: Incorrect item value
`[Int]`	`1`	`[1]`
`[Int]`	`null`	`null`
`[[Int]]`	`[[1], [2, 3]]`	`[[1], [2, 3]]`
`[[Int]]`	`[1, 2, 3]`	Error: Incorrect item value
`[[Int]]`	`1`	`[[1]]`
`[[Int]]`	`null`	`null`

3.12 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 selection set, a field may be omitted and the selection set 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.

> *Note*
> When a *field error* is raised on a non-null value, the error propagates to the parent field. For more information on this process, see Errors and Non-Null Fields within the Execution section.

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 *request 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.

A non-null argument cannot be omitted:

Counter Example № 86

```
{
  fieldWithNonNullArg
}
```

The value **null** cannot be provided to a non-null argument:

Counter Example № 87

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

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

Example № 88

```
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.

Type Validation

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

3.12.1 Combining List and Non-Null

The List and Non-Null wrapping types can compose, representing more complex types. The rules for result coercion and input coercion of Lists and Non-Null types apply in a recursive fashion.

For example if the inner item type of a List is Non-Null (e.g. `[T!]`), then that List may not contain any **null** items. However if the inner type of a Non-Null is a List (e.g. `[T]!`), then **null** is not accepted however an empty list is accepted.

Following are examples of result coercion with various types and values:

Expected Type	Internal Value	Coerced Result
`[Int]`	`[1, 2, 3]`	`[1, 2, 3]`

Expected Type	Internal Value	Coerced Result
`[Int]`	`null`	`null`
`[Int]`	`[1, 2, null]`	`[1, 2, null]`
`[Int]`	`[1, 2, Error]`	`[1, 2, null]` (With logged error)
`[Int]!`	`[1, 2, 3]`	`[1, 2, 3]`
`[Int]!`	`null`	Error: Value cannot be null
`[Int]!`	`[1, 2, null]`	`[1, 2, null]`
`[Int]!`	`[1, 2, Error]`	`[1, 2, null]` (With logged error)
`[Int!]`	`[1, 2, 3]`	`[1, 2, 3]`
`[Int!]`	`null`	`null`
`[Int!]`	`[1, 2, null]`	`null` (With logged coercion error)
`[Int!]`	`[1, 2, Error]`	`null` (With logged error)
`[Int!]!`	`[1, 2, 3]`	`[1, 2, 3]`
`[Int!]!`	`null`	Error: Value cannot be null
`[Int!]!`	`[1, 2, null]`	Error: Item cannot be null
`[Int!]!`	`[1, 2, Error]`	Error: Error occurred in item

3.13 Directives

DirectiveDefinition :
Description$_{opt}$ **directive** **@** *Name* *ArgumentsDefinition*$_{opt}$ **repeatable**$_{opt}$ **on** *DirectiveLocations*

DirectiveLocations :
DirectiveLocations **|** *DirectiveLocation*
|$_{opt}$ *DirectiveLocation*

DirectiveLocation :
ExecutableDirectiveLocation
TypeSystemDirectiveLocation

ExecutableDirectiveLocation : **one of**
QUERY
MUTATION

- SUBSCRIPTION
- FIELD
- FRAGMENT_DEFINITION
- FRAGMENT_SPREAD
- INLINE_FRAGMENT
- VARIABLE_DEFINITION

TypeSystemDirectiveLocation **: one of**

- SCHEMA
- SCALAR
- OBJECT
- FIELD_DEFINITION
- ARGUMENT_DEFINITION
- INTERFACE
- UNION
- ENUM
- ENUM_VALUE
- INPUT_OBJECT
- INPUT_FIELD_DEFINITION

A GraphQL schema describes directives which are used to annotate various parts of a GraphQL document as an indicator that they should be evaluated differently by a validator, executor, or client tool such as a code generator.

Built-in Directives

A *built-in directive* is any directive defined within this specification.

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

GraphQL implementations that support the type system definition language must provide the `@deprecated` directive if representing deprecated portions of the schema.

GraphQL implementations that support the type system definition language should provide the `@specifiedBy` directive if representing custom scalar definitions.

When representing a GraphQL schema using the type system definition language any *built-in directive* may be omitted for brevity.

When introspecting a GraphQL service all provided directives, including any *built-in directive*, must be included in the set of returned directives.

Custom Directives

GraphQL services and client tooling may provide any additional *custom directive* beyond those defined in this document. Directives are the preferred way to extend GraphQL with custom or experimental behavior.

> *Note*
> When defining a *custom directive*, it is recommended to prefix the directive's name to make its scope of usage clear and to prevent a collision with *built-in directive* which may be specified by future versions of this document (which will not include `_` in their name). For example, a *custom directive* used by Facebook's GraphQL service should be named `@fb_auth` instead of `@auth`. This is especially recommended for proposed additions to this specification which can change during the RFC process. For example a work in progress version of `@live` should be named `@rfc_live`.

Directives must only be used in the locations they are declared to belong in. In this example, a directive is defined which can be used to annotate a field:

Example № 89

```
directive @example on FIELD

fragment SomeFragment on SomeType {
  field @example
}
```

Directive locations may be defined with an optional leading `|` character to aid formatting when representing a longer list of possible locations:

Example № 90

```
directive @example on
  | FIELD
  | FRAGMENT_SPREAD
  | INLINE_FRAGMENT
```

Directives can also be used to annotate the type system definition language as well, which can be a useful tool for supplying additional metadata in order to generate GraphQL execution services, produce client generated runtime code, or many other useful extensions of the GraphQL semantics.

In this example, the directive `@example` annotates field and argument definitions:

Example № 91

```
directive @example on FIELD_DEFINITION | ARGUMENT_DEFINITION

type SomeType {
  field(arg: Int @example): String @example
}
```

A directive may be defined as repeatable by including the "repeatable" keyword. Repeatable directives are often useful when the same directive should be used with different arguments at a single location, especially in cases where additional information needs to be provided to a type or schema extension via a directive:

Example № 92

```
directive @delegateField(name: String!) repeatable on OBJECT | INTERFACE

type Book @delegateField(name: "pageCount") @delegateField(name: "author") {
  id: ID!
}

extend type Book @delegateField(name: "index")
```

While defining a directive, it must not reference itself directly or indirectly:

Counter Example № 93

```
directive @invalidExample(arg: String @invalidExample) on ARGUMENT_DEFINITION
```

Note

The order in which directives appear may be significant, including repeatable directives.

Validation

1. A directive definition must not contain the use of a directive which references itself directly.
2. A directive definition must not contain the use of a directive which references itself indirectly by referencing a Type or Directive which transitively includes a reference to this directive.
3. The directive must not have a name which begins with the characters "__" (two underscores).
4. For each argument of the directive:
    1. The argument must not have a name which begins with the characters "__" (two underscores).
    2. The argument must have a unique name within that directive; no two arguments may share the same name.
    3. The argument must accept a type where IsInputType(*argumentType*) returns **true**.

3.13.1 @skip

```
directive @skip(if: Boolean!) on FIELD | FRAGMENT_SPREAD | INLINE_FRAGMENT
```

The `@skip` *built-in 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 only be queried if the variable `$someTest` has the value `false`.

Example № 94

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

3.13.2 @include

```
directive @include(if: Boolean!) on FIELD | FRAGMENT_SPREAD | INLINE_FRAGMENT
```

The `@include` *built-in 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 only be queried if the variable `$someTest` has the value `true`

Example № 95

```
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 on the same 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.

3.13.3 @deprecated

```
directive @deprecated(
  reason: String = "No longer supported"
) on FIELD_DEFINITION | ARGUMENT_DEFINITION | INPUT_FIELD_DEFINITION | ENUM_VALUE
```

The `@deprecated` *built-in directive* is used within the type system definition language to indicate deprecated portions of a GraphQL service's schema, such as deprecated fields on a type, arguments on a

field, input fields on an input type, or values of an enum type.

Deprecations include a reason for why it is deprecated, which is formatted using Markdown syntax (as specified by CommonMark).

In this example type definition, `oldField` is deprecated in favor of using `newField` and `oldArg` is deprecated in favor of using `newArg`.

Example № 96

```
type ExampleType {
  newField: String
  oldField: String @deprecated(reason: "Use `newField`.")

  anotherField(
    newArg: String
    oldArg: String @deprecated(reason: "Use `newArg`.")
  ): String
}
```

The `@deprecated` directive must not appear on required (non-null without a default) arguments or input object field definitions.

Counter Example № 97

```
type ExampleType {
  invalidField(
    newArg: String
    oldArg: String! @deprecated(reason: "Use `newArg`.")
  ): String
}
```

To deprecate a required argument or input field, it must first be made optional by either changing the type to nullable or adding a default value.

3.13.4 @specifiedBy

```
directive @specifiedBy(url: String!) on SCALAR
```

The `@specifiedBy` *built-in directive* is used within the type system definition language to provide a *scalar specification URL* for specifying the behavior of custom scalar types. The URL should point to a human-readable specification of the data format, serialization, and coercion rules. It must not appear on built-in scalar types.

Note

> Details on implementing a GraphQL scalar specification can be found in the scalars.graphql.org implementation guide.

In this example, a custom scalar type for `UUID` is defined with a URL pointing to the relevant IETF specification.

Example № 98

```
scalar UUID @specifiedBy(url: "https://tools.ietf.org/html/rfc4122")
```

4 Introspection

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

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

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

Example № 99

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

A request containing the operation:

Example № 100

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

would produce the result:

Example № 101

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

Reserved Names

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.

Otherwise, any *Name* within a GraphQL type system must not start with two underscores "__".

4.1 Type Name Introspection

GraphQL supports type name introspection within any selection set in an operation, with the single exception of selections at the root of a subscription operation. Type name introspection is accomplished via the meta-field `__typename: String!` on any Object, Interface, or Union. It returns the name of the concrete Object type at that point during execution.

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

As a meta-field, `__typename` is implicit and does not appear in the fields list in any defined type.

> *Note*
> `__typename` may not be included as a root field in a subscription operation.

4.2 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
```

Like all meta-fields, these are implicit and do not appear in the fields list in the root type of the query operation.

First Class 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 service may return the `description` field using Markdown syntax (as specified by CommonMark). Therefore it is recommended that any tool that displays `description` use a CommonMark-compliant Markdown renderer.

Deprecation

To support the management of backwards compatibility, GraphQL fields, arguments, input fields, and enum values can indicate whether or not they are deprecated (`isDeprecated: Boolean`) along with 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.

Schema Introspection Schema

The schema introspection system is itself represented as a GraphQL schema. Below are the full set of type system definitions providing schema introspection, which are fully defined in the sections below.

```
type __Schema {
  description: String
  types: [__Type!]!
  queryType: __Type!
```

```
  mutationType: __Type
  subscriptionType: __Type
  directives: [__Directive!]!
}

type __Type {
  kind: __TypeKind!
  name: String
  description: String
  # must be non-null for OBJECT and INTERFACE, otherwise null.
  fields(includeDeprecated: Boolean = false): [__Field!]
  # must be non-null for OBJECT and INTERFACE, otherwise null.
  interfaces: [__Type!]
  # must be non-null for INTERFACE and UNION, otherwise null.
  possibleTypes: [__Type!]
  # must be non-null for ENUM, otherwise null.
  enumValues(includeDeprecated: Boolean = false): [__EnumValue!]
  # must be non-null for INPUT_OBJECT, otherwise null.
  inputFields(includeDeprecated: Boolean = false): [__InputValue!]
  # must be non-null for NON_NULL and LIST, otherwise null.
  ofType: __Type
  # may be non-null for custom SCALAR, otherwise null.
  specifiedByURL: String
}

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

type __Field {
  name: String!
  description: String
  args(includeDeprecated: Boolean = false): [__InputValue!]!
  type: __Type!
  isDeprecated: Boolean!
  deprecationReason: String
}

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

```
  isDeprecated: Boolean!
  deprecationReason: String
}

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

type __Directive {
  name: String!
  description: String
  locations: [__DirectiveLocation!]!
  args(includeDeprecated: Boolean = false): [__InputValue!]!
  isRepeatable: Boolean!
}

enum __DirectiveLocation {
  QUERY
  MUTATION
  SUBSCRIPTION
  FIELD
  FRAGMENT_DEFINITION
  FRAGMENT_SPREAD
  INLINE_FRAGMENT
  VARIABLE_DEFINITION
  SCHEMA
  SCALAR
  OBJECT
  FIELD_DEFINITION
  ARGUMENT_DEFINITION
  INTERFACE
  UNION
  ENUM
  ENUM_VALUE
  INPUT_OBJECT
  INPUT_FIELD_DEFINITION
}
```

4.2.1 The __Schema Type

The `__Schema` type is returned from the `__schema` meta-field and provides all information about the schema of a GraphQL service.

Fields:

- `description` may return a String or **null**.
- `queryType` is the root type of a query operation.
- `mutationType` is the root type of a mutation operation, if supported. Otherwise **null**.
- `subscriptionType` is the root type of a subscription operation, if supported. Otherwise **null**.
- `types` must return the set of all named types contained within this schema. Any named type which can be found through a field of any introspection type must be included in this set.
- `directives` must return the set of all directives available within this schema including all built-in directives.

4.2.2 The __Type Type

`__Type` is at the core of the type introspection system, it represents all types in the system: both named types (e.g. Scalars and Object types) and type modifiers (e.g. List and Non-Null types).

Type modifiers are used to modify the type presented in the field `ofType`. This modified type may recursively be a modified type, representing lists, non-nullables, and combinations thereof, ultimately modifying a named type.

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

Each sub-section below defines the expected fields of `__Type` given each possible value of the `__TypeKind` enum:

- `"SCALAR"`
- `"OBJECT"`
- `"INTERFACE"`
- `"UNION"`
- `"ENUM"`
- `"INPUT_OBJECT"`
- `"LIST"`
- `"NON_NULL"`

Scalar

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

Also represents Custom scalars which may provide `specifiedByURL` as a *scalar specification URL*.

Fields:

- `kind` must return `__TypeKind.SCALAR`.
- `name` must return a String.
- `description` may return a String or **null**.
- `specifiedByURL` may return a String (in the form of a URL) for custom scalars, otherwise must be **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` must return the set of fields that can be selected for this type.
  - Accepts the argument `includeDeprecated` which defaults to **false**. If **true**, deprecated fields are also returned.
- `interfaces` must return the set of interfaces that an object implements (if none, `interfaces` must return the empty set).
- 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 named fields where each implementing field type is equal to or a sub-type of (covariant) the interface type. 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` must return the set of fields required by this interface.
  - Accepts the argument `includeDeprecated` which defaults to **false**. If **true**, deprecated fields are also returned.
- `interfaces` must return the set of interfaces that an interface implements (if none, `interfaces` must return the empty set).
- `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` must return the set of enum values as a 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 defined as a list of named input values. They are only used as inputs to arguments and variables and cannot be a field return type.

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

> *Example № 102*

```
input 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` must return the set of input fields as a list of `__InputValue`.
  - Accepts the argument `includeDeprecated` which defaults to **false**. If **true**, deprecated input fields are also returned.
- 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.

The modified type in the `ofType` field may itself be a modified type, allowing the representation of Lists of Lists, or Lists of Non-Nulls.

Fields:

- `kind` must return `__TypeKind.LIST`.
- `ofType` must return a type of any kind.
- 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.

The modified type in the `ofType` field may itself be a modified List type, allowing the representation of Non-Null of Lists. However it must not be a modified Non-Null type to avoid a redundant Non-Null of Non-Null.

Fields:

- `kind` must return `__TypeKind.NON_NULL`.
- `ofType` must return a type of any kind except Non-Null.

- All other fields must return **null**.

4.2.3 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.
  - Accepts the argument `includeDeprecated` which defaults to **false**. If **true**, deprecated arguments are also returned.
- `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.

4.2.4 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**.
- `isDeprecated` returns **true** if this input field or argument should no longer be used, otherwise **false**.
- `deprecationReason` optionally provides a reason why this input field or argument is deprecated.

4.2.5 The __EnumValue Type

The `__EnumValue` type represents one of possible values of an enum.

Fields:

- `name` must return a String
- `description` may return a String or **null**
- `isDeprecated` returns **true** if this enum value should no longer be used, otherwise **false**.
- `deprecationReason` optionally provides a reason why this enum value is deprecated.

4.2.6 The __Directive Type

The `__Directive` type represents a directive that a service supports.

This includes both any *built-in directive* and any *custom directive*.

Individual directives may only be used in locations that are explicitly supported. All possible locations are listed in the `__DirectiveLocation` enum:

- `"QUERY"`
- `"MUTATION"`
- `"SUBSCRIPTION"`
- `"FIELD"`
- `"FRAGMENT_DEFINITION"`
- `"FRAGMENT_SPREAD"`
- `"INLINE_FRAGMENT"`
- `"VARIABLE_DEFINITION"`
- `"SCHEMA"`
- `"SCALAR"`
- `"OBJECT"`
- `"FIELD_DEFINITION"`
- `"ARGUMENT_DEFINITION"`
- `"INTERFACE"`
- `"UNION"`
- `"ENUM"`
- `"ENUM_VALUE"`
- `"INPUT_OBJECT"`
- `"INPUT_FIELD_DEFINITION"`

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.
  - Accepts the argument `includeDeprecated` which defaults to **false**. If **true**, deprecated arguments are also returned.
- `isRepeatable` must return a Boolean that indicates if the directive may be used repeatedly at a single location.

5 Validation

A GraphQL service 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 algorithms in the Execution section, however that result may be ambiguous, surprising, or unexpected relative to a request containing validation errors, so execution should only occur for valid 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 evolves 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:

Example № 103

```
type Query {
  dog: Dog
  findDog(searchBy: FindDogInput): Dog
}

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!
  pets: [Pet!]
}

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

input FindDogInput {
  name: String
  owner: String
}
```

5.1 Documents

5.1.1 Executable Definitions

Formal Specification

- For each definition *definition* in the document:
  - *definition* must be *ExecutableDefinition* (it must not be *TypeSystemDefinitionOrExtension*).

Explanatory Text

GraphQL execution will only consider the executable definitions Operation and Fragment. Type system definitions and extensions are not executable, and are not considered during execution.

To avoid ambiguity, a document containing *TypeSystemDefinitionOrExtension* is invalid for execution.

GraphQL documents not intended to be directly executed may include *TypeSystemDefinitionOrExtension*.

For example, the following document is invalid for execution since the original executing schema may not know about the provided type extension:

Counter Example № 104

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

```
    color
  }
}

extend type Dog {
  color: String
}
```

5.2 Operations

5.2.1 Named Operation Definitions

5.2.1.1 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:

Example № 105

```
query getDogName {
  dog {
    name
  }
}

query getOwnerName {
  dog {
```

```
    owner {
      name
    }
  }
}
```

While this document is invalid:

Counter Example № 106

```
query getName {
  dog {
    name
  }
}

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

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

Counter Example № 107

```
query dogOperation {
  dog {
    name
  }
}

mutation dogOperation {
  mutateDog {
    id
  }
}
```

5.2.2 Anonymous Operation Definitions

5.2.2.1 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:

Example № 108

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

While this document is invalid:

Counter Example № 109

```
{
  dog {
    name
  }
}

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

5.2.3 Subscription Operation Definitions

5.2.3.1 Single Root Field

Formal Specification

- Let *subscriptionType* be the root Subscription type in *schema*.
- For each subscription operation definition *subscription* in the document:
  - Let *selectionSet* be the top level selection set on *subscription*.
  - Let *variableValues* be the empty set.
  - Let *groupedFieldSet* be the result of CollectFields(*subscriptionType*, *selectionSet*, *variableValues*).
  - *groupedFieldSet* must have exactly one entry, which must not be an introspection field.

Explanatory Text

Subscription operations must have exactly one root field.

Valid examples:

Example № 110

```
subscription sub {
  newMessage {
    body
    sender
  }
}
```

Example № 111

```
subscription sub {
  ...newMessageFields
}

fragment newMessageFields on Subscription {
  newMessage {
    body
    sender
  }
}
```

Invalid:

Counter Example № 112

```
subscription sub {
  newMessage {
    body
    sender
  }
  disallowedSecondRootField
}
```

Counter Example № 113

```
subscription sub {
  ...multipleSubscriptions
}

fragment multipleSubscriptions on Subscription {
  newMessage {
    body
    sender
  }
  disallowedSecondRootField
}
```

The root field of a subscription operation must not be an introspection field. The following example is also invalid:

Counter Example № 114

```
subscription sub {
  __typename
}
```

Note

While each subscription must have exactly one root field, a document may contain any number of operations, each of which may contain different root fields. When executed, a document containing multiple subscription operations must provide the operation name as described in GetOperation().

5.3 Fields

5.3.1 Field Selections

Field selections must exist on Object, Interface, and Union 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:

Counter Example № 115

```
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:

Example № 116

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

and the following is invalid:

Counter Example № 117

```
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:

Example № 118

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

But the following is invalid:

Counter Example № 119

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

5.3.2 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. If *typeA* or *typeB* is nullable, return false.
   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. If *typeA* or *typeB* is not List, return false.
   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. If *typeA* and *typeB* are the same type return true, otherwise return false.
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. If SameResponseShape(*subfieldA*, *subfieldB*) is false, return false.
10. Return 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.

During execution, the simultaneous execution of fields with the same response name is accomplished by MergeSelectionSets() and CollectFields().

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:

Example № 120

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

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

The following is not able to merge:

Counter Example № 121

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

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

For example the following correctly merge:

Example № 122

```
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:

Counter Example № 123

```
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:

Example № 124

```
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, leaf types must not differ. In this example, `someValue` might be a `String` or an `Int`:

Counter Example № 125

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

5.3.3 Leaf Field Selections

Formal Specification

- For each *selection* in the document:
  - Let *selectionType* be the unwrapped result type of *selection*.
  - If *selectionType* is a scalar or enum:
    - 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

A field subselection is not allowed on leaf fields. A leaf field is any field with a scalar or enum unwrapped type.

The following is valid.

Example № 126

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

The following is invalid.

Counter Example № 127

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

Conversely, non-leaf fields must have a field subselection. A non-leaf field is any field with an object, interface, or union unwrapped type.

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

Example № 128

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

The following examples are invalid because they include non-leaf fields without a field subselection.

Counter Example № 129

```
query directQueryOnObjectWithoutSubFields {
  human
}

query directQueryOnInterfaceWithoutSubFields {
  pet
}

query directQueryOnUnionWithoutSubFields {
  catOrDog
}
```

However the following example is valid since it includes a field subselection.

Example № 130

```
query directQueryOnObjectWithSubFields {
  human {
    name
  }
}
```

5.4 Arguments

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

5.4.1 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:

Example № 131

```
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`.

Counter Example № 132

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

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

Counter Example № 133

```
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:

Example № 134

```
type Arguments {
  multipleRequirements(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]
  optionalNonNullBooleanArgField(optionalBooleanArg: Boolean! = false): Boolean!
}

extend type Query {
```

```
  arguments: Arguments
}
```

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

Example № 135

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

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

5.4.2 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*.

5.4.2.1 Required 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 *argumentDefinition* in *argumentDefinitions*:
    - Let *type* be the expected type of *argumentDefinition*.
    - Let *defaultValue* be the default value of *argumentDefinition*.
    - If *type* is Non-Null and *defaultValue* does not exist:
      - Let *argumentName* be the name of *argumentDefinition*.
      - 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. An argument is required if the argument type is non-null and does not have a default value. Otherwise, the argument is optional.

For example the following are valid:

Example № 136

```
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 fragment is valid:

Example № 137

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

but this is not valid on a required argument.

Counter Example № 138

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

Providing the explicit value **null** is also not valid since required arguments always have a non-null type.

Counter Example № 139

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

5.5 Fragments

5.5.1 Fragment Declarations

5.5.1.1 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:

Example № 140

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

fragment fragmentOne on Dog {
  name
}

fragment fragmentTwo on Dog {
  owner {
```

```
    name
  }
}
```

While this document is invalid:

Counter Example № 141

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

fragment fragmentOne on Dog {
  name
}

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

5.5.1.2 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 fragment is invalid.

For example the following fragments are valid:

Example № 142

```
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:

Counter Example № 143

```
fragment notOnExistingType on NotInSchema {
  name
}

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

5.5.1.3 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:

Example № 144

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

```
fragment fragOnInterface on Pet {
  name
}

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

and the following are invalid:

Counter Example № 145

```
fragment fragOnScalar on Int {
  something
}

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

5.5.1.4 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 document.

For example the following is an invalid document:

Counter Example № 146

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

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

5.5.2 Fragment Spreads

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

5.5.2.1 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. It is a validation error if the target of a spread is not defined.

Counter Example № 147

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

5.5.2.2 Fragment Spreads Must Not Form Cycles

Formal Specification

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

DetectFragmentCycles(*fragmentDefinition*, *visited*) :

1. Let *spreads* be all fragment spread descendants of *fragmentDefinition*.
2. For each *spread* in *spreads*:
   a. *visited* must not contain *spread*.
   b. Let *nextVisited* be the set including *spread* and members of *visited*.
   c. Let *nextFragmentDefinition* be the target of *spread*.
   d. DetectFragmentCycles(*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:

Counter Example № 148

```
{
  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:

Example № 149

```
{
  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:

Counter Example № 150

```
{
  dog {
    ...dogFragment
  }
}

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

fragment ownerFragment on Human {
  name
  pets {
    ...dogFragment
  }
}
```

5.5.2.3 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.

5.5.2.3.1 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

Example № 151

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

and the following is invalid

Counter Example № 152

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

5.5.2.3.2 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

Example № 153

```
fragment petNameFragment on Pet {
  name
}

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

is valid because *Dog* implements Pet.

Likewise

Example № 154

```
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.

5.5.2.3.3 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:

Example № 155

```
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:

Counter Example № 156

```
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.

5.5.2.3.4 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

Example № 157

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

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

```
  }
}
```

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

However

Counter Example № 158

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

fragment sentientFragment on Sentient {
  name
}
```

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

Interface Spreads in Implemented Interface Scope

Additionally, an interface type fragment can always be spread into an interface scope which it implements.

In the example below, the `...resourceFragment` fragments spreads is valid, since `Resource` implements `Node`.

Example № 159

```
interface Node {
  id: ID!
}

interface Resource implements Node {
  id: ID!
  url: String
}

fragment interfaceWithInterface on Node {
  ...resourceFragment
}

fragment resourceFragment on Resource {
  url
}
```

5.6 Values

5.6.1 Values of Correct Type

Formal Specification

- For each input Value *value* in the document:
  - Let *type* be the type expected in the position *value* is found.
  - *value* must be coercible to *type*.

Explanatory Text

Literal values must be compatible with the type expected in the position they are found as per the coercion rules defined in the Type System chapter.

The type expected in a position includes the type defined by the argument a value is provided for, the type defined by an input object field a value is provided for, and the type of a variable definition a default value is provided for.

The following examples are valid use of value literals:

Example № 160

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

fragment coercedIntIntoFloatArg on Arguments {
  # Note: The input coercion rules for Float allow Int literals.
  floatArgField(floatArg: 123)
}

query goodComplexDefaultValue($search: FindDogInput = { name: "Fido" }) {
  findDog(searchBy: $search) {
    name
  }
}
```

Non-coercible values (such as a String into an Int) are invalid. The following examples are invalid:

Counter Example № 161

```
fragment stringIntoInt on Arguments {
  intArgField(intArg: "123")
}

query badComplexValue {
  findDog(searchBy: { name: 123 }) {
    name
  }
}
```

5.6.2 Input Object Field Names

Formal Specification

- For each Input Object Field *inputField* in the document:
  - Let *inputFieldName* be the Name of *inputField*.
  - Let *inputFieldDefinition* be the input field definition provided by the parent input object type named *inputFieldName*.
  - *inputFieldDefinition* must exist.

Explanatory Text

Every input field provided in an input object value must be defined in the set of possible fields of that input object's expected type.

For example the following example input object is valid:

Example № 162

```
{
  findDog(searchBy: { name: "Fido" }) {
    name
  }
}
```

While the following example input-object uses a field "favoriteCookieFlavor" which is not defined on the expected type:

Counter Example № 163

```
{
  findDog(searchBy: { favoriteCookieFlavor: "Bacon" }) {
    name
  }
}
```

5.6.3 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 document will not pass validation.

Counter Example № 164

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

5.6.4 Input Object Required Fields

Formal Specification

- For each Input Object in the document:
  - Let *fields* be the fields provided by that Input Object.
  - Let *fieldDefinitions* be the set of input field definitions of that Input Object.

- For each *fieldDefinition* in *fieldDefinitions*:
  - Let *type* be the expected type of *fieldDefinition*.
  - Let *defaultValue* be the default value of *fieldDefinition*.
  - If *type* is Non-Null and *defaultValue* does not exist:
    - Let *fieldName* be the name of *fieldDefinition*.
    - Let *field* be the input field in *fields* named *fieldName*.
    - *field* must exist.
    - Let *value* be the value of *field*.
    - *value* must not be the **null** literal.

Explanatory Text

Input object fields may be required. Much like a field may have required arguments, an input object may have required fields. An input field is required if it has a non-null type and does not have a default value. Otherwise, the input object field is optional.

5.7 Directives

5.7.1 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 services define what directives they support. For each usage of a directive, the directive must be available on that service.

5.7.2 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 services 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 service has declared support for.

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

Counter Example № 165

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

5.7.3 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* and are not repeatable.
  - 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

GraphQL allows directives that are defined as `repeatable` to be used more than once on the definition they apply to, possibly with different arguments. In contrast, if a directive is not `repeatable`, then only one occurrence of it is allowed per location.

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

Counter Example № 166

```
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 operation and on the same named field:

Example № 167

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

5.8 Variables

5.8.1 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.

Counter Example № 168

```
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:

Example № 169

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

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

fragment HouseTrainedFragment on Query {
  dog {
    isHouseTrained(atOtherHomes: $atOtherHomes)
  }
}
```

5.8.2 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*.
    - IsInputType(*variableType*) must be **true**.

Explanatory Text

Variables can only be input types. Objects, unions, and interfaces cannot be used as inputs.

For these examples, consider the following type system additions:

Example № 170

```
extend type Query {
  booleanList(booleanListArg: [Boolean!]): Boolean
}
```

The following operations are valid:

Example № 171

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

query takesComplexInput($search: FindDogInput) {
  findDog(searchBy: $search) {
    name
  }
}

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

The following operations are invalid:

Counter Example № 172

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

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

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

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

5.8.3 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:

Example № 173

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

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

By contrast the following document is invalid:

Counter Example № 174

```
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:

Example № 175

```
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 document is invalid.

Counter Example № 176

```
query variableIsNotDefinedUsedInSingleFragment {
  dog {
    ...isHouseTrainedFragment
  }
}

fragment isHouseTrainedFragment on Dog {
  isHouseTrained(atOtherHomes: $atOtherHomes)
}
```

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

Counter Example № 177

```
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.

Example № 178

```
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:

Counter Example № 179

```
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.

5.8.4 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:

Counter Example № 180

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

because $*atOtherHomes* is not referenced.

These rules apply to transitive fragment spreads as well:

Example № 181

```
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:

Counter Example № 182

```
query variableNotUsedWithinFragment($atOtherHomes: Boolean) {
  dog {
    ...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.

Counter Example № 183

```
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.

5.8.5 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 *variableName* be the name of *variableUsage*.
    - Let *variableDefinition* be the *VariableDefinition* named *variableName* defined within *operation*.
    - IsVariableUsageAllowed(*variableDefinition*, *variableUsage*) must be **true**.

IsVariableUsageAllowed(*variableDefinition*, *variableUsage*) **:**

1. Let *variableType* be the expected type of *variableDefinition*.
2. Let *locationType* be the expected type of the *Argument*, *ObjectField*, or *ListValue* entry where *variableUsage* is located.
3. If *locationType* is a non-null type AND *variableType* is NOT a non-null type:
   a. Let *hasNonNullVariableDefaultValue* be **true** if a default value exists for *variableDefinition* and is not the value **null**.
   b. Let *hasLocationDefaultValue* be **true** if a default value exists for the *Argument* or *ObjectField* where *variableUsage* is located.
   c. If *hasNonNullVariableDefaultValue* is NOT **true** AND *hasLocationDefaultValue* is NOT **true**, return **false**.
   d. Let *nullableLocationType* be the unwrapped nullable type of *locationType*.
   e. Return AreTypesCompatible(*variableType*, *nullableLocationType*).
4. Return AreTypesCompatible(*variableType*, *locationType*).

AreTypesCompatible(*variableType*, *locationType*) **:**

1. If *locationType* is a non-null type:
   a. If *variableType* is NOT a non-null type, return **false**.
   b. Let *nullableLocationType* be the unwrapped nullable type of *locationType*.
   c. Let *nullableVariableType* be the unwrapped nullable type of *variableType*.
   d. Return AreTypesCompatible(*nullableVariableType*, *nullableLocationType*).
2. Otherwise, if *variableType* is a non-null type:
   a. Let *nullableVariableType* be the nullable type of *variableType*.
   b. Return AreTypesCompatible(*nullableVariableType*, *locationType*).
3. Otherwise, if *locationType* is a list type:
   a. If *variableType* is NOT a list type, return **false**.
   b. Let *itemLocationType* be the unwrapped item type of *locationType*.

c. Let *itemVariableType* be the unwrapped item type of *variableType*.

d. Return AreTypesCompatible(*itemVariableType*, *itemLocationType*).

4. Otherwise, if *variableType* is a list type, return **false**.
5. Return **true** if *variableType* and *locationType* are identical, otherwise **false**.

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:

Counter Example № 184

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

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

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

Counter Example № 185

```
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.

Counter Example № 186

```
query booleanArgQuery($booleanArg: Boolean) {
  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:

Example № 187

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

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

Counter Example № 188

```
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!]`.

Allowing Optional Variables When Default Values Exist

A notable exception to typical variable type compatibility is allowing a variable definition with a nullable type to be provided to a non-null location as long as either that variable or that location provides a default value.

In the example below, an optional variable `$booleanArg` is allowed to be used in the non-null argument `optionalBooleanArg` because the field argument is optional since it provides a default value in the schema.

Example № 189

```
query booleanArgQueryWithDefault($booleanArg: Boolean) {
  arguments {
    optionalNonNullBooleanArgField(optionalBooleanArg: $booleanArg)
  }
}
```

In the example below, an optional variable `$booleanArg` is allowed to be used in the non-null argument (`nonNullBooleanArg`) because the variable provides a default value in the operation. This behavior is explicitly supported for compatibility with earlier editions of this specification. GraphQL authoring tools may wish to report this as a warning with the suggestion to replace `Boolean` with `Boolean!` to avoid ambiguity.

Example № 190

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

Note
The value **null** could still be provided to such a variable at runtime. A non-null argument must raise a *field error* if provided a **null** value.

6 Execution

A GraphQL service 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* which must contain GraphQL *OperationDefinition* and may contain *FragmentDefinition*.
- 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.

Note
GraphQL requests do not require any specific serialization format or transport mechanism. Message serialization and transport mechanisms should be chosen by the implementing service.

6.1 Executing Requests

To execute a request, the executor must have a parsed *Document* 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*).
5. Otherwise if *operation* is a subscription operation:
   a. Return Subscribe(*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 raise a *request error* requiring *operationName*.
2. Otherwise:
   a. Let *operation* be the Operation named *operationName* in *document*.
   b. If *operation* was not found, raise a *request error*.
   c. Return *operation*.

6.1.1 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.

6.1.2 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 *request 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 *variablesDefinition* be the variables defined by *operation*.
3. For each *variableDefinition* in *variablesDefinition*:
   a. Let *variableName* be the name of *variableDefinition*.
   b. Let *variableType* be the expected type of *variableDefinition*.
   c. Assert: IsInputType(*variableType*) must be **true**.
   d. Let *defaultValue* be the default value for *variableDefinition*.
   e. Let *hasValue* be **true** if *variableValues* provides a value for the name *variableName*.
   f. Let *value* be the value provided in *variableValues* for the name *variableName*.
   g. If *hasValue* is not **true** and *defaultValue* exists (including **null**):
      i. Add an entry to *coercedValues* named *variableName* with the value *defaultValue*.
   h. Otherwise if *variableType* is a Non-Nullable type, and either *hasValue* is not **true** or *value* is **null**, raise a *request error*.
   i. Otherwise if *hasValue* is true:
      i. If *value* is **null**:
         1. Add an entry to *coercedValues* named *variableName* with the value **null**.
      ii. Otherwise:
         1. If *value* cannot be coerced according to the input coercion rules of *variableType*, raise a *request error*.
         2. Let *coercedValue* be the result of coercing *value* according to the input coercion rules of *variableType*.
         3. Add an entry to *coercedValues* named *variableName* with the value *coercedValue*.
4. Return *coercedValues*.

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

6.2 Executing Operations

The type system, as described in the "Type System" section of the spec, must provide a query root operation type. If mutations or subscriptions are supported, it must also provide a mutation or subscription root operation type, respectively.

6.2.1 Query

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

An initial value may be provided when executing a query operation.

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 the list of all *field error* raised while executing the selection set.
6. Return an unordered map containing *data* and *errors*.

6.2.2 Mutation

If the operation is a mutation, the result of the operation is the result of executing the operation'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 the list of all *field error* raised while executing the selection set.
6. Return an unordered map containing *data* and *errors*.

6.2.3 Subscription

If the operation is a subscription, the result is an event stream called the "Response Stream" where each event in the event stream is the result of executing the operation for each new event on an underlying "Source Stream".

Executing a subscription operation creates a persistent function on the service that maps an underlying Source Stream to a returned Response Stream.

Subscribe(*subscription*, *schema*, *variableValues*, *initialValue*) :

1. Let *sourceStream* be the result of running CreateSourceEventStream(*subscription*, *schema*, *variableValues*, *initialValue*).
2. Let *responseStream* be the result of running MapSourceToResponseEvent(*sourceStream*, *subscription*, *schema*, *variableValues*)
3. Return *responseStream*.

> *Note*
> In a large-scale subscription system, the Subscribe() and ExecuteSubscriptionEvent() algorithms may be run on separate services to maintain predictable scaling properties. See the section below on Supporting Subscriptions at Scale.

As an example, consider a chat application. To subscribe to new messages posted to the chat room, the client sends a request like so:

Example № 191

```
subscription NewMessages {
  newMessage(roomId: 123) {
    sender
    text
  }
}
```

While the client is subscribed, whenever new messages are posted to chat room with ID "123", the selection for "sender" and "text" will be evaluated and published to the client, for example:

Example № 192

```
{
  "data": {
    "newMessage": {
      "sender": "Hagrid",
      "text": "You're a wizard!"
    }
  }
}
```

The “new message posted to chat room” could use a “Pub-Sub” system where the chat room ID is the “topic” and each “publish” contains the sender and text.

Event Streams

An event stream represents a sequence of discrete events over time which can be observed. As an example, a “Pub-Sub” system may produce an event stream when “subscribing to a topic”, with an event occurring on that event stream for each “publish” to that topic. Event streams may produce an infinite sequence of events or may complete at any point. Event streams may complete in response to an error or simply because no more events will occur. An observer may at any point decide to stop observing an event stream by cancelling it, after which it must receive no more events from that event stream.

Supporting Subscriptions at Scale

Supporting subscriptions is a significant change for any GraphQL service. Query and mutation operations are stateless, allowing scaling via cloning of GraphQL service instances. Subscriptions, by contrast, are stateful and require maintaining the GraphQL document, variables, and other context over the lifetime of the subscription.

Consider the behavior of your system when state is lost due to the failure of a single machine in a service. Durability and availability may be improved by having separate dedicated services for managing subscription state and client connectivity.

Delivery Agnostic

GraphQL subscriptions do not require any specific serialization format or transport mechanism. GraphQL specifies algorithms for the creation of the response stream, the content of each payload on that stream, and the closing of that stream. There are intentionally no specifications for message acknowledgement, buffering, resend requests, or any other quality of service (QoS) details. Message serialization, transport mechanisms, and quality of service details should be chosen by the implementing service.

6.2.3.1 Source Stream

A Source Stream represents the sequence of events, each of which will trigger a GraphQL execution corresponding to that event. Like field value resolution, the logic to create a Source Stream is application-specific.

CreateSourceEventStream(*subscription*, *schema*, *variableValues*, *initialValue*) :

1. Let *subscriptionType* be the root Subscription type in *schema*.
2. Assert: *subscriptionType* is an Object type.
3. Let *selectionSet* be the top level Selection Set in *subscription*.
4. Let *groupedFieldSet* be the result of CollectFields(*subscriptionType*, *selectionSet*, *variableValues*).
5. If *groupedFieldSet* does not have exactly one entry, raise a *request error*.
6. Let *fields* be the value of the first entry in *groupedFieldSet*.
7. Let *fieldName* be the name of the first entry in *fields*. Note: This value is unaffected if an alias is used.
8. Let *field* be the first entry in *fields*.
9. Let *argumentValues* be the result of CoerceArgumentValues(*subscriptionType*, *field*, *variableValues*)
10. Let *fieldStream* be the result of running ResolveFieldEventStream(*subscriptionType*, *initialValue*, *fieldName*, *argumentValues*).
11. Return *fieldStream*.

ResolveFieldEventStream(*subscriptionType*, *rootValue*, *fieldName*, *argumentValues*) :

1. Let *resolver* be the internal function provided by *subscriptionType* for determining the resolved event stream of a subscription field named *fieldName*.
2. Return the result of calling *resolver*, providing *rootValue* and *argumentValues*.

> *Note*
> This ResolveFieldEventStream() algorithm is intentionally similar to ResolveFieldValue() to enable consistency when defining resolvers on any operation type.

6.2.3.2 Response Stream

Each event in the underlying Source Stream triggers execution of the subscription selection set using that event as a root value.

MapSourceToResponseEvent(*sourceStream*, *subscription*, *schema*, *variableValues*) :

1. Return a new event stream *responseStream* which yields events as follows:

2. For each *event* on *sourceStream*:
    a. Let *response* be the result of running ExecuteSubscriptionEvent(*subscription*, *schema*, *variableValues*, *event*).
    b. Yield an event containing *response*.
3. When *responseStream* completes: complete this event stream.

ExecuteSubscriptionEvent(*subscription*, *schema*, *variableValues*, *initialValue*) :

1. Let *subscriptionType* be the root Subscription type in *schema*.
2. Assert: *subscriptionType* is an Object type.
3. Let *selectionSet* be the top level Selection Set in *subscription*.
4. Let *data* be the result of running ExecuteSelectionSet(*selectionSet*, *subscriptionType*, *initialValue*, *variableValues*) *normally* (allowing parallelization).
5. Let *errors* be the list of all *field error* raised while executing the selection set.
6. Return an unordered map containing *data* and *errors*.

> *Note*
> The ExecuteSubscriptionEvent() algorithm is intentionally similar to ExecuteQuery() since this is how each event result is produced.

6.2.3.3 Unsubscribe

Unsubscribe cancels the Response Stream when a client no longer wishes to receive payloads for a subscription. This may in turn also cancel the Source Stream. This is also a good opportunity to clean up any other resources used by the subscription.

Unsubscribe(*responseStream*) :

1. Cancel *responseStream*

6.3 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 defined:
      i. Let *responseValue* be ExecuteField(*objectType*, *objectValue*, *fieldType*, *fields*, *variableValues*).
      ii. Set *responseValue* as the value for *responseKey* in *resultMap*.
4. Return *resultMap*.

> *Note*
> *resultMap* is ordered by which fields appear first in the operation. This is explained in greater detail in the Field Collection section below.

Errors and Non-Null Fields

If during ExecuteSelectionSet() a field with a non-null *fieldType* raises a *field error* then that error must propagate to this entire selection set, either resolving to **null** if allowed or further propagated to a parent field.

If this occurs, any sibling fields which have not yet executed or have not yet yielded a value may be cancelled to avoid unnecessary work.

> *Note*
> See Handling Field Errors for more about this behavior.

6.3.1 Normal and Serial Execution

Normally the executor can execute the entries in a grouped field set in whatever order it chooses (normally 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 service 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:

Example № 193

```
{
  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, starting with the first appearing field textually.

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:

Example № 194

```
{
  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:

Example № 195

```
{
  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:

Example № 196

```
{
  "first": {
    "theNumber": 1
  },
  "second": {
    "theNumber": 3
  },
  "third": {
    "theNumber": 2
  }
}
```

6.3.2 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 (the alias if defined, otherwise the field name). This ensures all fields with the same response key (including those in 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`:

Example № 197

```
{
  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* is 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* (the alias if defined, otherwise the field name).
      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*, *variableValues*, *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**.

> *Note*
> The steps in CollectFields() evaluating the `@skip` and `@include` directives may be applied in either order since they apply commutatively.

6.4 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 *fieldName* be the field name of *field*.
3. Let *argumentValues* be the result of CoerceArgumentValues(*objectType*, *field*, *variableValues*)
4. Let *resolvedValue* be ResolveFieldValue(*objectType*, *objectValue*, *fieldName*, *argumentValues*).
5. Return the result of CompleteValue(*fieldType*, *fields*, *resolvedValue*, *variableValues*).

6.4.1 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.

At each argument position in an operation 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 *hasValue* be **true** if *argumentValues* provides a value for the name *argumentName*.
    e. Let *argumentValue* be the value provided in *argumentValues* for the name *argumentName*.
    f. If *argumentValue* is a *Variable*:
        i. Let *variableName* be the name of *argumentValue*.
        ii. Let *hasValue* be **true** if *variableValues* provides a value for the name *variableName*.
        iii. Let *value* be the value provided in *variableValues* for the name *variableName*.
    g. Otherwise, let *value* be *argumentValue*.
    h. If *hasValue* is not **true** and *defaultValue* exists (including **null**):
        i. Add an entry to *coercedValues* named *argumentName* with the value *defaultValue*.
    i. Otherwise if *argumentType* is a Non-Nullable type, and either *hasValue* is not **true** or *value* is **null**, raise a *field error*.

j. Otherwise if *hasValue* is true:

i. If *value* is **null**:

1. Add an entry to *coercedValues* named *argumentName* with the value **null**.

ii. Otherwise, if *argumentValue* is a *Variable*:

1. Add an entry to *coercedValues* named *argumentName* with the value *value*.

iii. Otherwise:

1. If *value* cannot be coerced according to the input coercion rules of *argumentType*, raise a *field error*.
2. Let *coercedValue* be the result of coercing *value* according to the input coercion rules of *argumentType*.
3. Add an entry to *coercedValues* named *argumentName* 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 operations must only allow usage of variables of appropriate types.

6.4.2 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.

6.4.3 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**, raise a *field error*.
   d. Return *completedResult*.
2. If *result* is **null** (or another internal value similar to **null** such as **undefined**), return **null**.
3. If *fieldType* is a List type:
   a. If *result* is not a collection of values, raise 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 CoerceResult(*fieldType*, *result*).
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).

Coercing Results

The primary purpose of value completion is to ensure that the values returned by field resolvers are valid according to the GraphQL type system and a service's schema. This "dynamic type checking" allows GraphQL to provide consistent guarantees about returned types atop any service's internal runtime.

See the Scalars Result Coercion and Serialization sub-section for more detailed information about how GraphQL's built-in scalars coerce result values.

CoerceResult(*leafType*, *value*) :

1. Assert *value* is not **null**.
2. Return the result of calling the internal method provided by the type system for determining the "result coercion" of *leafType* given the value *value*. This internal method must return a valid value for the type and not **null**. Otherwise raise a *field error*.

> *Note*
> If a field resolver returns **null** then it is handled within CompleteValue() before CoerceResult() is called. Therefore both the input and output of CoerceResult() must not be **null**.

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 field of the same name is 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 operation illustrating parallel fields with the same name with sub-selections.

Example № 198

```
{
  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*.

6.4.4 Handling Field Errors

A *field error* is an error raised from a particular field during value resolution or coercion. While these errors should be reported in the response, they are "handled" by producing a partial response.

> *Note*
> This is distinct from a *request error* which results in a response with no data.

If a field error is raised while resolving a field, it is handled as though the field returned **null**, and the 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 a field error was raised), and that field is of a `Non-Null` type, then a field error is raised. The error must be added to the `"errors"` list in the response.

If the field returns **null** because of a field 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 a `List` type wraps a `Non-Null` type, and one of the elements of that list resolves to **null**, then the entire list must resolve to **null**. If the `List` type is also wrapped in a `Non-Null`, the field error continues to propagate upwards.

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

7 Response

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

A response may contain both a partial response as well as a list of errors in the case that any *field error* was raised on a field and was replaced with **null**.

7.1 Response Format

A response to a GraphQL request must be a map.

If the request raised any errors, the response map must contain an entry with key `errors`. The value of this entry is described in the "Errors" section. If the request completed without raising any errors, this entry must not be present.

If the request included execution, the response map must contain an entry with key `data`. The value of this entry is described in the "Data" section. If the request failed before execution, due to a syntax error, missing information, or validation error, 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 services and clients, the top level response map must not contain any entries other than the three described above.

> *Note*
> When `errors` is present in the response, it may be helpful for it to appear first when serialized to make it more clear when errors are present in a response during debugging.

7.1.1 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 query root operation type; if the operation was a mutation, this output will be an object of the mutation root operation type.

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

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

7.1.2 Errors

The `errors` entry in the response is a non-empty list of errors raised during the *request*, where each error is a map of data described by the error result format below.

If present, the `errors` entry in the response must contain at least one error. If no errors were raised during the request, the `errors` entry must not be present in the result.

If the `data` entry in the response is not present, the `errors` entry must be present. It must contain at least one *request error* indicating why no data was able to be returned.

If the `data` entry in the response is present (including if it is the value **null**), the `errors` entry must be present if and only if one or more *field error* was raised during execution.

Request Errors

A *request error* is an error raised during a *request* which results in no response data. Typically raised before execution begins, a request error may occur due to a parse grammar or validation error in the *Document*, an inability to determine which operation to execute, or invalid input values for variables.

A request error is typically the fault of the requesting client.

If a request error is raised, the `data` entry in the response must not be present, the `errors` entry must include the error, and request execution should be halted.

Field Errors

A *field error* is an error raised during the execution of a particular field which results in partial response data. This may occur due to an internal error during value resolution or failure to coerce the resulting value.

A field error is typically the fault of a GraphQL service.

If a field error is raised, execution attempts to continue and a partial result is produced (see Handling Field Errors). The `data` entry in the response must be present. The `errors` entry should include this error.

Error Result Format

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.

If an error can be associated to a particular field in the GraphQL result, it must contain an entry with the key `path` that details the path of the response field which experienced the error. This allows clients to identify whether a `null` result is intentional or caused by a runtime error.

This field should be a list of path segments starting at the root of the response and ending with the field associated with the error. Path segments that represent fields should be strings, and path segments that represent list indices should be 0-indexed integers. If the error happens in an aliased field, the path to the error should use the aliased name, since it represents a path in the response, not in the request.

For example, if fetching one of the friends' names fails in the following operation:

Example № 199

```
{
  hero(episode: $episode) {
    name
    heroFriends: friends {
      id
      name
    }
  }
}
```

The response might look like:

Example № 200

```
{
  "errors": [
```

```
    {
      "message": "Name for character with ID 1002 could not be fetched.",
      "locations": [{ "line": 6, "column": 7 }],
      "path": ["hero", "heroFriends", 1, "name"]
    }
  ],
  "data": {
    "hero": {
      "name": "R2-D2",
      "heroFriends": [
        {
          "id": "1000",
          "name": "Luke Skywalker"
        },
        {
          "id": "1002",
          "name": null
        },
        {
          "id": "1003",
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

If the field which experienced an error was declared as `Non-Null`, the `null` result will bubble up to the next nullable field. In that case, the `path` for the error should include the full path to the result field where the error was raised, even if that field is not present in the response.

For example, if the `name` field from above had declared a `Non-Null` return type in the schema, the result would look different but the error reported would be the same:

Example № 201

```
{
  "errors": [
    {
      "message": "Name for character with ID 1002 could not be fetched.",
      "locations": [{ "line": 6, "column": 7 }],
      "path": ["hero", "heroFriends", 1, "name"]
    }
  ],
  "data": {
    "hero": {
      "name": "R2-D2",
      "heroFriends": [
```

```
        {
          "id": "1000",
          "name": "Luke Skywalker"
        },
        null,
        {
          "id": "1003",
          "name": "Leia Organa"
        }
      ]
    }
  }
}
```

GraphQL services may provide an additional entry to errors with key `extensions`. This entry, if set, must have a map as its value. This entry is reserved for implementors to add additional information to errors however they see fit, and there are no additional restrictions on its contents.

Example № 202

```
{
  "errors": [
    {
      "message": "Name for character with ID 1002 could not be fetched.",
      "locations": [{ "line": 6, "column": 7 }],
      "path": ["hero", "heroFriends", 1, "name"],
      "extensions": {
        "code": "CAN_NOT_FETCH_BY_ID",
        "timestamp": "Fri Feb 9 14:33:09 UTC 2018"
      }
    }
  ]
}
```

GraphQL services should not provide any additional entries to the error format since they could conflict with additional entries that may be added in future versions of this specification.

Note

Previous versions of this spec did not describe the `extensions` entry for error formatting. While non-specified entries are not violations, they are still discouraged.

Counter Example № 203

```
{
  "errors": [
    {
      "message": "Name for character with ID 1002 could not be fetched.",
```

```
      "locations": [{ "line": 6, "column": 7 }],
      "path": ["hero", "heroFriends", 1, "name"],
      "code": "CAN_NOT_FETCH_BY_ID",
      "timestamp": "Fri Feb 9 14:33:09 UTC 2018"
    }
  ]
}
```

7.2 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 at least support representations of the following four primitives:

- Map
- List
- String
- Null

A serialization format should also support the following primitives, each representing one of the common GraphQL scalar types, however a string or simpler primitive may be used as a substitute if any are not directly supported:

- Boolean
- Int
- Float
- Enum Value

This is not meant to be an exhaustive list of what a serialization format may encode. For example custom scalars representing a Date, Time, URI, or number with a different precision may be represented in whichever relevant format a given serialization format may support.

7.2.1 JSON Serialization

JSON is the most common serialization format for GraphQL. Though as mentioned above, GraphQL does not require a specific serialization format.

When using JSON as a serialization of GraphQL responses, the following JSON values should be used to encode the related GraphQL values:

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

> *Note*
> For consistency and ease of notation, examples of responses are given in JSON format throughout this document.

7.2.2 Serialized Map Ordering

Since the result of evaluating a selection set is ordered, the serialized Map of results should preserve this order by writing the map entries in the same order as those fields were requested as defined by selection set execution. Producing a serialized 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.

Serialization formats which represent an ordered map should preserve the order of requested fields as defined by CollectFields() in the Execution section. Serialization formats which only represent unordered maps but where order is still implicit in the serialization's textual order (such as JSON) should preserve the order of requested fields textually.

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

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.

> *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.

A 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.

A.1 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 (ie. `=` or `terminal`), and prose typically describing a specific Unicode code-point `"Space (U+0020)"`. Sequences of Unicode characters only appear in syntactic grammars and represent a *Name* token of that specific sequence.

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 :
  ListOfLetterA `a`
  `a`

A.2 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 source text sequence of Unicode characters first produces a sequence of lexical tokens according to the lexical grammar which then produces abstract syntax tree (AST) according to the syntactical grammar.

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 **::**
  Letter$_{\text{list}}$

A Syntactical grammar production describes non-terminal "rules" by patterns of terminal Tokens. *WhiteSpace* and other *Ignored* sequences may appear before or after any terminal *Token*. A syntactical grammar production is distinguished by a one colon `:` definition.

Sentence **:**
  Word$_{\text{list}}$ **.**

A.3 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 following production means that the non-terminal *SafeWord* may be replaced by any sequence of characters that could replace *Word* provided that the same sequence of characters could not

replace *SevenCarlinWords*.

SafeWord :
  Word but not *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`

Lookahead Restrictions

A grammar production may specify that certain characters or tokens are not permitted to follow it by using the pattern [lookahead ∉ *NotAllowed*]. Lookahead restrictions are often used to remove ambiguity from the grammar.

The following example makes it clear that $Letter_{list}$ must be greedy, since *Word* cannot be followed by yet another *Letter*.

Word ::
  $Letter_{list}$ [lookahead ∉ *Letter*]

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* *Adverb*
  Noun *Verb*

A subscript suffix "$Symbol_{list}$" is shorthand for a list of one or more of that symbol, represented as an additional recursive production.

As an example:

Book :
  Cover $Page_{list}$ *Cover*

is shorthand for

Book :
  Cover *Page_list* *Cover*

Page_list :
  Page_list *Page*
  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$_{[?Param]}$
  [if Param] *D*
  [if not Param] *E*

is shorthand for

Example :
  A
  B_param
  C
  E

Example_param :
  A
  B_param
  C_param
  D

A.4 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*$_{\text{list}}$ `"`

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

A.5 Algorithms

This specification describes some algorithms used by the static and runtime semantics, they're defined in the form of a function-like syntax with the algorithm's name and the arguments it accepts along with a list of algorithmic steps to take in the order listed. Each step may establish references to other values, check various conditions, call other algorithms, and eventually return a value representing the outcome of the algorithm for the provided arguments.

For example, the following example describes an algorithm named *Fibonacci* which accepts a single argument *number*. The algorithm's steps produce the next number in the Fibonacci sequence:

Fibonacci(*number*) :

1. If *number* is `0`:
    a. Return `1`.
2. If *number* is `1`:
    a. Return `2`.
3. Let *previousNumber* be *number* - `1`.
4. Let *previousPreviousNumber* be *number* - `2`.
5. Return Fibonacci(*previousNumber*) + Fibonacci(*previousPreviousNumber*).

> *Note*
> Algorithms described in this document are written to be easy to understand. Implementers are encouraged to include equivalent but optimized implementations.

B Appendix: Grammar Summary

B.1 Source Text

SourceCharacter ::
  Any Unicode scalar value

B.2 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 ::
  # $CommentChar_{list,\ opt}$ [lookahead ∉ *CommentChar*]

CommentChar ::
  SourceCharacter but not *LineTerminator*

Comma ::

  ,

B.3 Lexical Tokens

Token ::
  Punctuator
  Name
  IntValue
  FloatValue
  StringValue

Punctuator :: **one of**
  `! $ & ( ) ... : = @ [ ] { | }`

Name ::
  NameStart *NameContinue*$_{\text{list, opt}}$ [lookahead ∉ *NameContinue*]

NameStart ::
  Letter
  `_`

NameContinue ::
  Letter
  Digit
  `_`

Letter :: **one of**
  `A B C D E F G H I J K L M`
  `N O P Q R S T U V W X Y Z`
  `a b c d e f g h i j k l m`
  `n o p q r s t u v w x y z`

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

IntValue ::
  IntegerPart [lookahead ∉ {*Digit*, `.`, *NameStart*}]

IntegerPart ::
  NegativeSign$_{\text{opt}}$ `0`
  NegativeSign$_{\text{opt}}$ *NonZeroDigit* *Digit*$_{\text{list, opt}}$

NegativeSign ::

- `-`

NonZeroDigit ::
  Digit but not `0`

FloatValue ::
  IntegerPart *FractionalPart* *ExponentPart* [lookahead ∉ {*Digit*, `.`, *NameStart*}]
  IntegerPart *FractionalPart* [lookahead ∉ {*Digit*, `.`, *NameStart*}]
  IntegerPart *ExponentPart* [lookahead ∉ {*Digit*, `.`, *NameStart*}]

FractionalPart ::
  `.` $Digit_{list}$

ExponentPart ::
  ExponentIndicator $Sign_{opt}$ $Digit_{list}$

ExponentIndicator :: **one of**
  `e` `E`

Sign :: **one of**
  `+` `-`

StringValue ::
  `""` [lookahead ≠ `"`]
  `"` $StringCharacter_{list}$ `"`
  `"""` $BlockStringCharacter_{list, opt}$ `"""`

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

EscapedUnicode ::
  `{` $HexDigit_{list}$ `}`
  HexDigit *HexDigit* *HexDigit* *HexDigit*

HexDigit :: **one of**
  `0` `1` `2` `3` `4` `5` `6` `7` `8` `9`
  `A` `B` `C` `D` `E` `F`
  `a` `b` `c` `d` `e` `f`

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

BlockStringCharacter ::
  SourceCharacter but not `"""` or `\"""`

\"""

> *Note*
> Block string values are interpreted to exclude blank initial and trailing lines and uniform indentation with BlockStringValue().

B.4 Document Syntax

Document :
  Definition$_{\text{list}}$

Definition :
  ExecutableDefinition
  TypeSystemDefinitionOrExtension

ExecutableDocument :
  ExecutableDefinition$_{\text{list}}$

ExecutableDefinition :
  OperationDefinition
  FragmentDefinition

OperationDefinition :
  OperationType *Name*$_{\text{opt}}$ *VariablesDefinition*$_{\text{opt}}$ *Directives*$_{\text{opt}}$ *SelectionSet*
  SelectionSet

OperationType : **one of**
  `query` `mutation` `subscription`

SelectionSet :
  `{` *Selection*$_{\text{list}}$ `}`

Selection :
  Field
  FragmentSpread
  InlineFragment

Field :
  Alias$_{\text{opt}}$ *Name* *Arguments*$_{\text{opt}}$ *Directives*$_{\text{opt}}$ *SelectionSet*$_{\text{opt}}$

Alias :
  Name :

Arguments$_{[\text{Const}]}$:

(*Argument*[?Const]list)

Argument[Const] :

Name : *Value*[?Const]

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] :

[if not 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]

VariablesDefinition :
(*VariableDefinition*list)

VariableDefinition :
Variable : *Type* *DefaultValue*opt *Directives*[Const]opt

Variable :
$ *Name*

DefaultValue :
= *Value*[Const]

Type :
NamedType
ListType
NonNullType

NamedType :
Name

ListType :
[*Type*]

NonNullType :
NamedType !
ListType !

Directives[Const] :
Directive[?Const]list

Directive[Const] :
@ *Name* *Arguments*[?Const]opt

TypeSystemDocument :
*TypeSystemDefinition*list

TypeSystemDefinition :
SchemaDefinition
TypeDefinition
DirectiveDefinition

TypeSystemExtensionDocument :

TypeSystemDefinitionOrExtension$_{list}$

TypeSystemDefinitionOrExtension :
- *TypeSystemDefinition*
- *TypeSystemExtension*

TypeSystemExtension :
- *SchemaExtension*
- *TypeExtension*

SchemaDefinition :
- *Description*$_{opt}$ **schema** *Directives*$_{[Const]opt}$ **{** *RootOperationTypeDefinition*$_{list}$ **}**

SchemaExtension :
- **extend schema** *Directives*$_{[Const]opt}$ **{** *RootOperationTypeDefinition*$_{list}$ **}**
- **extend schema** *Directives*$_{[Const]}$ [lookahead ≠ **{**]

RootOperationTypeDefinition :
- *OperationType* **:** *NamedType*

Description :
- *StringValue*

TypeDefinition :
- *ScalarTypeDefinition*
- *ObjectTypeDefinition*
- *InterfaceTypeDefinition*
- *UnionTypeDefinition*
- *EnumTypeDefinition*
- *InputObjectTypeDefinition*

TypeExtension :
- *ScalarTypeExtension*
- *ObjectTypeExtension*
- *InterfaceTypeExtension*
- *UnionTypeExtension*
- *EnumTypeExtension*
- *InputObjectTypeExtension*

ScalarTypeDefinition :
- *Description*$_{opt}$ **scalar** *Name* *Directives*$_{[Const]opt}$

ScalarTypeExtension :
- **extend scalar** *Name* *Directives*$_{[Const]}$

ObjectTypeDefinition :

Description$_{opt}$ **type** *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ *FieldsDefinition*

Description$_{opt}$ **type** *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ [lookahead ≠ **{**]

ObjectTypeExtension :

extend type *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ *FieldsDefinition*

extend type *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]}$ [lookahead ≠ **{**]

extend type *Name* *ImplementsInterfaces* [lookahead ≠ **{**]

ImplementsInterfaces :

ImplementsInterfaces **&** *NamedType*

implements **&**$_{opt}$ *NamedType*

FieldsDefinition :

{ *FieldDefinition*$_{list}$ **}**

FieldDefinition :

Description$_{opt}$ *Name* *ArgumentsDefinition*$_{opt}$ **:** *Type* *Directives*$_{[Const]opt}$

ArgumentsDefinition :

(*InputValueDefinition*$_{list}$ **)**

InputValueDefinition :

Description$_{opt}$ *Name* **:** *Type* *DefaultValue*$_{opt}$ *Directives*$_{[Const]opt}$

InterfaceTypeDefinition :

Description$_{opt}$ **interface** *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ *FieldsDefinition*

Description$_{opt}$ **interface** *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ [lookahead ≠ **{**]

InterfaceTypeExtension :

extend interface *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]opt}$ *FieldsDefinition*

extend interface *Name* *ImplementsInterfaces*$_{opt}$ *Directives*$_{[Const]}$ [lookahead ≠ **{**]

extend interface *Name* *ImplementsInterfaces* [lookahead ≠ **{**]

UnionTypeDefinition :

Description$_{opt}$ **union** *Name* *Directives*$_{[Const]opt}$ *UnionMemberTypes*$_{opt}$

UnionMemberTypes :

UnionMemberTypes **|** *NamedType*

= **|**$_{opt}$ *NamedType*

UnionTypeExtension :

extend union *Name* *Directives*$_{[Const]opt}$ *UnionMemberTypes*

extend union *Name* *Directives*$_{[Const]}$

EnumTypeDefinition :

Description$_{opt}$ **enum** *Name* *Directives*$_{[Const]opt}$ *EnumValuesDefinition*

Description$_{opt}$ **enum** *Name* *Directives*$_{[Const]opt}$ [lookahead ≠ **{**]

EnumValuesDefinition :

{ *EnumValueDefinition*$_{list}$ **}**

EnumValueDefinition :

Description$_{opt}$ *EnumValue* *Directives*$_{[Const]opt}$

EnumTypeExtension :

extend enum *Name* *Directives*$_{[Const]opt}$ *EnumValuesDefinition*

extend enum *Name* *Directives*$_{[Const]}$ [lookahead ≠ **{**]

InputObjectTypeDefinition :

Description$_{opt}$ **input** *Name* *Directives*$_{[Const]opt}$ *InputFieldsDefinition*

Description$_{opt}$ **input** *Name* *Directives*$_{[Const]opt}$ [lookahead ≠ **{**]

InputFieldsDefinition :

{ *InputValueDefinition*$_{list}$ **}**

InputObjectTypeExtension :

extend input *Name* *Directives*$_{[Const]opt}$ *InputFieldsDefinition*

extend input *Name* *Directives*$_{[Const]}$ [lookahead ≠ **{**]

DirectiveDefinition :

Description$_{opt}$ **directive @** *Name* *ArgumentsDefinition*$_{opt}$ **repeatable**$_{opt}$ **on** *DirectiveLocations*

DirectiveLocations :

DirectiveLocations **|** *DirectiveLocation*

|$_{opt}$ *DirectiveLocation*

DirectiveLocation :

ExecutableDirectiveLocation

TypeSystemDirectiveLocation

ExecutableDirectiveLocation : **one of**

QUERY
MUTATION
SUBSCRIPTION
FIELD
FRAGMENT_DEFINITION
FRAGMENT_SPREAD
INLINE_FRAGMENT
VARIABLE_DEFINITION

TypeSystemDirectiveLocation : **one of**

- `SCHEMA`
- `SCALAR`
- `OBJECT`
- `FIELD_DEFINITION`
- `ARGUMENT_DEFINITION`
- `INTERFACE`
- `UNION`
- `ENUM`
- `ENUM_VALUE`
- `INPUT_OBJECT`
- `INPUT_FIELD_DEFINITION`

§ Index

Written in Spec Markdown.