Table of Contents

File: descriptor.proto

Message: FileDescriptorSet

The protocol compiler can output a FileDescriptorSet containing the .proto files it parses.

Field	Type	Rule	Description
file	FileDescriptorProto	repeated	

Message: FileDescriptorProto

Describes a complete .proto file.

Field	Type	Rule	Description
name	string	optional	file name, relative to root of source tree
package	string	optional	e.g. "foo", "foo.bar", etc.
dependency	string	repeated	Names of files imported by this file.
public_dependency	int32	repeated	Indexes of the public imported files in the dependency list above.
weak_dependency	int32	repeated	Indexes of the weak imported files in the dependency list. For Google-internal migration only. Do not use.
message_type	DescriptorProto	repeated	All top-level definitions in this file.
enum_type	EnumDescriptorProto	repeated	
service	ServiceDescriptorProto	repeated	
extension	FieldDescriptorProto	repeated	
options	FileOptions	optional	

Field	Type	Rule	Description
source_code_info	SourceCodeInfo	optional	This field contains optional information about the original source code. You may safely remove this entire field whithout harming runtime functionality of the descriptors -- the information is needed only by development tools.

Message: DescriptorProto

Describes a message type.

Field	Type	Rule	Description
name	string	optional	
field	FieldDescriptorProto	repeated	
extension	FieldDescriptorProto	repeated	
nested_type	DescriptorProto	repeated	
enum_type	EnumDescriptorProto	repeated	
extension_range	ExtensionRange	repeated	
options	MessageOptions	optional	

Message: DescriptorProto.ExtensionRange

Field	Type	Rule	Description
start	int32	optional	
end	int32	optional	

Message: FieldDescriptorProto

Describes a field within a message.

Field	Type	Rule	Description
name	string	optional	
number	int32	optional	
label	Label	optional	

Field	Type	Rule	Description
type	Type	optional	If type_name is set, this need not be set. If both this and type_name are set, this must be either TYPE_ENUM or TYPE_MESSAGE.
type_name	string	optional	For message and enum types, this is the name of the type. If the name starts with a '.', it is fully-qualified. Otherwise, C++-like scoping rules are used to find the type (i.e. first the nested types within this message are searched, then within the parent, on up to the root namespace).
extendee	string	optional	For extensions, this is the name of the type being extended. It is resolved in the same manner as type_name.
default_value	string	optional	For numeric types, contains the original text representation of the value. For booleans, "true" or "false". For strings, contains the default text contents (not escaped in any way). For bytes, contains the C escaped value. All bytes >= 128 are escaped. TODO(kenton): Base-64 encode?
options	FieldOptions	optional	

Enum: FieldDescriptorProto.Type

Element	Value	Description
TYPE_DOUBLE	0	0 is reserved for errors. Order is weird for historical reasons.
TYPE_FLOAT	1	
TYPE_INT64	2	Not ZigZag encoded. Negative numbers take 10 bytes. Use TYPE_SINT64 if negative values are likely.
TYPE_UINT64	3	
TYPE_INT32	4	Not ZigZag encoded. Negative numbers take 10 bytes. Use TYPE_SINT32 if negative values are likely.
TYPE_FIXED64	5	
TYPE_FIXED32	6	
TYPE_BOOL	7	
TYPE_STRING	8	

Element	Value	Description
TYPE_GROUP	9	Tag-delimited aggregate.
TYPE_MESSAGE	10	Length-delimited aggregate.
TYPE_BYTES	11	New in version 2.
TYPE_UINT32	12	
TYPE_ENUM	13	
TYPE_SFIXED32	14	
TYPE_SFIXED64	15	
TYPE_SINT32	16	Uses ZigZag encoding.
TYPE_SINT64	17	Uses ZigZag encoding.

Enum: FieldDescriptorProto.Label

Element	Value	Description
LABEL_OPTIONAL	0	0 is reserved for errors
LABEL_REQUIRED	1	
LABEL_REPEATED	2	TODO(sanjay): Should we add LABEL_MAP?

Message: EnumDescriptorProto

Describes an enum type.

Field	Type	Rule	Description
name	string	optional	
value	EnumValueDescriptorProto	repeated	
options	EnumOptions	optional	

Message: EnumValueDescriptorProto

Describes a value within an enum.

Field	Type	Rule	Description
name	string	optional	

Field	Type	Rule	Description
number	int32	optional	
options	EnumValueOptions	optional	

Message: ServiceDescriptorProto

Describes a service.

Field	Type	Rule	Description
name	string	optional	
method	MethodDescriptorProto	repeated	
options	ServiceOptions	optional	

Message: MethodDescriptorProto

Describes a method of a service.

Field	Type	Rule	Description
name	string	optional	
input_type	string	optional	Input and output type names. These are resolved in the same way as FieldDescriptorProto.type_name, but must refer to a message type.
output_type	string	optional	
options	MethodOptions	optional	

Message: FileOptions

Field	Type	Rule	Description
java_package	string	optional	Sets the Java package where classes generated from this .proto will be placed. By default, the proto package is used, but this is often inappropriate because proto packages do not normally start with backwards domain names.
java_outer_classname	string	optional	If set, all the classes from the .proto file are wrapped in a single outer class with the given name. This applies to both Proto1 (equivalent to the old "--one_java_file" option) and Proto2

Field	Type	Rule	Description
			(where a .proto always translates to a single class, but you may want to explicitly choose the class name).
java_multiple_files	bool	optional	If set true, then the Java code generator will generate a separate .java file for each top-level message, enum, and service defined in the .proto file. Thus, these types will *not* be nested inside the outer class named by java_outer_classname. However, the outer class will still be generated to contain the file's getDescriptor() method as well as any top-level extensions defined in the file. [default = false]
java_generate_equals_and_hash	bool	optional	If set true, then the Java code generator will generate equals() and hashCode() methods for all messages defined in the .proto file. This is purely a speed optimization, as the AbstractMessage base class includes reflection-based implementations of these methods. [default = false]
optimize_for	OptimizeMode	optional	[default = SPEED]
go_package	string	optional	Sets the Go package where structs generated from this .proto will be placed. There is no default.
cc_generic_services	bool	optional	Should generic services be generated in each language? "Generic" services are not specific to any particular RPC system. They are generated by the main code generators in each language (without additional plugins). Generic services were the only kind of service generation supported by early versions of proto2. Generic services are now considered deprecated in favor of using plugins that generate code specific to your particular RPC system. Therefore, these default to false. Old code which depends on generic services should explicitly set them to true. [default = false]
java_generic_services	bool	optional	[default = false]

Field	Type	Rule	Description
py_generic_services	bool	optional	[default = false]
uninterpreted_option	UninterpretedOption	repeated	The parser stores options it doesn't recognize here. See above.

Enum: FileOptions.OptimizeMode

Generated classes can be optimized for speed or code size.

Element	Value	Description
SPEED	0	Generate complete code for parsing, serialization,
CODE_SIZE	1	etc. Use ReflectionOps to implement these methods.
LITE_RUNTIME	2	Generate code using MessageLite and the lite runtime.

Message: MessageOptions

Field	Type	Rule	Description
message_set_wire_format	bool	optional	Set true to use the old proto1 MessageSet wire format for extensions. This is provided for backwards-compatibility with the MessageSet wire format. You should not use this for any other reason: It's less efficient, has fewer features, and is more complicated. The message must be defined exactly as follows: message Foo { option message_set_wire_format = true; extensions 4 to max; } Note that the message cannot have any defined fields; MessageSets only have extensions. All extensions of your type must be singular messages; e.g. they cannot be int32s, enums, or repeated messages. Because this is an option, the above two restrictions are not enforced by the protocol compiler. [default = false]
no_standard_descriptor_accesso	bool	optional	Disables the generation of the standard "descriptor()" accessor, which can conflict with a field of the same name. This is meant

Field	Type	Rule	Description
			to make migration from proto1 easier; new code should avoid fields named "descriptor". [default = false]
uninterpreted_option	UninterpretedOption	repeated	The parser stores options it doesn't recognize here. See above.

Message: FieldOptions

Field	Type	Rule	Description
ctype	CType	optional	The ctype option instructs the C++ code generator to use a different representation of the field than it normally would. See the specific options below. This option is not yet implemented in the open source release -- sorry, we'll try to include it in a future version! [default = STRING]
packed	bool	optional	The packed option can be enabled for repeated primitive fields to enable a more efficient representation on the wire. Rather than repeatedly writing the tag and type for each element, the entire array is encoded as a single length-delimited blob.
lazy	bool	optional	Should this field be parsed lazily? Lazy applies only to message-type fields. It means that when the outer message is initially parsed, the inner message's contents will not be parsed but instead stored in encoded form. The inner message will actually be parsed when it is first accessed. This is only a hint. Implementations are free to choose whether to use eager or lazy parsing regardless of the value of this option. However, setting this option true suggests that the protocol author believes that using lazy parsing on this field is worth the additional bookkeeping overhead typically needed to implement it. This option does not affect the public interface of any generated code; all method signatures remain the same. Furthermore, thread-safety of the interface is not affected by this option; const

Field	Type	Rule	Description
			methods remain safe to call from multiple threads concurrently, while non-const methods continue to require exclusive access. Note that implementations may choose not to check required fields within a lazy sub-message. That is, calling IsInitialized() on the outher message may return true even if the inner message has missing required fields. This is necessary because otherwise the inner message would have to be parsed in order to perform the check, defeating the purpose of lazy parsing. An implementation which chooses not to check required fields must be consistent about it. That is, for any particular sub-message, the implementation must either *always* check its required fields, or *never* check its required fields, regardless of whether or not the message has been parsed. [default = false]
deprecated	bool	optional	Is this field deprecated? Depending on the target platform, this can emit Deprecated annotations for accessors, or it will be completely ignored; in the very least, this is a formalization for deprecating fields. [default = false]
experimental_map_key	string	optional	EXPERIMENTAL. DO NOT USE. For "map" fields, the name of the field in the enclosed type that is the key for this map. For example, suppose we have: message Item { required string name = 1; required string value = 2; } message Config { repeated Item items = 1 [experimental_map_key="name"]; } In this situation, the map key for Item will be set to "name". TODO: Fully-implement this, then remove the "experimental_" prefix.
weak	bool	optional	For Google-internal migration only. Do not use. [default = false]
uninterpreted_option	UninterpretedOption	repeated	The parser stores options it doesn't recognize here. See above.

Enum: FieldOptions.CType

Element	Value	Description
STRING	0	Default mode.
CORD	1	
STRING_PIECE	2	

Message: EnumOptions

Field	Type	Rule	Description
allow_alias	bool	optional	Set this option to false to disallow mapping different tag names to a same value. [default = true]
uninterpreted_option	UninterpretedOption	repeated	The parser stores options it doesn't recognize here. See above.

Message: EnumValueOptions

Field	Type	Rule	Description
uninterpreted_option	UninterpretedOption	repeated	The parser stores options it doesn't recognize here. See above.

Message: ServiceOptions

Field	Type	Rule	Description
uninterpreted_option	UninterpretedOption	repeated	The parser stores options it doesn't recognize here. See above.

Message: MethodOptions

Field	Type	Rule	Description
uninterpreted_option	UninterpretedOption	repeated	The parser stores options it doesn't recognize here. See above.

Message: UninterpretedOption

A message representing a option the parser does not recognize. This only appears in options protos created by the compiler::Parser class. DescriptorPool resolves these when building Descriptor objects. Therefore, options protos in descriptor objects (e.g. returned by Descriptor::options(), or produced by Descriptor::CopyTo()) will never have UninterpretedOptions in them.

Field	Type	Rule	Description
name	NamePart	repeated	
identifier_value	string	optional	The value of the uninterpreted option, in whatever type the tokenizer identified it as during parsing. Exactly one of these should be set.
positive_int_value	uint64	optional	
negative_int_value	int64	optional	
double_value	double	optional	
string_value	bytes	optional	
aggregate_value	string	optional	

Message: UninterpretedOption.NamePart

The name of the uninterpreted option. Each string represents a segment in a dot-separated name. is_extension is true iff a segment represents an extension (denoted with parentheses in options specs in .proto files). E.g.,{ ["foo", false], ["bar.baz", true], ["qux", false] } represents "foo.(bar.baz).qux".

Field	Type	Rule	Description
name_part	string	required	
is_extension	bool	required	

Message: SourceCodeInfo

Encapsulates information about the original source file from which a FileDescriptorProto was generated.

Field	Type	Rule	Description
location	Location	repeated	A Location identifies a piece of source code in a .proto file which corresponds to a particular definition. This information is intended to be useful to IDEs, code indexers, documentation generators, and similar tools. For example, say we have a file like: message Foo { optional string foo = 1; } Let's look at just the field definition: optional string foo = 1; ^ ^^ ^^ ^ ^^^ a bc de f ghi We have the following locations: span path represents [a,i) [4, 0, 2, 0] The whole field definition. [a,b) [4, 0, 2, 0, 4] The label (optional). [c,d) [4, 0,

Field	Type	Rule	Description
			2, 0, 5] The type (string). [e,f) [4, 0, 2, 0, 1] The name (foo). [g,h) [4, 0, 2, 0, 3] The number (1). Notes: - A location may refer to a repeated field itself (i.e. not to any particular index within it). This is used whenever a set of elements are logically enclosed in a single code segment. For example, an entire extend block (possibly containing multiple extension definitions) will have an outer location whose path refers to the "extensions" repeated field without an index. - Multiple locations may have the same path. This happens when a single logical declaration is spread out across multiple places. The most obvious example is the "extend" block again -- there may be multiple extend blocks in the same scope, each of which will have the same path. - A location's span is not always a subset of its parent's span. For example, the "extendee" of an extension declaration appears at the beginning of the "extend" block and is shared by all extensions within the block. - Just because a location's span is a subset of some other location's span does not mean that it is a descendent. For example, a "group" defines both a type and a field in a single declaration. Thus, the locations corresponding to the type and field and their components will overlap. - Code which tries to interpret locations should probably be designed to ignore those that it doesn't understand, as more types of locations could be recorded in the future.

Message: SourceCodeInfo.Location

Field	Type	Rule	Description
path	int32	repeated	Identifies which part of the FileDescriptorProto was defined at this location. Each element is a field number or an index. They form a path from the root FileDescriptorProto to the place where the definition. For example, this path: [4, 3, 2, 7, 1] refers to: file.message_type(3) // 4, 3 .field(7) // 2, 7 .name() // 1 This is because FileDescriptorProto.message_type

Field	Type	Rule	Description
			has field number 4: repeated DescriptorProto message_type = 4; and DescriptorProto.field has field number 2: repeated FieldDescriptorProto field = 2; and FieldDescriptorProto.name has field number 1: optional string name = 1; Thus, the above path gives the location of a field name. If we removed the last element: [4, 3, 2, 7] this path refers to the whole field declaration (from the beginning of the label to the terminating semicolon). [packed = true]
span	int32	repeated	Always has exactly three or four elements: start line, start column, end line (optional, otherwise assumed same as start line), end column. These are packed into a single field for efficiency. Note that line and column numbers are zero-based -- typically you will want to add 1 to each before displaying to a user. [packed = true]
leading_comments	string	optional	If this SourceCodeInfo represents a complete declaration, these are any comments appearing before and after the declaration which appear to be attached to the declaration. A series of line comments appearing on consecutive lines, with no other tokens appearing on those lines, will be treated as a single comment. Only the comment content is provided; comment markers (e.g. //) are stripped out. For block comments, leading whitespace and an asterisk will be stripped from the beginning of each line other than the first. Newlines are included in the output. Examples: optional int32 foo = 1; // Comment attached to foo. // Comment attached to bar. optional int32 bar = 2;

Field	Type	Rule	Description

			optional string baz = 3; // Comment attached to baz. // Another line attached to baz.
			
			// Comment attached to qux. // // Another line attached to qux. optional double qux = 4;
			
			optional string corge = 5; /* Block comment attached * to corge. Leading asterisks * will be removed. */ /* Block comment attached to * grault. */ optional int32 grault = 6;

trailing_comments	string	optional	

Scalar Value Types

A scalar message field can have one of the following types - the table shows the type specified in the .proto file, and the corresponding type in the automatically generated class:

Type	Notes	C++ Type	Java Type
double		double	double
float		float	float
int32	Uses variable-length encoding. Inefficient for encoding negative numbers - if your field is likely to have negative values, use sint32 instead.	int32	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers - if your field is likely to have negative values, use sint64 instead.	int64	long
uint32	Uses variable-length encoding.	uint32	int
uint64	Uses variable-length encoding.	uint64	long
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^28.	uint32	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^56.	uint64	long

Type	Notes	C++ Type	Java Type
sfixed32	Always four bytes..	int32	int
sfixed64	Always eight bytes.	int64	long
bool		bool	boolean
string	A string must always contain UTF-8 encoded or 7-bit ASCII text.	string	String
bytes	May contain any arbitrary sequence of bytes.	string	ByteString