JavaScript Generated Code

This page describes exactly what JavaScript code the protocol buffer compiler generates for any given protocol definition. Any differences between proto2 and proto3 generated code are highlighted. You should read the proto2 language guide (/web/20200928202524/https://developers.google.com/protocol-buffers/docs/proto) and/or the proto3 language guide (/web/20200928202524/https://developers.google.com/protocol-buffers/docs/proto3) before reading this document.

Compiler Invocation

The protocol buffer compiler produces JavaScript output when invoked with the `--js_out=` command-line flag. The parameter to the `--js_out=` option is the directory where you want the compiler to write your JavaScript output. The exact output depends on whether you want to use Closure-style imports or CommonJS-style imports; the compiler supports both.

rt for ES6-style imports is not implemented yet. Browsers can be supported by using Browserify, webpack, Closu
ler, or similar to resolve imports at compile time.

Closure Imports

By default, the compiler generates code with Closure-style imports. If you specify a **`library`** option when running the compiler, the compiler creates a single **`.js`** file with your specified library name. Otherwise the compiler generates a **`.js`** file for each *message* in your **`.proto`** file. The names of the output files are computed by taking the **`library`** value or message name (lowercased), with the following changes:

- A **`.js`** extension is added.
- The proto path (specified with the **`--proto_path=`** or **`-I`** command-line flag) is replaced with the output path (specified with the **`--js_out=`** flag).

So, for example, let's say you invoke the compiler as follows:

```
›c --proto_path=src --js_out=library=whizz/ponycopter,binary:build/gen src/foo.prot‹
```

The compiler will read the files **`src/foo.proto`** and **`src/bar/baz.proto`** and produce a single output file, **`build/gen/whizz/ponycopter.js`**. The compiler will automatically create the directory **`build/gen/whizz`** if necessary, but it will *not* create **`build`** or **`build/gen`**; they must already exist.

The generated file(s) **`goog.provide()`** all the types defined in your **`.proto`** file(s), and **`goog.require()`** many types in the core protocol buffers library and Google Closure library. Make sure your that your **`goog.provide()`** / **`goog.require()`** setup can find all of your generated code, the core library **`.js`** files (https://web.archive.org/web/20200928202524/https://github.com/protocolbuffers/protobuf/tree/master/js), and the Google Closure library itself.

You should be able to import your generated types with statements like:

```
 require('proto.my.package.MyMessage');

ıessage = proto.my.package.MyMessage();
```

CommonJS Imports

To specify that you want to use CommonJS-style imports instead of the default Closure style, you run the compiler with the **`import_style=commonjs`** option. The names of the output files are computed by taking the name of the each input **`.proto`** file and making two changes:

- The extension (**`.proto`**) is replaced with **`_pb.js`**.

- The proto path (specified with the `--proto_path=` or `-I` command-line flag) is replaced with the output path (specified with the `--js_out=` flag).

Specifying a `library` option is ignored with this import style.

So, for example, let's say you invoke the compiler as follows:

```
ıc --proto_path=src --js_out=import_style=commonjs,binary:build/gen src/foo.proto sı
```

The compiler will read the files `src/foo.proto` and `src/bar/baz.proto` and produce two output files: `build/gen/foo_pb.js` and `build/gen/bar/baz_pb.js`. The compiler will automatically create the directory `build/gen/bar` if necessary, but it will *not* create `build` or `build/gen`; they must already exist.

The generated code depends on the core runtime, which should be in a file called `google-protobuf.js`. If you installed protoc via `npm`, this file should already be built and available. If you are running from GitHub, you need to build it first by running:

```
 dist
```

You should be able to import your generated types with statements like:

```
ıessages = require('./messages_pb');

ıessage = new messages.MyMessage();
```

Compiler Options

The protocol buffer compiler for JavaScript has many options to customize its output in addition to the `library` and `import_style` options mentioned above. For example:

- `binary`: Using this option generates code that lets you serialize and deserialize your proto from the protocol buffers binary wire format. We recommend that you enable this option.

  ```
  --js_out=library=myprotos_lib.js,binary:.
  ```

- `error_on_name_conflict`: Using this option means a compiler error is returned if there are two types in your input that would generate output files with the same name.

As in the above examples, multiple options can be specified, separated by commas. You can see

a complete list of available options in `js_generator.h` (https://web.archive.org/web/20200928202524/https://github.com/protocolbuffers/protobuf/blob/master/src/google/protobuf/compiler/js/js_generator.h#L53)
.

Packages

Packages and Closure Imports

If you are using Closure-style imports and a `.proto` file contains a package declaration, the generated code uses the proto's `package` as part of the JavaScript namespace for your message types. For example, a proto package name of `example.high_score` results in a JavaScript namespace of `proto.example.high_score`.

```
provide('proto.example.high_score.Ponycopter');
```

Otherwise, if a `.proto` file does not contain a package declaration, the generated code just uses `proto` as the namespace for your message types, which is the root of the protocol buffers namespace.

Packages and CommonJS Imports

If you are using CommonJS-style imports, any package declarations in your `.proto` files are ignored by the compiler.

Messages

Given a simple message declaration:

```
ıge Foo {}
```

the protocol buffer compiler generates a class called `Foo`. `Foo` inherits from `jspb.Message` (https://web.archive.org/web/20200928202524/https://github.com/protocolbuffers/protobuf/blob/master/js/message.js)
.

You should *not* create your own `Foo` subclasses. Generated classes are not designed for subclassing and may lead to "fragile base class" problems.

Your generated class has accessors for all its fields (which we'll look at in the following sections) and the following methods that apply to the entire message:

- **`toObject()`**: Returns an object representation of the message, suitable for use in Soy templates. This method comes in static and instance versions. Field names that are reserved in JavaScript (https://web.archive.org/web/20200928202524/http://www.w3schools.com/js/js_reserved.asp) are renamed to `pb_`*`name`*. If you don't want to generate this method (for instance, if you're not going to use it and are concerned about code size), set **`jspb.Message.GENERATE_TO_OBJECT`** (https://web.archive.org/web/20200928202524/https://github.com/protocolbuffers/protobuf/blob/master/js/message.js) to false before code generation. Note that this representation is not the same as proto3's JSON representation (/web/20200928202524/https://developers.google.com/protocol-buffers/docs/proto3#json).
- **`cloneMessage()`**: Creates a deep clone of this message and its fields.

The following methods are also provided if you have enabled the `binary` option when generating your code:

- **`deserializeBinary()`**: Static method. Deserializes a message from protocol buffers binary wire format and returns a new populated message object. Does not preserve any unknown fields in the binary message.
- **`deserializeBinaryFromReader()`**: Static method. Deserializes a message in protocol buffers binary wire format from the provided `BinaryReader` (https://web.archive.org/web/20200928202524/https://github.com/protocolbuffers/protobuf/blob/master/js/binary/reader.js) into the provided message object. Does not preserve any unknown fields in the binary message.
- **`serializeBinary()`**: Serializes this message to protocol buffers binary wire format.
- **`serializeBinaryToWriter()`**: Serializes this message in protocol buffers binary wire format to the specified `BinaryWriter` (https://web.archive.org/web/20200928202524/https://github.com/protocolbuffers/protobuf/blob/master/js/binary/writer.js). This method has a static variant where you can serialize a specified message to the BinaryWriter.

Fields

The protocol buffer compiler generates accessors for each field in your protocol buffer message. The exact accessors depend on its type and whether it is a singular, repeated, map, or oneof field.

Note that the generated accessors always use camel-case naming, even if the field name in the `.proto` file uses lower-case with underscores (as it should (/web/20200928202524/https://developers.google.com/protocol-buffers/docs/style)). The case-conversion works as follows:

1. The first letter in the method name is always lower-case.
2. If the field name contains an underscore, the underscore is removed, and the following letter is capitalized.

Thus, the proto field `foo_bar_baz` has, for example, a `getFooBarBaz()` method.

Singular Scalar Fields (proto2)

For either of these field definitions:

```
ınal int32 foo = 1;
.red int32 foo = 1;
```

the compiler generates the following instance methods:

- `setFoo()`: Set the value of `foo`.
- `getFoo()`: Get the value of `foo`. If the field has not been set, returns the default value for its type.
- `hasFoo()`: Returns `true` if this field has been set.
- `clearFoo()`: Clears the value of this field: after this has been called `hasFoo()` returns `false` and `getFoo()` returns the default value.

Similar methods are generated for any of protocol buffers' scalar types (/web/20200928202524/https://developers.google.com/protocol-buffers/docs/proto#scalar).

Singular Scalar Fields (proto3)

For this field definition:

```
: foo = 1;
```

the compiler generates the following instance methods:

- `setFoo()`: Set the value of `foo`.
- `getFoo()`: Get the value of `foo`.

Similar methods are generated for any of protocol buffers' scalar types (/web/20200928202524/https://developers.google.com/protocol-buffers/docs/proto3#scalar).

Bytes Fields

For this field definition:

```
; foo = 1;
```

the compiler generates the same methods as for other scalar value types. The **`set..`** method accepts either a base-64 encoded string or a `Uint8Array`. The **`get..`** method returns whichever representation was set last. However, there are also special methods generated that allow you to coerce the returned representation to your preferred version:

- **`getFoo_asB64()`**: Returns the value of `foo` as a base-64 encoded string.
- **`getFoo_asU8()`**: Returns the value of `foo` as a `Uint8Array`.

Singular Message Fields

Given the message type:

```
ıge Bar {}
```

For a message with a `Bar` field:

```
'oto2
ıge Baz {
:ional Bar foo = 1;
 The generated code is the same result if required instead of optional.


'oto3
ıge Baz {
' foo = 1;
```

the compiler generates the following instance methods:

- **`setFoo()`**: Set the value of `foo`. When called with `undefined`, it is equivalent to calling **`clearFoo()`**.

- **`getFoo()`**: Get the value of **`foo`**. Returns **`undefined`** if the field has not been set.
- **`hasFoo()`**: Returns **`true`** if this field has been set. Equivalent to **`!!getFoo()`**.
- **`clearFoo()`**: Clears the value of this field to undefined.

Repeated Fields

For this message with a repeated field:

```
ıge Baz {
ıeated int32 foo = 1;
```

the compiler generates the following instance methods:

- **`setFooList()`**: Set the value of **`foo`** to the specified JavaScript array. Returns the message itself for chaining.
- **`addFoo()`**: Appends a value of **`foo`** to the end of the list of foos that was in the message. Returns the outer message for chaining **only if** the added value was a primitive. For added messages, returns the message that was added (b/138079947).
- **`getFooList()`**: Gets the value of **`foo`** as a JavaScript array. The returned array is never undefined and each element is never undefined. You should **not** mutate the list returned from this method.
- **`clearFooList()`**: Clears the value of this field to **`[]`**.

Map Fields

For this message with a map field:

```
ıge Bar {}

ıge Baz {
ı<string, Bar> foo = 1;
```

the compiler generates the following instance method:

- **`getFooMap()`**: Returns the **`Map`** (https://web.archive.org/web/20200928202524/https://github.com/protocolbuffers/protobuf/blob/master/js/map.js) containing **`foo`**'s key-value pairs. You can then use **`Map`** methods to interact with the map.

Oneof Fields

For this message with a oneof field:

```
ıge account;
ıge Profile {
:of avatar {
;tring image_url = 1;
›ytes image_data = 2;
```

The class corresponding to **`Profile`** will have accessor methods just like regular fields (**`getImageUrl()`**, **`getImageData()`**). However, unlike regular fields, at most one of the fields in a oneof can be set at a time, so setting one field will clear the others. Also note that if you are using proto3, the compiler generates **`has..`** and **`clear..`** accessors for oneof fields, even for scalar types.

In addition to the regular accessor methods, the compiler generates a special method to check which field in the oneof is set: for our example, the method is **`getAvatarCase()`**. The possible return values for this are defined in the **`AvatarCase`** enum:

```
›.account.Profile.AvatarCase = {
ıTAR_NOT_SET: 0,
ıGE_URL: 1,
ıGE_DATA: 2
```

Enumerations

Given an enumeration like:

```
ıge SearchRequest {
ım Corpus {
JNIVERSAL = 0;
/EB = 1;
:MAGES = 2;
.OCAL = 3;
IEWS = 4;
'RODUCTS = 5;
'IDEO = 6;
```

```
·pus corpus = 1;
```

the protocol buffer compiler generates a corresponding JavaScript enum.

```
).SearchRequest.Corpus = {
IVERSAL: 0,
3: 1,
AGES: 2,
CAL: 3,
VS: 4,
)DUCTS: 5,
)EO: 6
```

The compiler also generates getters and setters for enum fields, just like regular singular scalar fields. Note that in proto3, you can set an enum field to any value. In proto2, you should provide one of the specified enum values.

Any

Given an **Any** (/web/20200928202524/https://developers.google.com/protocol-buffers/docs/proto3#any) field like this:

```
·t "google/protobuf/any.proto";
ıge foo;

ıge Bar {}

ıge ErrorStatus {
·ing message = 1;
)gle.protobuf.Any details = 2;
```

In our generated code, the getter for the `details` field returns an instance of `proto.google.protobuf.Any`. This provides the following special methods:

```
:turns the fully qualified proto name of the packed message, if any.
·eturn {string|undefined}
```

```
).google.protobuf.Any.prototype.getTypeName;


ıcks the given message instance into this Any.
)aram {!Uint8Array} serialized The serialized data to pack.
)aram {string} name The fully qualified proto name of the packed message.
)aram {string=} opt_typeUrlPrefix the type URL prefix.

).google.protobuf.Any.prototype.pack;


:emplate T
ıpacks this Any into the given message object.
)aram {function(Uint8Array):T} deserialize Function that will deserialize
  the binary data properly.
)aram {string} name The expected type name of this message object.
:eturn {?T} If the name matched the expected name, returns the deserialized
  object, otherwise returns null.

).google.protobuf.Any.prototype.unpack;
```

Example:

```
:oring an arbitrary message type in Any.
: status = new proto.foo.ErrorStatus();
: any = new Any();
: binarySerialized = ...;
)ack(binarySerialized, 'foo.Bar');
)le.log(any.getTypeName());  // foo.Bar

:ading an arbitrary message from Any.
: bar = any.unpack(proto.foo.Bar.deserializeBinary, 'foo.Bar');
```

Last updated 2020-04-23 UTC.