

   

\*\*Protocol Buffers\*\* are a language-neutral, platform-neutral, extensible way of serializing structured data for use in communications protocols, data storage, and more, originally designed at Google ([see](https://developers.google.com/protocol-buffers/)). \*\*protobuf.js\*\* is a pure JavaScript implementation with [TypeScript](https://www.typescriptlang.org) support for [node.js](https://nodejs.org) and the browser. It's easy to use, blazingly fast and works out of the box with [.proto](https://developers.google.com/protocol-buffers/docs/proto) files! Contents -------- \* [Installation](#installation)

How to include protobuf.js in your project. \* [Usage](#usage)

A brief introduction to using the toolset. \* [Valid Message](#valid-message) \* [Toolset](#toolset)

\* [Examples](#examples)

A few examples to get you started. \* [Using .proto files](#using-proto-files) \* [Using JSON descriptors](#using-json-descriptors) \* [Using reflection only](#using-reflection-only) \* [Using custom classes](#using-custom-classes) \* [Using services](#using-services) \* [Usage with TypeScript](#usage-with-typescript)

\* [Command line](#command-line)

How to use the command line utility. \* [pbjs for JavaScript](#pbjs-for-javascript) \* [pbts for TypeScript](#pbts-for-typescript) \* [Reflection vs. static code](#reflection-vs-static-code) \* [Command line API](#command-line-api)

\* [Additional documentation](#additional-documentation)

A list of available documentation resources. \* [Performance](#performance)

A few internals and a benchmark on performance. \* [Compatibility](#compatibility)

Notes on compatibility regarding browsers and optional libraries. \* [Building](#building)

How to build the library and its components yourself. Installation --------------- ### node.js ``` $> npm install protobufjs [--save --save-prefix=~] ``` ```js var protobuf = require("protobufjs"); ``` \*\*Note\*\* that this library's versioning scheme is not semver-compatible for historical reasons. For guaranteed backward compatibility, always depend on `~6.A.B` instead of `^6.A.B` (hence the `--save-prefix` above). ### Browsers Development: ``` ``` Production: ``` ``` \*\*Remember\*\* to replace the version tag with the exact [release](https://github.com/dcodeIO/protobuf.js/tags) your project depends upon. The library supports CommonJS and AMD loaders and also exports globally as `protobuf`. ### Distributions Where bundle size is a factor, there are additional stripped-down versions of the [full library][dist-full] (~19kb gzipped) available that exclude certain functionality: \* When working with JSON descriptors (i.e. generated by [pbjs](#pbjs-for-javascript)) and/or reflection only, see the [light library][dist-light] (~16kb gzipped) that excludes the parser. CommonJS entry point is: ```js var protobuf = require("protobufjs/light"); ``` \* When working with statically generated code only, see the [minimal library][dist-minimal] (~6.5kb gzipped) that also excludes reflection. CommonJS entry point is: ```js var protobuf = require("protobufjs/minimal"); ``` [dist-full]: https://github.com/dcodeIO/protobuf.js/tree/master/dist [dist-light]: https://github.com/dcodeIO/protobuf.js/tree/master/dist/light [dist-minimal]: https://github.com/dcodeIO/protobuf.js/tree/master/dist/minimal Usage ----- Because JavaScript is a dynamically typed language, protobuf.js introduces the concept of a \*\*valid message\*\* in order to provide the best possible [performance](#performance) (and, as a side product, proper typings): ### Valid message > A valid message is an object (1) not missing any required fields and (2) exclusively composed of JS types understood by the wire format writer. There are two possible types of valid messages and the encoder is able to work with both of these for convenience: \* \*\*Message instances\*\* (explicit instances of message classes with default values on their prototype) always (have to) satisfy the requirements of a valid message by design and \* \*\*Plain JavaScript objects\*\* that just so happen to be composed in a way satisfying the requirements of a valid message as well. In a nutshell, the wire format writer understands the following types: | Field type | Expected JS type (create, encode) | Conversion (fromObject) |------------|-----------------------------------|------------------------ | s-/u-/int32

s-/fixed32 | `number` (32 bit integer) | value | 0 if signed

`value >>> 0` if unsigned | s-/u-/int64

s-/fixed64 | `Long`-like (optimal)

`number` (53 bit integer) | `Long.fromValue(value)` with long.js

`parseInt(value, 10)` otherwise | float

double | `number` | `Number(value)` | bool | `boolean` | `Boolean(value)` | string | `string` | `String(value)` | bytes | `Uint8Array` (optimal)

`Buffer` (optimal under node)

`Array.` (8 bit integers) | `base64.decode(value)` if a `string`

`Object` with non-zero `.length` is assumed to be buffer-like | enum | `number` (32 bit integer) | Looks up the numeric id if a `string` | message | Valid message | `Message.fromObject(value)` \* Explicit `undefined` and `null` are considered as not set if the field is optional. \* Repeated fields are `Array.`. \* Map fields are `Object.` with the key being the string representation of the respective value or an 8 characters long binary hash string for `Long`-likes. \* Types marked as \*optimal\* provide the best performance because no conversion step (i.e. number to low and high bits or base64 string to buffer) is required. ### Toolset With that in mind and again for performance reasons, each message class provides a distinct set of methods with each method doing just one thing. This avoids unnecessary assertions / redundant operations where performance is a concern but also forces a user to perform verification (of plain JavaScript objects that \*might\* just so happen to be a valid message) explicitly where necessary - for example when dealing with user input. \*\*Note\*\* that `Message` below refers to any message class. \* \*\*Message.verify\*\*(message: `Object`): `null|string`

verifies that a \*\*plain JavaScript object\*\* satisfies the requirements of a valid message and thus can be encoded without issues. Instead of throwing, it returns the error message as a string, if any. ```js var payload = "invalid (not an object)"; var err = AwesomeMessage.verify(payload); if (err) throw Error(err); ``` \* \*\*Message.encode\*\*(message: `Message|Object` [, writer: `Writer`]): `Writer`

encodes a \*\*message instance\*\* or valid \*\*plain JavaScript object\*\*. This method does not implicitly verify the message and it's up to the user to make sure that the payload is a valid message. ```js var buffer = AwesomeMessage.encode(message).finish(); ``` \* \*\*Message.encodeDelimited\*\*(message: `Message|Object` [, writer: `Writer`]): `Writer`

works like `Message.encode` but additionally prepends the length of the message as a varint. \* \*\*Message.decode\*\*(reader: `Reader|Uint8Array`): `Message`

decodes a buffer to a \*\*message instance\*\*. If required fields are missing, it throws a `util.ProtocolError` with an `instance` property set to the so far decoded message. If the wire format is invalid, it throws an `Error`. ```js try { var decodedMessage = AwesomeMessage.decode(buffer); } catch (e) { if (e instanceof protobuf.util.ProtocolError) { // e.instance holds the so far decoded message with missing required fields } else { // wire format is invalid } } ``` \* \*\*Message.decodeDelimited\*\*(reader: `Reader|Uint8Array`): `Message`

works like `Message.decode` but additionally reads the length of the message prepended as a varint. \* \*\*Message.create\*\*(properties: `Object`): `Message`

creates a new \*\*message instance\*\* from a set of properties that satisfy the requirements of a valid message. Where applicable, it is recommended to prefer `Message.create` over `Message.fromObject` because it doesn't perform possibly redundant conversion. ```js var message = AwesomeMessage.create({ awesomeField: "AwesomeString" }); ``` \* \*\*Message.fromObject\*\*(object: `Object`): `Message`

converts any non-valid \*\*plain JavaScript object\*\* to a \*\*message instance\*\* using the conversion steps outlined within the table above. ```js var message = AwesomeMessage.fromObject({ awesomeField: 42 }); // converts awesomeField to a string ``` \* \*\*Message.toObject\*\*(message: `Message` [, options: `ConversionOptions`]): `Object`

converts a \*\*message instance\*\* to an arbitrary \*\*plain JavaScript object\*\* for interoperability with other libraries or storage. The resulting plain JavaScript object \*might\* still satisfy the requirements of a valid message depending on the actual conversion options specified, but most of the time it does not. ```js var object = AwesomeMessage.toObject(message, { enums: String, // enums as string names longs: String, // longs as strings (requires long.js) bytes: String, // bytes as base64 encoded strings defaults: true, // includes default values arrays: true, // populates empty arrays (repeated fields) even if defaults=false objects: true, // populates empty objects (map fields) even if defaults=false oneofs: true // includes virtual oneof fields set to the present field's name }); ``` For reference, the following diagram aims to display relationships between the different methods and the concept of a valid message:



> In other words: `verify` indicates that calling `create` or `encode` directly on the plain object will [result in a valid message respectively] succeed. `fromObject`, on the other hand, does conversion from a broader range of plain objects to create valid messages. ([ref](https://github.com/dcodeIO/protobuf.js/issues/748#issuecomment-291925749)) Examples -------- ### Using .proto files It is possible to load existing .proto files using the full library, which parses and compiles the definitions to ready to use (reflection-based) message classes: ```protobuf // awesome.proto package awesomepackage; syntax = "proto3"; message AwesomeMessage { string awesome\_field = 1; // becomes awesomeField } ``` ```js protobuf.load("awesome.proto", function(err, root) { if (err) throw err; // Obtain a message type var AwesomeMessage = root.lookupType("awesomepackage.AwesomeMessage"); // Exemplary payload var payload = { awesomeField: "AwesomeString" }; // Verify the payload if necessary (i.e. when possibly incomplete or invalid) var errMsg = AwesomeMessage.verify(payload); if (errMsg) throw Error(errMsg); // Create a new message var message = AwesomeMessage.create(payload); // or use .fromObject if conversion is necessary // Encode a message to an Uint8Array (browser) or Buffer (node) var buffer = AwesomeMessage.encode(message).finish(); // ... do something with buffer // Decode an Uint8Array (browser) or Buffer (node) to a message var message = AwesomeMessage.decode(buffer); // ... do something with message // If the application uses length-delimited buffers, there is also encodeDelimited and decodeDelimited. // Maybe convert the message back to a plain object var object = AwesomeMessage.toObject(message, { longs: String, enums: String, bytes: String, // see ConversionOptions }); }); ``` Additionally, promise syntax can be used by omitting the callback, if preferred: ```js protobuf.load("awesome.proto") .then(function(root) { ... }); ``` ### Using JSON descriptors The library utilizes JSON descriptors that are equivalent to a .proto definition. For example, the following is identical to the .proto definition seen above: ```json // awesome.json { "nested": { "AwesomeMessage": { "fields": { "awesomeField": { "type": "string", "id": 1 } } } } } ``` JSON descriptors closely resemble the internal reflection structure: | Type (T) | Extends | Type-specific properties |--------------------|--------------------|------------------------- | \*ReflectionObject\* | | options | \*Namespace\* | \*ReflectionObject\* | nested | Root | \*Namespace\* | \*\*nested\*\* | Type | \*Namespace\* | \*\*fields\*\* | Enum | \*ReflectionObject\* | \*\*values\*\* | Field | \*ReflectionObject\* | rule, \*\*type\*\*, \*\*id\*\* | MapField | Field | \*\*keyType\*\* | OneOf | \*ReflectionObject\* | \*\*oneof\*\* (array of field names) | Service | \*Namespace\* | \*\*methods\*\* | Method | \*ReflectionObject\* | type, \*\*requestType\*\*, \*\*responseType\*\*, requestStream, responseStream \* \*\*Bold properties\*\* are required. \*Italic types\* are abstract. \* `T.fromJSON(name, json)` creates the respective reflection object from a JSON descriptor \* `T#toJSON()` creates a JSON descriptor from the respective reflection object (its name is used as the key within the parent) Exclusively using JSON descriptors instead of .proto files enables the use of just the light library (the parser isn't required in this case). A JSON descriptor can either be loaded the usual way: ```js protobuf.load("awesome.json", function(err, root) { if (err) throw err; // Continue at "Obtain a message type" above }); ``` Or it can be loaded inline: ```js var jsonDescriptor = require("./awesome.json"); // exemplary for node var root = protobuf.Root.fromJSON(jsonDescriptor); // Continue at "Obtain a message type" above ``` ### Using reflection only Both the full and the light library include full reflection support. One could, for example, define the .proto definitions seen in the examples above using just reflection: ```js ... var Root = protobuf.Root, Type = protobuf.Type, Field = protobuf.Field; var AwesomeMessage = new Type("AwesomeMessage").add(new Field("awesomeField", 1, "string")); var root = new Root().define("awesomepackage").add(AwesomeMessage); // Continue at "Create a new message" above ... ``` Detailed information on the reflection structure is available within the [API documentation](#additional-documentation). ### Using custom classes Message classes can also be extended with custom functionality and it is also possible to register a custom constructor with a reflected message type: ```js ... // Define a custom constructor function AwesomeMessage(properties) { // custom initialization code ... } // Register the custom constructor with its reflected type (\*) root.lookupType("awesomepackage.AwesomeMessage").ctor = AwesomeMessage; // Define custom functionality AwesomeMessage.customStaticMethod = function() { ... }; AwesomeMessage.prototype.customInstanceMethod = function() { ... }; // Continue at "Create a new message" above ``` (\*) Besides referencing its reflected type through `AwesomeMessage.$type` and `AwesomeMesage#$type`, the respective custom class is automatically populated with: \* `AwesomeMessage.create` \* `AwesomeMessage.encode` and `AwesomeMessage.encodeDelimited` \* `AwesomeMessage.decode` and `AwesomeMessage.decodeDelimited` \* `AwesomeMessage.verify` \* `AwesomeMessage.fromObject`, `AwesomeMessage.toObject` and `AwesomeMessage#toJSON` Afterwards, decoded messages of this type are `instanceof AwesomeMessage`. Alternatively, it is also possible to reuse and extend the internal constructor if custom initialization code is not required: ```js ... // Reuse the internal constructor var AwesomeMessage = root.lookupType("awesomepackage.AwesomeMessage").ctor; // Define custom functionality AwesomeMessage.customStaticMethod = function() { ... }; AwesomeMessage.prototype.customInstanceMethod = function() { ... }; // Continue at "Create a new message" above ``` ### Using services The library also supports consuming services but it doesn't make any assumptions about the actual transport channel. Instead, a user must provide a suitable RPC implementation, which is an asynchronous function that takes the reflected service method, the binary request and a node-style callback as its parameters: ```js function rpcImpl(method, requestData, callback) { // perform the request using an HTTP request or a WebSocket for example var responseData = ...; // and call the callback with the binary response afterwards: callback(null, responseData); } ``` Below is a working example with a typescript implementation using grpc npm package. ```ts const grpc = require('grpc') const Client = grpc.makeGenericClientConstructor({}) const client = new Client( grpcServerUrl, grpc.credentials.createInsecure() ) const rpcImpl = function(method, requestData, callback) { client.makeUnaryRequest( method.name, arg => arg, arg => arg, requestData, callback ) } ``` Example: ```protobuf // greeter.proto syntax = "proto3"; service Greeter { rpc SayHello (HelloRequest) returns (HelloReply) {} } message HelloRequest { string name = 1; } message HelloReply { string message = 1; } ``` ```js ... var Greeter = root.lookup("Greeter"); var greeter = Greeter.create(/\* see above \*/ rpcImpl, /\* request delimited? \*/ false, /\* response delimited? \*/ false); greeter.sayHello({ name: 'you' }, function(err, response) { console.log('Greeting:', response.message); }); ``` Services also support promises: ```js greeter.sayHello({ name: 'you' }) .then(function(response) { console.log('Greeting:', response.message); }); ``` There is also an [example for streaming RPC](https://github.com/dcodeIO/protobuf.js/blob/master/examples/streaming-rpc.js). Note that the service API is meant for clients. Implementing a server-side endpoint pretty much always requires transport channel (i.e. http, websocket, etc.) specific code with the only common denominator being that it decodes and encodes messages. ### Usage with TypeScript The library ships with its own [type definitions](https://github.com/dcodeIO/protobuf.js/blob/master/index.d.ts) and modern editors like [Visual Studio Code](https://code.visualstudio.com/) will automatically detect and use them for code completion. The npm package depends on [@types/node](https://www.npmjs.com/package/@types/node) because of `Buffer` and [@types/long](https://www.npmjs.com/package/@types/long) because of `Long`. If you are not building for node and/or not using long.js, it should be safe to exclude them manually. #### Using the JS API The API shown above works pretty much the same with TypeScript. However, because everything is typed, accessing fields on instances of dynamically generated message classes requires either using bracket-notation (i.e. `message["awesomeField"]`) or explicit casts. Alternatively, it is possible to use a [typings file generated for its static counterpart](#pbts-for-typescript). ```ts import { load } from "protobufjs"; // respectively "./node\_modules/protobufjs" load("awesome.proto", function(err, root) { if (err) throw err; // example code const AwesomeMessage = root.lookupType("awesomepackage.AwesomeMessage"); let message = AwesomeMessage.create({ awesomeField: "hello" }); console.log(`message = ${JSON.stringify(message)}`); let buffer = AwesomeMessage.encode(message).finish(); console.log(`buffer = ${Array.prototype.toString.call(buffer)}`); let decoded = AwesomeMessage.decode(buffer); console.log(`decoded = ${JSON.stringify(decoded)}`); }); ``` #### Using generated static code If you generated static code to `bundle.js` using the CLI and its type definitions to `bundle.d.ts`, then you can just do: ```ts import { AwesomeMessage } from "./bundle.js"; // example code let message = AwesomeMessage.create({ awesomeField: "hello" }); let buffer = AwesomeMessage.encode(message).finish(); let decoded = AwesomeMessage.decode(buffer); ``` #### Using decorators The library also includes an early implementation of [decorators](https://www.typescriptlang.org/docs/handbook/decorators.html). \*\*Note\*\* that decorators are an experimental feature in TypeScript and that declaration order is important depending on the JS target. For example, `@Field.d(2, AwesomeArrayMessage)` requires that `AwesomeArrayMessage` has been defined earlier when targeting `ES5`. ```ts import { Message, Type, Field, OneOf } from "protobufjs/light"; // respectively "./node\_modules/protobufjs/light.js" export class AwesomeSubMessage extends Message { @Field.d(1, "string") public awesomeString: string; } export enum AwesomeEnum { ONE = 1, TWO = 2 } @Type.d("SuperAwesomeMessage") export class AwesomeMessage extends Message { @Field.d(1, "string", "optional", "awesome default string") public awesomeField: string; @Field.d(2, AwesomeSubMessage) public awesomeSubMessage: AwesomeSubMessage; @Field.d(3, AwesomeEnum, "optional", AwesomeEnum.ONE) public awesomeEnum: AwesomeEnum; @OneOf.d("awesomeSubMessage", "awesomeEnum") public which: string; } // example code let message = new AwesomeMessage({ awesomeField: "hello" }); let buffer = AwesomeMessage.encode(message).finish(); let decoded = AwesomeMessage.decode(buffer); ``` Supported decorators are: \* \*\*Type.d(typeName?: `string`)\*\*   \*(optional)\*

annotates a class as a protobuf message type. If `typeName` is not specified, the constructor's runtime function name is used for the reflected type. \* \*\*Field.d<T>(fieldId: `number`, fieldType: `string | Constructor`, fieldRule?: `"optional" | "required" | "repeated"`, defaultValue?: `T`)\*\*

annotates a property as a protobuf field with the specified id and protobuf type. \* \*\*MapField.d<T extends { [key: string]: any }>(fieldId: `number`, fieldKeyType: `string`, fieldValueType. `string | Constructor<{}>`)\*\*

annotates a property as a protobuf map field with the specified id, protobuf key and value type. \* \*\*OneOf.d<T extends string>(...fieldNames: `string[]`)\*\*

annotates a property as a protobuf oneof covering the specified fields. Other notes: \* Decorated types reside in `protobuf.roots["decorated"]` using a flat structure, so no duplicate names. \* Enums are copied to a reflected enum with a generic name on decorator evaluation because referenced enum objects have no runtime name the decorator could use. \* Default values must be specified as arguments to the decorator instead of using a property initializer for proper prototype behavior. \* Property names on decorated classes must not be renamed on compile time (i.e. by a minifier) because decorators just receive the original field name as a string. \*\*ProTip!\*\* Not as pretty, but you can [use decorators in plain JavaScript](https://github.com/dcodeIO/protobuf.js/blob/master/examples/js-decorators.js) as well. Command line ------------ \*\*Note\*\* that moving the CLI to [its own package](./cli) is a work in progress. At the moment, it's still part of the main package. The command line interface (CLI) can be used to translate between file formats and to generate static code as well as TypeScript definitions. ### pbjs for JavaScript ``` Translates between file formats and generates static code. -t, --target Specifies the target format. Also accepts a path to require a custom target. json JSON representation json-module JSON representation as a module proto2 Protocol Buffers, Version 2 proto3 Protocol Buffers, Version 3 static Static code without reflection (non-functional on its own) static-module Static code without reflection as a module -p, --path Adds a directory to the include path. -o, --out Saves to a file instead of writing to stdout. --sparse Exports only those types referenced from a main file (experimental). Module targets only: -w, --wrap Specifies the wrapper to use. Also accepts a path to require a custom wrapper. default Default wrapper supporting both CommonJS and AMD commonjs CommonJS wrapper amd AMD wrapper es6 ES6 wrapper (implies --es6) closure A closure adding to protobuf.roots where protobuf is a global -r, --root Specifies an alternative protobuf.roots name. -l, --lint Linter configuration. Defaults to protobuf.js-compatible rules: eslint-disable block-scoped-var, no-redeclare, no-control-regex, no-prototype-builtins --es6 Enables ES6 syntax (const/let instead of var) Proto sources only: --keep-case Keeps field casing instead of converting to camel case. Static targets only: --no-create Does not generate create functions used for reflection compatibility. --no-encode Does not generate encode functions. --no-decode Does not generate decode functions. --no-verify Does not generate verify functions. --no-convert Does not generate convert functions like from/toObject --no-delimited Does not generate delimited encode/decode functions. --no-beautify Does not beautify generated code. --no-comments Does not output any JSDoc comments. --force-long Enforces the use of 'Long' for s-/u-/int64 and s-/fixed64 fields. --force-number Enforces the use of 'number' for s-/u-/int64 and s-/fixed64 fields. --force-message Enforces the use of message instances instead of plain objects. usage: pbjs [options] file1.proto file2.json ... (or pipe) other | pbjs [options] - ``` For production environments it is recommended to bundle all your .proto files to a single .json file, which minimizes the number of network requests and avoids any parser overhead (hint: works with just the \*\*light\*\* library): ``` $> pbjs -t json file1.proto file2.proto > bundle.json ``` Now, either include this file in your final bundle: ```js var root = protobuf.Root.fromJSON(require("./bundle.json")); ``` or load it the usual way: ```js protobuf.load("bundle.json", function(err, root) { ... }); ``` Generated static code, on the other hand, works with just the \*\*minimal\*\* library. For example ``` $> pbjs -t static-module -w commonjs -o compiled.js file1.proto file2.proto ``` will generate static code for definitions within `file1.proto` and `file2.proto` to a CommonJS module `compiled.js`. \*\*ProTip!\*\* Documenting your .proto files with `/\*\* ... \*/`-blocks or (trailing) `/// ...` lines translates to generated static code. ### pbts for TypeScript ``` Generates TypeScript definitions from annotated JavaScript files. -o, --out Saves to a file instead of writing to stdout. -g, --global Name of the global object in browser environments, if any. --no-comments Does not output any JSDoc comments. Internal flags: -n, --name Wraps everything in a module of the specified name. -m, --main Whether building the main library without any imports. usage: pbts [options] file1.js file2.js ... (or) other | pbts [options] - ``` Picking up on the example above, the following not only generates static code to a CommonJS module `compiled.js` but also its respective TypeScript definitions to `compiled.d.ts`: ``` $> pbjs -t static-module -w commonjs -o compiled.js file1.proto file2.proto $> pbts -o compiled.d.ts compiled.js ``` Additionally, TypeScript definitions of static modules are compatible with their reflection-based counterparts (i.e. as exported by JSON modules), as long as the following conditions are met: 1. Instead of using `new SomeMessage(...)`, always use `SomeMessage.create(...)` because reflection objects do not provide a constructor. 2. Types, services and enums must start with an uppercase letter to become available as properties of the reflected types as well (i.e. to be able to use `MyMessage.MyEnum` instead of `root.lookup("MyMessage.MyEnum")`). For example, the following generates a JSON module `bundle.js` and a `bundle.d.ts`, but no static code: ``` $> pbjs -t json-module -w commonjs -o bundle.js file1.proto file2.proto $> pbjs -t static-module file1.proto file2.proto | pbts -o bundle.d.ts - ``` ### Reflection vs. static code While using .proto files directly requires the full library respectively pure reflection/JSON the light library, pretty much all code but the relatively short descriptors is shared. Static code, on the other hand, requires just the minimal library, but generates additional source code without any reflection features. This also implies that there is a break-even point where statically generated code becomes larger than descriptor-based code once the amount of code generated exceeds the size of the full respectively light library. There is no significant difference performance-wise as the code generated statically is pretty much the same as generated at runtime and both are largely interchangeable as seen in the previous section. | Source | Library | Advantages | Tradeoffs |--------|---------|------------|----------- | .proto | full | Easily editable

Interoperability with other libraries

No compile step | Some parsing and possibly network overhead | JSON | light | Easily editable

No parsing overhead

Single bundle (no network overhead) | protobuf.js specific

Has a compile step | static | minimal | Works where `eval` access is restricted

Fully documented

Small footprint for small protos | Can be hard to edit

No reflection

Has a compile step ### Command line API Both utilities can be used programmatically by providing command line arguments and a callback to their respective `main` functions: ```js var pbjs = require("protobufjs/cli/pbjs"); // or require("protobufjs/cli").pbjs / .pbts pbjs.main([ "--target", "json-module", "path/to/myproto.proto" ], function(err, output) { if (err) throw err; // do something with output }); ``` Additional documentation ------------------------ #### Protocol Buffers \* [Google's Developer Guide](https://developers.google.com/protocol-buffers/docs/overview) #### protobuf.js \* [API Documentation](https://protobufjs.github.io/protobuf.js) \* [CHANGELOG](https://github.com/dcodeIO/protobuf.js/blob/master/CHANGELOG.md) \* [Frequently asked questions](https://github.com/dcodeIO/protobuf.js/wiki) on our wiki #### Community \* [Questions and answers](http://stackoverflow.com/search?tab=newest&q=protobuf.js) on StackOverflow Performance ----------- The package includes a benchmark that compares protobuf.js performance to native JSON (as far as this is possible) and [Google's JS implementation](https://github.com/google/protobuf/tree/master/js). On an i7-2600K running node 6.9.1 it yields: ``` benchmarking encoding performance ... protobuf.js (reflect) x 541,707 ops/sec ±1.13% (87 runs sampled) protobuf.js (static) x 548,134 ops/sec ±1.38% (89 runs sampled) JSON (string) x 318,076 ops/sec ±0.63% (93 runs sampled) JSON (buffer) x 179,165 ops/sec ±2.26% (91 runs sampled) google-protobuf x 74,406 ops/sec ±0.85% (86 runs sampled) protobuf.js (static) was fastest protobuf.js (reflect) was 0.9% ops/sec slower (factor 1.0) JSON (string) was 41.5% ops/sec slower (factor 1.7) JSON (buffer) was 67.6% ops/sec slower (factor 3.1) google-protobuf was 86.4% ops/sec slower (factor 7.3) benchmarking decoding performance ... protobuf.js (reflect) x 1,383,981 ops/sec ±0.88% (93 runs sampled) protobuf.js (static) x 1,378,925 ops/sec ±0.81% (93 runs sampled) JSON (string) x 302,444 ops/sec ±0.81% (93 runs sampled) JSON (buffer) x 264,882 ops/sec ±0.81% (93 runs sampled) google-protobuf x 179,180 ops/sec ±0.64% (94 runs sampled) protobuf.js (reflect) was fastest protobuf.js (static) was 0.3% ops/sec slower (factor 1.0) JSON (string) was 78.1% ops/sec slower (factor 4.6) JSON (buffer) was 80.8% ops/sec slower (factor 5.2) google-protobuf was 87.0% ops/sec slower (factor 7.7) benchmarking combined performance ... protobuf.js (reflect) x 275,900 ops/sec ±0.78% (90 runs sampled) protobuf.js (static) x 290,096 ops/sec ±0.96% (90 runs sampled) JSON (string) x 129,381 ops/sec ±0.77% (90 runs sampled) JSON (buffer) x 91,051 ops/sec ±0.94% (90 runs sampled) google-protobuf x 42,050 ops/sec ±0.85% (91 runs sampled) protobuf.js (static) was fastest protobuf.js (reflect) was 4.7% ops/sec slower (factor 1.0) JSON (string) was 55.3% ops/sec slower (factor 2.2) JSON (buffer) was 68.6% ops/sec slower (factor 3.2) google-protobuf was 85.5% ops/sec slower (factor 6.9) ``` These results are achieved by \* generating type-specific encoders, decoders, verifiers and converters at runtime \* configuring the reader/writer interface according to the environment \* using node-specific functionality where beneficial and, of course \* avoiding unnecessary operations through splitting up [the toolset](#toolset). You can also run [the benchmark](https://github.com/dcodeIO/protobuf.js/blob/master/bench/index.js) ... ``` $> npm run bench ``` and [the profiler](https://github.com/dcodeIO/protobuf.js/blob/master/bench/prof.js) yourself (the latter requires a recent version of node): ``` $> npm run prof [iterations=10000000] ``` Note that as of this writing, the benchmark suite performs significantly slower on node 7.2.0 compared to 6.9.1 because moths. Compatibility ------------- \* Works in all modern and not-so-modern browsers except IE8. \* Because the internals of this package do not rely on `google/protobuf/descriptor.proto`, options are parsed and presented literally. \* If typed arrays are not supported by the environment, plain arrays will be used instead. \* Support for pre-ES5 environments (except IE8) can be achieved by [using a polyfill](https://github.com/dcodeIO/protobuf.js/blob/master/scripts/polyfill.js). \* Support for [Content Security Policy](https://w3c.github.io/webappsec-csp/)-restricted environments (like Chrome extensions without [unsafe-eval](https://developer.chrome.com/extensions/contentSecurityPolicy#relaxing-eval)) can be achieved by generating and using static code instead. \* If a proper way to work with 64 bit values (uint64, int64 etc.) is required, just install [long.js](https://github.com/dcodeIO/long.js) alongside this library. All 64 bit numbers will then be returned as a `Long` instance instead of a possibly unsafe JavaScript number ([see](https://github.com/dcodeIO/long.js)). \* For descriptor.proto interoperability, see [ext/descriptor](https://github.com/dcodeIO/protobuf.js/tree/master/ext/descriptor) Building -------- To build the library or its components yourself, clone it from GitHub and install the development dependencies: ``` $> git clone https://github.com/dcodeIO/protobuf.js.git $> cd protobuf.js $> npm install ``` Building the respective development and production versions with their respective source maps to `dist/`: ``` $> npm run build ``` Building the documentation to `docs/`: ``` $> npm run docs ``` Building the TypeScript definition to `index.d.ts`: ``` $> npm run types ``` ### Browserify integration By default, protobuf.js integrates into any browserify build-process without requiring any optional modules. Hence: \* If int64 support is required, explicitly require the `long` module somewhere in your project as it will be excluded otherwise. This assumes that a global `require` function is present that protobuf.js can call to obtain the long module. If there is no global `require` function present after bundling, it's also possible to assign the long module programmatically: ```js var Long = ...; protobuf.util.Long = Long; protobuf.configure(); ``` \* If you have any special requirements, there is [the bundler](https://github.com/dcodeIO/protobuf.js/blob/master/scripts/bundle.js) for reference. \*\*License:\*\* [BSD 3-Clause License](https://opensource.org/licenses/BSD-3-Clause)