Sui Full Node gRPC

This content describes an alpha/beta feature or service. These early stage features and services are in active development, so details are likely to change.

This feature or service is currently available in

Sui Full node gRPC API will replace the JSON-RPC on Full nodes, such that JSON-RPC will be deprecated when gRPC API is generally available.

Protobuf definitions of public Sui core types.

This file contains a complete set of protobuf definitions for all of the public sui core types. All sui types are intended to have a 1:1 mapping to a protobuf message defined in this file and be able to roundtrip to/from their rust and protobuf definitions assuming a sufficiently up-to-date version of both these definitions.

For more information on the types these proto messages correspond with, see the documentation for their rust versions defined in the sui-sdk-types library.

These message definitions use protobuf version 3 (proto3). In proto3, fields that are primitives (that is, they are not a message) and are not present on the wire are zero-initialized. To gain the ability to detect field presence , these definitions follow the convention of having all fields marked optional , and wrapping repeated fields in a message as needed.

A new JWK.

Unique identifier for an account on the Sui blockchain.

An Address is a 32-byte pseudonymous identifier used to uniquely identify an account and asset-ownership on the Sui blockchain. Often, human-readable addresses are encoded in hexadecimal with a 0x prefix. For example, this is a valid Sui address: 0x02a212de6a9dfa3a69e22387acfbafbb1a9e591bd9d636e7895dcfc8de05f331 .

Address is denied for this coin type.

An argument to a programmable transaction command.

Expire old JWKs.

Update the set of valid JWKs.

Message that represents a type that is serialized and encoded using the BCS format.

A point on the BN254 elliptic curve.

A transaction that was cancelled.

Set of cancelled transactions.

System transaction used to change the epoch.

Input/output state of an object that was changed during execution.

A commitment made by a checkpoint.

The committed to contents of a checkpoint.

Version 1 of CheckpointContents .

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

A single command in a programmable transaction.

An error with an argument to a command.

Set of objects that were congested, leading to the transaction's cancellation.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

32-byte output of hashing a Sui structure using the Blake2b256 hash function.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

An event.

The status of an executed transaction.

An error that can occur during the execution of a transaction.

Summary of gas charges.

Storage is charged independently of computation. There are three parts to the storage charges:

When looking at a gas cost summary the amount charged to the user is computation_cost + storage_cost - storage_rebate and that is the amount that is deducted from the gas coins. non_refundable_storage_fee is collected from the objects being mutated/deleted and it is tracked by the system in storage funds.

Objects deleted, including the older versions of objects mutated, have the storage field on the objects added up to a pool of "potential rebate". This rebate then is reduced by the "nonrefundable rate" such that: potential_rebate(storage cost of deleted/mutated objects) = storage_rebate + non_refundable_storage_fee

Payment information for executing a transaction.

An object part of the initial chain state.

The genesis transaction.

A signed 128-bit integer encoded in little-endian using 16-bytes.

A Move identifier.

Identifiers are only valid if they conform to the following ABNF:

An input to a user transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Indicates that an object was modified at a specific version.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

Error that occurred in Move.

Location in Move bytecode where an error occurred.s

Module defined by a package.

A Move package.

A Move struct.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

An argument type for a nested result.

An object on the Sui blockchain.

Object data, either a package or struct.

Information about the old version of the object.

Unique identifier for an object on the Sui blockchain.

An ObjectId is a 32-byte identifier used to uniquely identify an object on the Sui blockchain.

Reference to an object.

An object reference with owner information.

Object write, including all of mutated, created, unwrapped.

Enum of different types of ownership for an object.

Package ID does not match PackageId in upgrade ticket.

An error with a upgrading a package.

Package write.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Read-only shared object from the input.

A RoaringBitmap. See RoaringFormatSpec for the specification for the serialized format of RoaringBitmap s.

A shared object input.

A basic signature.

Can either be an ed25519, secp256k1, or secp256r1 signature with corresponding public key.

A size error.

Command to split a single coin object into multiple coins.

Type information for a Move struct.

System package.

A transaction.

Version 1 of Transaction .

The output or effects of executing a transaction.

Version 1 of TransactionEffects .

Version 2 of TransactionEffects .

Events emitted during the successful execution of a transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Type argument error.

Identifies a struct and the module it was defined in.

Type of a Move value.

An unsigned 128-bit integer encoded in little-endian using 16-bytes.

An unsigned 256-bit integer encoded in little-endian using 32-bytes.

A shared object that wasn't changed during execution.

Command to upgrade an already published package.

Upgraded package info for the linkage table.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

Object version assignment from consensus.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

The sui.node.v2 package contains API definitions for services that are expected to run on Fullnodes.

The delta, or change, in balance for an address for a particular Coin type.

Indicates the finality of the executed transaction.

Request message for NodeService.ExecuteTransaction .

Note: You must provide only one of transaction or transaction_bcs .

Response message for NodeService.ExecuteTransaction .

An object used by or produced from a transaction.

A transaction, with all of its inputs and outputs.

Request message for NodeService.GetCheckpoint .

At most, provide one of sequence_number or digest . An error is returned if you attempt to provide both. If you provide neither, the service returns the latest executed checkpoint.

Response message for NodeService.GetCheckpoint .

Request message for NodeService.GetCommittee.

Response message for NodeService.GetCommittee .

Request message for NodeService.GetFullCheckpoint .

At most, provide one of sequence_number or digest . An error is returned if you provide both. If you provide neither, the service returns the latest executed checkpoint.

Response message for NodeService.GetFullCheckpoint .

Request message for NodeService.GetNodeInfo .

Response message for NodeService.GetNodeInfo .

Request message for NodeService.GetObject .

Response message for NodeService.GetObject .

Request message for NodeService.GetTransaction .

Response message for NodeService.GetTransactio n.

An input to a user transaction.

An error with an argument to a command.

An error that can occur during the execution of a transaction.

The status of an executed transaction.

Location in Move bytecode where an error occurred.

An error with upgrading a package.

A size error.

Type argument error.

A commitment made by a checkpoint.

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Input/output state of an object that was changed during execution.

The effects of executing a transaction.

A shared object that wasn't changed during execution.

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/types/types.proto

Protobuf definitions of public Sui core types.

This file contains a complete set of protobuf definitions for all of the public sui core types. All sui types are intended to have a 1:1 mapping to a protobuf message defined in this file and be able to roundtrip to/from their rust and protobuf definitions assuming a sufficiently up-to-date version of both these definitions.

For more information on the types these proto messages correspond with, see the documentation for their rust versions defined in the sui-sdk-types library.

These message definitions use protobuf version 3 (proto3). In proto3, fields that are primitives (that is, they are not a message) and are not present on the wire are zero-initialized. To gain the ability to detect field presence , these definitions follow the convention of having all fields marked optional , and wrapping repeated fields in a message as needed.

A new JWK.

Unique identifier for an account on the Sui blockchain.

An Address is a 32-byte pseudonymous identifier used to uniquely identify an account and asset-ownership on the Sui blockchain. Often, human-readable addresses are encoded in hexadecimal with a 0x prefix. For example, this is a valid Sui address: 0x02a212de6a9dfa3a69e22387acfbafbb1a9e591bd9d636e7895dcfc8de05f331 .

Address is denied for this coin type.

An argument to a programmable transaction command.

Expire old JWKs.

Update the set of valid JWKs.

Message that represents a type that is serialized and encoded using the BCS format.

A point on the BN254 elliptic curve.

A transaction that was cancelled.

Set of cancelled transactions.

System transaction used to change the epoch.

Input/output state of an object that was changed during execution.

A commitment made by a checkpoint.

The committed to contents of a checkpoint.

Version 1 of CheckpointContents .

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

A single command in a programmable transaction.

An error with an argument to a command.

Set of objects that were congested, leading to the transaction's cancellation.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

32-byte output of hashing a Sui structure using the Blake2b256 hash function.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

An event.

The status of an executed transaction.

An error that can occur during the execution of a transaction.

Summary of gas charges.

Storage is charged independently of computation. There are three parts to the storage charges:

When looking at a gas cost summary the amount charged to the user is computation_cost + storage_cost - storage_rebate and that is the amount that is deducted from the gas coins. non_refundable_storage_fee is collected from the objects being mutated/deleted and it is tracked by the system in storage funds.

Objects deleted, including the older versions of objects mutated, have the storage field on the objects added up to a pool of "potential rebate". This rebate then is reduced by the "nonrefundable rate" such that: potential_rebate(storage cost of deleted/mutated objects) = storage_rebate + non_refundable_storage_fee

Payment information for executing a transaction.

An object part of the initial chain state.

The genesis transaction.

A signed 128-bit integer encoded in little-endian using 16-bytes.

A Move identifier.

Identifiers are only valid if they conform to the following ABNF:

An input to a user transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Indicates that an object was modified at a specific version.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

Error that occurred in Move.

Location in Move bytecode where an error occurred.s

Module defined by a package.

A Move package.

A Move struct.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

An argument type for a nested result.

An object on the Sui blockchain.

Object data, either a package or struct.

Information about the old version of the object.

Unique identifier for an object on the Sui blockchain.

An ObjectId is a 32-byte identifier used to uniquely identify an object on the Sui blockchain.

Reference to an object.

An object reference with owner information.

Object write, including all of mutated, created, unwrapped.

Enum of different types of ownership for an object.

Package ID does not match PackageId in upgrade ticket.

An error with a upgrading a package.

Package write.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Read-only shared object from the input.

A RoaringBitmap. See RoaringFormatSpec for the specification for the serialized format of RoaringBitmap s.

A shared object input.

A basic signature.

Can either be an ed25519, secp256k1, or secp256r1 signature with corresponding public key.

A size error.

Command to split a single coin object into multiple coins.

Type information for a Move struct.

System package.

A transaction.

Version 1 of Transaction .

The output or effects of executing a transaction.

Version 1 of TransactionEffects .

Version 2 of TransactionEffects .

Events emitted during the successful execution of a transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Type argument error.

Identifies a struct and the module it was defined in.

Type of a Move value.

An unsigned 128-bit integer encoded in little-endian using 16-bytes.

An unsigned 256-bit integer encoded in little-endian using 32-bytes.

A shared object that wasn't changed during execution.

Command to upgrade an already published package.

Upgraded package info for the linkage table.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

Object version assignment from consensus.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

The sui.node.v2 package contains API definitions for services that are expected to run on Fullnodes.

The delta, or change, in balance for an address for a particular Coin type.

Indicates the finality of the executed transaction.

Request message for NodeService.ExecuteTransaction .

Note: You must provide only one of transaction or transaction_bcs .

Response message for NodeService.ExecuteTransaction .

An object used by or produced from a transaction.

A transaction, with all of its inputs and outputs.

Request message for NodeService.GetCheckpoint .

At most, provide one of sequence_number or digest . An error is returned if you attempt to provide both. If you provide neither, the service returns the latest executed checkpoint.

Response message for NodeService.GetCheckpoint .

Request message for NodeService.GetCommittee.

Response message for NodeService.GetCommittee .

Request message for NodeService.GetFullCheckpoint .

At most, provide one of sequence_number or digest . An error is returned if you provide both. If you provide neither, the service returns the latest executed checkpoint.

Response message for NodeService.GetFullCheckpoint .

Request message for NodeService.GetNodeInfo .

Response message for NodeService.GetNodeInfo .

Request message for NodeService.GetObject .

Response message for NodeService.GetObject .

Request message for NodeService.GetTransaction .

Response message for NodeService.GetTransactio n.

An input to a user transaction.

An error with an argument to a command.

An error that can occur during the execution of a transaction.

The status of an executed transaction.

Location in Move bytecode where an error occurred.

An error with upgrading a package.

A size error.

Type argument error.

A commitment made by a checkpoint.

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Input/output state of an object that was changed during execution.

The effects of executing a transaction.

A shared object that wasn't changed during execution.

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation.

Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the

Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/types/signature_scheme.proto

The sui.node.v2 package contains API definitions for services that are expected to run on Fullnodes.

The delta, or change, in balance for an address for a particular Coin type.

Indicates the finality of the executed transaction.

Request message for NodeService.ExecuteTransaction .

Note: You must provide only one of transaction or transaction_bcs .

Response message for NodeService.ExecuteTransaction .

An object used by or produced from a transaction.

A transaction, with all of its inputs and outputs.

Request message for NodeService.GetCheckpoint .

At most, provide one of sequence_number or digest . An error is returned if you attempt to provide both. If you provide neither, the service returns the latest executed checkpoint.

Response message for NodeService.GetCheckpoint .

Request message for NodeService.GetCommittee.

Response message for NodeService.GetCommittee .

Request message for NodeService.GetFullCheckpoint .

At most, provide one of sequence_number or digest . An error is returned if you provide both. If you provide neither, the service returns the latest executed checkpoint.

Response message for NodeService.GetFullCheckpoint .

Request message for NodeService.GetNodeInfo .

Response message for NodeService.GetNodeInfo .

Request message for NodeService.GetObject .

Response message for NodeService.GetObject .

Request message for NodeService.GetTransaction .

Response message for NodeService.GetTransactio n.

An input to a user transaction.

An error with an argument to a command.

An error that can occur during the execution of a transaction.

The status of an executed transaction.

Location in Move bytecode where an error occurred.

An error with upgrading a package.

A size error.

Type argument error.

A commitment made by a checkpoint.

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Input/output state of an object that was changed during execution.

The effects of executing a transaction.

A shared object that wasn't changed during execution.

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/node/v2/node_service.proto

The sui.node.v2 package contains API definitions for services that are expected to run on Fullnodes.

The delta, or change, in balance for an address for a particular Coin type.

Indicates the finality of the executed transaction.

Request message for NodeService.ExecuteTransaction .

Note: You must provide only one of transaction or transaction_bcs .

Response message for NodeService.ExecuteTransaction .

An object used by or produced from a transaction.

A transaction, with all of its inputs and outputs.

Request message for NodeService.GetCheckpoint .

At most, provide one of sequence_number or digest . An error is returned if you attempt to provide both. If you provide neither, the service returns the latest executed checkpoint.

Response message for NodeService.GetCheckpoint .

Request message for NodeService.GetCommittee.

Response message for NodeService.GetCommittee .

Request message for NodeService.GetFullCheckpoint .

At most, provide one of sequence_number or digest . An error is returned if you provide both. If you provide neither, the service returns the latest executed checkpoint.

Response message for NodeService.GetFullCheckpoint .

Request message for NodeService.GetNodeInfo .

Response message for NodeService.GetNodeInfo .

Request message for NodeService.GetObject .

Response message for NodeService.GetObject .

Request message for NodeService.GetTransaction .

Response message for NodeService.GetTransactio n.

An input to a user transaction.

An error with an argument to a command.

An error that can occur during the execution of a transaction.

The status of an executed transaction.

Location in Move bytecode where an error occurred.

An error with upgrading a package.

A size error.

Type argument error.

A commitment made by a checkpoint.

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Input/output state of an object that was changed during execution.

The effects of executing a transaction.

A shared object that wasn't changed during execution.

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; [https://www.googleapis.com/oauth2/v3/certs>](https://www.googleapis.com/oauth2/v3/certs)). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/input.proto

An input to a user transaction.

An error with an argument to a command.

An error that can occur during the execution of a transaction.

The status of an executed transaction.

Location in Move bytecode where an error occurred.

An error with upgrading a package.

A size error.

Type argument error.

A commitment made by a checkpoint.

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Input/output state of an object that was changed during execution.

The effects of executing a transaction.

A shared object that wasn't changed during execution.

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt;

https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1;

end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/execution_status.proto

An error with an argument to a command.

An error that can occur during the execution of a transaction.

The status of an executed transaction.

Location in Move bytecode where an error occurred.

An error with upgrading a package.

A size error.

Type argument error.

A commitment made by a checkpoint.

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Input/output state of an object that was changed during execution.

The effects of executing a transaction.

A shared object that wasn't changed during execution.

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/checkpoint_summary.proto

A commitment made by a checkpoint.

A header for a checkpoint on the Sui blockchain.

On the Sui network, checkpoints define the history of the blockchain. They are quite similar to the concept of blocks used by other blockchains like Bitcoin or Ethereum. The Sui blockchain, however, forms checkpoints after transaction execution has already happened to provide a certified history of the chain, instead of being formed before execution.

Checkpoints commit to a variety of state, including but not limited to:

CheckpointSummary s themselves don't directly include all of the previous information but they are the top-level type by which all the information is committed to transitively via cryptographic hashes included in the summary. CheckpointSummary s are signed and certified by a quorum of the validator committee in a given epoch to allow verification of the chain's state.

Data, which when included in a CheckpointSummary , signals the end of an Epoch .

Input/output state of an object that was changed during execution.

The effects of executing a transaction.

A shared object that wasn't changed during execution.

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by [gRPC](https://github.com/grpc) . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the [API Design Guide](https://cloud.google.com/apis/design/errors) .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z .

Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/effects.proto

Input/output state of an object that was changed during execution.

The effects of executing a transaction.

A shared object that wasn't changed during execution.

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/transaction_execution_service.proto

Response message for NodeService.ExecuteTransaction .

Indicates the finality of the executed transaction.

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/transaction.proto

A new JWK.

Expire old JWKs.

Update the set of valid JWKs.

A transaction that was canceled.

Set of canceled transactions.

System transaction used to change the epoch.

A single command in a programmable transaction.

Consensus commit prologue system transaction.

This message can represent V1, V2, and V3 prologue types.

Version assignments performed by consensus.

Set of operations run at the end of the epoch to close out the current epoch and start the next one.

Operation run at the end of an epoch.

Payment information for executing a transaction.

The genesis transaction.

A JSON web key.

Struct that contains info for a JWK. A list of them for different kinds can be retrieved from the JWK endpoint (for example, &#lt; https://www.googleapis.com/oauth2/v3/certs>). The JWK is used to verify the JWT token.

Key to uniquely identify a JWK.

Command to build a Move vector out of a set of individual elements.

Command to merge multiple coins of the same type into a single coin.

Command to call a Move function.

Functions that can be called by a MoveCall command are those that have a function signature that is either entry or public (which don't have a reference return type).

A user transaction.

Contains a series of native commands and Move calls where the results of one command can be used in future commands.

Command to publish a new Move package.

Randomness update.

Command to split a single coin object into multiple coins.

System package.

A transaction.

A TTL for a transaction.

Transaction type.

Command to transfer ownership of a set of objects to an address.

Command to upgrade an already published package.

Object version assignment from consensus.

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil {

... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/owner.proto

Enum of different types of ownership for an object.

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/object.proto

Module defined by a package.

An object on the Sui blockchain.

Identifies a struct and the module it was defined in.

Upgraded package info for the linkage table.

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/object_reference.proto

Reference to an object.

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/balance_change.proto

The delta, or change, in balance for an address for a particular Coin type.

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/epoch.proto

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by [gRPC](https://github.com/grpc) . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the [API Design Guide](https://cloud.google.com/apis/design/errors) .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/ledger_service.proto

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/executed_transaction.proto

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/argument.proto

An argument to a programmable transaction command.

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/checkpoint.proto

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by

"Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/gas_cost_summary.proto

Summary of gas charges.

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/bcs.proto

Bcs contains an arbitrary type that is serialized using the BCS format as well as a name that identifies the type of the serialized value.

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/event.proto

An event.

Events emitted during the successful execution of a transaction.

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/checkpoint_contents.proto

The committed to contents of a checkpoint.

Transaction information committed to in a checkpoint.

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/protocol_config.proto

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/signature_scheme.proto

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2beta/signature.proto

A G1 point.

A G2 point.

Aggregated signature from members of a multisig committee.

A multisig committee.

A member in a multisig committee.

Set of valid public keys for multisig committee members.

A signature from a member of a multisig committee.

A passkey authenticator.

See struct.PasskeyAuthenticator for more information on the requirements on the shape of the client_data_json field.

A signature from a user.

An aggregated signature from multiple validators.

The validator set for a particular epoch.

A member of a validator committee.

A zklogin authenticator.

A claim of the iss in a zklogin proof.

A zklogin groth16 proof and the required inputs to perform proof verification.

A zklogin groth16 proof.

Public key equivalent for zklogin authenticators.

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC

APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2alpha/live_data_service.proto

Metadata for a coin type

Information about a coin type's 0x2::coin::TreasuryCap and its total available supply

Request message for NodeService.GetCoinInfo .

Response message for NodeService.GetCoinInfo .

Request message for NodeService.ListDynamicFields

Response message for NodeService.ListDynamicFields

Information about a regulated coin, which indicates that it makes use of the transfer deny list.

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

sui/rpc/v2alpha/subscription_service.proto

Request message for SubscriptionService.SubscribeCheckpoints

Response message for SubscriptionService.SubscribeCheckpoints

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

google/rpc/status.proto

The Status type defines a logical error model that is suitable for different programming environments, including REST APIs and RPC APIs. It is used by gRPC . Each Status message contains three pieces of data: error code, error message, and error details.

You can find out more about this error model and how to work with it in the API Design Guide .

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

google/rpc/error_details.proto

Describes violations in a client request. This error type focuses on the syntactic aspects of the request.

A message type used to describe a single bad request field.

Describes additional debugging info.

Describes the cause of the error with structured details.

Example of an error when contacting the "pubsub.googleapis.com" API when it is not enabled:

This response indicates that the pubsub.googleapis.com API is not enabled.

Example of an error that is returned when attempting to create a Spanner instance in a region that is out of stock:

Provides links to documentation or for performing an out of band action.

For example, if a quota check failed with an error indicating the calling project hasn't enabled the accessed service, this can contain a URL pointing directly to the right place in the developer console to flip the bit.

Describes a URL link.

Provides a localized error message that is safe to return to the user which can be attached to an RPC error.

Describes what preconditions have failed.

For example, if an RPC failed because it required the Terms of Service to be acknowledged, it could list the terms of service violation in the PreconditionFailure message.

A message type used to describe a single precondition failure.

Describes how a quota check failed.

For example if a daily limit was exceeded for the calling project, a service could respond with a QuotaFailure detail containing the project id and the description of the quota limit that was exceeded. If the calling project hasn't enabled the service in the developer console, then a service could respond with the project id and set service_disabled to true.

Also see RetryInfo and Help types for other details about handling a quota failure.

A message type used to describe a single quota violation. For example, a daily quota or a custom quota that was exceeded.

Contains metadata about the request that clients can attach when filing a bug or providing other forms of feedback.

Describes the resource that is being accessed.

Describes when the clients can retry a failed request. Clients could ignore the recommendation here or retry when this information is missing from error responses.

It's always recommended that clients should use exponential backoff when retrying.

Clients should wait until retry_delay amount of time has passed since receiving the error response before retrying. If retrying requests also fail, clients should use an exponential backoff scheme to gradually increase the delay between retries based on retry_delay , until either a maximum number of retries have been reached or a maximum retry delay cap has been reached.

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

google/protobuf/timestamp.proto

A Timestamp represents a point in time independent of any time zone or calendar, represented as seconds and fractions of seconds at nanosecond resolution in UTC Epoch time. It is encoded using the Proleptic Gregorian Calendar which extends the Gregorian calendar backwards to year one. It is encoded assuming all minutes are 60 seconds long, i.e. leap seconds are "smeared" so that no leap second table is needed for interpretation. Range is from 0001-01-01T00:00:00Z to 9999-12-31T23:59:59.999999999Z . Restricting to that range ensures that conversion to and from RFC 3339 date strings is possible. See https://www.ietf.org/rfc/rfc3339.txt .

Example 1: Compute Timestamp from POSIX time() .

Example 2: Compute Timestamp from POSIX gettimeofday() .

Example 3: Compute Timestamp from Win32 GetSystemTimeAsFileTime() .

Example 4: Compute Timestamp from Java System.currentTimeMillis() .

Example 5: Compute Timestamp from current time in Python.

In JSON format, the Timestamp type is encoded as a string in the RFC 3339 format. That is, the format is {year}-{month}-{day}T{hour}:{min}:{sec}[.{frac_sec}]Z where {year} is always expressed using four digits while {month} , {day} , {hour} , {min} , and {sec} are zero-padded to two digits each. The fractional seconds, which can go up to 9 digits (so up to 1 nanosecond resolution), are optional. The "Z" suffix indicates the timezone ("UTC"); the timezone is required, though only UTC (as indicated by "Z") is presently supported.

For example, 2017-01-15T01:30:15.01Z encodes 15.01 seconds past 01:30 UTC on January 15, 2017.

In JavaScript, you can convert a Date object to this format using the standard toISOString() method. In Python, you can convert a standard datetime.datetime object to this format using strftime with the time format spec %Y-%m-%dT%H:%M:%S.%fZ . Likewise, in Java, you can use the Joda Time's ISODateTimeFormat.dateTime() to obtain a formatter capable of generating timestamps in this format.

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

google/protobuf/field_mask.proto

FieldMask represents a set of symbolic field paths, for example:

paths: "f.a" paths: "f.b.d"

Here f represents a field in some root message, a and b fields in the message found in f , and d a field found in the message in f.b .

Field masks are used to specify a subset of fields that should be returned by a get operation or modified by an update operation. Field masks also have a custom JSON encoding (see below).

When used in the context of a projection, a response message or sub-message is filtered by the API to only contain those fields as specified in the mask. For example, if the mask in the previous example is applied to a response message as follows:

f { a : 22 b { d : 1 x : 2 } y : 13 } z: 8

The result will not contain specific values for fields x,y and z (their value will be set to the default, and omitted in proto text output):

f { a : 22 b { d : 1 } }

A repeated field is not allowed except at the last position of a paths string.

If a FieldMask object is not present in a get operation, the operation applies to all fields (as if a FieldMask of all fields had been specified).

Note that a field mask does not necessarily apply to the top-level response message. In case of a REST get operation, the field mask applies directly to the response, but in case of a REST list operation, the mask instead applies to each individual message in the returned resource list. In case of a REST custom method, other definitions may be used. Where the mask applies will be clearly documented together with its declaration in the API. In any case, the effect on the returned resource/resources is required behavior for APIs.

A field mask in update operations specifies which fields of the targeted resource are going to be updated. The API is required to only change the values of the fields as specified in the mask and leave the others untouched. If a resource is passed in to describe the updated values, the API ignores the values of all fields not covered by the mask.

If a repeated field is specified for an update operation, new values will be appended to the existing repeated field in the target resource. Note that a repeated field is only allowed in the last position of a paths string.

If a sub-message is specified in the last position of the field mask for an update operation, then new value will be merged into the existing sub-message in the target resource.

For example, given the target message:

f { b { d: 1 x: 2 } c: [1] }

And an update message:

f { b { d: 10 } c: [2] }

then if the field mask is:

paths: ["f.b", "f.c"]

then the result will be:

f { b { d: 10 x: 2 } c: [1, 2] }

An implementation may provide options to override this default behavior for repeated and message fields.

In order to reset a field's value to the default, the field must be in the mask and set to the default value in the provided resource. Hence, in order to reset all fields of a resource, provide a default instance of the resource and set all fields in the mask, or do not provide a mask as described below.

If a field mask is not present on update, the operation applies to all fields (as if a field mask of all fields has been specified). Note that in the presence of schema evolution, this may mean that fields the client does not know and has therefore not filled into the request will be reset to their default. If this is unwanted behavior, a specific service may require a client to always specify a field mask, producing an error if not.

As with get operations, the location of the resource which describes the updated values in the request message depends on the operation kind. In any case, the effect of the field mask is required to be honored by the API.

The HTTP kind of an update operation which uses a field mask must be set to PATCH instead of PUT in order to satisfy HTTP semantics (PUT must only be used for full updates).

In JSON, a field mask is encoded as a single string where paths are separated by a comma. Fields name in each path are converted to/from lower-camel naming conventions.

As an example, consider the following message declarations:

message Profile { User user = 1; Photo photo = 2; } message User { string display_name = 1; string address = 2; }

In proto a field mask for Profile may look as such:

mask { paths: "user.display_name" paths: "photo" }

In JSON, the same mask is represented as below:

{ mask: "user.displayName,photo" }

Field masks treat fields in oneofs just as regular fields. Consider the following message:

message SampleMessage { oneof test_oneof { string name = 4; SubMessage sub_message = 9; } }

The field mask can be:

mask { paths: "name" }

Or:

mask { paths: "sub_message" }

Note that oneof type names ("test_oneof" in this case) cannot be used in paths.

The implementation of any API method which has a FieldMask type field in the request should verify the included field paths, and return an INVALID_ARGUMENT error if any path is unmappable.

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

google/protobuf/duration.proto

A Duration represents a signed, fixed-length span of time represented as a count of seconds and fractions of seconds at nanosecond resolution. It is independent of any calendar and concepts like "day" or "month". It is related to Timestamp in that the difference between two Timestamp values is a Duration and it can be added or subtracted from a Timestamp. Range is approximately +-10,000 years.

Example 1: Compute Duration from two Timestamps in pseudo code.

Timestamp start = ...; Timestamp end = ...; Duration duration = ...;

duration.seconds = end.seconds - start.seconds; duration.nanos = end.nanos - start.nanos;

if (duration.seconds &#lt; 0 && duration.nanos > 0) { duration.seconds += 1; duration.nanos -= 1000000000; } else if (duration.seconds > 0 && duration.nanos &#lt; 0) { duration.seconds -= 1; duration.nanos += 1000000000; }

Example 2: Compute Timestamp from Timestamp + Duration in pseudo code.

Timestamp start = ...; Duration duration = ...; Timestamp end = ...;

end.seconds = start.seconds + duration.seconds; end.nanos = start.nanos + duration.nanos;

if (end.nanos &#lt; 0) { end.seconds -= 1; end.nanos += 1000000000; } else if (end.nanos >= 1000000000) { end.seconds += 1; end.nanos -= 1000000000; }

Example 3: Compute Duration from datetime.timedelta in Python.

td = datetime.timedelta(days=3, minutes=10) duration = Duration() duration.FromTimedelta(td)

In JSON format, the Duration type is encoded as a string rather than an object, where the string ends in the suffix "s" (indicating seconds) and is preceded by the number of seconds, with nanoseconds expressed as fractional seconds. For example, 3 seconds with 0 nanoseconds should be encoded in JSON format as "3s", while 3 seconds and 1 nanosecond should be expressed in JSON format as "3.000000001s", and 3 seconds and 1 microsecond should be expressed in JSON format as "3.000001s".

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil {

... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

google/protobuf/any.proto

Any contains an arbitrary serialized protocol buffer message along with a URL that describes the type of the serialized message.

Protobuf library provides support to pack/unpack Any values in the form of utility functions or additional generated methods of the Any type.

Example 1: Pack and unpack a message in C++.

Foo foo = ...; Any any; any.PackFrom(foo); ... if (any.UnpackTo(&foo)) { ... }

Example 2: Pack and unpack a message in Java.

Foo foo = ...; Any any = Any.pack(foo); ... if (any.is(Foo.class)) { foo = any.unpack(Foo.class); } // or ... if (any.isSameTypeAs(Foo.getDefaultInstance())) { foo = any.unpack(Foo.getDefaultInstance()); }

Example 3: Pack and unpack a message in Python.

foo = Foo(...) any = Any() any.Pack(foo) ... if any.Is(Foo.DESCRIPTOR): any.Unpack(foo) ...

Example 4: Pack and unpack a message in Go

foo := &pb.Foo{...} any, err := anypb.New(foo) if err != nil { ... } ... foo := &pb.Foo{} if err := any.UnmarshalTo(foo); err != nil { ... }

The pack methods provided by protobuf library will by default use 'type.googleapis.com/full.type.name' as the type URL and the unpack methods only use the fully qualified type name after the last '/' in the type URL, for example "foo.bar.com/x/y.z" will yield type name "y.z".

The JSON representation of an Any value uses the regular representation of the deserialized, embedded message, with an additional field @type which contains the type URL. Example:

package google.profile; message Person { string first_name = 1; string last_name = 2; }

{ "@type": "type.googleapis.com/google.profile.Person", "firstName": &#lt;string>, "lastName": &#lt;string> }

If the embedded message type is well-known and has a custom JSON representation, that representation will be embedded adding a field value which holds the custom JSON in addition to the @type field. Example (for message [google.protobuf.Duration][]):

{ "@type": "type.googleapis.com/google.protobuf.Duration", "value": "1.212s" }

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

google/protobuf/empty.proto

A generic empty message that you can re-use to avoid defining duplicated empty messages in your APIs. A typical example is to use it as the request or the response type of an API method. For instance:

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.

Scalar Value Types

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.

Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.

Uses variable-length encoding.

Uses variable-length encoding.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.

Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.

Always four bytes. More efficient than uint32 if values are often greater than 2^28.

Always eight bytes. More efficient than uint64 if values are often greater than 2^56.

Always four bytes.

Always eight bytes.

A string must always contain UTF-8 encoded or 7-bit ASCII text.

May contain any arbitrary sequence of bytes.