grpc-interceptor

Dan Hipschman

Apr 14, 2023

CONTENTS

CHAPTER

ONE

REFERENCE

1.1 grpc_interceptor

Simplified Python gRPC interceptors.

class grpc_interceptor.**AsyncExceptionToStatusInterceptor**(*status_on_unknown_exception: Optional[grpc.StatusCode] = None*)

An interceptor that catches exceptions and sets the RPC status and details.

This is the async analogy to ExceptionToStatusInterceptor. Please see that class' documentation for more information.

async handle_exception(*ex: Exception*, *request_or_iterator: Any*, *context: grpc.ServicerContext*, *method_name: str*) → None

Override this if extending ExceptionToStatusInterceptor.

This will get called when an exception is raised while handling the RPC.

Parameters

- **ex** – The exception that was raised.
- **request_or_iterator** – The RPC request, as a protobuf message if it is a unary request, or an iterator of protobuf messages if it is a streaming request.
- **context** – The servicer context. You probably want to call context.abort(...)
- **method_name** – The name of the RPC being called.

Raises

- **This method must raise and cannot return, as in general there's no** –
- **meaningful RPC response to return if an exception has occurred. You can** –
- **raise the original exception, ex, or something else.** –

async intercept(*method: Callable*, *request_or_iterator: Any*, *context: grpc.ServicerContext*, *method_name: str*) → Any
Do not call this directly; use the interceptor kwarg on grpc.server().

class grpc_interceptor.**AsyncServerInterceptor**
Base class for asyncio server-side interceptors.

To implement an interceptor, subclass this class and override the intercept method.

abstract async intercept(*method: Callable*, *request: Any*, *context: grpc.aio._base_server.ServicerContext*, *method_name: str*) → Any
Override this method to implement a custom interceptor.

You should call await method(request, context) to invoke the next handler (either the RPC method implementation, or the next interceptor in the list).

Parameters

- **method** – Either the RPC method implementation, or the next interceptor in the chain.
- **request** – The RPC request, as a protobuf message.
- **context** – The ServicerContext pass by gRPC to the service.
- **method_name** – A string of the form "/protobuf.package.Service/Method"

Returns This should generally return the result of await method(request, context), which is typically the RPC method response, as a protobuf message. The interceptor is free to modify this in some way, however.

async intercept_service(*continuation*, *handler_call_details*)
Implementation of grpc.aio.ServerInterceptor.

This is not part of the grpc_interceptor.AsyncServerInterceptor API, but must have a public name. Do not override it, unless you know what you're doing.

class grpc_interceptor.**ClientCallDetails**(*method: str*, *timeout: Optional[float]*, *metadata: Optional[Sequence[Tuple[str, Union[str, bytes]]]]*, *credentials: Optional[grpc.CallCredentials]*, *wait_for_ready: Optional[bool]*, *compression: Any*)
Describes an RPC to be invoked.

See https://grpc.github.io/grpc/python/grpc.html#grpc.ClientCallDetails

class grpc_interceptor.**ClientInterceptor**
Base class for client-side interceptors.

To implement an interceptor, subclass this class and override the intercept method.

abstract intercept(*method: Callable*, *request_or_iterator: Any*, *call_details: grpc.ClientCallDetails*) → grpc_interceptor.client.ClientInterceptorReturnType
Override this method to implement a custom interceptor.

This method is called for all unary and streaming RPCs. The interceptor implementation should call *method* using a *grpc.ClientCallDetails* and the *request_or_iterator* object as parameters. The *request_or_iterator* parameter may be type checked to determine if this is a singluar request for unary RPCs or an iterator for client-streaming or client-server streaming RPCs.

Parameters

- **method** – A function that proceeds with the invocation by executing the next interceptor in the chain or invoking the actual RPC on the underlying channel.

- **request_or_iterator** – RPC request message or iterator of request messages for streaming requests.
- **call_details** – Describes an RPC to be invoked.

Returns

The type of the return should match the type of the return value received by calling *method*. This is an object that is both a Call for the RPC and a Future.

The actual result from the RPC can be got by calling *.result()* on the value returned from *method*.

intercept_stream_stream(*continuation: Callable*, *call_details: grpc.ClientCallDetails*, *request_iterator: Iterator[Any]*)
Implementation of grpc.StreamStreamClientInterceptor.

This is not part of the grpc_interceptor.ClientInterceptor API, but must have a public name. Do not override it, unless you know what you're doing.

intercept_stream_unary(*continuation: Callable*, *call_details: grpc.ClientCallDetails*, *request_iterator: Iterator[Any]*)
Implementation of grpc.StreamUnaryClientInterceptor.

This is not part of the grpc_interceptor.ClientInterceptor API, but must have a public name. Do not override it, unless you know what you're doing.

intercept_unary_stream(*continuation: Callable*, *call_details: grpc.ClientCallDetails*, *request: Any*)
Implementation of grpc.UnaryStreamClientInterceptor.

This is not part of the grpc_interceptor.ClientInterceptor API, but must have a public name. Do not override it, unless you know what you're doing.

intercept_unary_unary(*continuation: Callable*, *call_details: grpc.ClientCallDetails*, *request: Any*)
Implementation of grpc.UnaryUnaryClientInterceptor.

This is not part of the grpc_interceptor.ClientInterceptor API, but must have a public name. Do not override it, unless you know what you're doing.

class grpc_interceptor.**ExceptionToStatusInterceptor**(*status_on_unknown_exception: Optional[grpc.StatusCode] = None*)
An interceptor that catches exceptions and sets the RPC status and details.

ExceptionToStatusInterceptor will catch any subclass of GrpcException and set the status code and details on the gRPC context. You can also extend this and override the handle_exception method to catch other types of exceptions, and handle them in different ways. E.g., you can catch and handle exceptions that don't derive from GrpcException. Or you can set rich error statuses with context.abort_with_status().

Parameters **status_on_unknown_exception** – Specify what to do if an exception which is not a subclass of GrpcException is raised. If None, do nothing (by default, grpc will set the status to UNKNOWN). If not None, then the status code will be set to this value. It must not be OK. The details will be set to the value of repr(e), where e is the exception. In any case, the exception will be propagated.

Raises **ValueError** – If status_code is OK.

handle_exception(*ex: Exception*, *request_or_iterator: Any*, *context: grpc.ServicerContext*, *method_name: str*) → None
Override this if extending ExceptionToStatusInterceptor.

This will get called when an exception is raised while handling the RPC.

Parameters

- **ex** – The exception that was raised.
- **request_or_iterator** – The RPC request, as a protobuf message if it is a unary request, or an iterator of protobuf messages if it is a streaming request.
- **context** – The servicer context. You probably want to call context.abort(...)
- **method_name** – The name of the RPC being called.

Raises

- **This method must raise and cannot return, as in general there's no** –
- **meaningful RPC response to return if an exception has occurred. You can** –
- **raise the original exception, ex, or something else.** –

intercept(*method: Callable*, *request_or_iterator: Any*, *context: grpc.ServicerContext*, *method_name: str*) → Any

Do not call this directly; use the interceptor kwarg on grpc.server().

class grpc_interceptor.**MethodName**(*package: str*, *service: str*, *method: str*)

Represents a gRPC method name.

gRPC methods are defined by three parts, represented by the three attributes.

package

This is defined by the *package foo.bar;* designation in the protocol buffer definition, or it could be defined by the protocol buffer directory structure, depending on the language (see https://developers.google.com/protocol-buffers/docs/proto3#packages).

service

This is the service name in the protocol buffer definition (e.g., *service SearchService { ... }*.

method

This is the method name. (e.g., *rpc Search(...) returns (...);*).

property fully_qualified_service

Return the service name prefixed with the package.

Example

```
>>> MethodName("foo.bar", "SearchService", "Search").fully_qualified_service
'foo.bar.SearchService'
```

class grpc_interceptor.**ServerInterceptor**

Base class for server-side interceptors.

To implement an interceptor, subclass this class and override the intercept method.

abstract intercept(*method: Callable*, *request_or_iterator: Any*, *context: grpc.ServicerContext*, *method_name: str*) → Any

Override this method to implement a custom interceptor.

You should call method(request, context) to invoke the next handler (either the RPC method implementation, or the next interceptor in the list).

Parameters

- **method** – Either the RPC method implementation, or the next interceptor in the chain.
- **request_or_iterator** – The RPC request, as a protobuf message if it is a unary request, or an iterator of protobuf messages if it is a streaming request.
- **context** – The ServicerContext pass by gRPC to the service.
- **method_name** – A string of the form "/protobuf.package.Service/Method"

Returns This should generally return the result of method(request, context), which is typically the RPC method response, as a protobuf message, or an iterator of protobuf messages for streaming responses. The interceptor is free to modify this in some way, however.

intercept_service(*continuation*, *handler_call_details*)
Implementation of grpc.ServerInterceptor.

This is not part of the grpc_interceptor.ServerInterceptor API, but must have a public name. Do not override it, unless you know what you're doing.

grpc_interceptor.**parse_method_name**(*method_name: str*) → grpc_interceptor.server.MethodName
Parse a method name into package, service and endpoint components.

Parameters **method_name** – A string of the form "/foo.bar.SearchService/Search", as passed to ServerInterceptor.intercept().

Returns A MethodName object.

Example

```
>>> parse_method_name("/foo.bar.SearchService/Search")
MethodName(package='foo.bar', service='SearchService', method='Search')
```

1.2 grpc_interceptor.exceptions

Exceptions for ExceptionToStatusInterceptor.

See https://grpc.github.io/grpc/core/md_doc_statuscodes.html for the source of truth on status code meanings.

exception grpc_interceptor.exceptions.**Aborted**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)
The operation was aborted.

Typically this is due to a concurrency issue such as a sequencer check failure or transaction abort. See the guidelines on other exceptions for deciding between FAILED_PRECONDITION, ABORTED, and UNAVAILABLE.

exception grpc_interceptor.exceptions.**AlreadyExists**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)
The entity that a client attempted to create already exists.

E.g., a file or directory that a client is trying to create already exists.

exception grpc_interceptor.exceptions.**Cancelled**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)
The operation was cancelled, typically by the caller.

exception grpc_interceptor.exceptions.**DataLoss**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

Unrecoverable data loss or corruption.

exception grpc_interceptor.exceptions.**DeadlineExceeded**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

The deadline expired before the operation could complete.

For operations that change the state of the system, this error may be returned even if the operation has completed successfully. For example, a successful response from a server could have been delayed long.

exception grpc_interceptor.exceptions.**FailedPrecondition**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

The operation failed because the system is in an invalid state for execution.

For example, the directory to be deleted is non-empty, an rmdir operation is applied to a non-directory, etc. Service implementors can use the following guidelines to decide between FAILED_PRECONDITION, ABORTED, and UNAVAILABLE: (a) Use UNAVAILABLE if the client can retry just the failing call. (b) Use ABORTED if the client should retry at a higher level (e.g., when a client-specified test-and-set fails, indicating the client should restart a read-modify-write sequence). (c) Use FAILED_PRECONDITION if the client should not retry until the system state has been explicitly fixed. E.g., if an "rmdir" fails because the directory is non-empty, FAILED_PRECONDITION should be returned since the client should not retry unless the files are deleted from the directory.

exception grpc_interceptor.exceptions.**GrpcException**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

Base class for gRPC exceptions.

Generally you would not use this class directly, but rather use a subclass representing one of the standard gRPC status codes (see: https://grpc.github.io/grpc/core/md_doc_statuscodes.html for the official list).

status_code

A grpc.StatusCode other than OK. The only use case for this is if gRPC adds a new status code that isn't represented by one of the subclasses of GrpcException. Must not be OK, because gRPC will not raise an RpcError to the client if the status code is OK.

details

A string with additional informantion about the error.

Parameters

- **details** – If not None, specifies a custom error message.
- **status_code** – If not None, sets the status code.

Raises **ValueError** – If status_code is OK.

property status_string

Return status_code as a string.

Returns The status code as a string.

Example

```
>>> GrpcException(status_code=StatusCode.NOT_FOUND).status_string
'NOT_FOUND'
```

exception grpc_interceptor.exceptions.**Internal**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

Internal errors.

This means that some invariants expected by the underlying system have been broken. This error code is reserved for serious errors.

exception grpc_interceptor.exceptions.**InvalidArgument**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

The client specified an invalid argument.

Note that this differs from FAILED_PRECONDITION. INVALID_ARGUMENT indicates arguments that are problematic regardless of the state of the system (e.g., a malformed file name).

exception grpc_interceptor.exceptions.**NotFound**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

Some requested entity (e.g., file or directory) was not found.

Note to server developers: if a request is denied for an entire class of users, such as gradual feature rollout or undocumented whitelist, NOT_FOUND may be used. If a request is denied for some users within a class of users, such as user-based access control, PERMISSION_DENIED must be used.

exception grpc_interceptor.exceptions.**OutOfRange**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

The operation was attempted past the valid range.

E.g., seeking or reading past end-of-file. Unlike INVALID_ARGUMENT, this error indicates a problem that may be fixed if the system state changes. For example, a 32-bit file system will generate INVALID_ARGUMENT if asked to read at an offset that is not in the range [0,2^32-1], but it will generate OUT_OF_RANGE if asked to read from an offset past the current file size. There is a fair bit of overlap between FAILED_PRECONDITION and OUT_OF_RANGE. We recommend using OUT_OF_RANGE (the more specific error) when it applies so that callers who are iterating through a space can easily look for an OUT_OF_RANGE error to detect when they are done.

exception grpc_interceptor.exceptions.**PermissionDenied**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

The caller does not have permission to execute the specified operation.

PERMISSION_DENIED must not be used for rejections caused by exhausting some resource (use RESOURCE_EXHAUSTED instead for those errors). PERMISSION_DENIED must not be used if the caller can not be identified (use UNAUTHENTICATED instead for those errors). This error code does not imply the request is valid or the requested entity exists or satisfies other pre-conditions.

exception grpc_interceptor.exceptions.**ResourceExhausted**(*details: Optional[str] = None*, *status_code: Optional[grpc.StatusCode] = None*)

Some resource has been exhausted.

Perhaps a per-user quota, or perhaps the entire file system is out of space.

exception grpc_interceptor.exceptions.**Unauthenticated**(*details: Optional[str] = None, status_code: Optional[grpc.StatusCode] = None*)

The request does not have valid authentication credentials for the operation.

exception grpc_interceptor.exceptions.**Unavailable**(*details: Optional[str] = None, status_code: Optional[grpc.StatusCode] = None*)

The service is currently unavailable.

This is most likely a transient condition, which can be corrected by retrying with a backoff. Note that it is not always safe to retry non-idempotent operations.

exception grpc_interceptor.exceptions.**Unimplemented**(*details: Optional[str] = None, status_code: Optional[grpc.StatusCode] = None*)

The operation is not implemented or is not supported/enabled in this service.

exception grpc_interceptor.exceptions.**Unknown**(*details: Optional[str] = None, status_code: Optional[grpc.StatusCode] = None*)

Unknown error.

For example, this error may be returned when a Status value received from another address space belongs to an error space that is not known in this address space. Also errors raised by APIs that do not return enough error information may be converted to this error.

1.3 grpc_interceptor.testing

A framework for testing interceptors.

class grpc_interceptor.testing.**DummyRequest**

class grpc_interceptor.testing.**DummyResponse**

class grpc_interceptor.testing.**DummyService**(*special_cases: Dict[str, Callable[[str, grpc.ServicerContext], str]]*)

A gRPC service used for testing.

Parameters **special_cases** – A dictionary where the keys are strings, and the values are functions that take and return strings. The functions can also raise exceptions. When the Execute method is given a string in the dict, it will call the function with that string instead, and return the result. This allows testing special cases, like raising exceptions.

Execute(*request: dummy_pb2.DummyRequest, context: grpc.ServicerContext*) → dummy_pb2.DummyResponse

Echo the input, or take on of the special cases actions.

ExecuteClientServerStream(*request_iter: Iterable[dummy_pb2.DummyRequest], context: grpc.ServicerContext*) → Iterable[dummy_pb2.DummyResponse]

Stream input to output.

ExecuteClientStream(*request_iter: Iterable[dummy_pb2.DummyRequest], context: grpc.ServicerContext*) → dummy_pb2.DummyResponse

Iterate over the input and concatenates the strings into the output.

ExecuteServerStream(*request: dummy_pb2.DummyRequest*, *context: grpc.ServicerContext*) → Iterable[dummy_pb2.DummyResponse]

Stream one character at a time from the input.

grpc_interceptor.testing.**dummy_client**(*special_cases: Dict[str, Callable[[str, grpc.ServicerContext], str]]*, *interceptors: Optional[List[grpc_interceptor.server.ServerInterceptor]] = None*, *client_interceptors: Optional[List[grpc_interceptor.client.ClientInterceptor]] = None*, *aio_server: bool = False*, *aio_client: bool = False*, *aio_read_write: bool = False*)

A context manager that returns a gRPC client connected to a DummyService.

grpc_interceptor.testing.**raises**(*e: Exception*) → Callable

Return a function that raises the given exception when called.

Parameters **e** – The exception to be raised.

Returns A function that can take any arguments, and raises the given exception.

CHAPTER

TWO

LICENSE

MIT License

Copyright (c) 2020 Dan Hipschman

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

The primary aim of this project is to make Python gRPC interceptors simple. The Python `grpc` package provides service interceptors, but they're a bit hard to use because of their flexibility. The `grpc` interceptors don't have direct access to the request and response objects, or the service context. Access to these are often desired, to be able to log data in the request or response, or set status codes on the context.

The secondary aim of this project is to keep the code small and simple. Code you can read through and understand quickly gives you confidence and helps debug issues. When you install this package, you also don't want a bunch of other packages that might cause conflicts within your project. Too many dependencies slow down installation as well

as runtime (fresh imports take time). Hence, a goal of this project is to keep dependencies to a minimum. The only core dependency is the `grpc` package, and the `testing` extra includes `protobuf` as well.

The `grpc_interceptor` package provides the following:

- A `ServerInterceptor` base class, to make it easy to define your own server-side interceptors. Do not confuse this with the `grpc.ServerInterceptor` class.
- An `AsyncServerInterceptor` base class, which is the analogy for async server-side interceptors.
- An `ExceptionToStatusInterceptor` interceptor, so your service can raise exceptions that set the gRPC status code correctly (rather than the default of every exception resulting in an `UNKNOWN` status code). This is something for which pretty much any service will have a use.
- An `AsyncExceptionToStatusInterceptor` interceptor, which is the analogy for async `ExceptionToStatusInterceptor`.
- A `ClientInterceptor` base class, to make it easy to define your own client-side interceptors. Do not confuse this with the `grpc.ClientInterceptor` class. (Note, there is currently no async analogy to `ClientInterceptor`, though contributions are welcome.)
- An optional testing framework. If you're writing your own interceptors, this is useful. If you're just using `ExceptionToStatusInterceptor` then you don't need this.

CHAPTER

THREE

INSTALLATION

To install just the interceptors:

```
$ pip install grpc-interceptor
```

To also install the testing framework:

```
$ pip install grpc-interceptor[testing]
```

CHAPTER

FOUR

USAGE

4.1 Server Interceptors

To define your own server interceptor (we can use a simplified version of `ExceptionToStatusInterceptor` as an example):

```
from grpc_interceptor import ServerInterceptor
from grpc_interceptor.exceptions import GrpcException

class ExceptionToStatusInterceptor(ServerInterceptor):

    def intercept(
        self,
        method: Callable,
        request: Any,
        context: grpc.ServicerContext,
        method_name: str,
    ) -> Any:
        """Override this method to implement a custom interceptor.

        You should call method(request, context) to invoke the
        next handler (either the RPC method implementation, or the
        next interceptor in the list).

        Args:
            method: The next interceptor, or method implementation.
            request: The RPC request, as a protobuf message.
            context: The ServicerContext pass by gRPC to the service.
            method_name: A string of the form
                "/protobuf.package.Service/Method"

        Returns:
            This should generally return the result of
            method(request, context), which is typically the RPC
            method response, as a protobuf message. The interceptor
            is free to modify this in some way, however.
        """
        try:
            return method(request, context)
        except GrpcException as e:
            context.set_code(e.status_code)
            context.set_details(e.details)
            raise
```

Then inject your interceptor when you create the `grpc` server:

```
interceptors = [ExceptionToStatusInterceptor()]
server = grpc.server(
    futures.ThreadPoolExecutor(max_workers=10),
    interceptors=interceptors
)
```

To use `ExceptionToStatusInterceptor`:

```
from grpc_interceptor.exceptions import NotFound

class MyService(my_pb2_grpc.MyServiceServicer):
    def MyRpcMethod(
        self, request: MyRequest, context: grpc.ServicerContext
    ) -> MyResponse:
        thing = lookup_thing()
        if not thing:
            raise NotFound("Sorry, your thing is missing")
        ...
```

This results in the gRPC status status code being set to `NOT_FOUND`, and the details `"Sorry, your thing is missing"`. This saves you the hassle of catching exceptions in your service handler, or passing the context down into helper functions so they can call `context.abort` or `context.set_code`. It allows the more Pythonic approach of just raising an exception from anywhere in the code, and having it be handled automatically.

4.1.1 Server Streaming Interceptors

The above example shows how to write an interceptor for a unary-unary RPC. Server streaming RPCs need to be handled a little differently because `method(request, context)` will return a generator. Hence, the code won't actually run until you iterate over it. Hence, if we were to continue the example of catching exceptions from RPCs, we would need to do something like this:

```
class ExceptionToStatusInterceptor(ServerInterceptor):

    def intercept(
        self,
        method: Callable,
        request: Any,
        context: grpc.ServicerContext,
        method_name: str,
    ) -> Any:
        try:
            for response in method(request, context):
                yield response
        except GrpcException as e:
            context.set_code(e.status_code)
            context.set_details(e.details)
            raise
```

However, this will *only* work for server streaming RPCs. In order to work with both unary and streaming RPCs, you'll need to handle the unary case and streaming case separately, like this:

```
class ExceptionToStatusInterceptor(ServerInterceptor):

    def intercept(self, method, request, context, method_name):
```

(continues on next page)

(continued from previous page)

```
        # Call the RPC. It could be either unary or streaming
        try:
            response_or_iterator = method(request, context)
        except GrpcException as e:
            # If it was unary, then any exception raised would be caught
            # immediately, so handle it here.
            context.set_code(e.status_code)
            context.set_details(e.details)
            raise
        # Check if it's streaming
        if hasattr(response_or_iterator, "__iter__"):
            # Now we know it's a server streaming RPC, so the actual RPC method
            # hasn't run yet. Delegate to a helper to iterate over it so it runs.
            # The helper needs to re-yield the responses, and we need to return
            # the generator that produces.
            return self._intercept_streaming(response_or_iterator)
        else:
            # For unary cases, we are done, so just return the response.
            return response_or_iterator

    def _intercept_streaming(self, iterator):
        try:
            for resp in iterator:
                yield resp
        except GrpcException as e:
            context.set_code(e.status_code)
            context.set_details(e.details)
            raise
```

4.1.2 Async Server Interceptors

Async interceptors are similar to sync ones, but there are two things of which you need to be aware.

First, async server streaming RPCs that are implemented with `async def + yield` cannot be awaited. When you call such a method, you get back an `async_generator`. This is not `await`-able (though you can `async for` loop over it). This is contrary to a unary RPC is implemented with `async def + return`. That results in a coroutine when called, which you *can* `await`.

All this is to say that you mustn't await `method(request, context)` in an async interceptor immediately. First, check if it's an `async_generator`. You can do this by checking for the presence of the `__aiter__` attribute.

Here's an async version of our running `ExceptionToStatusInterceptor` example:

```
from grpc_interceptor.exceptions import GrpcException
from grpc_interceptor.server import AsyncServerInterceptor

class AsyncExceptionToStatusInterceptor(AsyncServerInterceptor):

    async def intercept(
        self,
        method: Callable,
        request_or_iterator: Any,
        context: grpc.ServicerContext,
        method_name: str,
    ) -> Any:
        try:
```

(continues on next page)

(continued from previous page)

```
            response_or_iterator = method(request_or_iterator, context)
            if not hasattr(response_or_iterator, "__aiter__"):
                # Unary, just await and return the response
                return await response_or_iterator
        except GrpcException as e:
            await context.set_code(e.status_code)
            await context.set_details(e.details)
            raise

        # Server streaming responses, delegate to an async generator helper.
        # Note that we do NOT await this.
        return self._intercept_streaming(response_or_iterator, context)

    async def _intercept_streaming(self, iterator, context):
        try:
            async for r in iterator:
                yield r
        except GrpcException as e:
            await context.set_code(e.status_code)
            await context.set_details(e.details)
            raise
```

The second thing you must be aware of with async RPCs, is that an alternate streaming API was added. With this API, instead of writing a server streaming RPC with `async def + yield`, you write it as `async def + return`, but it returns `None`. The way it streams responses is by calling `await context.write(...)` for each response it streams. Similarly, client streaming can be achieved by calling `await context.read()` instead of iterating over the request object.

If you must support RPC services written using this new API, then you must be aware that a server streaming RPC could return `None`. In that case it will not be an `async_generator` even though it's streaming. You will also need your own solution to get access to the streaming response objects. For example, you could wrap the `context` object that you pass to `method(request, context)`, so that you can capture `read` and `write` calls.

4.2 Client Interceptors

We will use an invocation metadata injecting interceptor as an example of defining a client interceptor:

```
from grpc_interceptor import ClientCallDetails, ClientInterceptor

class MetadataClientInterceptor(ClientInterceptor):

    def intercept(
        self,
        method: Callable,
        request_or_iterator: Any,
        call_details: grpc.ClientCallDetails,
    ):
        """Override this method to implement a custom interceptor.

        This method is called for all unary and streaming RPCs. The interceptor
        implementation should call `method` using a `grpc.ClientCallDetails` and the
        `request_or_iterator` object as parameters. The `request_or_iterator`
        parameter may be type checked to determine if this is a singluar request
        for unary RPCs or an iterator for client-streaming or client-server streaming
```

(continues on next page)

(continued from previous page)

```
        RPCs.

        Args:
            method: A function that proceeds with the invocation by executing the next
                interceptor in the chain or invoking the actual RPC on the underlying
                channel.
            request_or_iterator: RPC request message or iterator of request messages
                for streaming requests.
            call_details: Describes an RPC to be invoked.

        Returns:
            The type of the return should match the type of the return value received
            by calling `method`. This is an object that is both a
            `Call <https://grpc.github.io/grpc/python/grpc.html#grpc.Call>`_ for the
            RPC and a `Future <https://grpc.github.io/grpc/python/grpc.html#grpc.
↪Future>`_.

            The actual result from the RPC can be got by calling `.result()` on the
            value returned from `method`.
        """
        new_details = ClientCallDetails(
            call_details.method,
            call_details.timeout,
            [("authorization", "Bearer mysecrettoken")],
            call_details.credentials,
            call_details.wait_for_ready,
            call_details.compression,
        )

        return method(request_or_iterator, new_details)
```

Now inject your interceptor when you create the `grpc` channel:

```
interceptors = [MetadataClientInterceptor()]
with grpc.insecure_channel("grpc-server:50051") as channel:
    channel = grpc.intercept_channel(channel, *interceptors)
    ...
```

Client interceptors can also be used to retry RPCs that fail due to specific errors, or a host of other use cases. There are some basic approaches in the tests to get you started.

Note: The `method` in a client interceptor is a `continuation` as described in the client interceptor section of the gRPC docs. When you invoke the continuation, you get a future back, which resolves to either the result, or exception. This is different than invoking a client stub, which returns the result directly. If the interceptor needs the value returned by the call, or to catch exceptions, then you'll need to do `future = method(request_or_iterator, call_details)`, followed by `future.result()`. Check out the tests for examples.

CHAPTER

FIVE

TESTING

The testing framework provides an actual gRPC service and client, which you can inject interceptors into. This allows end-to-end testing, rather than mocking things out (such as the context). This can catch interactions between your interceptors and the gRPC framework, and also allows chaining interceptors.

The crux of the testing framework is the `dummy_client` context manager. It provides a client to a gRPC service, which by defaults echos the `input` field of the request to the `output` field of the response.

You can also provide a `special_cases` dict which tells the service to call arbitrary functions when the input matches a key in the dict. This allows you to test things like exceptions being thrown.

Here's an example (again using `ExceptionToStatusInterceptor`):

```
from grpc_interceptor import ExceptionToStatusInterceptor
from grpc_interceptor.exceptions import NotFound
from grpc_interceptor.testing import dummy_client, DummyRequest, raises

def test_exception():
    special_cases = {"error": raises(NotFound())}
    interceptors = [ExceptionToStatusInterceptor()]
    with dummy_client(special_cases=special_cases, interceptors=interceptors) as␣
↪client:
        # Test a happy path first
        assert client.Execute(DummyRequest(input="foo")).output == "foo"
        # And now a special case
        with pytest.raises(grpc.RpcError) as e:
            client.Execute(DummyRequest(input="error"))
        assert e.value.code() == grpc.StatusCode.NOT_FOUND
```

CHAPTER

SIX

LIMITATIONS

Known limitations:

- Async client interceptors are not implemented.
- The `read` / `write` API for async streaming technically works, but you'll need to roll your own solution to get access to streaming request and response objects.

Contributions or requests are welcome for any limitations you may find.

PYTHON MODULE INDEX

g

INDEX

M

N

O

P

R

S

U