How to write tests for long-running operations

The Google Cloud client libraries for Rust have helpers that simplify interaction with long-running operations (henceforth, LROs).

Simulating the behavior of LROs in tests involves understanding the details these helpers hide. This guide shows how to do that.

Prerequisites

This guide assumes you are familiar with the previous chapters:

- Working with long-running operations
- How to write tests using a client

Tests for automatic polling

Let's say our application code awaits `lro::Poller::until_done()`. In previous sections, we called this "automatic polling".

```rust
// An example application function that automatically polls.
//
// It starts an LRO, awaits the result, and processes it.
pub async fn my_automatic_poller(
    client: &speech::client::Speech,
    project_id: &str,
) -> Result<Option<wkt::Duration>> {
    use speech::Poller;
    client
        .batch_recognize(format!(
            "projects/{project_id}/locations/global/recognizers/_"
        ))
        .poller()
        .until_done()
        .await
        .map(|r| r.total_billed_duration)
}
```

Note that our application only cares about the final result of the LRO. We do not need to test how it handles intermediate results from polling the LRO. Our tests can simply return the final result of the LRO from the mock.

Creating the `longrunning::model::Operation`

Let's say we want our call to result in the following response.

```rust
fn expected_duration() -> Option<wkt::Duration> {
    Some(wkt::Duration::clamp(100, 0))
}

fn expected_response() -> BatchRecognizeResponse {
    BatchRecognizeResponse::new().set_total_billed_duration(expected_duration())
}
```

You may have noticed that the stub returns a `longrunning::model::Operation`, not a `BatchRecognizeResponse`. We need to pack our desired response into the `Operation::result`.

```
    fn make_finished_operation(response: &BatchRecognizeResponse) -> Result<Operation>
{
        let any = wkt::Any::try_from(response).map_err(Error::serde)?;
        let operation = Operation::new()
            .set_done(true)
            .set_result(OperationResult::Response(any.into()));
        Ok(operation)
    }
```

Note also that we set the `done` field to `true`. This indicates to the `Poller` that the operation has completed, thus ending the polling loop.

```
            .set_done(true)
```

Test code

Now we are ready to write our test.

First we define our mock class, which implements the `speech::stub::Speech` trait.

```
    mockall::mock! {
        #[derive(Debug)]
        Speech {}
        impl speech::stub::Speech for Speech {
            async fn batch_recognize(&self, req: BatchRecognizeRequest, _options:
gax::options::RequestOptions) -> Result<Operation>;
        }
    }
```

Now in our test we create our mock, and set expectations on it.

```
        let mut mock = MockSpeech::new();
        mock.expect_batch_recognize()
            .return_once(|_, _| make_finished_operation(&expected_response()));
```

Finally, we create a client from the mock, call our function, and verify the response.

```
        // Create a client, implemented by our mock.
        let client = speech::client::Speech::from_stub(mock);

        // Call our function which automatically polls.
        let billed_duration = my_automatic_poller(&client, "my-project").await?;

        // Verify the final result of the LRO.
        assert_eq!(billed_duration, expected_duration());
```

Tests for manual polling with intermediate metadata

Let's say our application code manually polls, and does some processing on partial updates.

```rust
pub struct BatchRecognizeResult {
    pub progress_updates: Vec<i32>,
    pub billed_duration: Result<Option<wkt::Duration>>,
}

// An example application function that manually polls.
//
// It starts an LRO. It consolidates the polling results, whether full or
// partial.
//
// In this case, it is the `BatchRecognize` RPC. If we get a partial update,
// we extract the `progress_percent` field. If we get a final result, we
// extract the `total_billed_duration` field.
pub async fn my_manual_poller(
    client: &speech::client::Speech,
    project_id: &str,
) -> BatchRecognizeResult {
    use speech::Poller;
    let mut progress_updates = Vec::new();
    let mut poller = client
        .batch_recognize(format!(
            "projects/{project_id}/locations/global/recognizers/_"
        ))
        .poller();
    while let Some(p) = poller.poll().await {
        match p {
            speech::PollingResult::Completed(r) => {
                let billed_duration = r.map(|r| r.total_billed_duration);
                return BatchRecognizeResult {
                    progress_updates,
                    billed_duration,
                };
            }
            speech::PollingResult::InProgress(m) => {
                if let Some(metadata) = m {
                    // This is a silly application. Your application likely
                    // performs some task immediately with the partial
                    // update, instead of storing it for after the operation
                    // has completed.
                    progress_updates.push(metadata.progress_percent);
```

```
                    }
                }
                speech::PollingResult::PollingError(e) => {
                    return BatchRecognizeResult {
                        progress_updates,
                        billed_duration: Err(Error::from(e)),
                    };
                }
            }
            tokio::time::sleep(std::time::Duration::from_millis(500)).await;
        }

        // We can only get here if `poll()` returns `None`, but it only returns
        // `None` after it returned `PollingResult::Completed`. Therefore this
        // is never reached.
        unreachable!("loop should exit via the `Completed` branch.");
    }
```

We want to simulate how our application acts when it receives intermediate metadata. We can achieve this by returning in-progress operations from our mock.

Creating the `longrunning::model::Operation`

The `BatchRecognize` RPC returns partial results in the form of a `speech::model::OperationMetadata`. Like before, we will need to pack this into the returned `longrunning::model::Operation`, but this time into the `Operation::metadata` field.

```
    fn make_partial_operation(progress: i32) -> Result<Operation> {
        let metadata = OperationMetadata::new().set_progress_percent(progress);
        let any = wkt::Any::try_from(&metadata).map_err(Error::serde)?;
        let operation = Operation::new().set_metadata(Some(any));
        Ok(operation)
    }
```

Test code

First we define our mock class, which implements the `speech::stub::Speech` trait. Note that we override `get_operation()`. We will see why shortly.

```
    mockall::mock! {
        #[derive(Debug)]
        Speech {}
        impl speech::stub::Speech for Speech {
            async fn batch_recognize(&self, req: BatchRecognizeRequest, _options:
gax::options::RequestOptions) -> Result<Operation>;
            async fn get_operation(&self, req: GetOperationRequest, _options:
gax::options::RequestOptions) -> Result<Operation>;
        }
    }
```

Now in our test we create our mock, and set expectations on it.

```
        let mut seq = mockall::Sequence::new();
        let mut mock = MockSpeech::new();
        mock.expect_batch_recognize()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| make_partial_operation(25));
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| make_partial_operation(50));
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| make_partial_operation(75));
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| make_finished_operation(&expected_response()));
```

These expectations will return partial results (25%, 50%, 75%), then return our desired final outcome.

Now a few things you probably noticed.

1. The first expectation is set on `batch_recognize()`, whereas all subsequent expectations are set on `get_operation()`.

   The initial `BatchRecognize` RPC starts the LRO on the server-side. The server returns some identifier for the LRO. This is the `name` field which is omitted from the test code, for simplicity.

   From then on, the client library just polls the status of that LRO. It does this using the `GetOperation` RPC.

   That is why we set expectations on different RPCs for the initial response vs. all subsequent responses.

2. Expectations are set in a sequence.

   This allows `mockall` to verify the order of the calls. It is also necessary to determine which `expect_get_operation` is matched.

Finally, we create a client from the mock, call our function, and verify the response.

```
        // Create a client, implemented by our mock.
        let client = speech::client::Speech::from_stub(mock);

        // Call our function which manually polls.
        let result = my_manual_poller(&client, "my-project").await;

        // Verify the partial metadata updates, and the final result.
        assert_eq!(result.progress_updates, [25, 50, 75]);
        assert_eq!(result.billed_duration?, expected_duration());
```

Simulating errors

Errors can arise in an LRO from a few places.

If your application uses automatic polling, the following cases are all equivalent: `until_done()` returns the error in the `Result`, regardless of where it originated. Simulating an error starting an LRO will yield the simplest test.

» Simulating an error starting an LRO

The simplest way to simulate an error is to have the initial request fail with an error.

```
        mock.expect_batch_recognize()
            .return_once(|_, _| Err(Error::other("failed to start operation")));
```

For manual polling, an error starting an LRO is returned via the completed branch. This ends the polling loop.

```
                speech::PollingResult::Completed(r) => {
```

Simulating an LRO resulting in an error

If you need to simulate an LRO resulting in an error, after intermediate metadata is returned, we need to return the error in the final `longrunning::model::Operation`.

```
fn make_failed_operation(status: rpc::model::Status) -> Result<Operation> {
    let operation = Operation::new()
        .set_done(true)
        .set_result(OperationResult::Error(status.into()));
    Ok(operation)
}
```

We set our expectations to return the `Operation` from `get_operation` as before.

```
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| {
                // This is a common error for `Create*` RPCs, which are often
                // LROs. It is less applicable to `BatchRecognize` in practice.
                let status = rpc::model::Status::default()
                    .set_code(gax::error::rpc::Code::AlreadyExists as i32)
                    .set_message("resource already exists");
                make_failed_operation(status)
            });
```

An LRO ending in an error will be returned via the completed branch. This ends the polling loop.

```
                speech::PollingResult::Completed(r) => {
```

Simulating a polling error

Polling loops can also exit because the polling policy has been exhausted. When this happens, the client library can not say definitively whether the LRO has completed or not.

If your application has custom logic to deal with this case, we can exercise it by returning an error from the `get_operation` expectation.

```
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| Err(Error::other("could not poll operation")));
```

An LRO ending with a polling error will be returned via the polling error branch.

```
                speech::PollingResult::PollingError(e) => {
```

Note that the stubbed out client does not have an associated polling policy. The polling loop will terminate on the first error, even if the error is typically considered transient.

Automatic polling - Full test

```rust
use gax::Result;
use gax::error::Error;
use google_cloud_gax as gax;
use google_cloud_longrunning as longrunning;
use google_cloud_speech_v2 as speech;
use google_cloud_wkt as wkt;
use longrunning::model::Operation;
use longrunning::model::operation::Result as OperationResult;
use speech::model::{BatchRecognizeRequest, BatchRecognizeResponse};

// Example application code that is under test
mod my_application {
    use super::*;

    // An example application function that automatically polls.
    //
    // It starts an LRO, awaits the result, and processes it.
    pub async fn my_automatic_poller(
        client: &speech::client::Speech,
        project_id: &str,
    ) -> Result<Option<wkt::Duration>> {
        use speech::Poller;
        client
            .batch_recognize(format!(
                "projects/{project_id}/locations/global/recognizers/_"
            ))
            .poller()
            .until_done()
            .await
            .map(|r| r.total_billed_duration)
    }
}

#[cfg(test)]
mod test {
    use super::my_application::*;
    use super::*;
```

```
    mockall::mock! {
        #[derive(Debug)]
        Speech {}
        impl speech::stub::Speech for Speech {
            async fn batch_recognize(&self, req: BatchRecognizeRequest, _options:
gax::options::RequestOptions) -> Result<Operation>;
        }
    }

    fn expected_duration() -> Option<wkt::Duration> {
        Some(wkt::Duration::clamp(100, 0))
    }

    fn expected_response() -> BatchRecognizeResponse {
        BatchRecognizeResponse::new().set_total_billed_duration(expected_duration())
    }

    fn make_finished_operation(response: &BatchRecognizeResponse) -> Result<Operation>
{
        let any = wkt::Any::try_from(response).map_err(Error::serde)?;
        let operation = Operation::new()
            .set_done(true)
            .set_result(OperationResult::Response(any.into()));
        Ok(operation)
    }

    #[tokio::test]
    async fn automatic_polling() -> Result<()> {
        // Create a mock, and set expectations on it.
        let mut mock = MockSpeech::new();
        mock.expect_batch_recognize()
            .return_once(|_, _| make_finished_operation(&expected_response()));

        // Create a client, implemented by our mock.
        let client = speech::client::Speech::from_stub(mock);

        // Call our function which automatically polls.
        let billed_duration = my_automatic_poller(&client, "my-project").await?;

        // Verify the final result of the LRO.
```

```
        assert_eq!(billed_duration, expected_duration());

        Ok(())
    }
}
```

Manual polling with intermediate metadata - Full test

```rust
use gax::Result;
use gax::error::Error;
use google_cloud_gax as gax;
use google_cloud_longrunning as longrunning;
use google_cloud_speech_v2 as speech;
use google_cloud_wkt as wkt;
use longrunning::model::operation::Result as OperationResult;
use longrunning::model::{GetOperationRequest, Operation};
use speech::model::{BatchRecognizeRequest, BatchRecognizeResponse, OperationMetadata};

// Example application code that is under test
mod my_application {
    use super::*;

    pub struct BatchRecognizeResult {
        pub progress_updates: Vec<i32>,
        pub billed_duration: Result<Option<wkt::Duration>>,
    }

    // An example application function that manually polls.
    //
    // It starts an LRO. It consolidates the polling results, whether full or
    // partial.
    //
    // In this case, it is the `BatchRecognize` RPC. If we get a partial update,
    // we extract the `progress_percent` field. If we get a final result, we
    // extract the `total_billed_duration` field.
    pub async fn my_manual_poller(
        client: &speech::client::Speech,
        project_id: &str,
    ) -> BatchRecognizeResult {
        use speech::Poller;
        let mut progress_updates = Vec::new();
        let mut poller = client
            .batch_recognize(format!(
                "projects/{project_id}/locations/global/recognizers/_"
            ))
```

```
                .poller();
            while let Some(p) = poller.poll().await {
                match p {
                    speech::PollingResult::Completed(r) => {
                        let billed_duration = r.map(|r| r.total_billed_duration);
                        return BatchRecognizeResult {
                            progress_updates,
                            billed_duration,
                        };
                    }
                    speech::PollingResult::InProgress(m) => {
                        if let Some(metadata) = m {
                            // This is a silly application. Your application likely
                            // performs some task immediately with the partial
                            // update, instead of storing it for after the operation
                            // has completed.
                            progress_updates.push(metadata.progress_percent);
                        }
                    }
                    speech::PollingResult::PollingError(e) => {
                        return BatchRecognizeResult {
                            progress_updates,
                            billed_duration: Err(Error::from(e)),
                        };
                    }
                }
                tokio::time::sleep(std::time::Duration::from_millis(500)).await;
            }

            // We can only get here if `poll()` returns `None`, but it only returns
            // `None` after it returned `PollingResult::Completed`. Therefore this
            // is never reached.
            unreachable!("loop should exit via the `Completed` branch.");
        }
    }

    #[cfg(test)]
    mod test {
        use super::my_application::*;
        use super::*;
```

```
    mockall::mock! {
        #[derive(Debug)]
        Speech {}
        impl speech::stub::Speech for Speech {
            async fn batch_recognize(&self, req: BatchRecognizeRequest, _options:
gax::options::RequestOptions) -> Result<Operation>;
            async fn get_operation(&self, req: GetOperationRequest, _options:
gax::options::RequestOptions) -> Result<Operation>;
        }
    }

    fn expected_duration() -> Option<wkt::Duration> {
        Some(wkt::Duration::clamp(100, 0))
    }

    fn expected_response() -> BatchRecognizeResponse {
        BatchRecognizeResponse::new().set_total_billed_duration(expected_duration())
    }

    fn make_finished_operation(response: &BatchRecognizeResponse) -> Result<Operation>
{
        let any = wkt::Any::try_from(response).map_err(Error::serde)?;
        let operation = Operation::new()
            .set_done(true)
            .set_result(OperationResult::Response(any.into()));
        Ok(operation)
    }

    fn make_partial_operation(progress: i32) -> Result<Operation> {
        let metadata = OperationMetadata::new().set_progress_percent(progress);
        let any = wkt::Any::try_from(&metadata).map_err(Error::serde)?;
        let operation = Operation::new().set_metadata(Some(any));
        Ok(operation)
    }

    #[tokio::test]
    async fn manual_polling_with_metadata() -> Result<()> {
        let mut seq = mockall::Sequence::new();
        let mut mock = MockSpeech::new();
        mock.expect_batch_recognize()
            .once()
```

```
            .in_sequence(&mut seq)
            .returning(|_, _| make_partial_operation(25));
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| make_partial_operation(50));
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| make_partial_operation(75));
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| make_finished_operation(&expected_response()));

        // Create a client, implemented by our mock.
        let client = speech::client::Speech::from_stub(mock);

        // Call our function which manually polls.
        let result = my_manual_poller(&client, "my-project").await;

        // Verify the partial metadata updates, and the final result.
        assert_eq!(result.progress_updates, [25, 50, 75]);
        assert_eq!(result.billed_duration?, expected_duration());

        Ok(())
    }
}
```

Simulating errors - Full tests

```rust
use gax::Result;
use gax::error::Error;
use google_cloud_gax as gax;
use google_cloud_longrunning as longrunning;
use google_cloud_rpc as rpc;
use google_cloud_speech_v2 as speech;
use google_cloud_wkt as wkt;
use longrunning::model::operation::Result as OperationResult;
use longrunning::model::{GetOperationRequest, Operation};
use speech::model::{BatchRecognizeRequest, OperationMetadata};

// Example application code that is under test
mod my_application {
    use super::*;

    pub struct BatchRecognizeResult {
        pub progress_updates: Vec<i32>,
        pub billed_duration: Result<Option<wkt::Duration>>,
    }

    // An example application function that manually polls.
    //
    // It starts an LRO. It consolidates the polling results, whether full or
    // partial.
    //
    // In this case, it is the `BatchRecognize` RPC. If we get a partial update,
    // we extract the `progress_percent` field. If we get a final result, we
    // extract the `total_billed_duration` field.
    pub async fn my_manual_poller(
        client: &speech::client::Speech,
        project_id: &str,
    ) -> BatchRecognizeResult {
        use speech::Poller;
        let mut progress_updates = Vec::new();
        let mut poller = client
            .batch_recognize(format!(
                "projects/{project_id}/locations/global/recognizers/_"
```

```
                ))
                .poller();
            while let Some(p) = poller.poll().await {
                match p {
                    speech::PollingResult::Completed(r) => {
                        let billed_duration = r.map(|r| r.total_billed_duration);
                        return BatchRecognizeResult {
                            progress_updates,
                            billed_duration,
                        };
                    }
                    speech::PollingResult::InProgress(m) => {
                        if let Some(metadata) = m {
                            // This is a silly application. Your application likely
                            // performs some task immediately with the partial
                            // update, instead of storing it for after the operation
                            // has completed.
                            progress_updates.push(metadata.progress_percent);
                        }
                    }
                    speech::PollingResult::PollingError(e) => {
                        return BatchRecognizeResult {
                            progress_updates,
                            billed_duration: Err(Error::from(e)),
                        };
                    }
                }
                tokio::time::sleep(std::time::Duration::from_millis(500)).await;
            }

            // We can only get here if `poll()` returns `None`, but it only returns
            // `None` after it returned `PollingResult::Completed`. Therefore this
            // is never reached.
            unreachable!("loop should exit via the `Completed` branch.");
        }
    }

    #[cfg(test)]
    mod test {
        use super::my_application::*;
        use super::*;
```

```
    mockall::mock! {
        #[derive(Debug)]
        Speech {}
        impl speech::stub::Speech for Speech {
            async fn batch_recognize(&self, req: BatchRecognizeRequest, _options:
gax::options::RequestOptions) -> Result<Operation>;
            async fn get_operation(&self, req: GetOperationRequest, _options:
gax::options::RequestOptions) -> Result<Operation>;
        }
    }

    fn make_partial_operation(progress: i32) -> Result<Operation> {
        let metadata = OperationMetadata::new().set_progress_percent(progress);
        let any = wkt::Any::try_from(&metadata).map_err(Error::serde)?;
        let operation = Operation::new().set_metadata(Some(any));
        Ok(operation)
    }

    fn make_failed_operation(status: rpc::model::Status) -> Result<Operation> {
        let operation = Operation::new()
            .set_done(true)
            .set_result(OperationResult::Error(status.into()));
        Ok(operation)
    }

    #[tokio::test]
    async fn error_starting_lro() -> Result<()> {
        let mut mock = MockSpeech::new();
        mock.expect_batch_recognize()
            .return_once(|_, _| Err(Error::other("failed to start operation")));

        // Create a client, implemented by our mock.
        let client = speech::client::Speech::from_stub(mock);

        // Call our function which manually polls.
        let result = my_manual_poller(&client, "my-project").await;

        // Verify the the final result.
        assert!(result.billed_duration.is_err());
```

```
    Ok(())
}

#[tokio::test]
async fn lro_ending_in_error() -> Result<()> {
    let mut seq = mockall::Sequence::new();
    let mut mock = MockSpeech::new();
    mock.expect_batch_recognize()
        .once()
        .in_sequence(&mut seq)
        .returning(|_, _| make_partial_operation(25));
    mock.expect_get_operation()
        .once()
        .in_sequence(&mut seq)
        .returning(|_, _| make_partial_operation(50));
    mock.expect_get_operation()
        .once()
        .in_sequence(&mut seq)
        .returning(|_, _| make_partial_operation(75));
    mock.expect_get_operation()
        .once()
        .in_sequence(&mut seq)
        .returning(|_, _| {
            // This is a common error for `Create*` RPCs, which are often
            // LROs. It is less applicable to `BatchRecognize` in practice.
            let status = rpc::model::Status::default()
                .set_code(gax::error::rpc::Code::AlreadyExists as i32)
                .set_message("resource already exists");
            make_failed_operation(status)
        });

    // Create a client, implemented by our mock.
    let client = speech::client::Speech::from_stub(mock);

    // Call our function which manually polls.
    let result = my_manual_poller(&client, "my-project").await;

    // Verify the partial metadata updates, and the final result.
    assert_eq!(result.progress_updates, [25, 50, 75]);
    assert!(result.billed_duration.is_err());
```

```
        Ok(())
    }

    #[tokio::test]
    async fn polling_loop_error() -> Result<()> {
        let mut seq = mockall::Sequence::new();
        let mut mock = MockSpeech::new();
        mock.expect_batch_recognize()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| make_partial_operation(25));
        mock.expect_get_operation()
            .once()
            .in_sequence(&mut seq)
            .returning(|_, _| Err(Error::other("could not poll operation")));

        // Create a client, implemented by our mock.
        let client = speech::client::Speech::from_stub(mock);

        // Call our function which manually polls.
        let result = my_manual_poller(&client, "my-project").await;

        // Verify the partial metadata updates, and the final result.
        assert_eq!(result.progress_updates, [25]);
        assert!(result.billed_duration.is_err());

        Ok(())
    }
}
```