How to write tests using the Storage client

The Google Cloud Client Libraries for Rust provide a way to stub out the real client implementations, so a mock can be injected for testing.

Applications can use mocks to write controlled, reliable unit tests that do not involve network calls, and do not incur billing.

In this guide, you will learn:

- Why the design of the `Storage` client deviates from the design of other Google Cloud clients
- How to write testable interfaces using the `Storage` client
- How to mock reads
- How to mock writes

This guide is specifically for mocking the `Storage` client. For a generic mocking guide (which applies to the `StorageControl` client), see How to write tests using a client.

Design rationale

Other clients

Most clients, such as `StorageControl` hold a boxed, `dyn`-compatible implementation of the stub trait internally. They use dynamic dispatch to forward requests from the client to their stub (which could be the real implementation or a mock).

Because dynamic dispatch is used, the exact type of the stub does not need to be known by the compiler. The clients do not need to be templated on their stub.

`Storage` client

In order to have a `dyn`-compatible trait, the size of all types must be known.

The `Storage` client has complex types in its interfaces.

- `write_object` accepts a generic payload.
- `read_object` returns a stream-like thing.

Thus, if we wanted to use the same dynamic dispatch approach for the `Storage` client, we would have to end up boxing all generics / trait `impl`s. Each box is an extra heap allocation, plus the dynamic dispatch.

Because we want the `Storage` client to be as performant as possible, we decided it was preferable to template the client on a non-`dyn`-compatible, concrete implementation of the stub trait.

Testable interfaces

Applications that do not need to test their code can simply write all interfaces in terms of `Storage`. The default `T` is the real implementation of the client.

```
pub async fn my_function(_client: Storage) {}
```

Applications that need to test their code, should write their interfaces in terms of the generic `T`.

```
pub async fn my_testable_function<T>(_client: Storage<T>)
where
    T: gcs::stub::Storage + 'static,
{
}
```

Mocking reads

Let's say we have an application function which downloads an object and counts how many newlines it contains.

```
// Downloads an object from GCS and counts the total lines.
pub async fn count_newlines<T>(
    client: &Storage<T>,
    bucket_id: &str,
    object_id: &str,
) -> gcs::Result<usize>
where
    T: gcs::stub::Storage + 'static,
{
    let mut count = 0;
    let mut reader = client
        .read_object(format!("projects/_/buckets/{bucket_id}"), object_id)
        .set_generation(42)
        .send()
        .await?;
    while let Some(buffer) = reader.next().await.transpose()? {
        count += buffer.into_iter().filter(|c| *c == b'\n').count();
    }
    Ok(count)
}
```

We want to test our code against a known response from the server. We can do this by faking the `ReadObjectResponse`.

A `ReadObjectResponse` is essentially a stream of bytes. You can create a fake `ReadObjectResponse` in tests by supplying a payload to `ReadObjectResponse::from_source`. The library accepts the same payload types as `Storage::write_object`.

```
    fn fake_response(size: usize) -> ReadObjectResponse {
        let mut contents = String::new();
        for i in 0..size {
            contents.push_str(&format!("{i}\n"))
        }
        ReadObjectResponse::from_source(ObjectHighlights::default(),
bytes::Bytes::from(contents))
    }
```

We define the mock as usual, using `mockall`.

```
    mockall::mock! {
        #[derive(Debug)]
        Storage {}
        impl gcs::stub::Storage for Storage {
            async fn read_object(&self, _req: ReadObjectRequest, _options:
RequestOptions) -> Result<ReadObjectResponse>;
            async fn write_object_buffered<P: StreamingSource + Send + Sync +
'static>(
                &self,
                _payload: P,
                _req: WriteObjectRequest,
                _options: RequestOptions,
            ) -> Result<Object>;
            async fn write_object_unbuffered<P: StreamingSource + Seek + Send +
Sync + 'static>(
                &self,
                _payload: P,
                _req: WriteObjectRequest,
                _options: RequestOptions,
            ) -> Result<Object>;
        }
    }
```

We write a unit test, which calls into our `count_newlines` function.

```
    #[tokio::test]
    async fn test_count_lines() -> anyhow::Result<()> {
        let mut mock = MockStorage::new();
        mock.expect_read_object().return_once({
            move |r, _| {
                // Verify contents of the request
                assert_eq!(r.generation, 42);
                assert_eq!(r.bucket, "projects/_/buckets/my-bucket");
                assert_eq!(r.object, "my-object");

                // Return a `ReadObjectResponse`
                Ok(fake_response(100))
            }
        });
        let client = gcs::client::Storage::from_stub(mock);

        let count = count_newlines(&client, "my-bucket", "my-object").await?;
        assert_eq!(count, 100);

        Ok(())
    }
```

Mocking writes

Let's say we have an application function which uploads an object from memory.

```
// Uploads an object to GCS.
pub async fn upload<T>(client: &Storage<T>, bucket_id: &str, object_id: &str) ->
gcs::Result<Object>
where
    T: gcs::stub::Storage + 'static,
{
    client
        .write_object(
            format!("projects/_/buckets/{bucket_id}"),
            object_id,
            "payload",
        )
        .set_if_generation_match(42)
        .send_unbuffered()
        .await
}
```

To test this function, we define the mock as usual, using `mockall`.

```
    mockall::mock! {
        #[derive(Debug)]
        Storage {}
        impl gcs::stub::Storage for Storage {
            async fn read_object(&self, _req: ReadObjectRequest, _options:
RequestOptions) -> Result<ReadObjectResponse>;
            async fn write_object_buffered<P: StreamingSource + Send + Sync +
'static>(
                &self,
                _payload: P,
                _req: WriteObjectRequest,
                _options: RequestOptions,
            ) -> Result<Object>;
            async fn write_object_unbuffered<P: StreamingSource + Seek + Send +
Sync + 'static>(
                &self,
                _payload: P,
                _req: WriteObjectRequest,
                _options: RequestOptions,
            ) -> Result<Object>;
        }
    }
```

We write a unit test, which calls into our `upload` function.

```
    #[tokio::test]
    async fn test_upload() -> anyhow::Result<()> {
        let mut mock = MockStorage::new();
        mock.expect_write_object_unbuffered()
            .return_once(
                |_payload: Payload<BytesSource>, r, _| {
                    // Verify contents of the request
                    assert_eq!(r.spec.if_generation_match, Some(42));
                    let o = r.spec.resource.unwrap_or_default();
                    assert_eq!(o.bucket, "projects/_/buckets/my-bucket");
                    assert_eq!(o.name, "my-object");

                    // Return the object
                    Ok(Object::default()
                        .set_bucket("projects/_/buckets/my-bucket")
                        .set_name("my-object")
                        .set_generation(42))
                },
            );
        let client = gcs::client::Storage::from_stub(mock);

        let object = upload(&client, "my-bucket", "my-object").await?;
        assert_eq!(object.generation, 42);

        Ok(())
    }
```

Details

Because our function calls `send_unbuffered()`, we should use the corresponding `write_object_unbuffered()`.

```
        mock.expect_write_object_unbuffered()
```

Generics in `mockall::mock!` are treated as different functions. We need to provide the exact payload type, so the compiler knows which function to use.

```
                |_payload: Payload<BytesSource>, r, _| {
```

Full application code and test suite

```rust
use gcs::client::Storage;
use gcs::model::Object;
use google_cloud_storage as gcs;

// Downloads an object from GCS and counts the total lines.
pub async fn count_newlines<T>(
    client: &Storage<T>,
    bucket_id: &str,
    object_id: &str,
) -> gcs::Result<usize>
where
    T: gcs::stub::Storage + 'static,
{
    let mut count = 0;
    let mut reader = client
        .read_object(format!("projects/_/buckets/{bucket_id}"), object_id)
        .set_generation(42)
        .send()
        .await?;
    while let Some(buffer) = reader.next().await.transpose()? {
        count += buffer.into_iter().filter(|c| *c == b'\n').count();
    }
    Ok(count)
}

// Uploads an object to GCS.
pub async fn upload<T>(client: &Storage<T>, bucket_id: &str, object_id: &str) ->
gcs::Result<Object>
where
    T: gcs::stub::Storage + 'static,
{
    client
        .write_object(
            format!("projects/_/buckets/{bucket_id}"),
            object_id,
            "payload",
        )
        .set_if_generation_match(42)
        .send_unbuffered()
        .await
}

#[cfg(test)]
mod tests {
    use super::{count_newlines, upload};
    use gcs::Result;
    use gcs::model::{Object, ReadObjectRequest};
    use gcs::model_ext::{ObjectHighlights, WriteObjectRequest};
    use gcs::read_object::ReadObjectResponse;
    use gcs::request_options::RequestOptions;
    use gcs::streaming_source::{BytesSource, Payload, Seek, StreamingSource};
```

```
    use google_cloud_storage as gcs;

    mockall::mock! {
        #[derive(Debug)]
        Storage {}
        impl gcs::stub::Storage for Storage {
            async fn read_object(&self, _req: ReadObjectRequest, _options:
RequestOptions) -> Result<ReadObjectResponse>;
            async fn write_object_buffered<P: StreamingSource + Send + Sync +
'static>(
                &self,
                _payload: P,
                _req: WriteObjectRequest,
                _options: RequestOptions,
            ) -> Result<Object>;
            async fn write_object_unbuffered<P: StreamingSource + Seek + Send +
Sync + 'static>(
                &self,
                _payload: P,
                _req: WriteObjectRequest,
                _options: RequestOptions,
            ) -> Result<Object>;
        }
    }

    fn fake_response(size: usize) -> ReadObjectResponse {
        let mut contents = String::new();
        for i in 0..size {
            contents.push_str(&format!("{i}\n"))
        }
        ReadObjectResponse::from_source(ObjectHighlights::default(),
bytes::Bytes::from(contents))
    }

    #[tokio::test]
    async fn test_count_lines() -> anyhow::Result<()> {
        let mut mock = MockStorage::new();
        mock.expect_read_object().return_once({
            move |r, _| {
                // Verify contents of the request
                assert_eq!(r.generation, 42);
                assert_eq!(r.bucket, "projects/_/buckets/my-bucket");
                assert_eq!(r.object, "my-object");

                // Return a `ReadObjectResponse`
                Ok(fake_response(100))
            }
        });
        let client = gcs::client::Storage::from_stub(mock);

        let count = count_newlines(&client, "my-bucket", "my-object").await?;
        assert_eq!(count, 100);

        Ok(())
```

```
    }

    #[tokio::test]
    async fn test_upload() -> anyhow::Result<()> {
        let mut mock = MockStorage::new();
        mock.expect_write_object_unbuffered()
            .return_once(
                |_payload: Payload<BytesSource>, r, _| {
                    // Verify contents of the request
                    assert_eq!(r.spec.if_generation_match, Some(42));
                    let o = r.spec.resource.unwrap_or_default();
                    assert_eq!(o.bucket, "projects/_/buckets/my-bucket");
                    assert_eq!(o.name, "my-object");

                    // Return the object
                    Ok(Object::default()
                        .set_bucket("projects/_/buckets/my-bucket")
                        .set_name("my-object")
                        .set_generation(42))
                },
            );
        let client = gcs::client::Storage::from_stub(mock);

        let object = upload(&client, "my-bucket", "my-object").await?;
        assert_eq!(object.generation, 42);

        Ok(())
    }
}
```