INTERNATIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT (IJSREM)
VOLUME: 05 ISSUE: 11 | NOV - 2021 SJIF RATING: 6.714 ISSN: 2582-3930

Testing Golang Spanner Interactions Using Test Doubles

Nilesh Jagnik
Mountain View, USA
nileshjagnik@gmail.com

***Abstract*—Almost all software systems require the use of databases for storing application data and state. As such, it is important to test the correctness of database operations made by various components of a system. A lack of testing can lead to bugs, errors and even outages in production. For systems using database services like Cloud Spanner, it is difficult to spawn a local database instance. This can make hermetic test setup quite difficult. To solve this problem, Spanner provides test doubles which can be used for unit and functional testing of database operations. These test doubles act like real Spanner servers but keep data in memory only and do not persist it. Although the performance and efficiency of these test doubles is quite low, they are well suited for hermetic correctness testing. In this paper, we review test doubles and the best practices associated with them. We then take a look at two test doubles offered by Spanner, with one of them being available only in Golang. We also cover the caveats associated with the use of these test doubles.**

Keywords—databases, software testing, test doubles, Cloud Spanner

I. INTRODUCTION

Databases are the backbone of software applications. They are useful for input data, metadata and application state. Databases allow separating the concerns of persisting data from logic that does computations using data. As such, databases have an important role in software design.

Databases can be of many types offering different features and capabilities. In applications where data points can have relationships between them, relational databases are often the right choice. For applications using Google Cloud for data storage, there are several relational database offerings available. Cloud Spanner is the optimal choice for a fully managed, scalable, multi-region database.

Along with choosing the right database service, developers should also test their applications. This includes unit testing for individual classes/modules and integration and functional testing for asserting end-to-end behavior. Because developers don't own Spanner, they may be tempted to mock/stub interactions with Spanner when writing tests. However, this may reduce the effectiveness of tests in detecting errors and bugs. This is because the real instance of Spanner may not behave like the mock/stub. In general, the use of mocks and stubs makes tests less effective and harder to setup and maintain.

A better idea would be to use a test instance provided by Spanner. Spanner provides a few lightweight database instances for testing. These implementations provide the same functionality as real Spanner, but not the same performance. They are meant only for testing correctness of an application's interactions with Spanner.

In this paper, we first look at different test doubles that can be used for testing, and why we would prefer one over the other. We then look at test doubles available for testing Spanner interactions in Golang, which has the best support for Spanner testing.

II. TEST DOUBLES

A test double can act the same as a real object in a test environment. There are a few types of test doubles that can be used to test interactions with external systems. Let us review them and look at best practices around the use of test doubles.

A. Stubs

A stub is a test double which executes no logic and returns a response, which can be configured during test setup. Stubs are used to mimic certain responses from an external library or service to get the test execution into an expected state. They don't do any of the actual work a real instance would do and cannot mimic state changes which may occur from interactions with real objects. Since stubs don't do any actual work, it is easy to manually create stubs that follow the real instance's interface. However, it is common to use a mocking framework to create stubs.

B. Mocks

A mock is test double which is similar to a stub, but provides some additional features. This includes the ability to track how many times the mock instance was called, what parameters values it was called with, etc. Mocks can used to test that interactions with the test doubles are correct and valid. Similar to stubs, mocks are useful in cases where no state changes are expected as a result of interaction with the real instance. E.g., A mock can be used to verify that a string was stored into a file, but it cannot be used to verify that the contents of the file are correctly written (since the mock will not actually write to the file). It is common to use mocking frameworks that can create mocks and stubs.

C. Fakes

Unlike stubs and mocks which use a mocking framework to create a no-op version of the real instance, fakes are lightweight implementations that follow the real object's

```
func TestUsingSpannerFake(t *testing.T) {
  // Start a new in-memory fake server.
  spanner_server, err := spannertest.NewServer(":0")
  assert.NilError(t, err)

  ctx := context.Background()
  db_path := "projects/P/instances/I/databases/D"

  // Dail into fake server with no security.
  conn, err := grpc.DialContext(
    ctx,
    spanner_server.Addr,
    grpc.WithTransportCredentials(
                  insecure.NewCredentials())
  )

  // Create Spanner client.
  client, err := spanner.NewClient(
    ctx,
    db_path,
    option.WithGRPCConn(conn)
  )
  assert.NilError(t, err)
  defer client.Close()

  // Use spanner client as normal
  ...
}
```

interface. Fakes are expected to behave exactly like the real object but not be as resource efficient or performant as the real object. This is considered fine since tests process a small amount of data and thus, do not require the performance or efficiency of the real instance. Fakes are normally owned and maintained by the owner of the real object so that clients have an easy way to test interactions with the real object.

D. Best Practices

Mocks and stubs can cause more problems than they solve. They make tests harder to understand and maintain. Fakes are better since the test code does not usually need a lot of setup and maintenance to use Fakes. However, all test doubles have one common issue – test doubles may behave differently to real instances because they are different implementations. Developers to use real objects wherever possible. If real objects are not available, they should prefer the use of fakes. The use of mocks and stubs should be limited to simple use cases where there is very less reliance on test double behavior.

III. SPANNER IN-MEMORY FAKE FOR UNIT TESTING

The spannertest package in Golang allows easy testing of Cloud Spanner interactions. This package creates a fake in-memory implementation of Spanner. The fake server can be started and closed inside of a unit test.

A. Usage

Using the spannertest package is simple and requires minimal setup. The core functionality of this package is provided by spannertest.NewServer(), which creates a new in-memory server. The server will be listening for gRPC connections on the provided address (without any security). Once the server is up, we can use gRPC to start a connection with this server. Note that this connection should be insecure. Finally, we use this connection to create a new Spanner client. The Spanner client can then be used by the code under test that makes Spanner calls. The rest of the code can just assume that real Spanner is being used.

Fig. 1. In-memory fake for unit testing in Golang (using spannertest).

B. Caveats

Although the spannertest package is great for easy unit testing of code interacting with Spanner, it is experimental and does not support all Spanner features. There are several read/write operations, validations and data types supported by Spanner, which are not supported by the in-memory fake. Any use of these operations in the code under test will raise Spanner exceptions. This could cause confusion and frustration, so knowing and documenting the limitations of this package is important when using it.

Another important thing to note is that the in-memory server does not actually save data anywhere except in memory, so data is lost after test execution.

IV. SPANNER EMULATOR FOR FUNCTIONAL TESTING

In addition to the spannertest package which is only available for unit testing Golang, there is also a generic emulator provided by Spanner. This emulator starts a local Spanner server on a user-specified port. Similar to spannertest, this emulator also provides an in-memory Spanner server. The emulator can be created without setting up billing and is free of cost. Since this emulator does not provide an easy way for creation as part of a unit test, it is more suited for local development and functional tests, where test environment setup can be handled manually or scripted into environment creation.

A. Setup

Using the Spanner emulator is similar to using spannertest, however the steps for creating the fake server are different. In this workflow the fake server is created using the gcloud command shown in Fig. 2.

```
# Starts a gRPC server that listens on localhost:9010
$ gcloud emulators spanner start
```

Fig. 2. Command for starting Spanner emulator.

B. Usage

Once the emulator is set up, the usage in test code is very similar to Fig. 1. The only difference being that the step for starting the server should be omitted. Instead, client libraries should set the SPANNER_EMULATOR_HOST environment variable. Spanner clients check for this variable to be set and automatically connect to the local emulator when this variable is set.

```
$ export SPANNER_EMULATOR_HOST=localhost:9010
```

Fig. 3. Setting the enviroment variable that forces connection to emulator.

C. Caveats

Similar to the spannertest package, there are some limitations in the local emulator's functionality. It does not support security, authentication and authorization, so any

logic around these cannot be tested using the emulator. In addition, it does not support profiles of queries. The ANALYZE statement is ignored by the emulator. Logging and monitoring are also not supported.

There are also some differences from production Spanner, e.g., the error messages generated by the emulator may not be same as production. The performance of the emulator is much lower as compared to production, so it is only suited for testing on smaller data sets. The emulator stores data only in memory so data is lost if the emulator is restarted or shut down.

CONCLUSION

Testing database interactions is mandatory for ensuring correctness in an application. For applications using Cloud Spanner, it is hard to spawn a real Spanner database inside a test environment. Test double best practices suggest the use of fakes in cases where real objects are not available. This is because other test doubles are manually created and provide no real guarantees of behaving like the real object. For this reason, Spanner provided fakes are the best option for testing database interactions. Spanner provides an in-memory instance for unit testing of Golang applications. There is also an emulator which is better suited to functional testing and can be used in any language. With both of these test doubles, there are caveats to be aware of, namely limited functionality and reduced performance. Both of these test doubles are in-memory and do not actually persist any data to storage.

REFERENCES

[1] Andrew Trenk, “Testing on the Toilet: Know Your Test Doubles (Jul 2013),” https://testing.googleblog.com/2013/07/testing-on-toilet-know-your-test-doubles.html

[2] Andrew Trenk, “Testing on the Toilet: Don’t Overuse Mocks (May 2013),” https://testing.googleblog.com/2013/05/testing-on-toilet-dont-overuse-mocks.html

[3] Laurence de Jong, “Cloud Spanner Testing in Go (Feb 2021),” https://ldej.nl/post/cloud-spanner-testing-in-go

[4] “spannertest (Oct 2021),” https://pkg.go.dev/cloud.google.com/go/spanner/spannertest

[5] “Cloud Spanner Emulator (Mar 2021),” https://github.com/GoogleCloudPlatform/cloud-spanner-emulator

[6] Peter Runge, “Testing a Spring Boot application with the Google Cloud Spanner Emulator (Nov 2020),” https://medium.com/google-cloud/testing-a-spring-boot-application-with-the-google-cloud-spanner-emulator-3c4d5d6b52fb

[7] “Cloud Spanner Emulator (Apr 2020),” https://opensource.googleblog.com/2020/04/cloud-spanner-emulator.html