DGP Testing Objectives

DGP's software development methodology, based on the scientific method, relies heavily on various types of testing in order to prove that the results that are periodically delivered are able to meet all of the applicable requirements. All of the testing functionality and applications needed to verify most of the NFR's have been designed and built into DGP itself. In general, DGP systems need to constantly be able to provide proof of the following:

- Prove the functionality is correct
- Prove the data is correct
- Prove (monitor) high performance and efficiency
    - End user testing and monitoring (end-to-end)
    - Long duration load tests (collect server resource metrics)

DGP's message-based APIs allow additional data to be sent to and from each API method, and this capability is what makes it possible to build unit tests and server performance monitoring into every API method. The results of the unit test(s) and server performance are returned as part of every API response message. This allows those results to be displayed to every user of the system, and is also the basis for the API Tester test harness application, which runs remote end-to-end tests of every API method in a system.

API Tester Test Harness

The API Tester test harness application has been designed and built specifically to test DGP's message-based APIs by running special test files. The test harness is used frequently by both developers and testers, and can also easily be used by other users of a system by granting them membership to a tester role.

Developers are responsible for creating the initial test file for each API method, and focus on positive tests of the functionality. Testers build on that and add a variety of negative tests to the test files of each method. The objective of these groups of test files is to test both positive and alternate paths through all the code of each API method. The test harness can run a single test file, all test files, or some selected groups of test files, as needed. Test files are grouped in folders to help simplify the scope (number of methods tested) per test run. The data from a test run can be saved to the current location, and also to a local Excel file for further analysis.

Software System Efficiency Problems

The API Tester is also able to repeatedly run the same set of test files for a given number of iterations to generate a synthetic workload for load testing. The easiest way to explain the importance of load tests is by providing some examples of serious problems with several software systems that would have been uncovered by load testing. The two examples are both .NET systems, one created by a business and the other a vendor's expensive low-code development platform. Both systems worked and performed well at a small scale during the normal testing performed to verify the functionality of the applications. The problems occurred after the systems were put into production and experienced heavy workloads for days at a time.

The custom software system was found to use more and more memory on the host server the longer it ran, behaving very much like a memory leak. Also, during the period of time it was running, the performance of the system would get slower and slower. By the second day, the system was running so slowly that it was basically unusable. By that time, the host server would usually run out of memory and crash. Similarly, custom applications built using a vendor's low-code development platform would behave in much the same way, although it generally took a longer time for it to cause their servers to crash. A lot of time was spent belatedly performing load testing for these systems to help identify the root causes of the problems and correct them, at great expense.

The lesson to be learned from these examples is that some performance, efficiency and scalability problems are not uncovered by normal low-volume testing, and will only occur under high workloads applied over long periods of time. However, it is usually difficult and expensive to perform load tests for most software systems. The ideal objective is to test an individual server to the point of failure, but generating a sufficient volume of synthetic workload to reach that point can be problematic. Using the smallest possible server with the least resources will help to make load testing less expensive, assuming the small server results can be extrapolated to larger servers. It is also important to consider the volume of data that will be generated during the load testing to make sure it will not exceed the storage capacity of the test server.

API Tester Load Testing

The API Tester test harness can run full regression tests of all the API methods in a system (or a selected subset) over and over again for a specified number of iterations. A sufficiently high number of iterations can take a day or longer to complete. Each such iteration simulates the traffic of X number of users, depending on the usage patterns of the system. Minimizing the network latency of these end-to-end tests between the test harness and the server will help maximize the workload applied to the test server, so the workstations running the API Tester and the test server should ideally be connected to the same LAN switch. The total volume of the synthetic workload can be increased to the desired level by running multiple instances of the API Tester application in parallel.

For DGP load tests, during the period of time that the API Tester application(s) are constantly running their iterations of test files, various resource utilization and GC metrics should be monitored and recorded on the test server. The desired behavior is that the resources used by the server will plateau at levels corresponding to the volume of the test workload, and that performance will remain high and not degrade over time.

API Tester Test Files

The test files are basically API request messages that have been saved as files, and then modified with some additional elements needed for the functionality of the API Tester test harness application.

```
<TestBatch>                                        - a collection of one or more test messages
  <TName>APIRole New Negative</TName>
  <TDescrip>Negative tests of the APIRole New method</TDescrip>
  <TGID>{{TGID}}</TGID>
  <TMsg>                                           - each test message is run sequentially by the test harness
    <TMUserName>{{TMUserName}}</TMUserName>        - the double curly braces indicate a template placeholder value
    <TMPassword>{{TMPassword}}</TMPassword>        - placeholder values are replaced with actual values in each test run
    <TMName>APIRole.New.base</TMName>
    <TMDescrip>empty input parameters</TMDescrip>
    <TMExpAuthCode>OK</TMExpAuthCode>              - the expected authentication code for the user account
    <Meth>                                         - each Meth element is the same as an actual API request message
      <MName>APIRole.New.base</MName>
      <PList>
        <Prm>
```

```
          <Name>RoleName</Name>
         <Val>
            <![CDATA[]]>
          </Val>
        </Prm>
        <Prm>
          <Name>RoleDescrip</Name>
          <Val>
            <![CDATA[]]>
          </Val>
        </Prm>
      </PList>
    </Meth>
    <RList>                                  - a list of one or more results returned by the test message
      <Result>
        <RName>APIRole.New.base_DEFAULT</RName>
        <ExpRCode>OK</ExpRCode>              - the expected Result Code
        <ExpDType>TEXT</ExpDType>            - the expected Data Type
        <ExpRVal></ExpRVal>                  - the expected Result Value
        <ValMatch></ValMatch>                - flag value indicating if the returned value matches the expected value
        <VarName></VarName>                  - a name for the result value returned by the test message, which can be used as if it were
      </Result>                              a template placeholder in test messages run later in the test batch.
    </RList>
  </TMsg>
</TestBatch>
```

The DGPDrive open-source code also contains all the test files used to test every API method in a DGPDrive system. These test files consist of two main types: the positive test “CRUD” files, along with negative test files that go along with them. Combined, they should ideally provide 100% test coverage of the code in each API method.

Test files will usually contain more than one API method call, each of which are executed sequentially by the API Tester test harness (no batch execution of the multiple method calls like a standard API Request message). This is done so that the results of one method call can be used as an input in later method calls within the same test file. This is accomplished by assigning a returned value to the VarName element in the Result section of the test file method call.

Doing so adds the return value to the collection of name/value pairs stored in a generic Dictionary. The named value can then be substituted as an input into test file method calls by enclosing it in double curly braces, for example {{VarName}}. Certain variable names such as {{TMUserName}} and {{TMPassword}} are standard in the API Tester test harness, in this case to run each test using the credentials of the user that connected to the location of a DGP system. A good place to see working examples of this feature are the *_CRUD_proc.xml test files that exist for each set of API methods corresponding to a database table, such as APIMethod_CRUD_proc.xml. These positive path test files are a sequential set of method calls that:

1. Create a new test record, returning its global ID (which is saved as a variable and used in later method calls).
2. Queries for the new record by ID value
3. Queries for the new record by Name value
4. Searches for the new record by Name (includes the Count and Search method calls for server-side pagination)
5. *Any other query API methods would also be called
6. Updates the test record to edit a value
7. *Optional read-after-write query of the edited value(s)
8. Queries for the history of the record (returns all edited and active versions)
9. Deletes the test record (soft delete, which is another update)
10. Calls a method to check for duplicate records in the table

All of the CRUD_proc.xml test files follow this basic pattern to call the various Create, Read, Update and Delete methods, for each database table, and also "cleans up" after itself by deleting the test record if the entire process runs successfully. These files also include some negative tests mixed in with the positive tests, such as attempting to create duplicate records, or update the wrong record, etc.

The negative test files call each of the API methods individually, first with empty input parameters to test the required field logic, and then with missing input parameters to test the lower-level API message logic. Those two sets of negative tests cover most of the error and exception handling logic of the API methods. Additional tests can pass in the wrong data types for methods that convert the default text inputs to other types such as numbers or dates – but those are fairly uncommon.

Note: The TestReplica test files test replication and count check methods between the TestDB1 and TestDB2 databases on a single computer. In some situations, this is easier to use than configuring replication between different locations (although that is preferable). Those test files in the TestReplica folder should be separated from the other test files, because those test files will not work correctly until they have been edited to use static values that are valid in a given location. Compatibility can only be tested manually by creating applications using many different programming languages running on many different platforms, all of which call a single instance of the web service API's.

Continuous Testing

As mentioned previously, in order to be able to easily (and continuously) prove that DGP systems are functioning correctly and performing well, unit tests and server-side performance monitoring have been permanently built into each API method. The message-based API web services are able to contain additional data compared to standard method calls, and this capability is used to return the results of the built-in unit test and the server-side performance measurement as part of each API response message. Any errors or poor performance are also logged using DGP's standard global logging functionality.

What this means in practice is that every API method call runs the built-in unit test and measures how long it takes to execute the method on the server. The results of the API method unit tests and the end-to-end performance are displayed to every user of a system in the status bar of the client applications as they use the system. In this way, every user is verifying the correct functionality and high performance of every API method, as they use the system in a production environment. Users that are members of the RemoteMonitor role save these end-to-end test and performance results to the SysMetrics database of the location they are connected to, effectively treating those users as remote monitors of the system. The number of accounts that are members of the RemoteMonitor role is used to control the amount of data that is collected.

Automated processes are used to incrementally replicate data from one location to another, and then also to incrementally scan the data in each location proving its correctness (scanning for missing records, duplicate records, and mismatched values). The same techniques are used to monitor the automated processes, which are themselves implemented as API methods that are periodically called/executed by a scheduler application. The same role membership mechanism applied to service accounts is used to limit the volume of data being collected from the scheduled execution of the automated processes. This role membership does not affect the two-stage logging of process data into their respective schedule database, event viewer and iteration log files, etc.