October 2021

Dependent Request Inference and Execution in Integration Tests

Patrick Blesi

Dependent Request Inference and Execution in Integration Tests

ABSTRACT

The test setup phase of an automated software test for an application programming interface (API) typically requires the creation of many dependent objects. Current techniques for test setup typically require a test writer to explicitly define dependencies between resources, which entails substantial manual effort to build the setup necessary to test an API request. This disclosure describes techniques to automatically assemble a dependency graph and build the setup and cleanup necessary to test an API request. An API service definition is analyzed to enumerate all remote procedure calls (RPCs) and infer the fields and RPCs required to successfully execute a request. Structured API definitions and heuristics are used to determine whether an API request has a dependency on other API requests. These dependencies are assembled into a dependency graph, the nodes of which are executed after being topologically sorted.

KEYWORDS

- Software testing
- Four-phase test
- Test setup
- Software under test (SUT)
- Remote procedure call (RPC)
- Application programming interface (API)
- Structured API
- API definition
- Dependency graph

BACKGROUND

An automated software test for testing an application programming interface (API) is typically structured into four phases [1] - setup, exercise, verify, and cleanup. (The cleanup phase is also referred to as teardown.) The setup phase creates the desired state for the software under

test (SUT) and is carried out in the form of API requests. A simple example of an API request for setup is the following: a resource must be created before its readability can be tested. Test setup is generally much more complicated than this simple example, and typically requires the creation of many dependent objects. Defining this setup phase takes time and effort when trying to test a given API request.

Current techniques for test setup typically require the test writer to explicitly define dependencies between resources, which in turn entails a high level of manual effort to build the setup necessary to test an API request.

DESCRIPTION

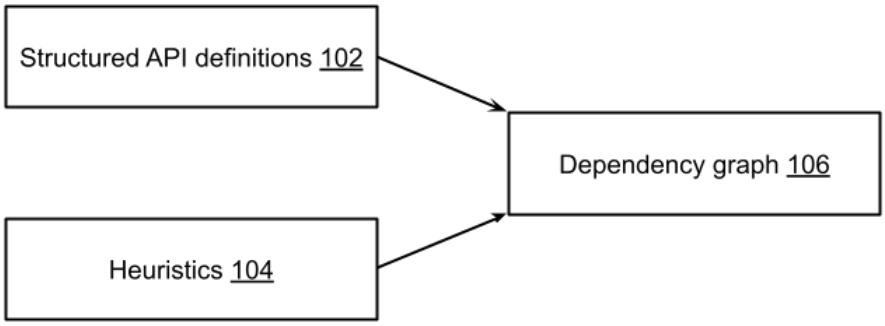

Fig. 1: Automatic assembly of a dependency graph

This disclosure describes techniques to automatically assemble a dependency graph and build the setup and cleanup necessary to test an API request. An API service definition is introspected (analyzed) to enumerate all remote procedure calls (RPCs) and infer the fields and RPCs required to successfully execute a request. Per the techniques, illustrated in Fig. 1, structured API definitions (102, e.g., service [4] and resource [5] proto definitions) and heuristics (104) are used to determine whether an API request has a dependency on other API requests. Dependencies may be other API requests or required fields that must be populated. Additionally,

a test-writer can add hints to indicate specific dependencies for a given API request. These dependencies are assembled into a dependency graph (106).

Parsing structured data and heuristics

The structured API definition can be parsed, e.g., using the proto descriptors API [6], with helper functions as necessary. Heuristics are used as part of the resolution procedure while deriving the dependency graph. Some examples include looking for specific naming patterns for remote procedure call (RPC) names and fields to determine how they should be resolved. An example heuristic is the use of the RPC name to determine if an RPC creates a resource. Another heuristic is to derive the name of a suspected cleanup RPC to identify an RPC used to clean up the corresponding created resource. These heuristics can be overridden to explicitly identify the RPC used to create or delete a resource.

Examples of attributes of the structured API definition used to derive the dependency graph include:

- The list of RPCs that comprise a service (e.g., as in [7]). These are used to identify the RPC request message (which is considered a dependency of an RPC graph node) and to identify RPCs (such as create-RPCs) that can be used to resolve field nodes (such as a field that references a resource).
- Required field annotation (e.g., as in [11]), used to determine if a field should be considered a dependency of its parent message (either the request message of an RPC or a message nested within the request message).
- Resource reference annotation (e.g., as in [9]), used to identify a field as one whose value should reference an RPC resource (usually resulting in an RPC node that is a dependency of the field with the annotation).

- HTTP annotations (e.g., as in [8]). Template variables are used to identify fields required in the request message.
- Resource annotation (e.g., as in [10]), used to identify the pattern of the name of a resource name when the dependent resource cannot be suitably created. Common examples are for project or location resources [12].
- Heuristics based on the names of RPCs or messages. For example, a special behavior can be invoked if a field has the name ‘name,’ or look for RPCs that start with ‘create’ or ‘delete.’

Dependency graph

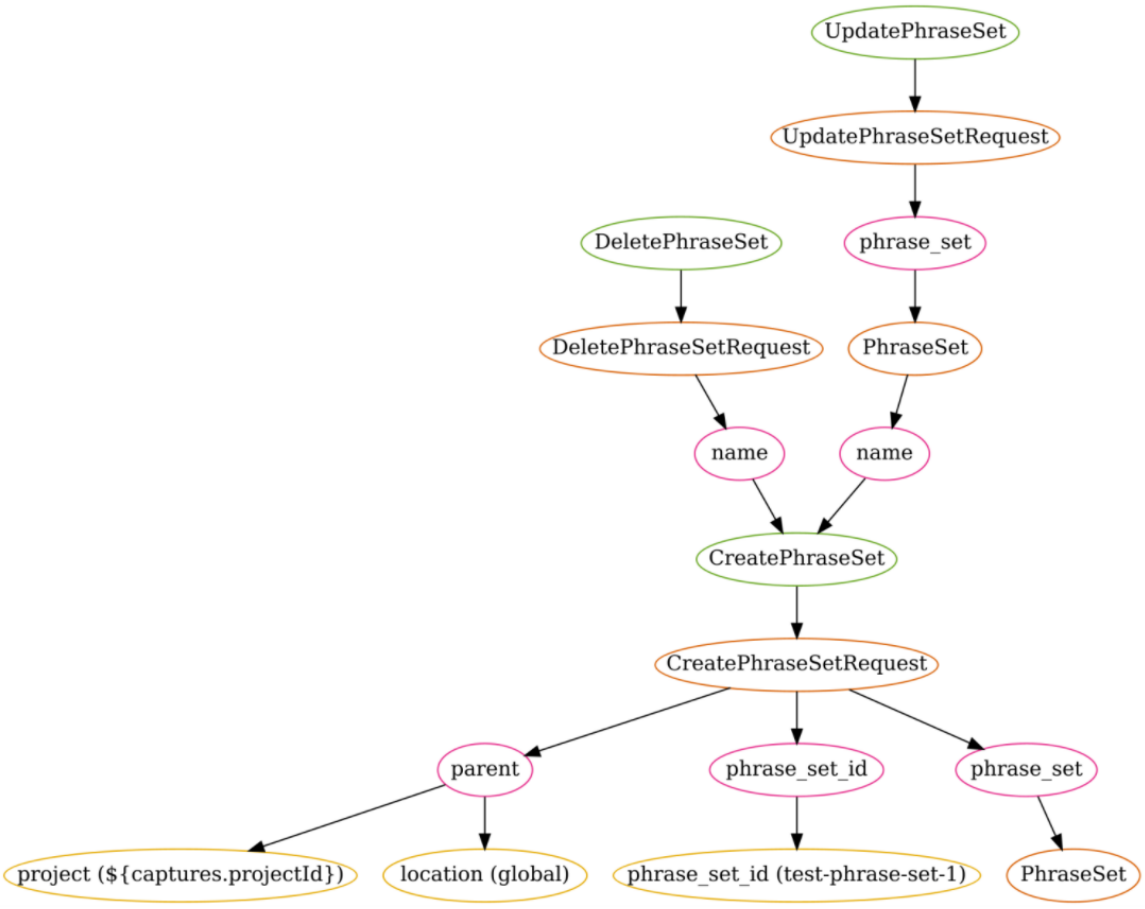

Fig. 2: An example dependency graph

Fig. 2 illustrates an example dependency graph, taken from an update-phrase-set RPC in a speech-to-text application. Each node in the graph has an ID, a set of parents, a set of children, and a payload corresponding to one of RPC (green), message (orange), field (pink), or static value (yellow). The payloads include metadata and values that can be used to resolve (derive child nodes) and to execute (obtain the value of) the nodes. For the static value nodes, the value in parentheses is the value of the constant (in this case set by override hints). An edge in the graph indicates that the child must be evaluated and have a value before the parent can be evaluated.

In this example graph, `UpdatePhraseSet` is the root RPC. There are also special branches of the graph that are used for deriving the clean phase of the test. In this case, `DeletePhraseSet` is a cleanup node. These nodes are collected during execution and executed (in reverse order from how they are encountered) after the root node in order to clean up all resources after test execution.

When testing an API request, the dependency graph is used to identify all dependencies necessary to execute the request. These dependencies are materialized in a form necessary to support the original request. A straightforward example of such materialization is to execute the dependent requests. The dependency graph can be used to write tests in any number of test-definition languages suitable for execution in various test execution engines.

Creation of the dependency graph

Fig. 3 illustrates an example process for the creation of a dependency graph. The dependency graph can be created by performing a breadth-first search, using the service definition and user-defined heuristics to identify subsequent nodes. A given root RPC is added to the graph as root node (302). It is also added to a queue of unresolved nodes (304). While there

are unresolved nodes in the queue (306), a node is taken off the queue and resolved to identify its dependent (child) nodes (308). The dependent nodes (and other unresolved nodes such as cleanup nodes) are added to the queue of unresolved nodes (310). Once the graph is derived, a topological sort can be used to visit each node and resolve its value.

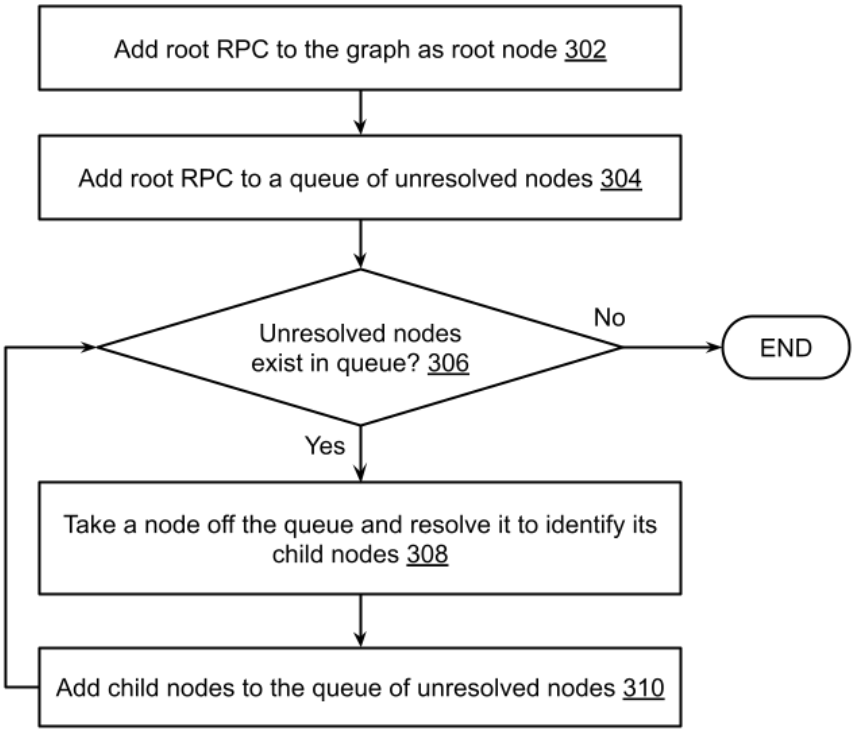

Fig. 3: Creating a dependency graph

Derivation of a node's dependent (child) nodes

The resolution of an unresolved node depends on its type, as illustrated by the following examples.

- RPC nodes have a single dependency, e.g., its request message.
- Message nodes are resolved by iterating through their fields and including as dependencies any fields that are deemed required.
- Field nodes use annotations to determine their value or provide a simple default value if a more suitable value cannot be determined.

- Static value nodes are resolved by definition.

For each of the above resolution types, the user can specify overrides to provide different values for each of the resolved nodes. For example, the user can specify the exact value of a message or a constant value of a field to prevent the resolution procedure from deriving the value in its default way. Overrides also exist for specifying if the user wants a field to be required without providing an explicit value.

Transforming a dependency graph into a sequence of executions

A dependency graph, once derived, is subjected to topological sorting to visit each of its nodes and resolve their value, as illustrated by the following examples.

- Static values are resolved by definition.
- Fields have various strategies for resolving their values including static value (takes the value of its constant child), RPC (evaluates the dependent RPC and takes its value from a field within the RPC response), and pattern (derives value from a template and dependent children).
- Messages are resolved by setting their fields according to each of their dependent field children.
- RPCs are resolved by collecting the necessary attributes to execute the RPC (e.g., http parameters, dependent request message) and, depending on the resolution method, executing the requests in the order that they are encountered.

In the `UpdatePhraseSet` example of Fig. 2, the four test phases can be:

- Setup: `CreatePhraseSet` (all descendent RPCs of the root node).
- RPC under test: `UpdatePhraseSet` (always the root node).
- Verification: any verifications can be interjected to assert on state or RPC under test.

- Cleanup: `DeletePhraseSet` (Cleanup RPCs executed in reverse order from where their corresponding Create RPCs are discovered in the graph).

A procedure that correctly exercises an API request, as described herein, provides a number of benefits, including:

- Horizontal testing can be implemented by tweaking the environment and asserting on the expected result.
- Requests can be executed to provide simple, happy-path integration testing.
- The provided RPC arguments can be tweaked to perform sad-path testing.
- RPC arguments can be programmatically tweaked to perform fuzz testing.
- RPCs can be repeatedly executed to perform load testing.

The described techniques enable test-writers to be more efficient by reducing the manual effort to determine the required setup and cleanup of their tests. The techniques are applicable to APIs that are defined in a structured manner, and whose validity is being tested in an automated fashion.

CONCLUSION

This disclosure describes techniques to automatically assemble a dependency graph and build the setup and cleanup necessary to test an API request. An API service definition is analyzed to enumerate all remote procedure calls (RPCs) and infer the fields and RPCs required to successfully execute a request. Structured API definitions and heuristics are used to determine whether an API request has a dependency on other API requests. These dependencies are assembled into a dependency graph, the nodes of which are executed after being topologically sorted.

REFERENCES

[1] http://xunitpatterns.com/Four%20Phase%20Test.html accessed Sep. 22, 2021.

[2] https://github.com/thoughtbot/factory_bot accessed Sep. 22, 2021.

[3] https://github.com/FactoryBoy/factory_boy accessed Sep. 22, 2021.

[4] https://github.com/googleapis/googleapis/blob/master/google/api/service.proto accessed Sep. 22, 2021.

[5] https://github.com/googleapis/googleapis/blob/master/google/api/resource.proto accessed Sep. 22, 2021.

[6] https://googleapis.dev/python/protobuf/latest/google/protobuf/descriptor.html accessed Sep. 22, 2021.

[7] https://developers.google.com/protocol-buffers/docs/proto#services) accessed Sep. 22, 2021.

[8] https://cloud.google.com/endpoints/docs/grpc/transcoding#map_a_get_method) accessed Sep. 22, 2021.

[9] https://github.com/googleapis/googleapis/blob/5eeefdc0e9e6d4789ee99c662c6aa1aa91dfd012/google/api/resource.proto#L28-L32 accessed Sep. 22, 2021.

[10] https://github.com/googleapis/googleapis/blob/5eeefdc0e9e6d4789ee99c662c6aa1aa91dfd012/google/api/resource.proto#L40-L66 accessed Sep. 22, 2021.

[11] https://github.com/googleapis/googleapis/blob/5eeefdc0e9e6d4789ee99c662c6aa1aa91dfd012/google/api/field_behavior.proto#L58-L61 accessed Sep. 22, 2021.

[12]

https://github.com/googleapis/googleapis/blob/master/google/cloud/common_resources.proto

accessed Sep. 22, 2021.