Service Builder Guide

Introduction

Services encapsulate behavior associated with a specific functional aspect of an application. Services expose that behavior via their interface, which consists of a list of procedures and/or Event Types which are available to consumers of the Service. Services may be shared with other applications via the Vantiq Catalog which allows their interface to be accessed remotely.

In this chapter we will cover how to use the service builder to define Vantiq services. The service builder organizes the definition of a service into 3 main categories (each with their own tab): Interface, Implementation, and Testing which we will cover in the following sections.

Once we've covered the basics of how to define a service, we'll explore more advanced topics such as service state management and creating a GenAI Agent.

Related Documents

To walk through building your first Service, follow the Introductory Tutorial. After completing the Introductory Tutorial, explore testing services in the Testing the Introductory Tutorial. You can also learn about Service State Management and building GenAI Agents.

Interface

The Service interface describes the externally facing behavior of the service. Its goal is to define *what* the service does, not *how* it does it (which is the domain of the service's implementation). The interface of a service consists of Inbound and Outbound Event Types, which describe the service's asynchronous behavior, and Procedures, which describe its synchronous behavior.

Understanding the interface of a service is useful to help lay out the design of the service prior to its implementation. In part because understanding the requirements of a service is the first step in defining its interface. It is also useful for other Vantiq users who need to integrate this Service into their own projects. For example, it is a service's interface which is published to the Catalog for reuse.

> The service builder supports a more "ad-hoc" development style where the service interface is constructed as its implementation is being defined. As a result, anytime a new public event type or procedure is defined in the implementation tab, the builder will automatically update the interface to include it.

Event Types

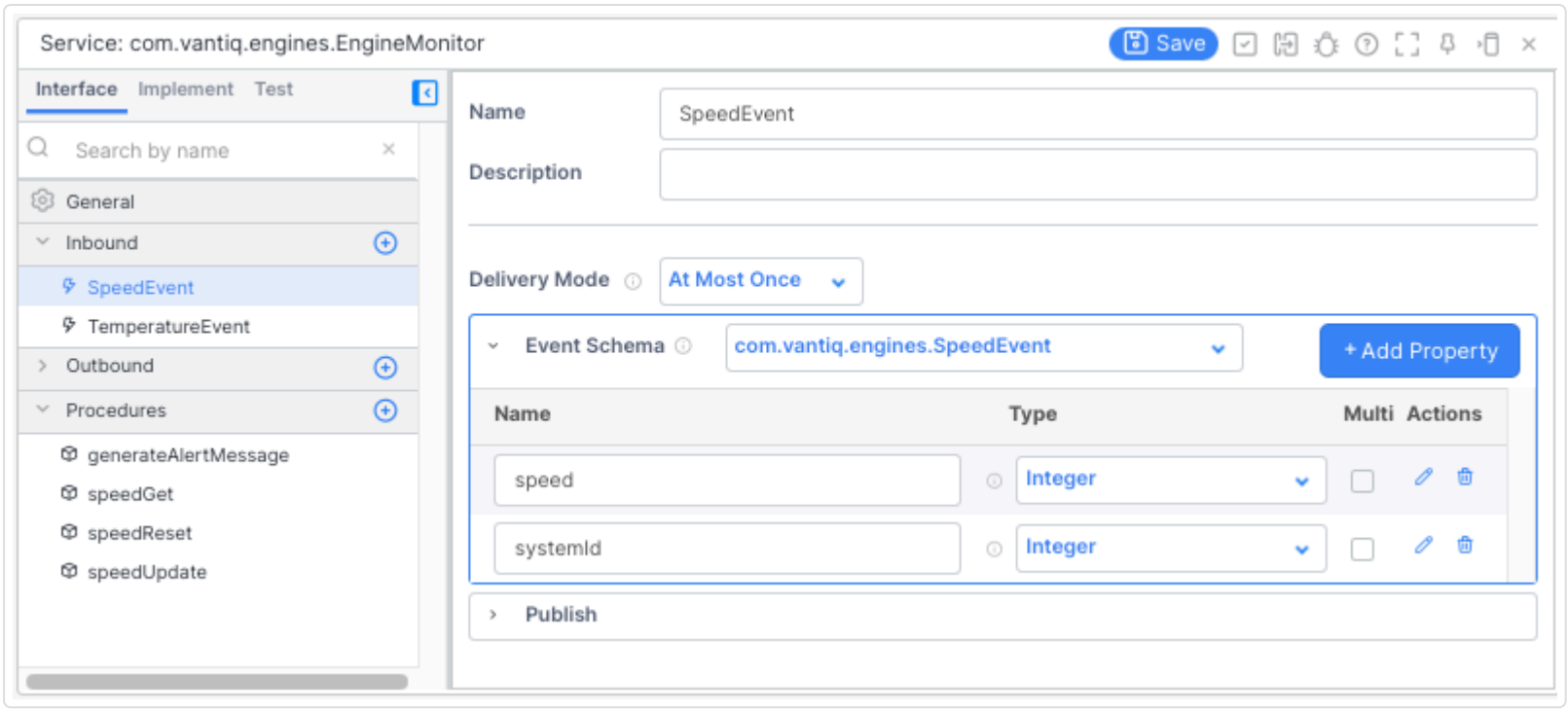
Service Event Types are used to describe the production and consumption of events by the owning Service. This provides consumers with an event-driven interaction for the functionality encapsulated by the Service. This makes it much more natural to leverage the Service as part of a larger Event Driven Architecture (EDA). All event types defined in the interface tab are "public" and thus are part of the service's interface. Event types are divided into two categories, based on the "direction" in which the events flow.

Inbound

Inbound event types represent events that flow from the consumer to the service for processing. Service consumers can send events using either the VAIL `PUBLISH` statement or a PublishToService task in a visual event handler (typically of another service). They have the following properties:

- **Name** – the name of the Event Type. Must be a legal identifier and unique for a given service.
- **Description** – the description for the Event Type.
- **Delivery Mode** – indicates whether or not the received events will be processed "reliably", if possible (see the Reliable Messaging Guide for more details). The possible values are:
  - *At Most Once* – the event is not reliable and might not be processed due to runtime errors.
  - *At Least Once* – the event will be processed reliably by the service. Note that if the sender of the event is not reliable, the event may still be lost if a failure occurs prior to its receipt by the service.
- **Event Schema** – a reference to a Type which describes the structure of the data associated with events of this type.

The service builder provides a way to directly "publish" the inbound event type to assist in development. An example of an inbound event type (taken from the tutorial) is `SpeedEvent` which carries data on an engine's current speed for processing by the service.

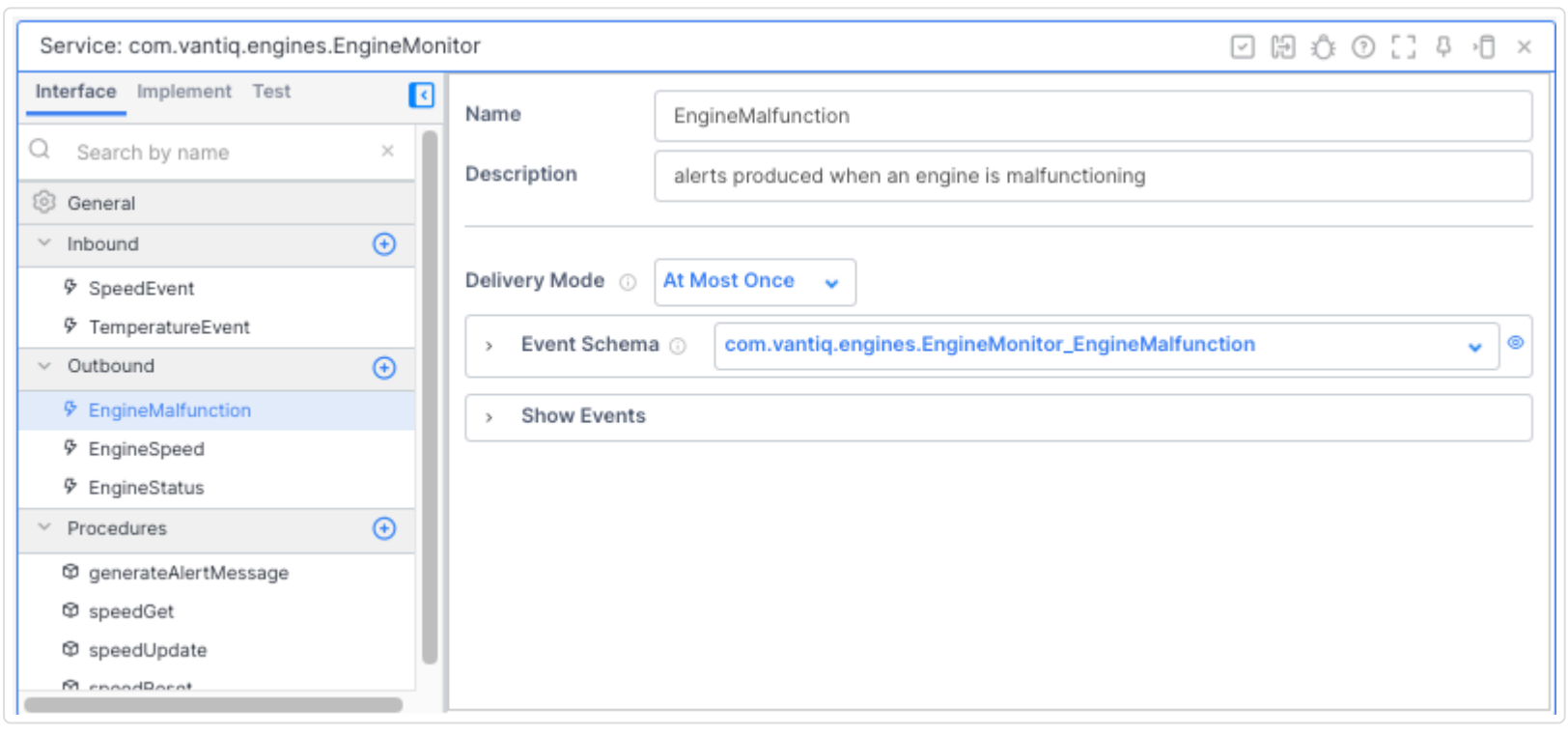


Outbound

Outbound event types represent events that flow from the service to the service consumer(s) (in an anonymous fashion). Service consumers can subscribe for these events by referencing them in the `WHEN` clause of a rule or as the triggering condition of a Visual Event Handler. They have the following properties:

- **Name** – the name of the Event Type. Must be a legal identifier and unique for a given service.
- **Description** – the description for the Event Type.
- **Delivery Mode** – indicates whether or not the events will be sent reliably (see the Reliable Messaging Guide for more details). The possible values are:
  - *At Most Once* – the event is not reliable and might not be delivered due to runtime errors.
  - *At Least Once* – the event will be sent reliably, service consumers are expected to acknowledge receipt of the event at the appropriate time.

The service builder provides a way to view events sent via the outbound event type to assist in development. An example of an outbound event type (taken from the tutorial) is `EngineMalfunction` which is generated when the serivce detects specific unsafe engine states. The event data contains details about the situation that has been detected.



Procedures

A service's procedures represent its synchronous behavior to the outside world. In the *Interface* tab, the focus is on the definition of each procedure's signature. This describes the parameters necessary to execute the Procedure and the Procedure's return type. This signature is used to generate a skeleton of the procedure's implementation. As development proceeds, the service builder ensures that the signature in the interface matches the one expressed by the implementation.

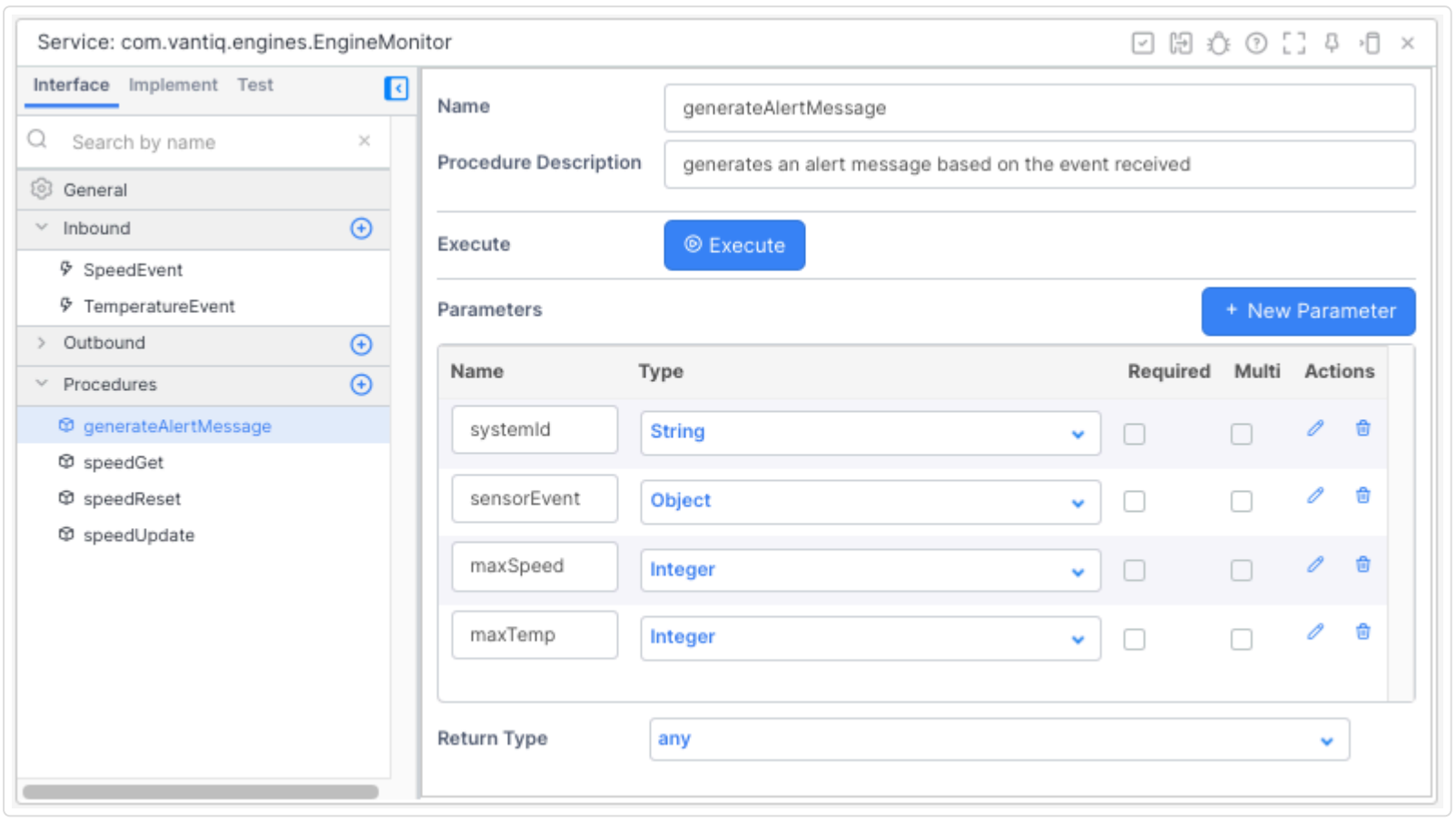
The Procedure Signatures ihave the following properties:

- `name` – the name of the Procedure. Must be a legal identifier and unique for a given Service.
- `description` – the description for the Procedure
- `parameters` – An array of parameter descriptors. Each parameter descriptor may define:
  - *name*: the name of the parameter. Must be a legal identifier and unique to the Procedure
  - *description*: a description of the parameter

  - *type*: The Type (either built-in or User Defined) that describes the schema for the value provided
  - *multi*: a boolean declaring that the value being passed must be an array
  - *required*: declares that the parameter is required and that any caller must supply a value (though that value can be null)
  - *default*: declares that if a value is not supplied for the parameter, then the specified value should be used.
- `returnType` – The Type (either built-in or User Defined) that describes the schema for the value returned by this Procedure

Although optional, it is highly encouraged to declare a type for all parameters and the return type. When present the system will validate that any values provided for the parameter are legal values of the declared type and will automatically perform any supported type conversions. If provided, the system will use the return type to validate and convert any value produced by the procedure. Since VAIL is dynamically typed, these validations/conversions occur at runtime. In addition, the Vantiq system will confirm that any requested operations can be performed and produce an error if they cannot (also known as “duck” typing).

For more information on defining Procedures in VAIL, please refer to the VAIL Reference Guide.



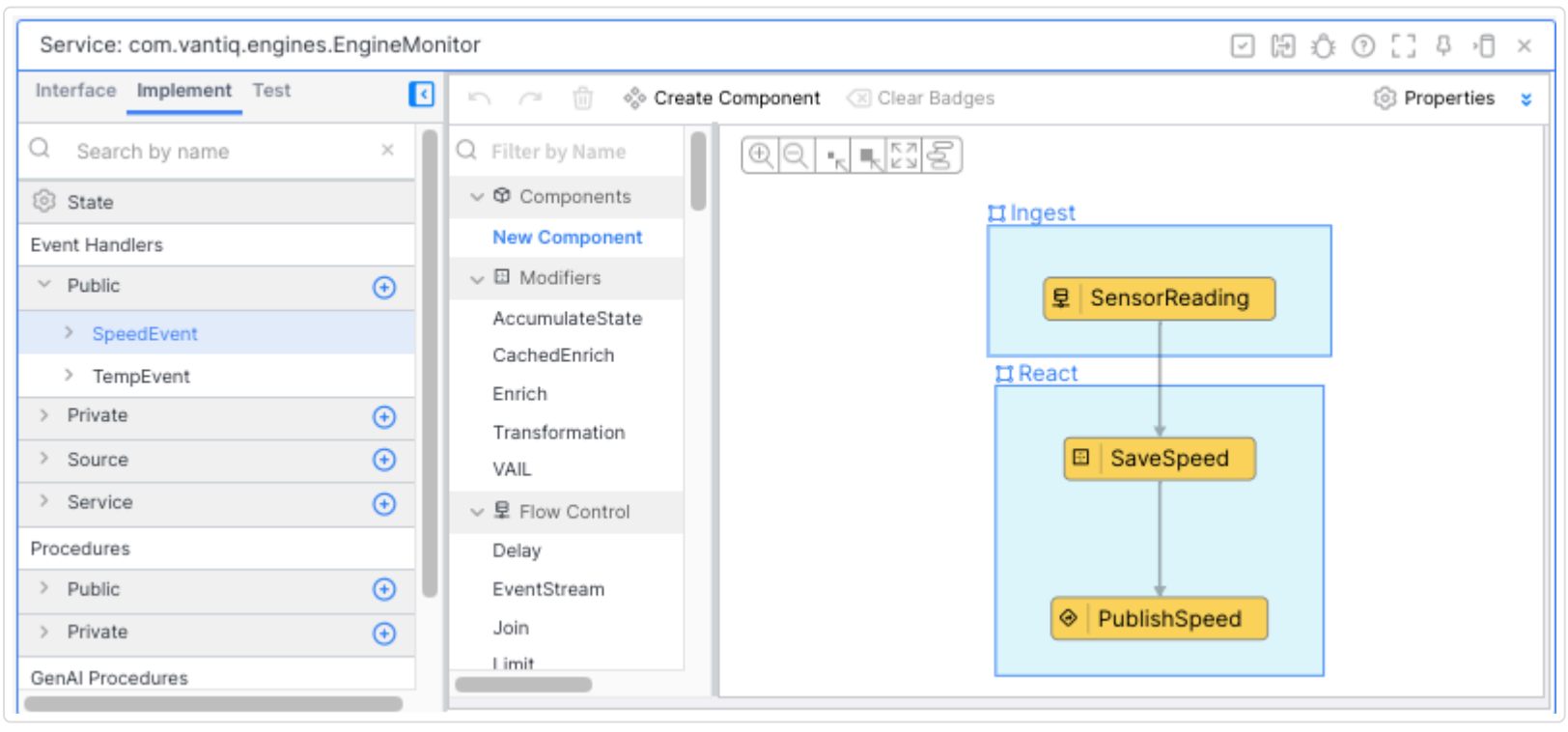
Implementation

The Service’s implementation describes (in detail) exactly *how* the service provides its public interface. The implementation also includes internal details which are not part of its interface (these are known as “private” artifacts). As with the interface these are divided into categories based on the “type” of functionality being defined, event handlers, procedures, and GenAI procedures.

Event Handlers

Public Event Handlers

Each Inbound Event Type in the service’s interface is implemented via the definition of a public “event handler”. These are either a Visual Event Handler (aka App) or a Rule (aka VAIL Event Handler). Here is an example of a visual event handler:



By default, each event handler will receive events from a single inbound event type (as shown above). However, when defining a visual event handler, it is possible to add additional EventStream tasks, each of which is bound to a different inbound event type. This allows the use of Joins and Merges within Visual Event Handlers and allows a single handler to process multiple Event Types. VAIL Event Handlers are limited to implementing a single Event Type.

Event handlers are considered to be part of the service's implementation and therefore are granted access to the service's private procedures. When invoking a single partition, private procedure the event handler code must explicitly supply a partitioning key as outlined in Partitioned Execution.

It is worth noting that Visual Event Handlers will automatically augment the Service definition given the implementation of that Event Handler. In particular, Visual Event Handlers generate outbound event types, state management procedures, and state Type definitions when required. For more information on Visual Event Handlers and how they interact with services, refer to the Visual Event Handler Builder Guide.

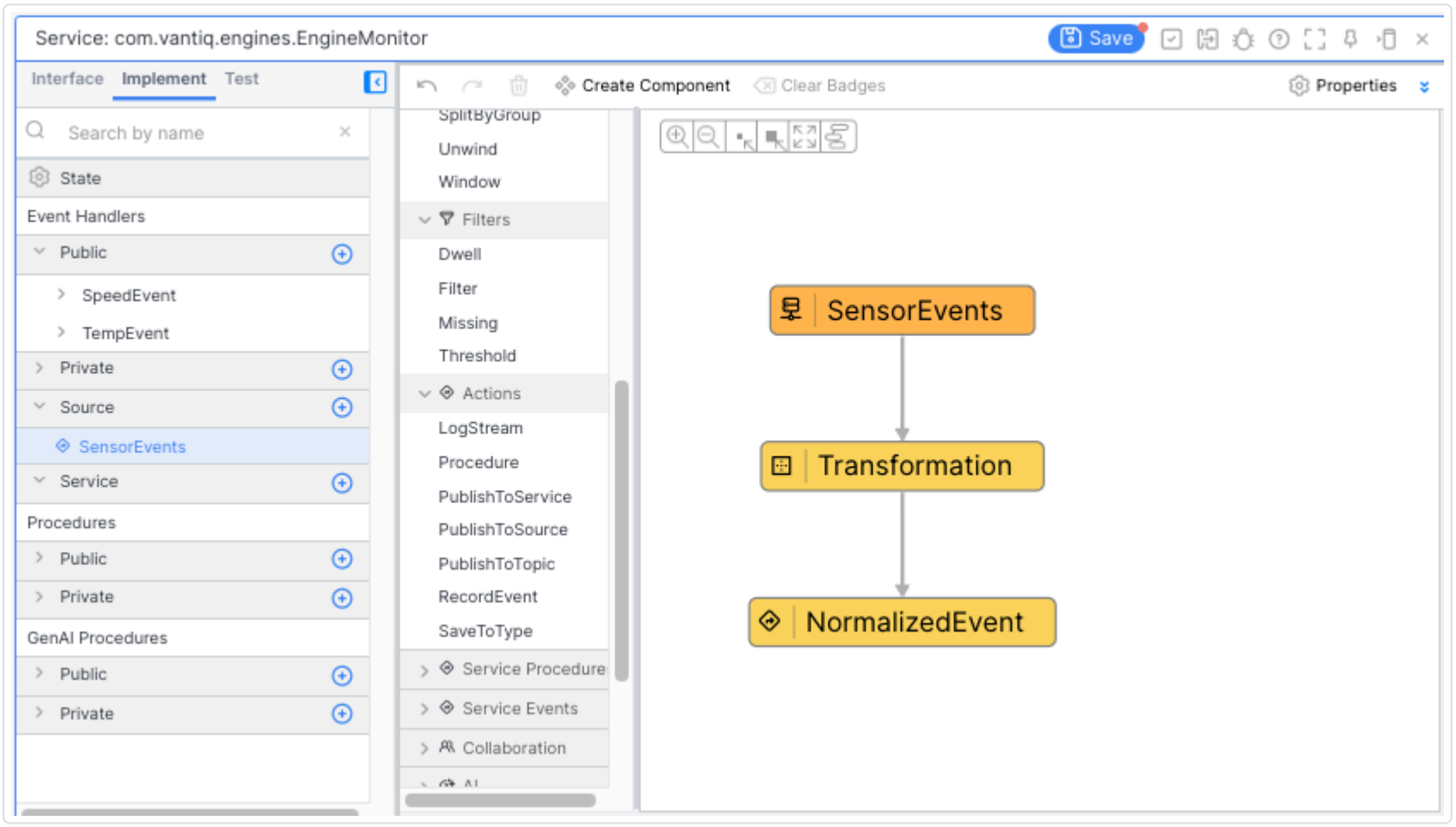
Private Event Handlers

In addition to the public inbound event types defined in a service's interface, a service may also define one or more private event handlers. This allows services to leverage event processing as part of its private implementation. As the name suggests, the inbound event type for a private event handler is visible only to the service itself. Otherwise, it can be used like any other event type.

Source and Service Event Handlers

Source and Service Event Handlers allow a service to process events received from a **Source** or **Service**. They are part of the service's implementation and are not visible to Service consumers. A common pattern for Source or Service Event Handlers is:

- Ingesting external events;
- Performing some processing or transformation;
- Exposing the result as an Outbound Event Type for other Services to trigger off of



Source and Service Event Handlers allow Services to act as a wrapper around Sources or other external events.

Event Routing

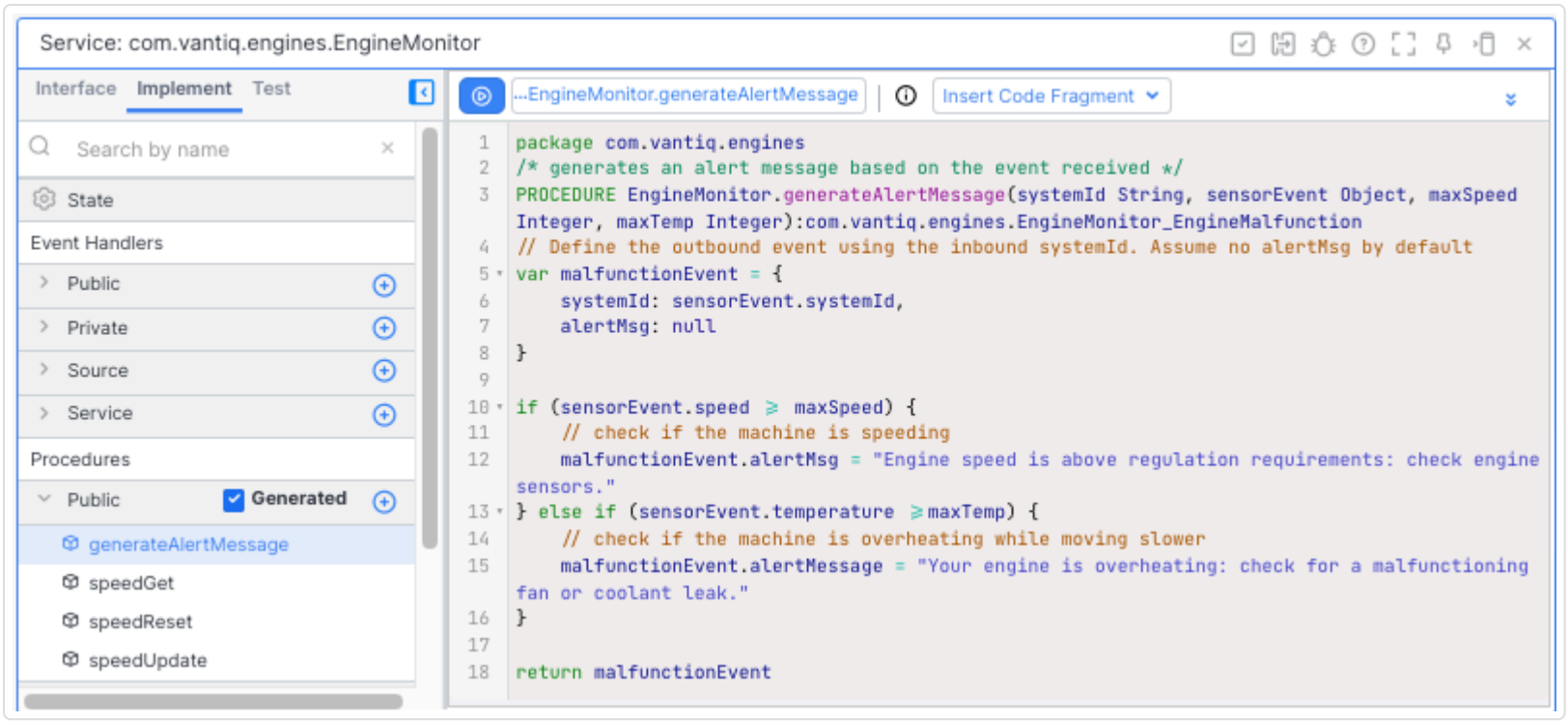
As mentioned in the section above, Source and Service Event Handlers *must* be triggered by the Event Type they implement. However, in many cases, it is useful for an Outbound Event Type of one Service to trigger processing on an Inbound Event Type of another. *Service Routes*, defined in the Design Modeler, are pathways from outbound event types to inbound event types. When a route exists between two event types, the system knows to automatically send events from the publishing Outbound Event Type to the ingesting Inbound Event Type. Because routes are defined dynamically in the Design Modeler, neither Service needs to know about the existence of the other. This also means that adding, updating, and removing routes is done without having to update any of the underlying Service definitions. This is particularly useful when composing Services installed via the Vantiq Catalog.

Procedures

Procedures provide an *imperative*, request/response approach to interacting with a service. The Procedure Signatures defined in the service's interface must have an associated *public* procedure for their implementation. The service can also define any number of *private* procedures for its own, internal use.

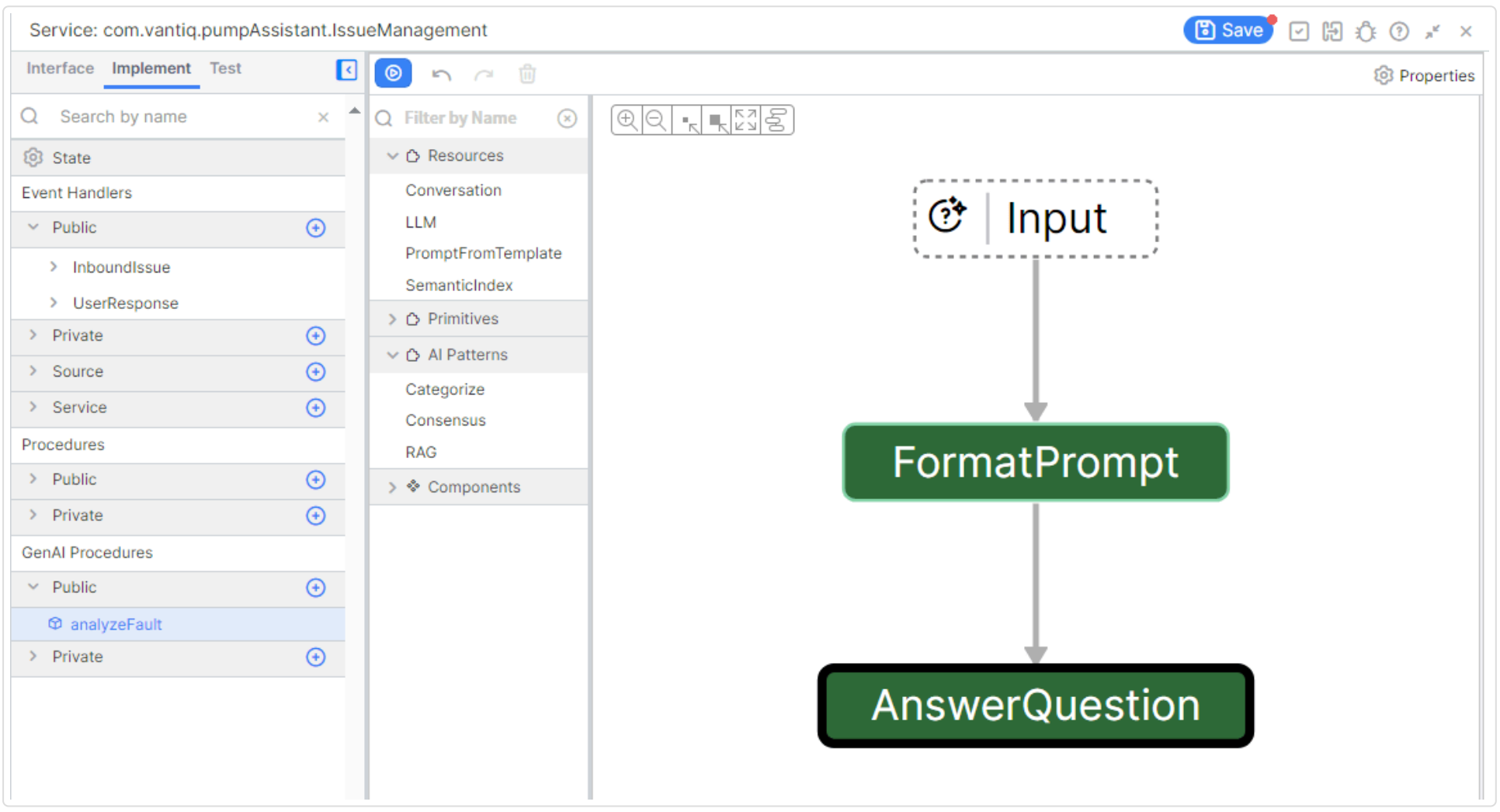
VAIL Procedures

One way to implement a service procedure is by using VAIL to describe its behavior. This provides a very general purpose approach and can be used to define procedures that implement effectively any algorithm. See the Service Procedures documentation for the exact VAIL syntax for defining Service Procedures. Here we see a public VAIL procedure being defined:



GenAI Procedures

Another way to implement service procedures is by creating a GenAI Procedure. GenAI Procedures are implemented using the GenAI Builder to describe a GenAI Flow. GenAI Flows are focused specifically on leveraging Generative AI (GenAI) technologies such as Large Language Models (LLMs) and Semantic Indexes to perform a variety of data analysis tasks. Here we see a public GenAI Procedure being defined:



GenAI Procedures are always defined with two parameters:

- input (`Any`) – the data to be processed by the GenAI Procedure.
- config (`Object`) – the runtime configuration values to be applied when executing the GenAI Procedure.

The return type of a GenAI procedure defaults to `Any`, but can be set to any legal type.

Service Tests

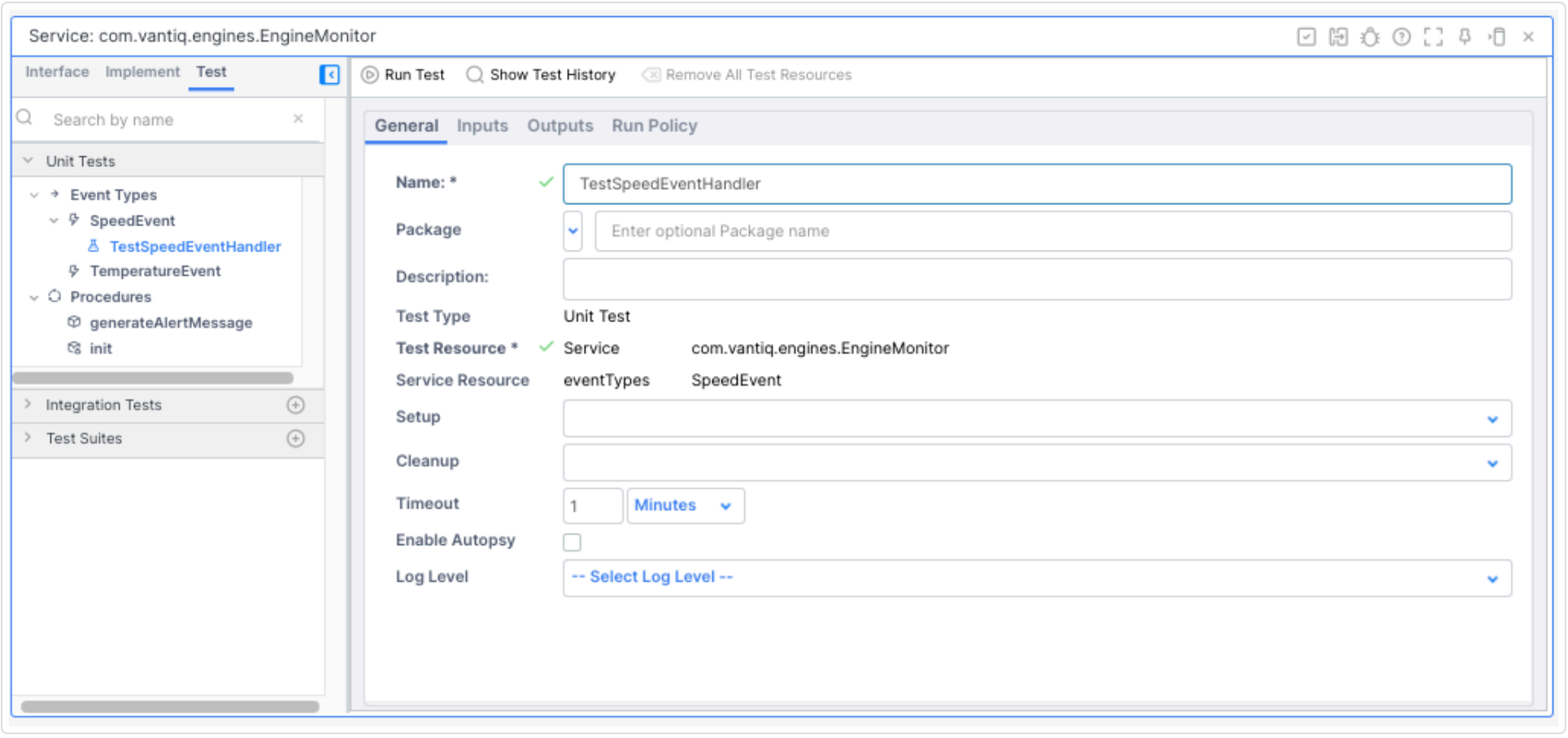
Service Tests are extensions of the regular Testing feature. However, Service Tests are specialized to pinpoint the functionality of individual Services and individual units within a Service.

Unit Tests

Service Unit Tests test the functionality of a single Service Procedure of Service Event Handler. Service Unit Tests will specify the test *resource* as the service (`"/services/<serviceName>"`) and will specify the *unitReference* based on the unit being tested. The *unitReference* is a property of the test in the format *{type: <unit type>, name: <unit name>}* where type is either *eventTypes* or *procedures* and the unit name is the name of the Event Type or Procedure being tested. Service Unit Tests are the only mechanism for testing Private Service Procedures.

Service Unit Tests require that the only input to the test is the Unit being tested. For inbound event types, this means the only test input is publishing to that inbound event type. For Procedures, this means the only test input is executing that Procedure. Service Unit Tests require that the only test outputs are Outbound Service Event Types, Service Procedures, or expected errors. Service Procedures may be used to validate the value of a particular state property. Any Procedure outputs will be executed at the end of the test after all other outputs have been received or the test timeout has expired.

For more information on Unit Tests, refer to the Testing Guide.



Integration Tests

Service Integration Tests will specify the test *resource* as the service (`"/services/<serviceName>"`). Service Integration Tests test the functionality of the Service as a whole. Service Integration Tests define a Service as the test resource rather than a Project. At runtime, a Service Integration Test will generate a Project that contains only the Service and its implementing resources. That project will be deployed to a Testing namespace so that it can be tested in isolation from the rest of the Project.

Service Integration Tests require the test to interact with the Service using only the Service interface. This will accurately simulate how consumers of the Service will use the Service. This means that the only valid test inputs are publishing to Inbound Service Event Types and executing public Service Procedures. The only valid test outputs are expecting events on Outbound Service Event Types, Service Procedures, or expected errors. Service Procedures may be used to validate the value of a particular state property. Any Procedure outputs will be executed at the end of the test after all other outputs have been received or the test timeout has expired.

For more information on Integration Tests, refer to the Testing Guide.