Service State Management

Introduction

An important consideration for any service is whether or how to manage the state needed to perform its function. In this document we will cover the facilities provided by the Vantiq platform to assist in service state management, along with some examples of their use.

Related Documents

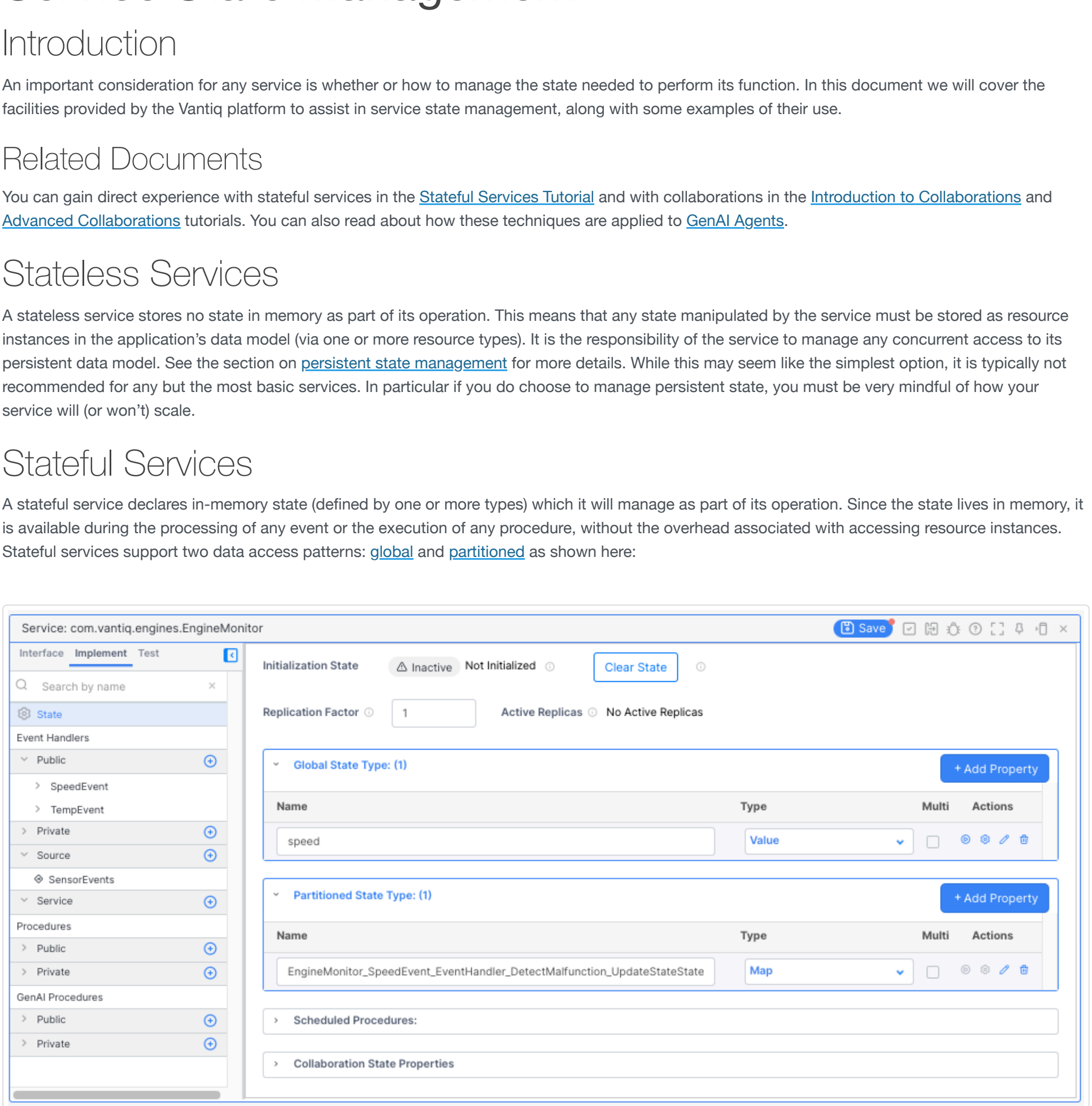
You can gain direct experience with stateful services in the Stateful Services Tutorial and with collaborations in the Introduction to Collaborations and Advanced Collaborations tutorials. You can also read about how these techniques are applied to GenAI Agents.

Stateless Services

A stateless service stores no state in memory as part of its operation. This means that any state manipulated by the service must be stored as resource instances in the application's data model (via one or more resource types). It is the responsibility of the service to manage any concurrent access to its persistent data model. See the section on persistent state management for more details. While this may seem like the simplest option, it is typically not recommended for any but the most basic services. In particular if you do choose to manage persistent state, you must be very mindful of how your service will (or won't) scale.

Stateful Services

A stateful service declares in-memory state (defined by one or more types) which it will manage as part of its operation. Since the state lives in memory, it is available during the processing of any event or the execution of any procedure, without the overhead associated with accessing resource instances. Stateful services support two data access patterns: global and partitioned as shown here:



> It is not possible to declare a stateful service using VAIL. Doing so requires the use of the Service Builder as shown above.

A given stateful service may use either or both of these approaches as necessary. It is the responsibility of the service to manage any concurrent access to this in-memory state. See the section on in-memory state management for more details.

Global State

If a service has global state, then there exists at runtime a single copy of this state associated with all instances of the service. While the exact location of this state is undefined, the system ensures that there is only one copy. The structure of the global state is defined by the declaration of a global state type. The properties of this type are available as implicitly defined variables.

These state variables are only accessible to procedures which are declared as "global" (via the procedure declaration). If a service has only global state, then it is known as a global service and all its procedures are implicitly declared as global.

Partitioned State

If a service has partitioned state, then the data will be spread across multiple partitions. The goal of partitioning the data is to allow for concurrent processing of that state. The model here is the same one used by distributed databases. By partitioning a service's state, we can divide it up between the members of a Vantiq cluster and allow each partition to be accessed independently. As with global state, the system ensures that procedures that need to

access the partitioned state are routed appropriately so that they run on the instance(s) associated with the targeted partition(s). Unlike global state, the partitioned state supports two access patterns:

- Single partition – each invocation of a single partition procedure executes on exactly one partition. Which partition is determined by the supplied "partitioning key" which by convention is the procedure's initial parameter. This key is also typically used to access specific values within the partition (see In-Memory State Management for more details).
- Multi partition – each invocation of a multi partition procedure executes on *all* of the partitions (potentially simultaneously). These procedures are typically used to perform various housekeeping tasks, such as initialization or transferring data from in-memory state) to persistent state. When executed directly they return a sequence of results that can be processed incrementally via a processing block.

The structure of the partitioned state is defined by the declaration of a partitioned state type. The properties of this type are available as implicitly defined variables. These state variables are only accessible to procedures which are declared as single or multi partition (via the procedure declaration). If a service has only partitioned state, then all of its procedures are implicitly declared as single partition.

Stateless Procedures

Sometimes it is useful to define a procedure that does not require state access as part of an otherwise stateful service. Doing this can simplify its usage and enable the creation of "utility" procedures that can be used independent of the service's state. This is done by using the *STATELESS* procedure modifier when defining a procedure.

State Initialization

All the properties of the declared state type(s) (both global and partitioned) will be implicitly assigned a value based on their declared type:

- `Object` – an empty `Object` instance.
- User Defined Types – an empty `Object` instance.
- `Map` – an instance created using Concurrent.Map().
- `Value` – an instance created using Concurrent.Value().
- `Integer` – the value `0`.
- `Real` – the value `0.0`.
- All other VAIL Scalar – the value `null`.

To perform custom initialization beyond simple creation of the properties, stateful services can have initializers for both the global and partitioned state. These are "well-known" procedures with the following characteristics:

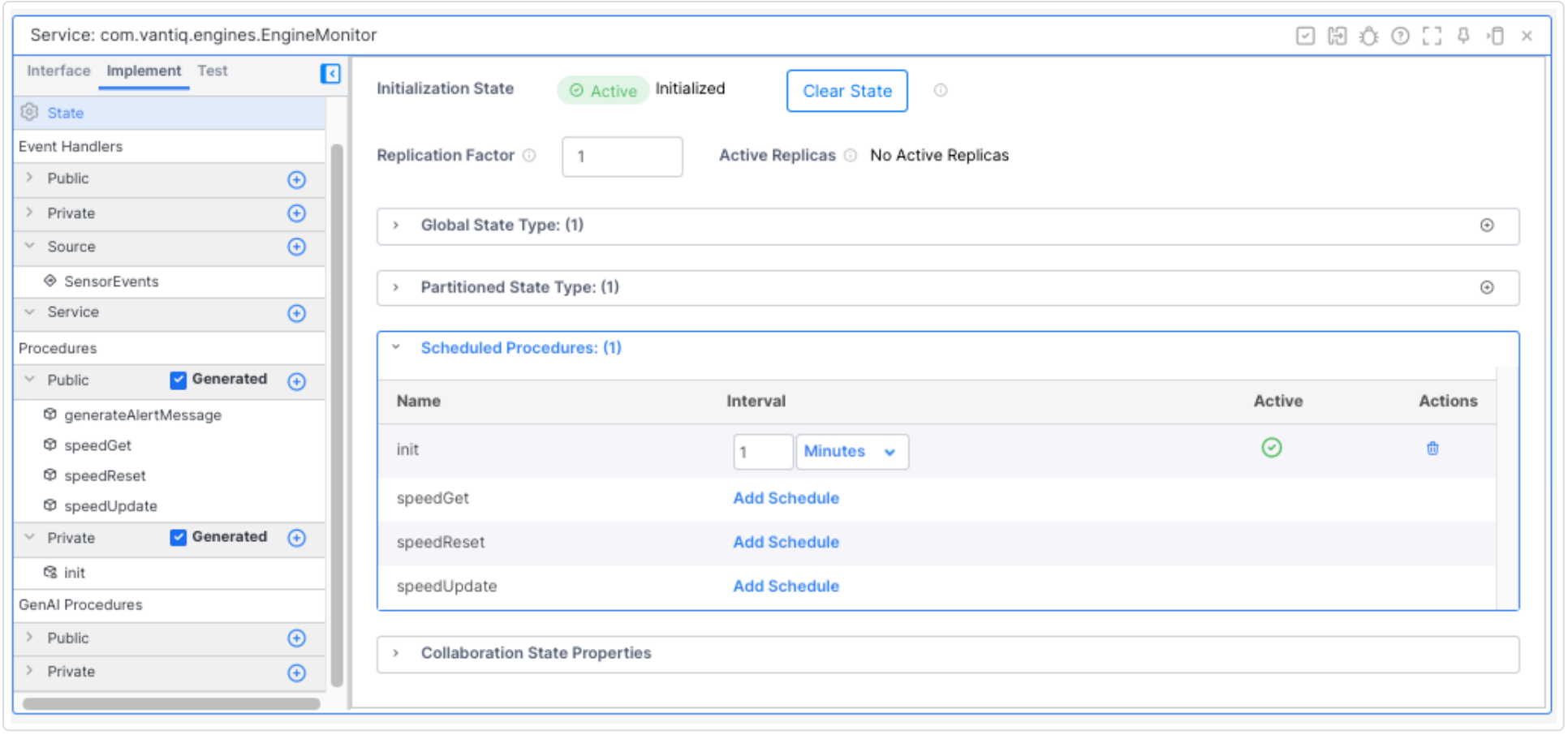
- They have fixed names – `initializeGlobalState` and `initializePartitionedState`. Each one is only permitted if the service has the state being initialized.
- They must be `PRIVATE` procedures.
- They must have no parameters.
- The initializers do not require an explicit state access modifier. If one is provided, then the global initializer must be marked as "global" and the partitioned initializer must be marked as "multi partition".
- They will be invoked implicitly by the system and may *not* be invoked explicitly.

The system invokes the initializers automatically any time it must create or recreate the associated state. For example, if the partitioned state is invalidated due to a change to the partitioned state type, then the system will recreate it and run the partitioned initializer.

Scheduled Procedures

Stateful services may declare *scheduled* procedures to help them perform periodic, background tasks, such as persisting state in resource instances or resetting statistics being computed. This is done by specifying a scheduling interval as part of the procedure declaration. The minimum supported interval is `1 minute` and the minimum increment is `1 minute`. Execution of scheduled procedures is aligned, meaning that all invocations start at the top of the hour and offset from there (so an interval of 10 minutes means 0, 10, 20, 30,… and an interval of 15 minutes means 0, 15, 30,…).

Scheduled procedures are not subject to execution time limits. However, the *next* invocation of the procedure will not occur until the *current* invocation completes. For example, a procedure scheduled to run every 10 minutes which takes 11 minutes to complete will run at 0, 20, 40, 0, etc… This is to ensure that there are no overlapping executions. Note that this only applies when a procedure is called by the scheduler. If scheduled procedures are called from any other context, they are subject to the standard controls.



Service Replication

Stateful services can be “replicated” in order to provide fault tolerance for the state that they manage. This is done by replicating writes to more than one member of the cluster. Then, if a member fails there will be an identical copy of its data available via the remaining members. Replication occurs transparently and has no impact on read semantics. A given write operation is not considered complete until it has been successfully replicated to all targets. Readers of the data see the effects of writes that completed prior to the read request.

The number of “copies” of the data is determined by the service’s “replication factor”. A replication factor (RF) of 1 (the default) means that there is only one instance of the data (the primary instance) and so the service is not replicated. A value of 2 or more indicates that there are multiple copies of the data, stored on multiple cluster members. When a failure occurs, the system will automatically switch to using one of the secondary copies of the data, until the failure has been resolved. If another failure occurs before this happens, then the system will attempt to switch again, assuming another copy of the data exists. In general, the system can recover from (RF-1) simultaneous failures without loss of data.

> The effective RF of a service is bounded by the size of the cluster in which it is running. For example, in a cluster with 3 members, the effective RF of any replicated service cannot exceed 3 (even if set higher).
>
> There is rarely a reason to set the RF to a value greater than `2`. An RF of `2` allows the service state to survive 99.95% of all cluster failures. Critically it ensures that the state will survive a rolling restart of the cluster (such as when a new version of the Vantiq platform is installed). Beyond this point the additional overhead is typically not worth the resulting benefit. For example, increasing RF from 2 to 3 doubles the possibility that a failure will occur during replication and limits throughput to that of the slowest replica.

The system attempts to make the use of replication as transparent as possible. However, there are a few “best practices” to keep in mind:

- Only state properties typed as `Map` or `Value` will be replicated. Properties with other types may exist, but their values will be lost in the event of a failure.
- The results of “direct assignment” to a state property cannot be replicated. Therefore this should be avoided outside of the state initializers. For example, to remove all the keys from a `Map` instance you should use: `myMap.clear()` and not `myMap = Concurrent.Map()`.

Collaborations

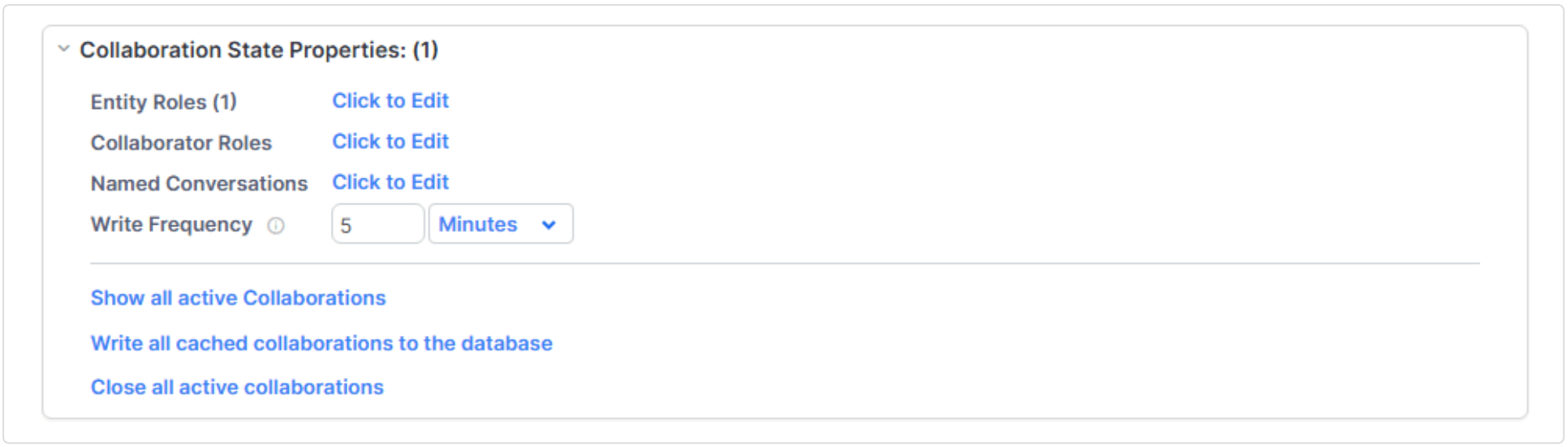
While it is certainly possible for a Vantiq service to simply record data (from either events or procedure invocations) in its state in order to present it to the user, it is much more powerful when services use the data to recognize important situations and take *action* as a result. These actions may involve GenAI Agents (or other services) exchanging information with each other, with the user, or coordinating the actions of multiple users. Some example include:

- A machine failure may be detected by the service but repair activities require collaboration with the technical specialist who can physically repair the machine. The service can provide context dependent suggestions and recommendations as well as timely information on the status of the machine, but the expertise of the technical specialist is required to minimize repair time and cost.
- In an application monitoring guests at a Casino in order to “comp” the best customers and increase their loyalty, the floor manager is a key collaborator. The floor manager can see the guests and easily assess whether the suggested comps will be well received by the guest. In cases where the floor manager recognizes unique issues, they can override the application generated offers to optimize the experience of the guest.
- An Agent monitoring the vital statistics of a patient may make recommendations regarding that patient’s care, but it must collaborate with the doctor and agents specializing in other domains (such as drug interactions) in order to ensure that those recommendations will be effective.

In all of these cases, the collaborative activity involves information about some application specific “entity” and may span relatively long time periods. Vantiq offers extensive capabilities simplifying the development and operation of collaborations between the services of a Real-time Business Application and its users, as well as collaborations among the application’s users.

Collaboration State Properties

Once you have decided that a service will be managing collaborations, the first step is to define the service's *Collaboration State Properties*. These are found in the *State* section of the *Implements* tab in the service builder as shown here:



Entity and Collaborator Roles

Roles are used to identify the responsibility of each participant in a collaboration. There are 2 distinct roles available:

- **Entity Roles** – bound to objects that represent the domain model entities impacted by each activity. By convention, Entity Roles refer to the entities impacted by the execution of the collaboration. At runtime, objects are bound to the Entity Roles identifying the actual entity instances (aka entities) on which the collaborations are operating.

- **Collaborator Roles** – bound to actual users that are tasked with executing each activity. Each collaborator role implicitly identifies a set of responsibilities for the user assigned to that role in the form of the activities in which the role participates.

Both entity and collaborator roles are defined by the service which is managing the collaborations. Once defined, a given role may be used by any of the service's event handlers.

Establishing Collaborations using Entities

When doing work associated with a collaboration, one of the first things a service must do is "establish" *which* collaboration instance it should be using. One common way to do this is by starting with the id of an entity and using that as a "foreign key" to find the correct collaboration instance (or possibly create a new one). This can be done programmatically using the entity role procedures or by using the EstablishCollaboration activity pattern in a visual event handler. When using an entity in this way, only collaboration instances created by the current service will be considered. Any other collaborations will be ignored. More formally we say that each service "owns" the collaborations that it manages and that entity roles are "scoped" to the defining service.

Named Conversations

Agents participating in a collaboration must be able to manage their conversation state as part of the collaboration instances. For this reason, each collaboration instance has a default conversation identified by its *collaborationId*. If a service needs to engage in more than one conversation, the user may declare additional conversation names. These can then be used in the `ActiveCollabsStartConversation` procedure. In a visual event handler the SubmitPrompt, AnswerQuestion and GenAI Flow activity patterns can be configured to participate in these conversations. They can also be used when invoking a GenAI Procedure.

The conversation state lifecycle corresponds to the collaboration instance in which it was created. When a new collaboration opens, the system starts referenced conversations with the ConversationMemory service. When a collaboration is written to the database, the system writes the current conversation state along with it. When a collaboration loads from the database the system re-establishes its conversation state if necessary. When a collaboration closes, the system ends its conversations in the ConversationMemory service and records the final state of all conversations in the closed instance.

Write Frequency

Collaborations are updated and managed entirely within the Service State. However, it can be useful for long-running collaborations to store the results to the database at some interval such that the service state does not become too large. Persisting the results can also serve as an audit log of historical collaborations. The write frequency determines the interval at which the service will persist the in-memory collaboration state to the database. By default, a service will persist its collaboration state every *5 minutes*.

Collaboration Management

Service State

Defining any collaboration roles or conversations will trigger generation of the collaboration service state and collaboration management procedures (using any of the Collaboration Management activity patterns in a VEH also triggers this). Collaboration state is managed in two partitioned state properties:

- `ActiveCollabs` (*Map[String, ArsCollaboration]*) – a mapping from the collaboration id to the associated collaboration instance.

- `CollabIdsByEntity` (*Map[String, String]*) – a mapping from an entity id to the corresponding collaboration id (used to facilitate collaboration establishment)

Collaboration instances have the following properties:

- **id** – a unique identifier for the Collaboration. The unique id is assigned by the system when the Collaboration is first created. The variable `collaborationId` is available for use as a configuration property to any task.
- **name** – the name of the collaboration (by default this will be the name of the creating event handler).
- **status** – the current state of the collaboration represented by one of the string values: *active*, *completed*, *failed*.
- **entities** – an object identifying the entity roles and the objects that are currently assigned to the entity role. The keys are the entity role names and the value is the role's current assignment.
- **collaborators** – an object identifying the current collaboration roles and the user assigned to each role. The keys are the collaborator role names and the value is the role's current assignment. The collaborators property represents the current snapshot of the participating collaborators. The participants may change over the course of collaboration execution.
- **conversations** – an object identifying the active conversations. The keys are the conversation "names" and the values are objects with the following properties:
  - **id** – the conversation id
  - **state** – the conversation's current state
  - **properties** – the conversation's properties
- **results** – results that are aggregated across activities. For example, a location tracking activity will save a list of the arrived users and the current locations for every other tracked user.

Procedures

In addition to the service state, the following collaboration management procedures are added to the service:

> In all cases the procedures will operate only on collaboration instances which are "owned" by the current service.

Public Procedures

- `ActiveCollabsCloseAll()` – Persists any collaborations in memory onto the database and immediately updates the status of all active collaborations (even those that weren't in memory) to *completed*.
- `ActiveCollabsGetActive(): system.collaborations ARRAY` – Returns all active collaboration instances. This will include persistent instances that are not currently in the cache.
- `ActiveCollabsGetById(collaborationId String REQUIRED, resultType String, allowInactive Boolean): Any` – Get the collaboration specified by the given id. Returns "null" if no such collaboration exists. By default this will return the entire collaboration instance. The *resultType* parameter can be used to request a specific collaboration property instead. It is an error to request an inactive collaboration instance, unless *allowInactive* is set to `true`.
- `ActiveCollabsGetCached(activeOnly Boolean DEFAULT true): system.collaborations ARRAY` – Returns all collaboration instances currently cached in memory by the service. By default this returns only active instances (unless *activeOnly* is set to `false`).
- `ActiveCollabsUpdateStatus(collaborationId String REQUIRED, status String REQUIRED)` – Updates the collaboration status to the specified value. This must be either *completed* or *failed*.
- `ActiveCollabsWriteAll()` – Persists each of the collaborations in memory onto the database. The Write Procedure will be scheduled to execute automatically every 5 minutes. Updating the *writeFrequency* for the Service will modify the interval at which the Write Procedure is executed.

Private Procedures

- `ActiveCollabsGenerateId(): String` – Generates a new collaboration id.
- `ActiveCollabsGetByEntityId(entityId String REQUIRED, entityRoleName String REQUIRED): system.collaborations` – Returns the active collaboration instance associated with the specified entity or `null` if none exists.
- `ActiveCollabsGetConversation(collaborationId String REQUIRED, conversationName String): String` – Returns the id of the named conversation from the specified collaboration instance. Returns the "default" conversation if no name is given.
- `ActiveCollabsGetOrCreate(collaborationId String REQUIRED, collaborationName String): Object` – Get the collaboration specified by the given id, creating one if it doesn't exist. The return value is an Object with the properties:
  - collaboration – the associated active collaboration instance.
  - isNew – "true" if the collaboration instance was created, otherwise "false".
- `ActiveCollabsGetOrCreateForEntity(entityId String REQUIRED, entityRoleName String REQUIRED, entity Object REQUIRED, initialCollaboration Object): Object` – Returns the active collaboration instance associated with the specified entity, creating one if it doesn't exist. The return value is an Object with the properties:
  - collaboration – the associated active collaboration instance.
  - isNew – "true" if the collaboration instance was created, otherwise "false".
- `ActiveCollabsSetEntity(collaborationId String REQUIRED, entityId String REQUIRED, entityRoleName String REQUIRED, entity Object REQUIRED)` – Set the given entity value in the collaboration with the specified id. Returns the updated collaboration.
- `ActiveCollabsStartConversation(collaborationId String REQUIRED, conversationName String, conversationProperties Object, collaborationName String): String` – Starts/continues the named conversation ("default" if none is specified) in the collaboration with the given id. The collaboration will be created if it does not currently exist. Returns the id of the conversation.

API Migration Notes:

The previous API included `ActiveCollabsGetAll` which was assumed to act like `ActiveCollabsGetActive`, but did not. That procedure is still available, but has been deprecated (and made private). Please migrate to either `ActiveCollabsGetActive` or `ActiveCollabsGetCached` prior to the next release.

The previous API directly exposed the "behavior" choices that are provided by the EstablishCollaboration activity pattern. This was very confusing. The updated API provides both a strict "getter", which will return `null` if no instance is found and a `getOrCreate` variant, which will create a new collaboration if none is found. These are provided in both `ById` and `ByEntity` variants (so you can start with either a collaboration id or an entity id). Previously, the entity based versions were generated independently for each task. These have been removed and any calls to them will need to be updated.

Entity Role Procedures

The default collaboration management procedures are focused primarily on directly managing collaborations. As such, most of them assume that you have the id of a specific collaboration on which to operate. Sometimes it is more convenient for a service to access the active collaborations via their associated entity id. To support this access style, the IDE offers the option to generate the following procedures for each defined entity role:

- `${entityRoleName}Activate(entityId String REQUIRED, entity ${entityType} REQUIRED, collaborationName String): String` – Activate a collaboration for the specified instance of the "${roleName}" entity role. Returns the id of the active collaboration.
- `${entityRoleName}Close(entityId String REQUIRED, asFailure Boolean)` – Close the active collaboration associated with the ${roleName} instance.
- `${entityRoleName}GetActive(forAllUsers Boolean DEFAULT false): Object Sequence` – Return details about all the currently active collaborations associated with the "${roleName}" entity role. The returned Object instances have the following properties:
  - collaborationId – the id of the collaboration used to manage the active ${roleName}.
  - entity – the current value associated with the active ${roleName}.
- `${entityRoleName}UpdateActive(entityId String REQUIRED, entity ${entityType} REQUIRED)` – Update the specified instance of the "${roleName}" entity role in its active collaboration (if one exists).

Generation of these procedures can be done for selected entity roles by clicking on the gear icon in the creation dialog as shown here: