Basic API Integration: FRAIHMWORK Health And Integrity API

Contents

[1 Introduction 2](#_Toc178164451)

[2 API Endpoints and Concepts 3](#_Toc178164452)

[2.1 Entities (“Actors”) 3](#_Toc178164453)

[2.2 Authentication Requests 3](#_Toc178164454)

[2.3 Component Requests 4](#_Toc178164455)

[2.3.1 Component Registration / Initial Reporting 4](#_Toc178164456)

[2.3.2 Component Updates 6](#_Toc178164457)

[2.3.3 Clearing Components 6](#_Toc178164458)

[2.3.4 Refreshing Components and Component Liveliness 6](#_Toc178164459)

[2.3.5 Field Descriptions 7](#_Toc178164460)

[2.4 Faults 9](#_Toc178164461)

[2.4.1 Fault Reporting 9](#_Toc178164462)

[2.4.2 Fault Updates 10](#_Toc178164463)

[2.4.3 Fault Clearing 11](#_Toc178164464)

[2.4.4 Refreshing Faults and Fault Timeouts 11](#_Toc178164465)

[2.4.5 Field Descriptions 12](#_Toc178164466)

[2.5 Registration 13](#_Toc178164467)

[2.6 Mitigation Handling API 14](#_Toc178164468)

[2.6.1 Mitigation Registration 15](#_Toc178164469)

[2.6.2 Mitigation Updates 16](#_Toc178164470)

[2.6.3 Retrieving Mitigation Details 17](#_Toc178164471)

[2.7 Return objects 18](#_Toc178164472)

[2.8 Specifications and Miscellaneous 19](#_Toc178164473)

[3 Configuration 20](#_Toc178164474)

[3.1 FRAIHMWORK Configuration Library 20](#_Toc178164475)

[3.1.1 Default Data Fields 20](#_Toc178164476)

[3.1.2 Representative Image 20](#_Toc178164477)

[3.2 Fault Library 20](#_Toc178164478)

[3.3 Timeouts 20](#_Toc178164479)

[4 Security 21](#_Toc178164480)

[5 Limitations And Future Work 22](#_Toc178164481)

[Appendix A. 23](#_Toc178164482)

# Introduction

Integration between an existing system and FRAIHMWORK means using the FRAIHMWORK provided APIs to relay component health information so that it can be processed by the FRAIHMWORK platform and ultimately displayed for a maintainer. This document will provide the high-level tasks that need to be completed to successfully integrate the two systems and provide a deeper explanation for the workflows and data concepts that are needed to make integration a success.

The scope of this document is limited to the APIs that we expect to be using for integrating these two platforms under this effort. Although other APIs are publicly exposed to other clients, they will not be necessary.

# API Endpoints and Concepts

All the API details are in the API specifications themselves. This section exists to provide context and explain some prerequisite concepts to help understand them.

## Entities (“Actors”)

Entities include actors, such as human users and non-human services and system components. Actor types along with their respective definitions are shown below in Figure 1.

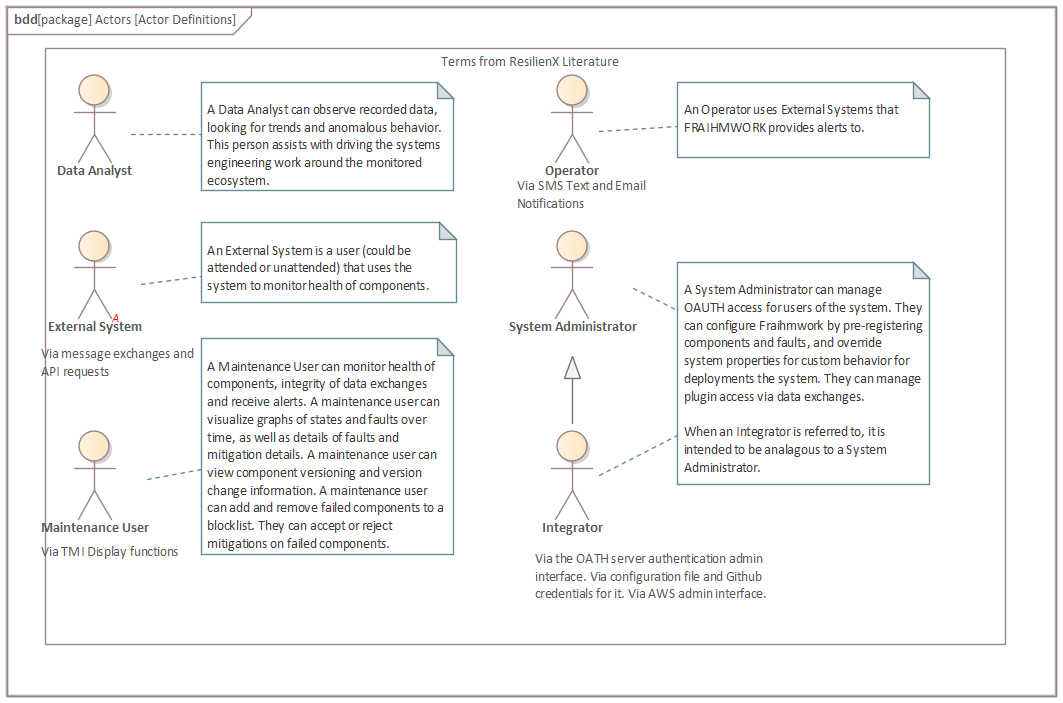


Figure 1: Actor Definitions

## Authentication Requests

Interaction with FRAIHMWORK via FRAIHMWORK API requests requires authentication using OAuth 2.0. Credentials are required for requesting an access token from the OAuth 2.0 server using an HTTPS request. Once the credentials are accepted by the OAuth server, an access token and some additional information is provided in the response from the access request. All API requests to FRAIHMWORK must include the access token, otherwise an authentication error will result. This activity is shown in the sequence diagram, below (Figure 2).

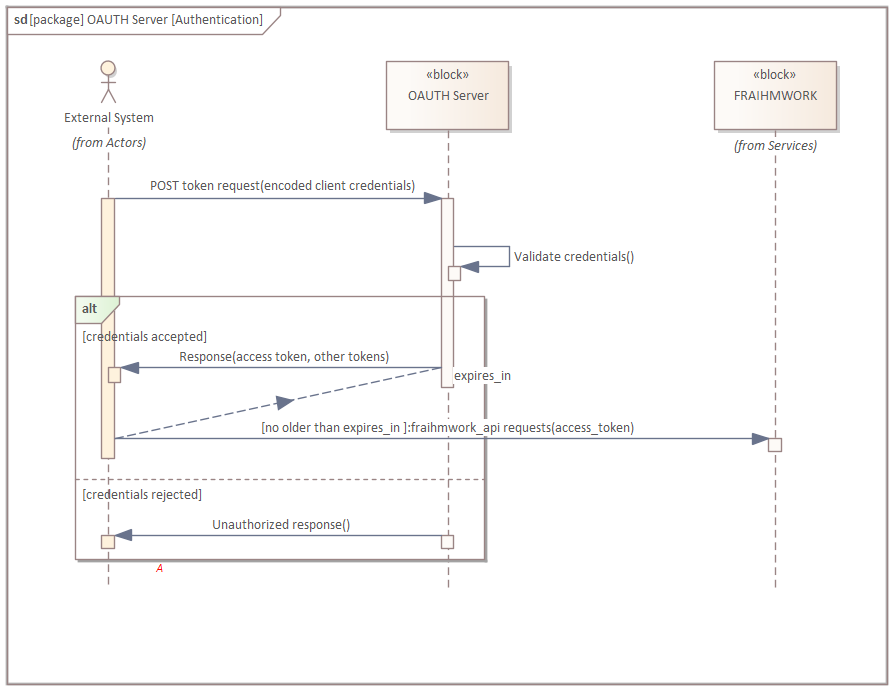


Figure 2: Authentication activity using OAuth 2.0

For more information on authentication, see section 4, Security.

## Component Requests

External systems are made up of components. A component can be a complete system, or it can be one piece of many that make up a complete system. In the context of the API, a component is a monitored entity that can report its status over HTTP/S. The component object (“Monitored Component”) in the API represents all the facets of the component that FRAIHMWORK wants to record, digest, and, finally, portray on the TMI Display.

### Component Registration / Initial Reporting

When an external system or component wants to be monitored by FRAIHMWORK, it initiates a connection and posts its current component data, via the FRAIHMWORK Health and Integrity API as shown in Figure 3 below. Additionally, components can be POSTed by other systems on behalf of these components. This is common in case of components that are monitored, but do not have the ability to self-monitor or self-register.

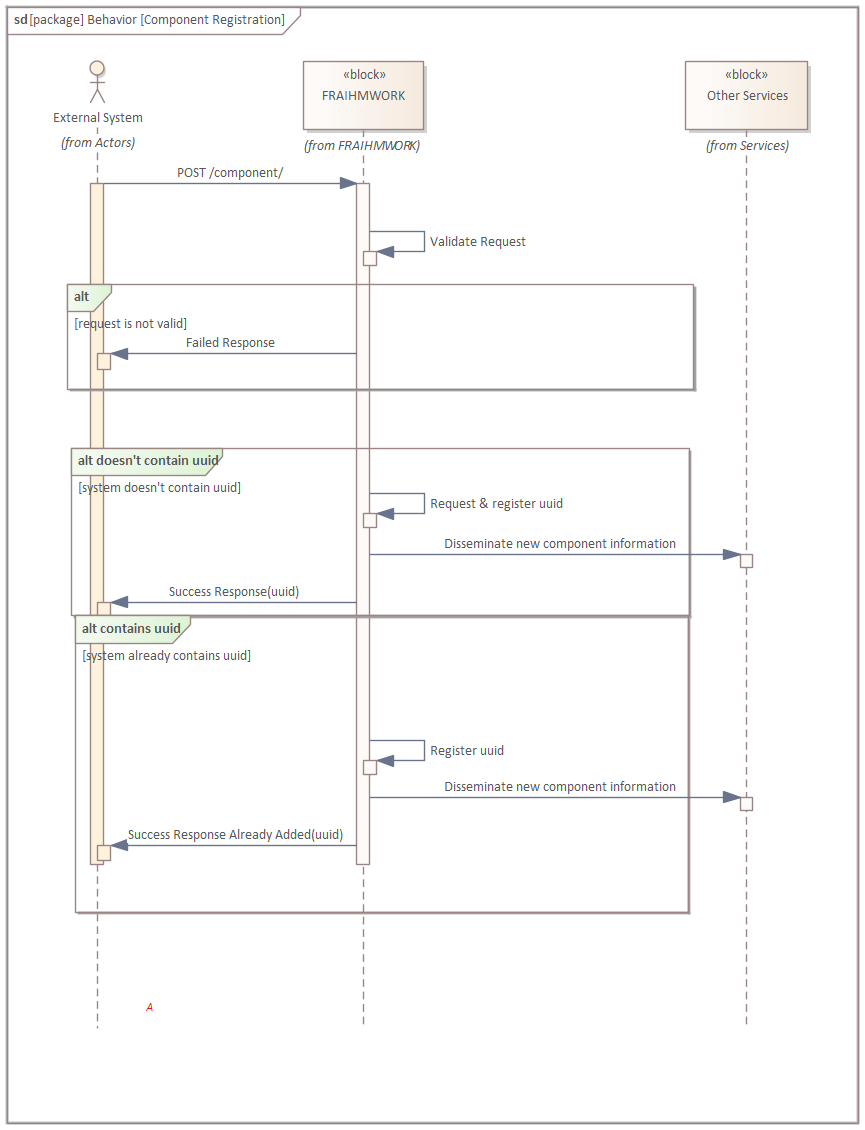


Figure 3: Initial Component Registration

If a component object is posted by an external system, then FRAIHMWORK will return the UUID of the newly registered component. If the component has a UUID that they already want to use, they can include it in the POST message and FRAIHMWORK will register that UUID instead of generating a new one.

### Component Updates

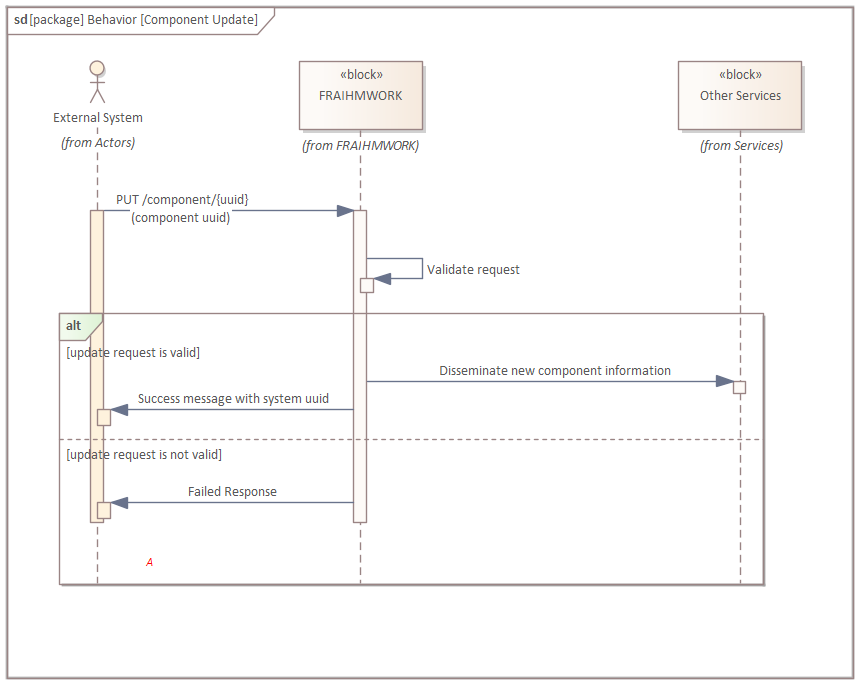


Figure 4: Subsequent Component Updates

For any subsequent updates to the component, due either to changes or as a keep alive measure, the HTTP PUT command is used in conjunction with the registered UUID, returned from the initial POST as seen in Figure 4.

### Clearing Components

The same pattern is used for deletion of components as is used with updating components, except the HTTP DELETE command is used instead.

### Refreshing Components and Component Liveliness

The FHAIS service requires a given component to be “refreshed” within a timeout period in the Monitored Component object data type. An empty POST request on the /component/{component uuid}/refresh is used for this action. If activity on that component is not provided to FHAIS, it will produce a liveliness fault. Additionally, a subsequent PUT on /component/{uuid} will also refresh component liveliness.

### Field Descriptions

Although some fields within the component object are self-explanatory, some require more detail and are further defined below.

#### State

Components can be described with one of the following states:

* STARTING\_UP: Used to denote a component that will be providing services soon, but not yet.
* ONLINE: Used to describe a component that is operating normally
* FAULTED: Used for a component that has one or more faults against it that is affecting the quality of service
* UNAVAILABLE: Used to describe a component that cannot be reached. Generally, this state will not be reported by a component, but it could be used to describe a subsystem that the component/system relies on. Components in this state are not subject to liveliness checks.

The states that describe a component are depicted below in the following notional state diagram. Note, FRAIHMWORK does not enforce the transitions in this state diagram, rather, it is for informational purposes on how these states should be regarded by an integrator.

Diagram

Description automatically generated

Figure 5 Monitored Component States

#### Versions

The data type provides several fields to describe the version of the component. These do not require strict interpretations, so long as they are consistently used to describe the same thing. Refer to semantic versioning as a widely accepted set of guidelines for versioning systems and components – [www.semver.org](http://www.semver.org).

#### Extra Properties (extensions)

The FHAIS API is designed to be flexible and allow for additional properties to be described in the extensions portion of the monitored component data type, if desired. Putting values here provides no guarantee of specific behavior within the FRAIHMWORK system but will be passed to any logging and data recording and will be passed to the display if customized rules are used for certain sites. For example, a radar system operating in a specific long range scanning mode may capture that in a ‘mode’ extra property so that it can be recorded by FRAIHMWORK (and possibly displayed if an adaptation was made for it on the display).

#### Parent

The concept of components and subcomponents exists in this API, and it is assumed that the client takes responsibility for reporting on all of them if it defines such a relationship. The field in the data type is optional and should only be used if there is an explicit ownership. In order to report on subcomponents, the process is the same, except that the UUID provided to the parent component should be used for this parent field. When there are subcomponents of a component, they should be registered from the highest level down, as shown in Figure 6.

|  |  |
| --- | --- |
|  | Figure 6: Subcomponent (“parent-child”) Registration |

This order of operations ensures that the UUIDs of the parent component(s) are available to fill out the parent field of any subsystems.

#### Time of Validity

This is a timestamp that the client can provide to explicitly state when the data provided in the message was last accurate. If left blank, then the timestamp of receipt is assumed to be the time of validity. Although it can be left empty, it is useful to fill out this information for error tracing.

## Faults

Faults are any off-nominal condition that occurs within a system and its components. Anything that is outside the bounds of perfectly normal operation could be considered a fault, whether it is an atypical observation that has no impact or a serious communication error with another system.

### Fault Reporting

The workflow for reporting and updating faults is nearly identical to that for reporting and updating component information.

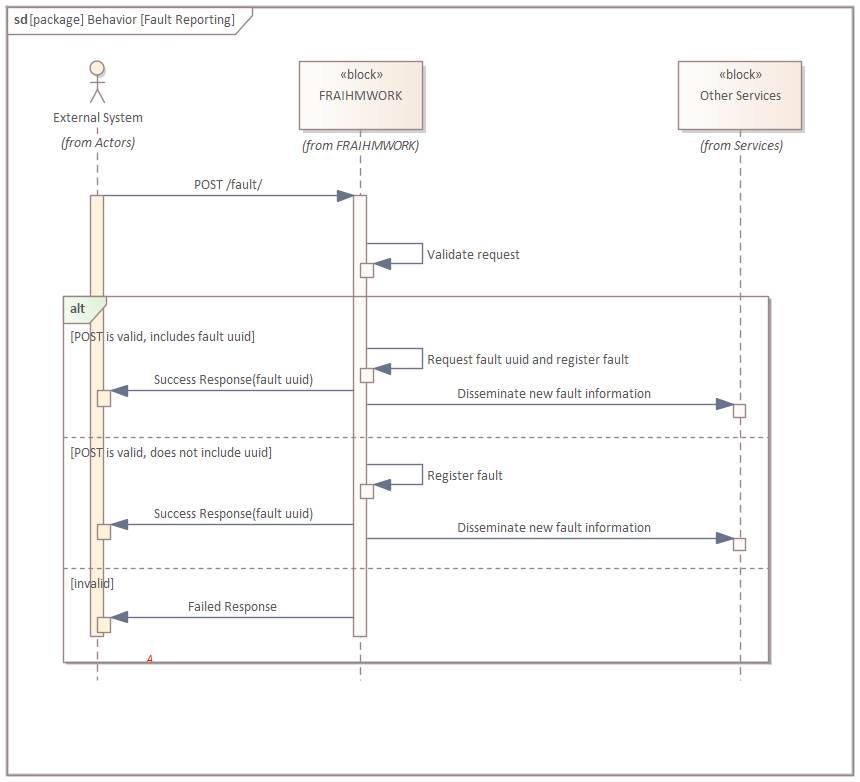


Figure 7: Fault Reporting

The fault is registered with a POST call, and FRAIHMWORK will attempt to honor the fault UUID if provided so long as it does not conflict with anything already registered in the system, otherwise, it will generate one for the fault instance resulting from the request. Registration of the fault UUID will happen under the hood, and the return value of the call will simply be a success message that includes the UUID that was accepted, or an error message with a null UUID field. A required parameter for the registration of the fault is the component UUID that the fault is concerned with.

### Fault Updates

To ensure that the fault is still active, FRAIHMWORK expects that updates or resends of the information will occur at certain intervals, or it will clear the fault due to inactivity.

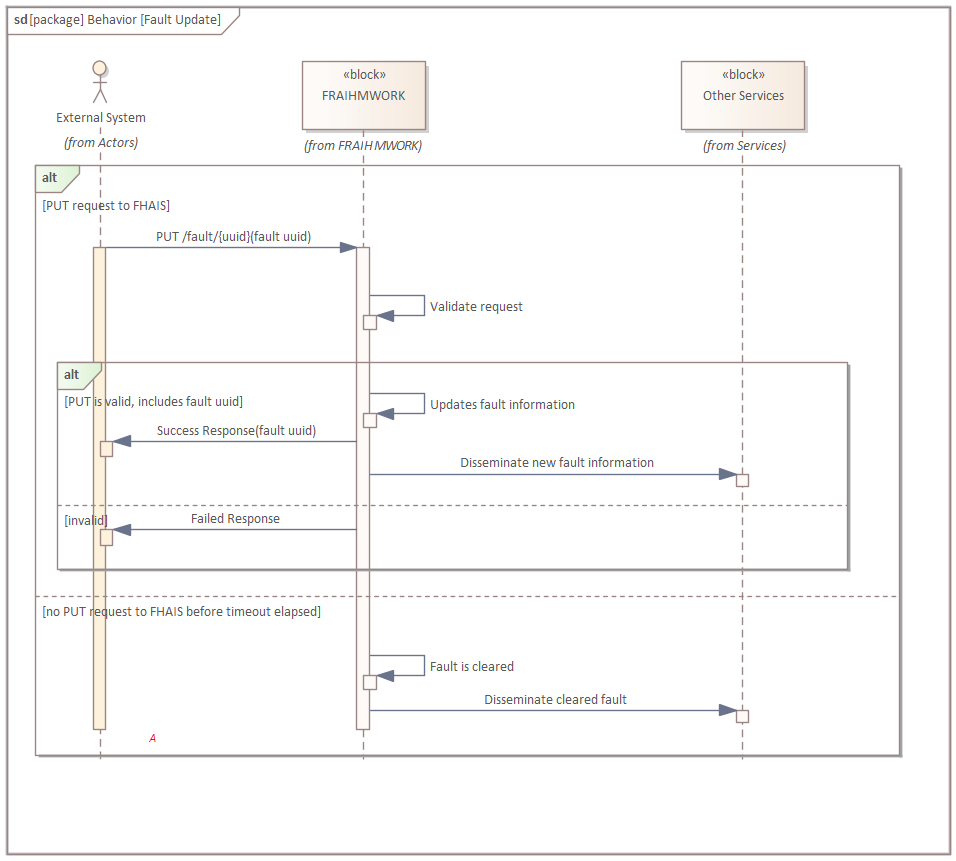


Figure 8: Fault Updating / Resending

To update or resend a fault, the HTTP PUT command is used with the UUID of the fault as a parameter. The dataflow is identical to that of resending or updating a component object from section 2.2.

### Fault Clearing

Faults need to be cleared when they are no longer active, and the way to do this is with the HTTP DELETE command with the UUID of the fault provided as a URL parameter.

### Refreshing Faults and Fault Timeouts

FRAIHMWORK expects that faults that persist in the environment should continue to be reported on. Therefore, a given fault needs to be “refreshed” within a timeout period in the Fault object data type. An empty POST request on the /fault/{uuid}/refresh (uuid denotes the fault uuid) is used for this action. PUT updates on the /fault/{uuid} endpoint also serve this purpose. If activity on that fault is not provided to FRAIHMWORK, the fault will be cleared automatically.

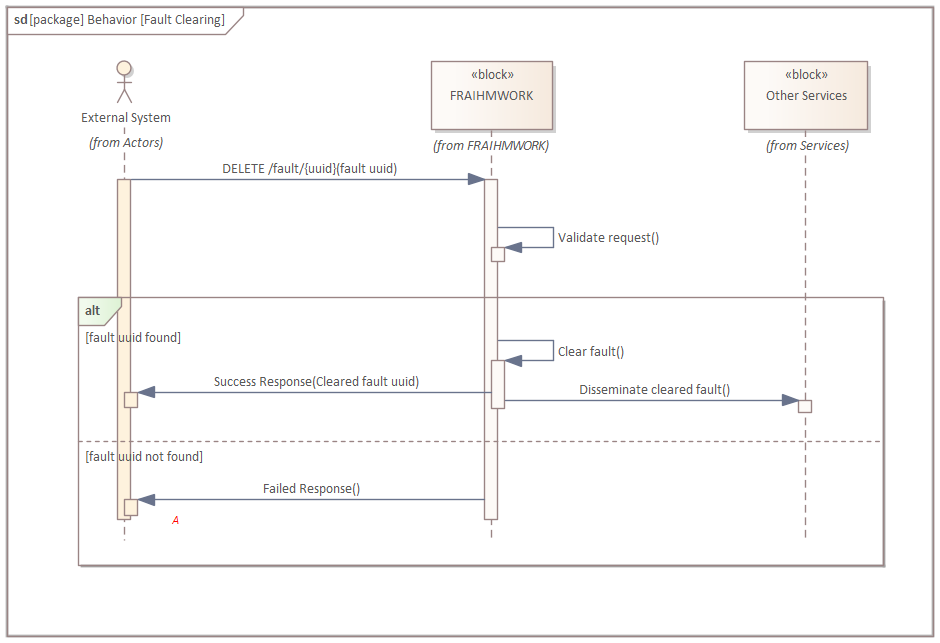


Figure 9: Fault clearing

Clearing a fault requires no additional payload of the latest fault status.

### Field Descriptions

Although some fields within the fault object are self-explanatory, some require more detail and are further defined below.

#### Severity

The severity of a fault indicates how impactful it is to the monitored component as a whole. This can range from “something the component notices” to “something that is causing failure within the component or system”. To capture this range, there are four levels of fault severities:

* LOW – The component is unaffected by this fault, but it is useful to log it for some reason, or could become a more severe fault over time
* MEDIUM – The component’s operational performance is suffering due to this fault, but the quality of information that it is producing is still reliable
* HIGH – The quality of service is affected and data is untrustworthy, due to it being incorrect, missing, or not meeting other necessary requirements
* UNKNOWN – This is generally reserved for when the source of the fault cannot be determined, and thus, the component does not understand its impact on operations. This should not be used in most use cases, as components should be aware of the faults that they can produce. In terms of fault evaluation within FRAIHMWORK, “UNKNOWN” is considered the highest severity.

#### Source

The UUID of the affected component that the fault is against. If this field is left blank, then it is equivalent to putting the UUID of the reporting system there.

#### Code

The fault code is a number that is a unique identifier for the type of fault that is being instantiated or referenced. A code should be persistent across all fault reporting, in that any two faults that have the same code should be the same type of problem. Conversely, two faults with different codes should not be reporting about the same exact problem. Partners should pre-plan their code mappings in the design phase (this is discussed in more detail in sections 3.2).

#### Time of Detection / Time of Validity

The time of detection is when the fault was first discovered. This should not change with any subsequent updates to the fault. If it is left blank, it is assumed that the time of receipt of the first POST message is the time of detection.

By contrast, the time of validity represents the most recent time that the fault was verified to still be active. For example, if the fault is caused by a failed hardware check that is performed every 3 minutes, the time of detection would be the first time it was detected, and the time of validity would change every 3 minutes when the check is made.

The time of validity is expected to be equal or later than the time of detection. If it is left blank, then it is assumed that the time of receipt of the most recent PUT message is the time of validity, which may or may not be the most accurate description.

## Registration

This API provides a mechanism for client systems to interact with the active list of components and faults within the FRAIHMWORK system. This is primarily useful to get the latest information about components that are monitored by FRAIHMWORK. It also provides the ability to generate new UUIDs for forthcoming API requests that use these as allocated identifiers.

Using an HTTP GET command on the /component endpoint will result in a full list of the active components in the system, which can be used to get the UUID information or other details for all components. Component information can also be requested by UUID by using GET on the /component/{uuid} endpoint as well.

Faults can be accessed similarly using the /fault/ and /fault/{uuid} endpoints. For more information, see the API specification for filtered queries and additional data type fields for the responses on these endpoints.

## Mitigation Handling API

A mitigation is one or more actions that can be taken by a system or person to reduce the effect of (or clear) a FRAIHMWORK fault or external off-nominal situation.

For example, if a network switch is unresponsive, FRAIHMWORK may provide steps to try and resolve the situation in the form of a mitigation. Performing the mitigation’s actions may resolve the situation entirely (clearing the fault) or lessen its impact (perhaps by activating a backup device).

The relationship between faults and mitigations can be seen in Figure 10.

Diagram

Description automatically generated

Figure 10: Block diagram - Relationship between mitigations and faults

As per the diagram, a fault can have zero or more mitigations associated to it. Also, a single mitigation can be associated to zero or more faults.

There are two types of resolutions to mitigations based on the actor who resolves it:

* + **Manual**: Manual mitigations are enacted by users through manual actions
  + **Automatic**: Automatic mitigations are resolved by a service. This is the default resolution of a mitigation.

During its lifecycle, a mitigation can be in different states as shown in Figure 11:

Diagram

Description automatically generated

Figure 11: Mitigation Lifecycle

* **RECOMMENDED**: The mitigation action has been recommended to an end user but is waiting for further instruction. This is the default initial mitigation state. From here, the state can only change to 'ACCEPTED', 'REVOKED', or 'DENIED'.
* **ACCEPTED**: The mitigation has been accepted by a user but has not yet been executed. This as well can be an initial state. Mitigations that start in this state are understood to have been added by a human, rather than an automated process. From here, the state should only change to 'STARTED', 'REVOKED', or 'FAILED'.
* **STARTED**: The mitigation action has started and is still in progress. From here, the state should only change to 'COMPLETE' or 'FAILED'.
* **COMPLETE**: The mitigation action has been fully resolved. This is a terminal state for the action.
* **FAILED**: The mitigation action was unsuccessful in completing its task or its completion resulted in an unsatisfactory mitigation. This is a terminal state for the mitigation.
* **REVOKED**: The executor of the mitigation can no longer perform the action, or the context has changed such that the mitigation would not be effective. It effectively cancels the mitigation and is a terminal state.
* **DENIED**: A user has declined the mitigation action. It is a terminal state for the mitigation.

The user can control or update the state of mitigation through the TMI display. For MANUAL resolutions, all state changes must be driven by the user, whereas for AUTOMATIC resolutions, the user must signal to the executor that they approve/deny the mitigation. If the mitigation is approved, the executor is responsible for taking the mitigation action and then updating the state.

A constraint on mitigation operations is that components and faults must be registered prior to being referenced The Mitigation API provides access to the user to perform the following mitigation operation.

### Mitigation Registration

Mitigations are identified by FRAIHMWORK by their issuer id, executor id, list of faults associated, description, and the time of issue. Here, the issuer is the component that recognizes and creates the mitigation, while the executor is the component running the mitigation process.

A mitigation is registered with a POST call for which FRAIHMWORK will generate a new UUID internally. The mitigation will be successfully registered with the UUID if no other mitigation is registered with the same issuer id, executor id, faults, description, and the time of issue.

As shown in Figure 12, to register a mitigation against one or more faults, the following steps can be executed:

Diagram

Description automatically generated

Figure 12: Mitigation Registration

1. Verify that there exists a registered component against which one or more faults may be registered.
2. Send a POST request at <base URL>/api/mitigation/v0/mitigation by applying the generated access token. The request body must be a JSON containing the mitigation details.

### Mitigation Updates

To update the mitigation state, the HTTP PUT command is used in conjunction with the registered mitigation uuid which was returned from the initial mitigation registration. Updates to the same mitigation must include the state of the mitigation along with the time of validity. If the time of validity is not provided, the time of receipt of the update message will be considered.

As shown in Figure 13, to update an available mitigation, send a PUT request at <base URL>/api/mitigation/v0/mitigation/{uuid} by applying the generated access token. The {uuid} refers to the UUID of the registered mitigation whose properties are to be updated. The request body should contain the state of the mitigation and time of validity (if desired).

Diagram

Description automatically generated

Figure 13: Mitigation Updates

If the mitigation exists, a success message will be returned with the mitigation uuid, whereas if the mitigation does not exist, the user will get a 404-error response.

### Retrieving Mitigation Details

The Mitigation Service API provides a mechanism for users to get the latest information related to mitigations registered with the FRAIHMWORK system. For example, suppose a user wanted to learn which mitigations are registered with FRAIHMWORK and then determine which one of the mitigations are accepted. The user can then determine who is performing the mitigation and what faults are being addressed through a single mitigation.

Diagram

Description automatically generated

Figure 14: Retrieve mitigation details

To receive a full list of all the mitigation registrations in the system, as seen in Figure 14, an HTTP GET command will need to be executed on <base URL>/api/mitigation/v0/mitigation/ endpoint. Each mitigation in the list of mitigations will contain the mitigation uuid and other details.

Suppose a user is interested in learning about a single mitigation, the mitigation can also be requested by mitigation UUID by performing a GET on the <base URL>/api/mitigation/v0/mitigation/{uuid} endpoint.

If a user is interested in filtering mitigations by their current state, performing a GET on the <base URL>/api/mitigation/v0/mitigation/state/{state} endpoint.

## Return objects

All the returned JSON objects contain a message that indicates success or failure, as well as the UUID that is relevant to the call that was made. For all component requests, the returned UUID is the UUID of the component information of data type ActiveComponent, being reported, or the new UUID generated by the post request. Similarly, the fault endpoint returns the ActiveFault data type, and the UUID is representative of the fault’s UUID that is being used to track it. If the UUID is null, this indicates that the component or fault was either not successfully created (in the case of an error), that it could not be found, or that it has been deleted.

## Specifications and Miscellaneous

All data sent to and returned from the APIs must be in JSON format as defined in [RFC 7159](https://tools.ietf.org/html/rfc7159). Therefore, it is expected that the HTTP ‘Accept Header’ is set to or includes JSON.

All timestamp information must be described as strings following ISO 8601 and must include time zone information if it is not presented as UTC.

The versioning of the APIs will follow SemVer 2.0.0 rules as described here <https://semver.org/>

# Configuration

In order to handle a wide variety of different clients, there are some assumptions that FRAIHMWORK must make in order to handle a number of different systems and components. These assumptions can be overridden with configuration, and clients may elect to provide configuration data if they have specific reporting requirements.

## FRAIHMWORK Configuration Library

The configuration library is a collection of descriptive configuration files that represent the minimum base of what is to be expected in each ecosystem. It is also where parameters can be entered to change how FRAIHMWORK monitors specific components and services.

### Default Data Fields

Many of the data fields that are present in the system information data object can be pre-populated in the configuration library entry. This is useful for when the system is present in the ecosystem’s configuration but is powered off or connectivity is disrupted. The data can still be presented as it is in the configuration entry, but the state will simply be changed to either OFFLINE or UNAVAILABLE depending on the circumstances.

### Representative Image

The Components Tier of the TMI Display can optionally show a picture of the component being monitored, or a logo or other image. The easiest way to provide this is with a .png image, though dynamic URLs for this could be used in the future.

## Fault Library

Beyond the basic name, code, and severity, which are all required fields in the Fault object, there are additional pre-configured fields that can be set to suggest general troubleshooting and remediations that can be taken by the maintainer. These will be shown on the TMI Display and provides the opportunity for much longer descriptions, what system reactions are being automatically taken, and what the maintainer can do to help remediate the problem. For this reason, it is important that fault codes are organized in a way such that they can uniquely identify different problems, so that they can be reused for this purpose.

## Timeouts

The system and fault endpoints both operate on the assumption that data is continually refreshed whenever changes occur. However, if a system is in a steady state, this could mean that there are no new data for hours or days. Therefore, timeouts can be configured to set a maximum amount of time that a client system will go before sending an update on its system or fault information. If this threshold is exceeded, then FRAIHMWORK assumes that connectivity has been disrupted and will generate a fault on its behalf.

# Security

The API specifies that the primary security protocol used for the endpoints is OAuth 2.0 (<https://tools.ietf.org/html/rfc6749>) authorization framework. The flow being used will be Client Credential, which is the typical use for interactions that do not involve an individual user. More detailed security information will be provided on a site-by-site basis depending on what, if any, authorization server exists. If none exists, FRAIHMWORK will provide its own service to stand in its place.

# Limitations And Future Work

At the time of writing, there are no inherent protections against bad actors within the ecosystem reporting an excess of faults, components, etc., nor are there explicit permissions set on objects created by a client. Future development on these APIs and the servers that resources can only be modified or deleted by the systems that initially create them.



This section outlines additional details to aid the user in the initial integration stages with FRAIHMWORK.

1. Capture OAuth Token
   1. Client ID
      1. Provided to you by ResilienX.
   2. Client Secret
      1. Provided to you by ResilienX.
   3. Retrieve URL
      1. Combination of the Endpoint and whichever FRAIHMWORK namespace you would like to register components for.
   4. Endpoint: POST /auth/realms/dev/protocol/openid-connect/token
   5. Returns: Bearer Token
   6. Note: The returned token must be used when creating the API client.
2. Check for Component Existence
   1. Get All Components
      1. Use GET on the component endpoint to capture all component names.
   2. Endpoint: GET /monitor/v1/component
   3. Returns: List of Components
   4. Note: Use the component names to determine if the partner software service already registered itself as a component at some point. Save the component's UUID to begin updating liveliness and post faults.
3. Create Component
   1. Name
      1. The name of the software service for each partner (e.g., AIS - Artemis).
   2. State
      1. State should be {"state": "ONLINE"} to indicate the software service is running.
   3. UUID
      1. The UUID may be left blank or it can be provided by the software service.
   4. Manufacturer
      1. A good place to add the partner name (e.g., AIS for the Artemis component).
   5. Timeout
      1. The most important parameter. Set this to a maximum of 10 seconds. This means that the software service must update the liveliness of the component within 10 seconds before a fault is created by FRAIHMWORK.
   6. Endpoint: POST /monitor/v1/component
   7. Returns: Component UUID
   8. Note: You must provide the bearer token returned when capturing the OAuth token.
4. Update Component Liveliness
   1. Name
      1. The name of the software service for each partner (e.g., AIS - Artemis).
   2. State
      1. State should be {"state": "ONLINE"} to indicate the software service is running.
   3. UUID
      1. The UUID may be left blank or it can be provided by the software service. It is recommended to include the UUID that was either used or returned by FRAIHMWORK when creating the component.
   4. Endpoint: PUT /monitor/v1/component
   5. Returns: Component UUID
   6. Note: This needs to be done within the "Timeout" seconds that were specified when the component was created.
5. Refresh OAuth Token
   1. Client ID
      1. Provided to you by ResilienX.
   2. Client Secret
      1. Provided to you by ResilienX.
   3. Retrieve URL
      1. Combination of the Endpoint and whichever FRAIHMWORK namespace you would like to register components for.
   4. Endpoint: POST /auth/realms/dev/protocol/openid-connect/token
   5. Returns: Bearer Token
   6. Note: Re-run the 'Capture OAuth Token' step to obtain a new token and replace the old one.
6. Create Fault
   1. Name
      1. The name of the software service for each partner (e.g., AIS - Artemis).
   2. Source
      1. The UUID may be left blank or it can be provided by the software service. It is recommended to include the UUID that was either used or returned by FRAIHMWORK when creating the component.
   3. Code
      1. Unique code for the fault. It must be an integer value.
   4. Severity
      1. Can either be LOW, MEDIUM, HIGH, or Unknown and indicates how detrimental the fault is for the partner software service.
   5. Description
      1. A high-level description of what the fault indicates.
   6. Timeout
      1. Number of seconds within which a fault update is expected by the partner service.
   7. Endpoint: POST /monitor/v1/fault
   8. Returns: Fault UUID
   9. Note: Specifying the source allows you to guarantee that you are posting the fault to the correct component in the rare case that a duplicate component name exists. Save the returned UUID for when this fault is to be deleted.
7. Update Fault
   1. Name
      1. The name of the software service for each partner (e.g., AIS - Artemis).
   2. Source
      1. The UUID may be left blank or it can be provided by the software service. It is recommended to include the UUID that was either used or returned by FRAIHMWORK when creating the component.
   3. Code
      1. Unique code for the fault. It must be an integer value.
   4. Severity
      1. Can either be LOW, MEDIUM, HIGH, or Unknown and indicates how detrimental the fault is for the partner software service.
   5. Endpoint: PUT /monitor/v1/fault
   6. Returns: Fault UUID
   7. Note: This needs to be done within "Timeout" seconds that were specified when the fault was created.
8. Delete Fault
   1. UUID
      1. The UUID of the fault that was created by the software service originally.
   2. Endpoint: DELETE /monitor/v1/fault
   3. Returns: Fault UUID
   4. Note: The UUID from "Create Fault" is to be utilized to delete the fault.