Events

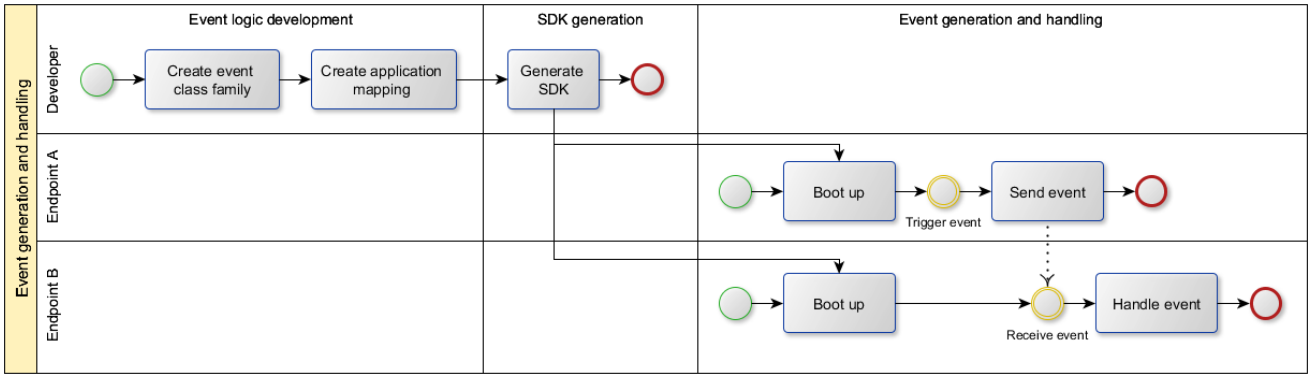
The Kaa Event subsystem enables generation of events on endpoints in near real-time fashion, handling those events on a Kaa server, and dispatching them to other endpoints that belong to the same user (potentially, across different applications). The Kaa event structure is determined by a configurable *event class schema*.

The Kaa Event subsystem provides the following features.

- Generation of the event object model and related API calls in the endpoint SDK
- Enforcement of data integrity and validity
- Efficient targeting of event recipients
- Efficient and compact serialization

It is the developer's responsibility to design event class schemas and make the client application interpret event data supplied by the endpoint library. The Kaa administrator, in turn, can provision those schemas into the Kaa server and generate the endpoint SDK.

Event generation and handling overview



Event class and event class schema

Each event is based on a particular event class (EC) that is defined by the corresponding event class schema. An EC is uniquely identified by a fully qualified name (FQN) and a tenant. In other words, there can be no two ECs with the same FQN within a single tenant.

An event class schema format is based on the Avro schema with the additional attribute *classType* that supports two values: *event* and *object* . Kaa uses the *classType* attribute to distinguish actual events from objects, which are reusable parts of events. This is useful for avoiding redundant methods in SDK API.

The following examples illustrate basic event class schemas.

- The simplest definition of an event with the com.company.project.SimpleEvent FQN and no data fields

```
{
    "namespace": "com.company.project",
    "type": "record",
    "classType": "event",
    "name": "SimpleEvent1",
    "fields": []
}
```

- The event definition with the com.company.project.SimpleEvent2 FQN and two data fields (*field1* and *field2*)

```
{
    "namespace": "com.company.project",
    "type": "record",
    "classType": "event",
    "name": "SimpleEvent2",
    "fields": [
  { "name": "field1", "type": "int"},
  { "name": "field2", "type": "string"}
 ]
}
```

- The event definition with the com.company.project.ComplexEvent FQN and two complex fields: com.company.project.SimpleRecordObject and com.company.project.SimpleEnumObject

```
[
    {
        "namespace": "com.company.project",
        "type": "enum",
        "classType": "object",
        "name": "SimpleEnumObject",
        "symbols" : ["ENUM_VALUE_1", "ENUM_VALUE_2", "ENUM_VALUE_3"]
    },
    {
        "namespace": "com.company.project",
        "type": "record",
        "classType": "object",
        "name": "SimpleRecordObject",
        "fields": [
            { "name": "field1", "type": "int" },
            { "name": "field2", "type": "string" }
        ]
    },
    {
        "namespace": "com.company.project",
        "type": "record",
        "classType": "event",
        "name": "ComplexEvent",
        "fields": [
            { "name": "field1", "type": "com.company.project.SimpleEnumObject"
},
            { "name": "field2", "type":
"com.company.project.SimpleRecordObject" }
        ]
    }
]
```

Event class families

ECs are grouped into event class families (ECF) by subject areas. ECFs are registered within the Kaa tenant together with the corresponding *event class family schemas*.

An ECF is uniquely identified by its name and/or class name and tenant. In other words, there cannot be two ECFs with the same name or same class name within a single tenant. Although this is quite a strict requirement, it helps prevent naming collisions during the SDK

generation.

To simplify the process of EC and ECF setup, Kaa Web UI automatically creates the ECF and corresponding EC entities based on the ECF name, class name and schema.

The following examples illustrate basic event class family schemas.

- The schema that contains two events with FQNs com.company.project.family1.SimpleEvent1 and com.company.project.family1.SimpleEvent2

```
[
    {
        "namespace": "com.company.project.family1",
        "name": "SimpleEvent1",
        "type": "record",
        "classType": "event",
        "fields": []
    },
    {
        "namespace": "com.company.project.family1",
        "name": "SimpleEvent2",
        "type": "record",
        "classType": "event",
        "fields": [
            { "name": "field1", "type": "int" },
            { "name": "field2", "type": "string" }
        ]
    }
]
```

- The schema that contains two events (with FQNs com.company.project.family2.ComplexEvent1 and com.company.project.family2.ComplexEvent2) that reuse the same complex field type com.company.project.family2.SimpleObject

```
[
    {
        "namespace": "com.company.project.family2",
        "name": "SimpleObject",
        "type": "record",
        "classType": "object",
        "fields": [
            { "name": "field1", "type": "int" },
            { "name": "field2", "type": "string" }
        ]
    },
    {
        "namespace": "com.company.project.family2",
        "name": "ComplexEvent1",
        "type": "record",
        "classType": "event",
        "fields": [
            { "name": "field1", "type":
"com.company.project.family2.SimpleObject" },
            { "name": "field2", "type": "int" }
        ]
    },
    {
        "namespace": "com.company.project.family2",
        "name": "ComplexEvent2",
        "type": "record",
        "classType": "event",
        "fields": [
            { "name": "field1", "type":
"com.company.project.family2.SimpleObject" },
            { "name": "field2", "type": "string" }
        ]
    }
]
```

Once the event class family schema is loaded into the Kaa application, the Control server automatically assigns it the version number. The user can define new versions of the ECF schema, whereas each version may contain different event classes, if necessary.

Event family mapping

One application can use multiple ECFs, while the same ECF can be used in multiple applications. In other words, the user can define ECFs that will be used by multiple applications. This is useful for controlling sources and sinks of particular events. For example, the user may want to implement the following rules:

- Application A should be able to send events with class E1 but does not need to receive them. Thus, application A is the **source** of E1.
- Application B should be able to receive events of class E1 but does not need to send them. Thus, application B is the **sink** of E1.
- Application C should be able to both receive and send events of class E1. Thus, application C is **both** the source and the sink of E1.

Once the application and ECF are created, the tenant administrator can create a mapping between these two entities by assigning a certain version of the ECF to the application. This mapping in Kaa is called *event family mapping*. Multiple ECFs (but not multiple versions of the same ECF) can be mapped to a single application.

By default, the application is mapped to each event of the ECF as both the source and the sink; however, the administrator can overwrite the default mapping. Once defined, the mapping cannot be changed in the future.

Event routing

Events can be sent to a single endpoint (unicast traffic) or to all the event sink endpoints of the given user (multicast traffic).

In case of a multicast event, the Kaa server relays the event to all endpoints registered as the corresponding EC sinks during the ECF mapping. If the user's endpoints are distributed over multiple Operation servers, the event is sent to all these Operation servers. Until being expired, thew event remains deliverable for the endpoints that were offline at the moment of the event generation.

In case of a unicast event, the Kaa server delivers the event to the target endpoint only if the endpoint was registered as the corresponding EC sink during the ECF mapping. The EP SDK supplies API to query the list of endpoints currently registered as the EC sinks under the given user.

Event sequence number

Sequence numbers are used to avoid duplication of events sent by endpoints. Each endpoint has its own sequence number, which is incremented by one with every event sent by this endpoint.

With the first sync request, the endpoint attempts to synchronize its event sequence number with the one stored at the Operation server. The server answers with either the sequence number of the latest event received from the endpoint or the number zero (if no events were received so far). If the number provided by the server differs from the number stored at the endpoint, the endpoint accepts the former and uses it as a starting number for new events.

SDK generation

During the SDK generation, the Control server generates the event object model and extends the APIs to support methods for sending events and registering event listeners. The generated SDK can support multiple ECFs, although it cannot simultaneously support multiple versions of the same ECF.
