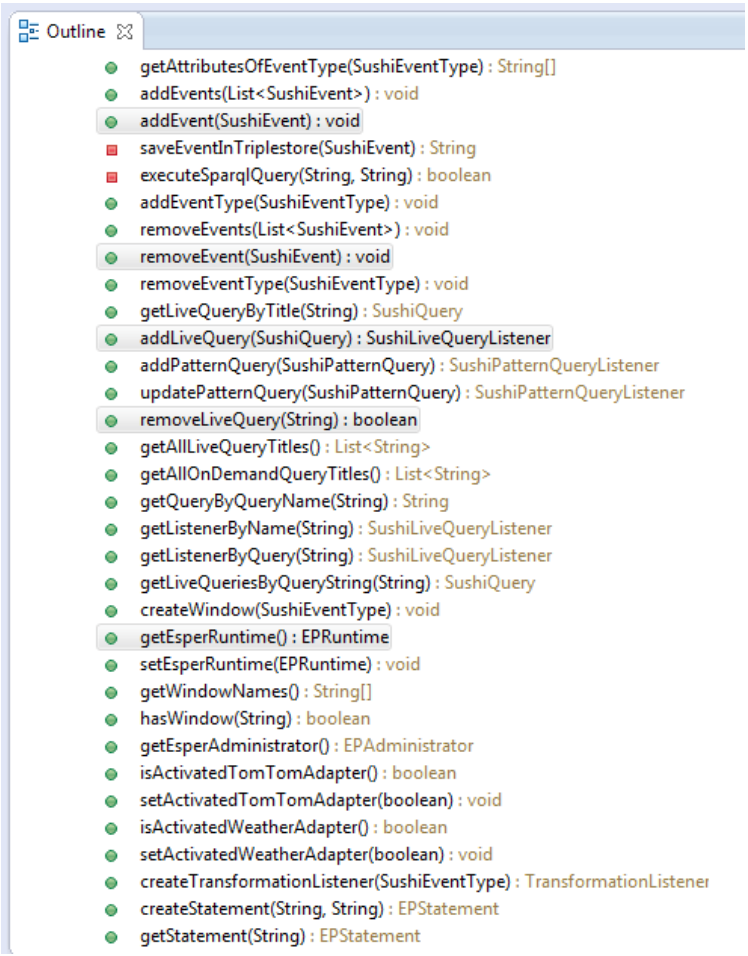
Integration of a new feature

This documentation is meant for developers and provides pointers to the existing source code. This shall enable developers to identify the right spots in the code to start the integration of their new feature. This document will explicitly talk about access to the event stream and integration into the existing database. The displayed examples are in no way enforced best practices but rather hints on how to start the integration.

Access to the event stream

One of the most common tasks when integrating a new feature is accessing the event stream and processing its events. One starting point in working with the incoming events can be found in the *StreamProcessingAdapter*[1].

Besides other features, it provides access to the *esperRuntime* and lets you add/remove events and queries to/from the platform (see class outline below).



[1] to be found in EapEventProcessing > src/main/java/hpi/eap/esper/StreamProcessingAdapter.java.

Assuming you want to execute a routine for EVERY event that is added to the platform[2], the *addEvent* method would be one spot you could add a call to your feature's routines to.

The *addEvent* method is mostly used by the *Broker*[3]. This class is a central mediator that forwards requests for event addition/removal to the *StreamProcessingAdapter*. Furthermore, it wraps other features around the event addition/removal, e.g. trigger a correlation for the event when it is added.

The broker is the medium to go through when events should be removed from or added to the platform, e.g. when importing events through a file upload. The broker methods itself are again another central point to add routines to that should be executed for all events that enter the platform or are supposed to be removed from it.

With these two classes as starting points, you can set your own hooks in the event processing procedure and manipulate it, e.g. by only allowing those events to pass on to the platform that fulfill certain criteria.

Other starting points for features related to event manipulation and processing are:

1. The *EapEvent* itself[4].
2. The existing importers[5] (for further extension or as examples for new importers).

Integration into the existing database

In order to add new data to the existing MySQL database, there are several steps to pay attention to.

Assuming you want to allow the user to add configuration files (e.g. via webservice call) that you need later in the process of evaluating incoming events. Assuming you added a method for the upload (e.g. by adding a new method to the webservice[6]), you need to create an object from the attributes the user gave you. One way of doing so would be a new Java class representing your configuration files. An example for such a class can be found in the

[2] Be aware that this provides overhead to the event processing since every event will go through this routine.
[3] To be found in SushiEsper > src/main/java/sushi/eventhandling/Broker.java.
[4] To be found in SushiCommon > src/main/java/sushi/event/SushiEvent.java.
[5] To be found in SushiImport > src/main/java/sushi/json/importer/JSONImporter.java.
[6] To be found in SushiWicket > src/main/java/de/hpi/EventProcessingPlatformWebservice.java.

EventMapping[7]class. As shown below, your class needs to extend the *Persistable* class. Furthermore, you can provide a name for the table that holds all individuals of this class in the database (i.e. provide a JPA annotation *@Table* above the class). Your class and its attributes need to be annotated according to the official JPA guide[8] (see an example below).

```
@Entity
@Table(name="EventMapping")
public class EventMapping extends Persistable{

    private static final long serialVersionUID = -7405308825216881596L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    protected int ID;

    @Basic(optional=false)
    @Column(unique=true,name="associated_event_type_name")
    protected String associatedEventType;

    @ElementCollection
    @MapKeyColumn(name="event_attribute")
    @Column(name="semantic_uri")
    @CollectionTable(name="mapping_values",joinColumns=@JoinColumn(name="mapping_id"))
    protected Map<String, String> mappingValues;

    public EventMapping() {}

    public EventMapping(Map<String, String> mappingValues, String associatedEventType) {
        this.mappingValues = mappingValues;
        this.associatedEventType = associatedEventType;
    }
```

When your new class is ready for persistence, you need to add it to the persistence configuration of the platform[9]. This ensures that all tables are created in the schema of the database on platform initialization. Please refer to the official developer guide of the platform for information on how to set up the persistence configuration in general. To add your class to the database schema, simply add a line `<class>hpi.eap.semantic.EventMapping</class>` (adjusted to your new class) to the persistence configuration to all other class definitions.

You can now use your persisted configurations in the code by retrieving them from the database. For that matter, you can either use native SQL queries addressing your defined table name in the class or use *Java Persistence Queries*[10] referring to the class name itself.

[7] To be found in EapSemantic > src/main/java/hpi/eap/semantic/EventMapping.java.
[8] To be found at http://www.objectdb.com/api/java/jpa/annotations .
[9] To be found in EapCommons > src/main/resources/META-INF/persistence.xml.
[10] To be found at http://docs.oracle.com/javaee/6/tutorial/doc/bnbtg.html .