IBM Proactive Technology Online (Proton) Programmer Guide

IBM Research – Haifa

Version 1.0.4: October 2014

Contents

Proton architectural principles

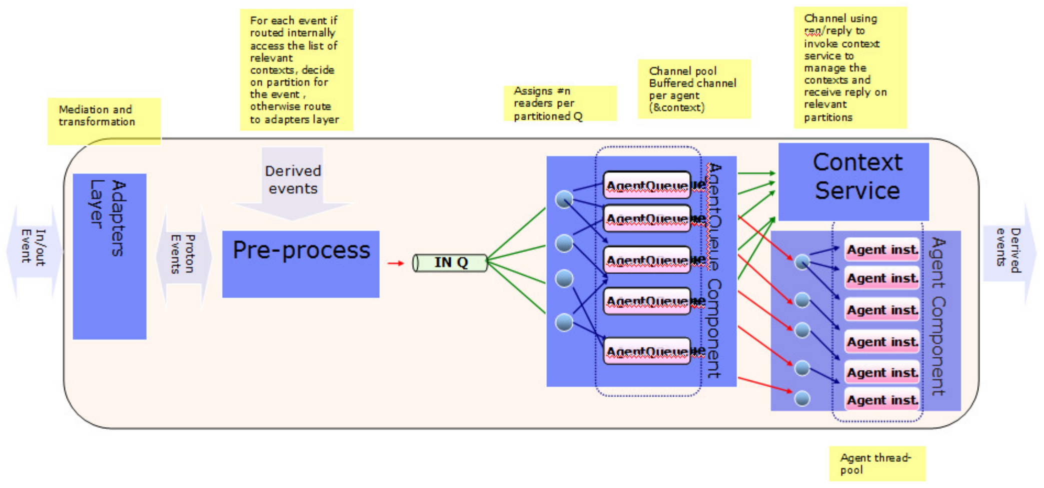
Proton conceptual architecture

The principles behind Proton architecture ensure scalable, fault-tolerant, parallel architecture while ensuring execution according to correctness schemes.
When we talk about parallel architecture we are trying to address the following issues:

- Partitioning of event load (processing multiple event instances simultaneously)
- Partitioning of logic (processing the same event instance simultaneously by different agents)

The challenge is that we need both sharding (data partitioning between different system instances to handle multiple independent operations simultaneously) and logic partitioning while two of those are co-dependent
We need to allow partitioning of event load and distribution of logic while taking into account that both the logic and event instances are co-dependent.
The architecture also ensures separation of logic and environment-specific services, to allow single Proton logic core which will be capable to run in different environments.

High level Architecture


Mediation and transformation
For each event if routed internally access the list of relevant contexts, decide on partition for the event , otherwise route to adapters layer
Assigns #n readers per partitioned Q
Channel pool Buffered channel per agent (&context)
Channel using req/reply to invoke context service to manage the contexts and receive reply on relevant partitions
In/out Event
Adapters Layer
Proton Events
Derived events
Pre-process
IN Q
AgentQueue
AgentQueue
AgentQueue
AgentQueue
AgentQueue
AgentQueue Component
Context Service
Agent inst.
Agent inst.
Agent inst.
Agent inst.
Agent inst.
Agent inst.
Agent Component
Derived events
Agent thread-pool

The incoming events are handled by multiple routing threads routing the events to relevant agent queues - according to the agent type an event is intended for and/or context type an event is intended for. After buffering time (either minimal or correctness-based) of the agent queue expires the event is handled by processing threads, sent to partitioning by context service and sent to EPA (Event Processing Agent) manager for processing by particular agent instance according to the agent's type and context partition.
This allows for: 1) handling of the same event instances simultaneously by different agents 2) handling multiple incoming instances in parallel but 3) saving order (either detection or occurrence) between dependent event instances.
Since we handle all relevant events for the same context partition in the same node we have data locality.

Derived events are driven back to the system in case they also act as input events to additional agents within EPN (Event Processing Network), and/or to output adapters for derived events intended for consumer

Proton Adapters

Overview

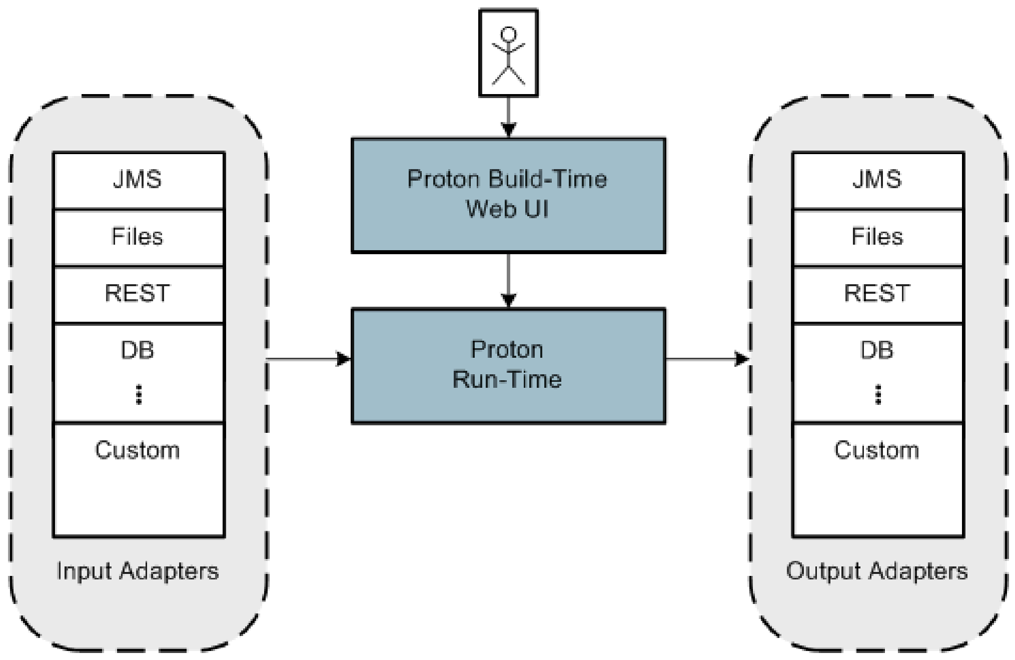


Figure 1: Adapters and Proton runtime

Proton semantic layer allows the user to define producers and consumers for event data (see Figure 1). Producers produce event data, and consumers consume the event data. The definitions of producers and consumer, which is specified during the application buildtime are translated into input and output adapters in Proton execution time.

The physical entities representing the logical entities of producers and consumers in Proton are adapter instances.

The adapters are environment-agnostic. The adapters use the environment-specific connectivity layer to connect to environment-specific Proton implementation.

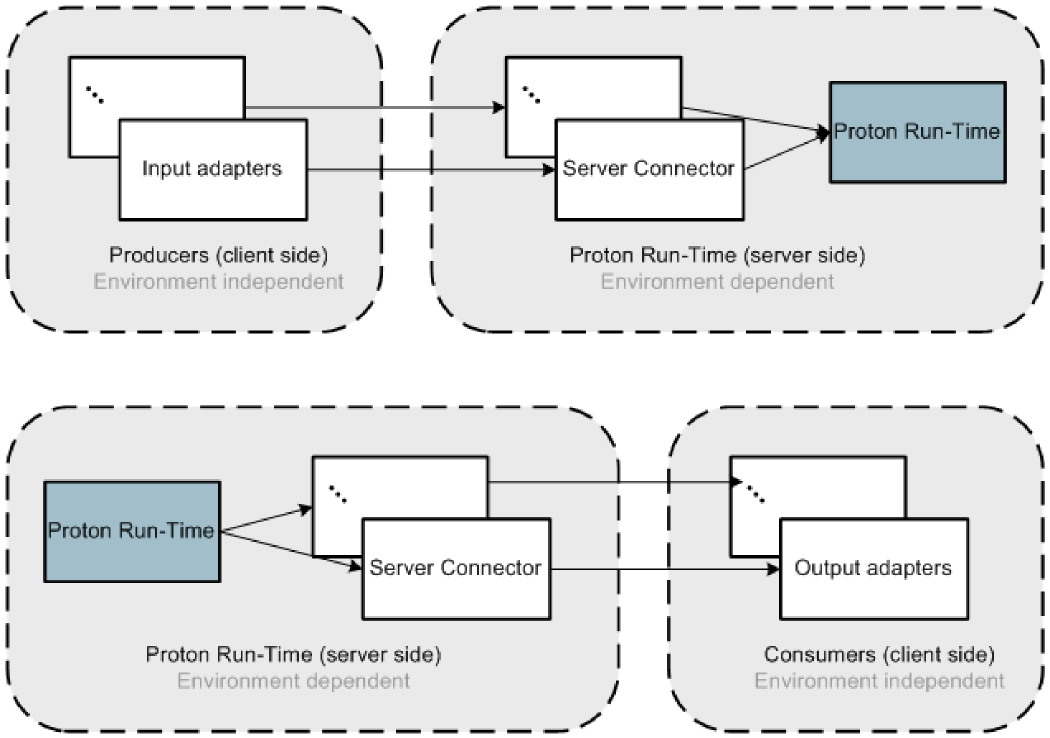


Figure 2: Adapters layer representation

As can be seen in Figure 2, for each producer an input adapter is defined, which defines how to pull the data from the source resource, how to format the data into Proton's object format before delivering it to the engine. The adapter is environment-agnostic but uses the environment-specific connector object, injected into the adapter during its creation, to connect to Proton runtime.

The consumers and their respective output adapters are operated in a push mode – each time an event is published by the runtime it is pushed through environment-specific server connectors to the appropriate consumers, represented by their output adapters, which publish the event in the appropriate format to the designated resource.

The server connectors are environment-specific, they hide the implementation of the connectivity layer from the adapters which allows them to be environment-agnostic.

The J2SE implementation of Proton runtime includes an input and output socket servers, which allow the input and output server connectors to communicate with the runtime using sockets mechanism.

Adapters design principles

As part of the Proton application design the user specifies the events producers as sources of event data and the event consumers as sinks for event data.
The specification of producer includes the resource from which the adapter pulls the information (whether this resource is a database, a file in a file system, a JMS queue, REST), and format settings which allow the adapter to transform the resource specific information to Proton event data object. The formatting depends on the kind of

resource we are dealing with – for file it can be a tagged file formatter, for JMS an object transformer.
For example, in a tag format, when the user defines the delimiter as ";" and the tagDataSeparator as "=", an event of type *ShipPosition* with attributes *ShipId, Long, lat* and *Speed* will be expected to arrive in the following format:
`Name=ShipPosition;ShipID=RTX33;Long=46;Lat=55;Speed=4.0;`

Likewise, the specification of consumer includes the resource to which the event created by Proton runtime should be published and a formatter describing on how to transform a Proton event data object into resource-specific object.

The design of adapter's layer satisfies the following principles:

- A producer is a logical entity which holds such specifications as the source of the event data, the format of the event data. The input adapter is the physical entity representing a producer, an entity which actually interacts with the resource and communicates event information to Proton runtime server.
- A consumer is a logical entity which holds such specifications as the sink for the vent data, the format of the sink event data. The output adapter is the physical representation of the consumer, it is an entity which is invoked by the Proton runtime when an event instance should be published to the resource.
- The input adapters all implement a standard interface, which is extendable for custom input adapter types and which allows to add new producers for custom-type resources.
- The output adapters all implement a standard interface, which is extendable for custom output adapter types and which allows to add new consumers for custom-type resources.
- A single event instance can have multiple consumers
- A producer can produce events of different types, a single event instance might serve as input to multiple logical agents within the event processing network, according to the network's specifications
- Producers operate in pull mode, each input adapter pulls the information from designated resource according to its specifications, each time processing the incremental additions in the resource.
- Consumers define a list of event types they are interested in, they can also specify a filter condition on each event type – only event instances satisfying this condition will be actually delivered to this consumer.
- Consumers operate in push mode, each time the Proton runtime publishes an event instance it is pushed to the relevant consumer.
- The producers and consumers are not directly connected, but the raw event's data supplied by a certain producer can be delivered to a consumer if the consumer specifies this event type in it's desirable events list.

Adapters design

A user defines a producer or consumer to act as event data supplier or consumer, he does so using Proton's buildtime tooling to create the appropriate metadata. The metadata defines the access information to the event data source or sink, and the information on how to format the data to/from Proton readable event object.
The adapters framework is built on the notion of extending a common abstract adapter (one for all input adapters and one for all output adapters). The adapter framework

provides all the sequence of adapter's lifecycle management (initialization, establishing connection to the server, processing of data, and shutdown) ,all that each specific adapter implementation have to supply is resource-specific implementations of **readData()** or **writeObject()** methods in order to pull the data from this resource or push the data to the resource (see Figure 3 for class diagram of the framework), as well as **createConfiguration()** adapter-specific method, which fetches the required adapter-specific properties from producer/consumer metadata and creates appropriate configuration object.

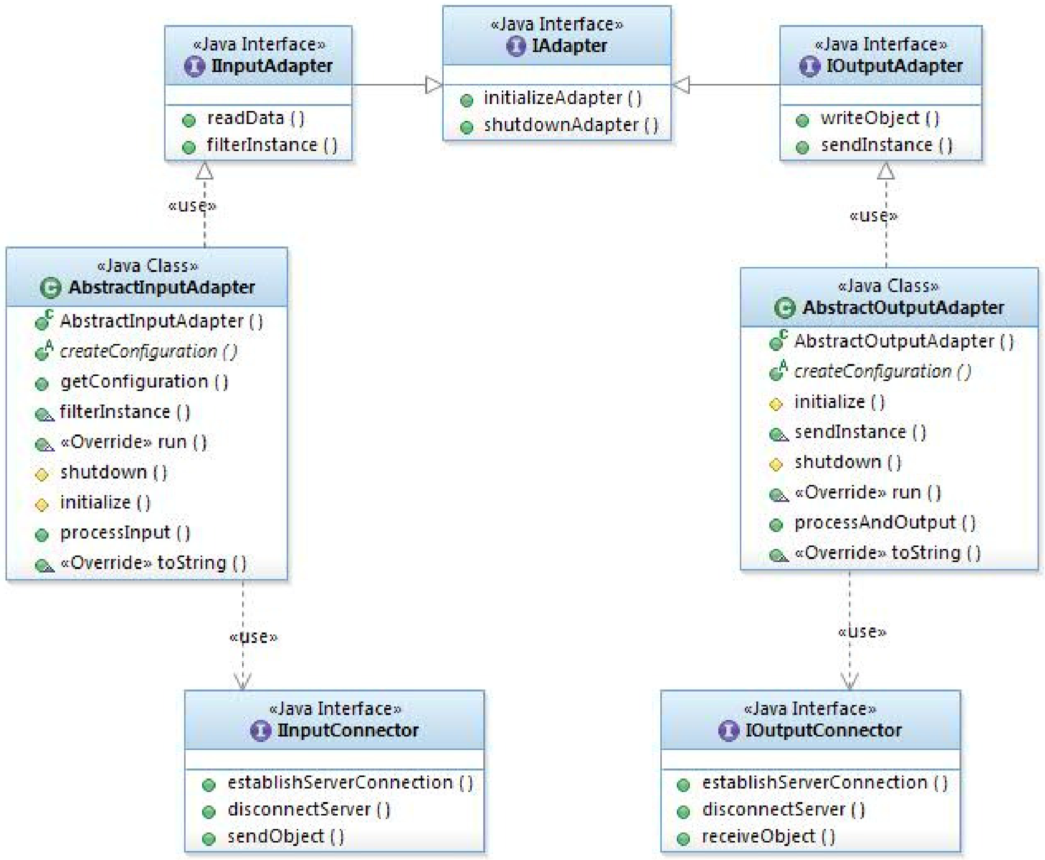


Figure 3 – Adapters framework class diagram

Input Adapters

Runtime

The adapter representing the producer is configured, upon startup it is supplied with server connector which handles all communication of the adapter with Proton runtime (see Figure 4 for initialization sequence diagram).

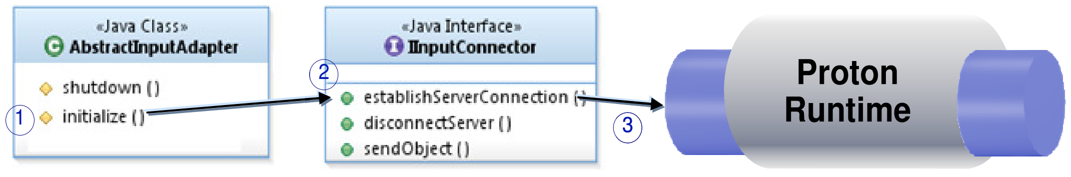


Figure 4 – Adapter initialization sequence-establishing connection to Proton server

Once the adapter starts running (see Figure 5 for processing sequence), it constantly polls the data source for changes (1,2), transforms an entry within the data source into Proton readable event object (3), and sends the information via server connector to the server (4), without being aware of the underlying communication infrastructure which the connector uses to establish the connection and send the data to the server.

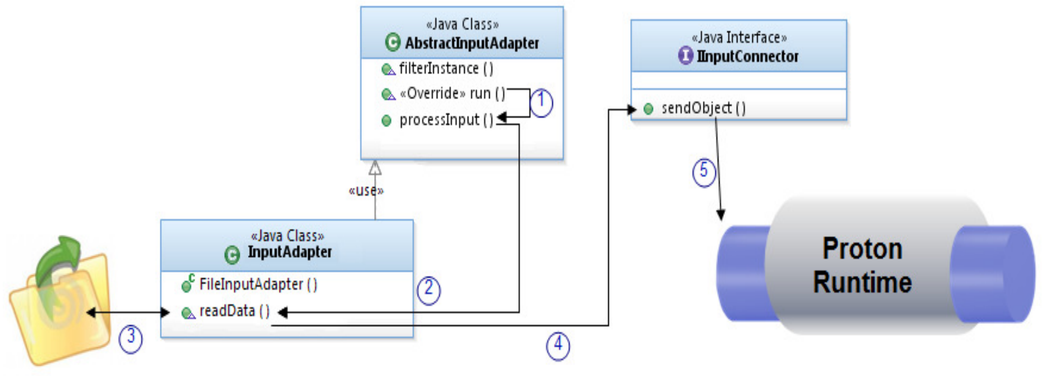
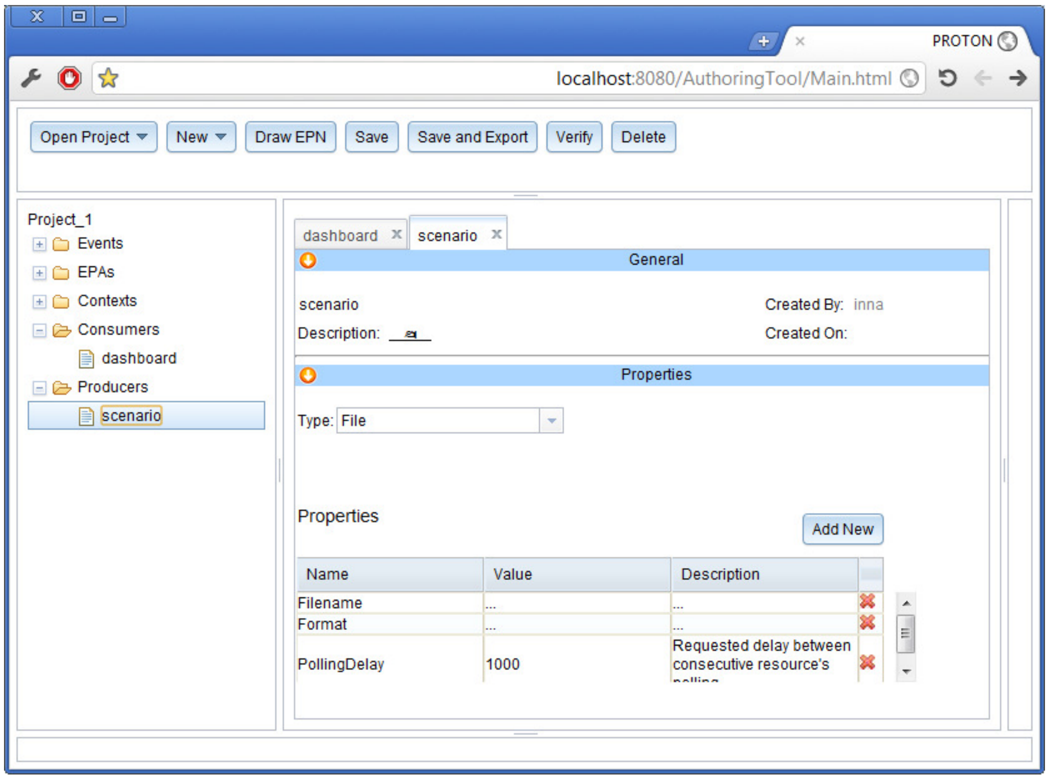


Figure 5 – Input adapter processing sequence

Definition

In order to define a producer we use the buildtime tool to choose the producer type and define suitable properties. Figure 6 depicts a producer definition screen in the Proton authoring tool.



scenario

Description:

Created By: inna

Created On:

Type: File

Properties

Name	Value	Description
Filename	...	...
Format	...	...
PollingDelay	1000	Requested delay between consecutive resource's polling

Figure 6 – buildtime representation of a producer definition

In order to define a producer we need to supply the following information:

- The general metadata including the name of the producer, the description, the createdDate and createdBy information.
- The type of the producer – at the moment supporting FILE , JMS adapters, planned support for REST, DB and custom adapters.
- Per specific chosen type – list of properties specifying the resource to access, the credentials to access the resource, and the formatter information
    - **File adapter: (see Figure 7)** the relevant properties include
        - The absolute path to the file representing the resource this file adapter is polling
        - Sending delay between sending available event instances to the system
        - Polling interval – the interval for two consecutive polls of the resource for updates
        - Formatter type for the entries within the file : at the moment supporting tag-delimited formatter
        - Properties of the tag-delimited formatter, including the delimiter ,and the tag-data separator characters

```
{"name":"scenario","type":"file","properties":
[
    {"name":"filename","value":"C:\\scenario.txt"},
    {"name":"sendingDelay","value":"5000"},
    {"name":"pollingInterval","value":"1000"},
    {"name":"formatter","value":"tag"},
    {"name":"delimiter","value":";"},
    {"name":"tagDataSeparator","value":"="}
],
"desription":"","createdDate":"","createdBy": "inna"}
```

Figure 7 – producer of file type JSON definition

- **JMS adapter (see Figure 8)** : the relevant properties include
  - The hostname of the server where the input JMS destination resides
  - The port to connect to on the server where the input JMS destination resides
  - The JNDI name of the connection factory object
  - The JNDI name of the destination object
  - Timeout – the timeout to wait for a new object on the JMS destination
  - Sending delay between sending available event instances to the system
  - Polling interval – the interval for two consecutive polls of the resource for updates

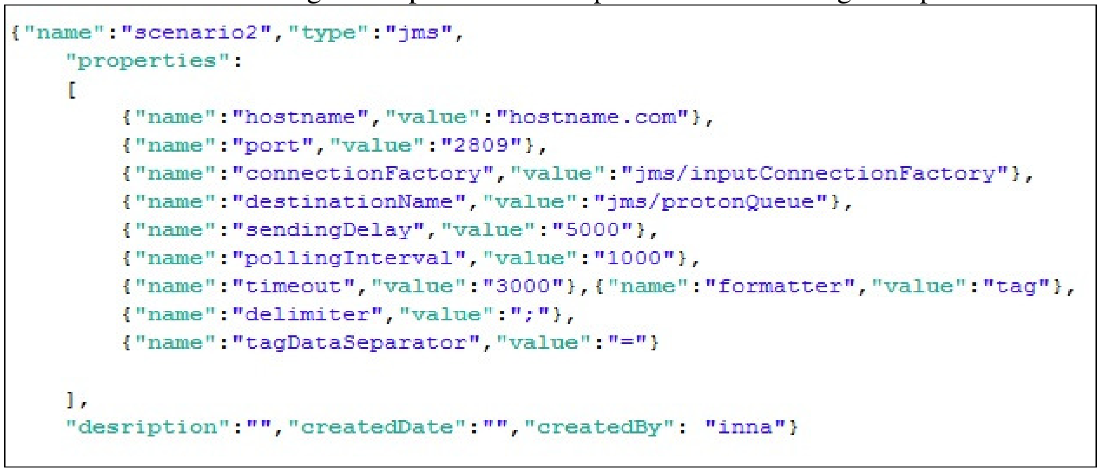
```
{"name":"scenario2","type":"jms",
    "properties":
    [
        {"name":"hostname","value":"hostname.com"},
        {"name":"port","value":"2809"},
        {"name":"connectionFactory","value":"jms/inputConnectionFactory"},
        {"name":"destinationName","value":"jms/inputQueue"},
        {"name":"sendingDelay","value":"5000"},
        {"name":"pollingInterval","value":"1000"},
        {"name":"timeout","value":"3000"}

    ],
    "desription":"","createdDate":"","createdBy": "inna"}
```

Figure 8- producer of type JMS object message JSON definition

If those are the only properties mentioned, the JMS producer assumes the JMS destination contains serializable objects which implement the **IObjectMessage** interface (see later in the description of interfaces)

We can specify additional options for the formatter, in which case the JMS adapter implementation assumes the JMS message is a tag-delimited text message with the specified formatting information. Additional properties for JMS producer which wishes to use formatted text messages are: (see Figure 9)

- Formatter – the formatter type (right now only tag-delimited messages are supported so the only option is 'tag')
- Delimiter – the delimiter string between the tag-data groups
- TagDataSeparator – the separator within the tag-data pair

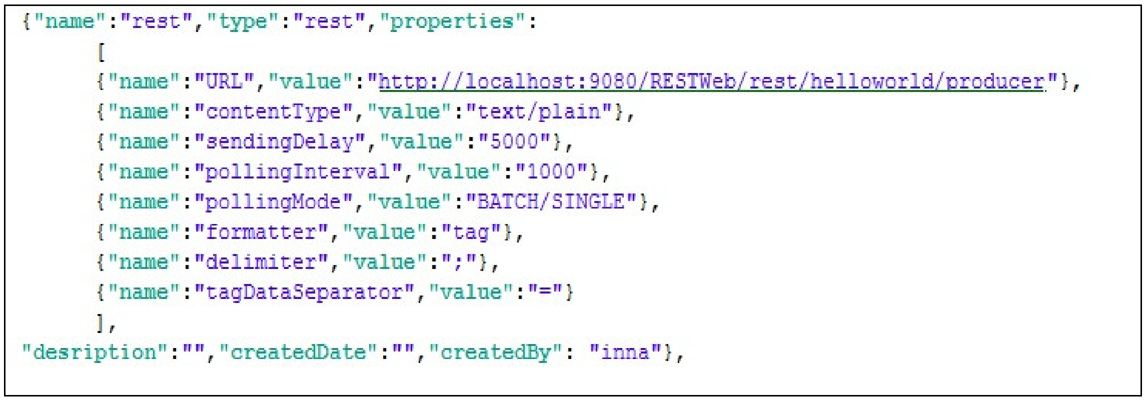


```
{"name":"scenario2","type":"jms",
    "properties":
    [
        {"name":"hostname","value":"hostname.com"},
        {"name":"port","value":"2809"},
        {"name":"connectionFactory","value":"jms/inputConnectionFactory"},
        {"name":"destinationName","value":"jms/protonQueue"},
        {"name":"sendingDelay","value":"5000"},
        {"name":"pollingInterval","value":"1000"},
        {"name":"timeout","value":"3000"},{"name":"formatter","value":"tag"},
        {"name":"delimiter","value":";"},
        {"name":"tagDataSeparator","value":"="}

    ],
    "desription":"","createdDate":"","createdBy": "inna"}
```

Figure 9 – producer of type JMS formatted text message JSON definition

The JMS producer supports the WebSphere 7.x api. In order to use the WebSphere api for JMS, WebShpere thin client libraries are needed to be added to the lib directory of Proton.

- **REST adapter** (see Figure 10), is a REST service provider that receives events from an external system (the producer) using the POST method. The relevant properties of this adapter include:
    - contentType - can be "text/plain", "text/xml", "application/xml", "application/json" etc. This defines the content the service accepts to receive and what formatter to apply.
    - Formatter properties - the same as in file. The user has to supply correct formatters to work with the defined content type, for example he will have to add formatters to deal with XML or JSON content. If the user defines formatter not suitable for contentType (for example he defines tag formatter for a content which is not plain text) he will get an exception.



```
{"name":"rest","type":"rest","properties":
      [
      {"name":"URL","value":"http://localhost:9080/RESTWeb/rest/helloworld/producer"},
      {"name":"contentType","value":"text/plain"},
      {"name":"sendingDelay","value":"5000"},
      {"name":"pollingInterval","value":"1000"},
      {"name":"pollingMode","value":"BATCH/SINGLE"},
      {"name":"formatter","value":"tag"},
      {"name":"delimiter","value":";"},
      {"name":"tagDataSeparator","value":"="}
      ],
"desription":"","createdDate":"","createdBy": "inna"},
```

Figure 10 – producer of type REST definition

Interfaces to implement

An input adapter needs to implement a few interfaces:

1. It should extend the **AbstractInputAdapter** abstract class, which in turn implements the **IInputAdapter** interface , the developer should provide implementation for the following methods:
   - **IInputAdapterConfiguration createConfiguration(ProducerMetadata metadata)** - creates the adapter-specific configuration object after extracting the relevant properties from the producer metadata object
   - **void initializeAdapter() -** a method existing in the abstract interface but which should be overloaded with any resource-specific initialization after the call to the parent's method (such as acquiring a handle to a file, opening a connection to datasource etc.)
   - **IEventInstance readData() –** which accesses the resource, pulls the event data and transforms each event data entry into Proton event instance object.
   - **void shutdownAdapter() –** a method existing in the abstract interface but which should be overloaded with any resource-specific shutdown actions after the call to the parent's method (such as closing a handle to a file, closing a connection to datasource etc.)
2. It should provide an adapter specific implementation of **IInputAdapterConfiguration** interface for an adapter-specific configuration object. This is basically a bean with getters method carrying adapter specific information which helps the adapter to access the specific resource (such as filename for file adapter, or hostname and port name for JMS server for the JMS adapter)

Output Adapters

Runtime

The adapter representing the consumer is configured, upon startup it is supplied with server connector which handles all communication of Proton runtime with the adapter (see Figure 11 for initialization sequence diagram).

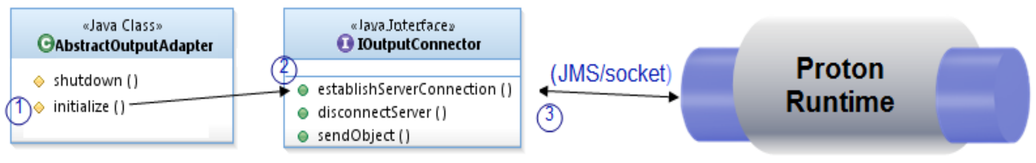


Figure 11 – Adapter initialization sequence-establishing connection to Proton server

The Proton runtime pushes all published events for the specific consumer to the consumer's connector object, where it is stored in the queue. (see Figure 12 step 1a). The output adapter accesses the queue and pulls the event objects from the queue. Once the adapter starts running (see Figure 12 for processing sequence), it constantly polls the server connector for new published event objects (1b,2), transform the event data entry received from Proton runtime into resource-specific format(3), and writes the transformed object to the destination resource (4).

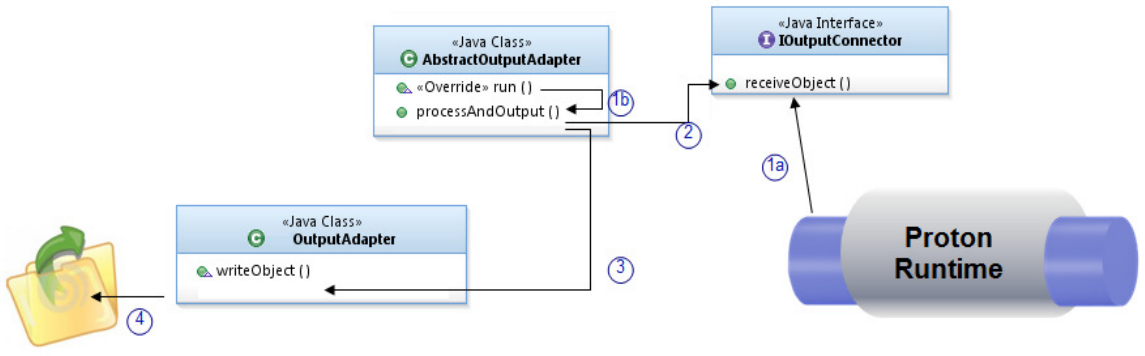
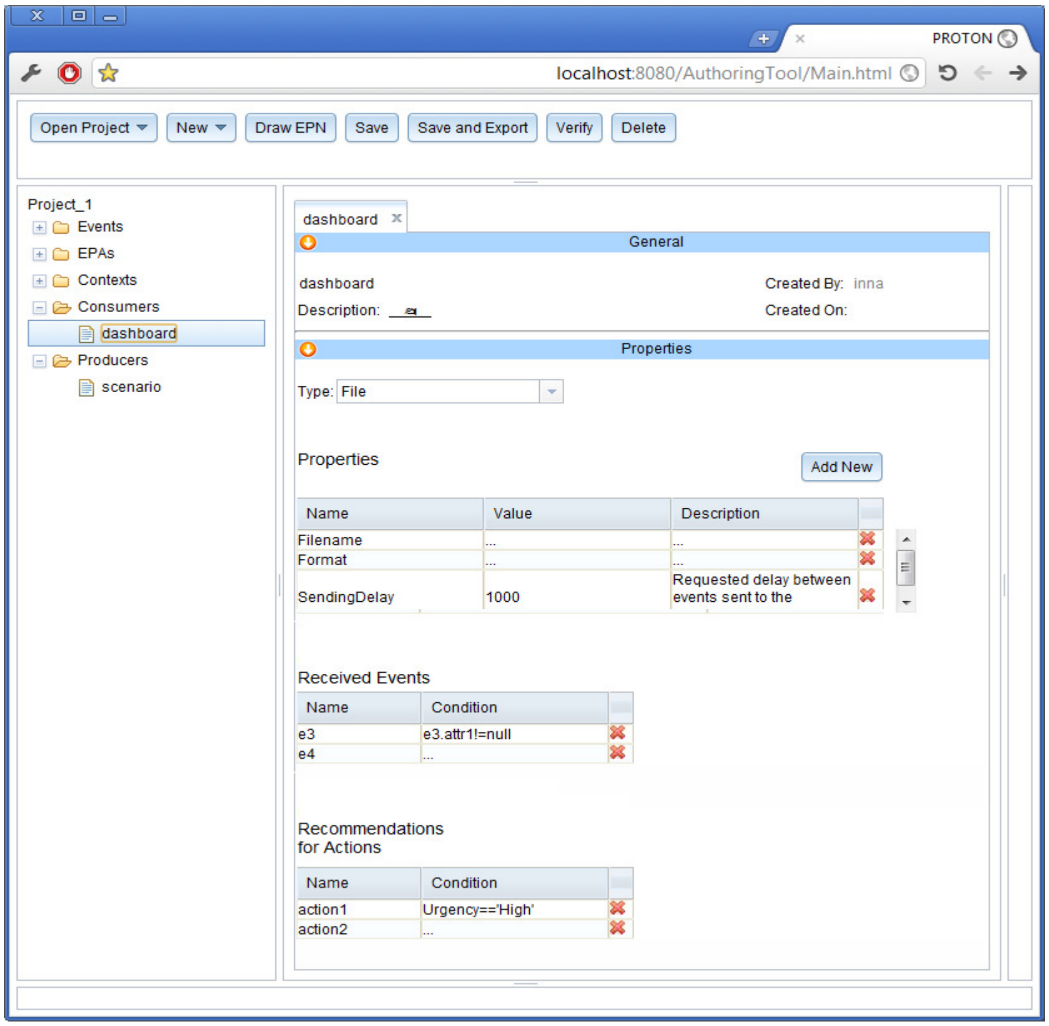


Figure 12– Output adapter processing sequence

Definition

In order to define a consumer we use the buildtime tool to choose the consumer type and define suitable properties. Figure 13 depicts a consumer definition screen in the Proton authoring tool.



dashboard

Description:

Created By: inna

Created On:

Type: File

Properties

Name	Value	Description
Filename	...	...
Format	...	...
SendingDelay	1000	Requested delay between events sent to the

Received Events

Name	Condition
e3	e3.attr1!=null
e4	...

Recommendations for Actions

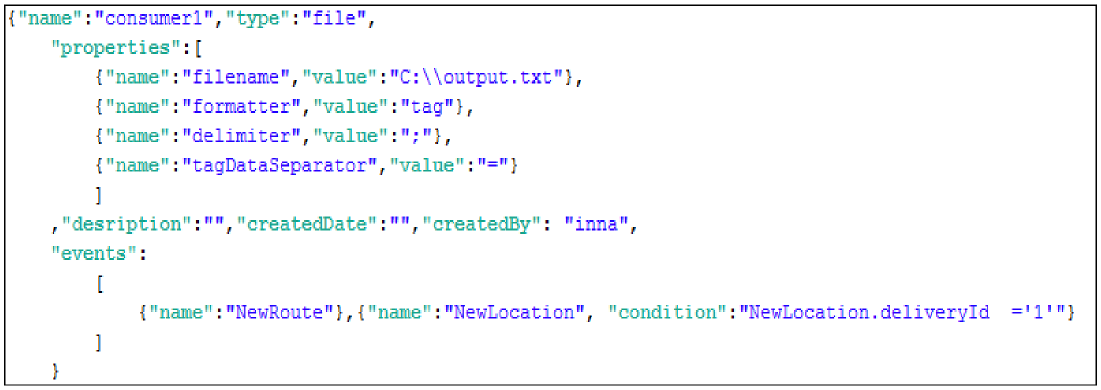
Name	Condition
action1	Urgency=='High'
action2	...

Figure 13– buildtime representation of a consumer definition

In order to define a consumer we need to supply the following information:

- The general metadata including the name of the consumer, the description, the createdDate and createdBy information.
- The type of the consumer – at the moment supporting FILE , JMS adapters, planned support for REST, DB and custom adapters.
- The list of all event types this consumer is interested to receive. For each event a filtering condition might also be specified – only instances satisfying this condition will be delivered to the consumer
- Per specific chosen type – list of properties specifying the resource to access, the credentials to access the resource, and the formatter information
    - **File adapter**: (see Figure 14) the relevant properties include
        - The absolute path to the file representing the resource this file adapter is writing to
        - Formatter type for the entries within the file : at the moment supporting tag-delimited formatter
        - Properties of the tag-delimited formatter, including the delimiter ,and the tag-data separator characters

        - A list of event types (either raw or derived) which should be delivered to this consumer

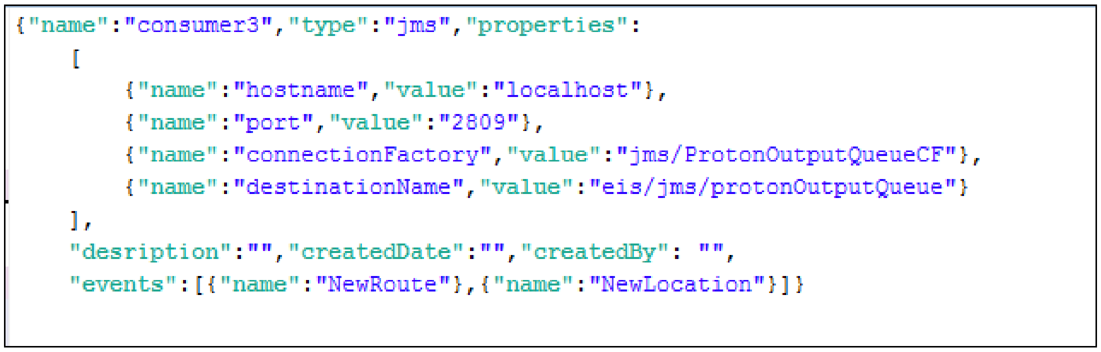


```
{"name":"consumer1","type":"file",
    "properties":[
        {"name":"filename","value":"C:\\output.txt"},
        {"name":"formatter","value":"tag"},
        {"name":"delimiter","value":";"},
        {"name":"tagDataSeparator","value":"="}
        ]
    ,"desription":"","createdDate":"","createdBy": "inna",
    "events":
        [
            {"name":"NewRoute"},{"name":"NewLocation", "condition":"NewLocation.deliveryId  ='1'"}
        ]
    }
```

Figure 14 – consumer of file type JSON definition

    - **JMS adapter** (see Figure 15) : the relevant properties include
        - The hostname of the server where the output JMS destination resides
        - The port to connect to on the server where the output JMS destination resides
        - The JNDI name of the connection factory object
        - The JNDI name of the destination object
        - A list of event types (either raw or derived) which should be delivered to this consumer

.



```
{"name":"consumer3","type":"jms","properties":
    [
        {"name":"hostname","value":"localhost"},
        {"name":"port","value":"2809"},
        {"name":"connectionFactory","value":"jms/ProtonOutputQueueCF"},
        {"name":"destinationName","value":"eis/jms/protonOutputQueue"}
    ],
    "desription":"","createdDate":"","createdBy": "",
    "events":[{"name":"NewRoute"},{"name":"NewLocation"}]}
```

Figure 15- consumer of type JMS object message JSON definition

If those are the only properties mentioned, the JMS consumer assumes the JMS destination will consume serializable objects which implement the **IObjectMessage** interface (see later in the description of interfaces) and creates an implementation instance of such interface which it places on the JMS destination.
We can specify additional options for the formatter, in which case the JMS adapter implementation assumes the JMS message is a tag-delimited text message with the specified formatting information.
Additional properties for JMS consumer which wishes to write formatted text messages to the JMS destination are: (see Figure 16)

- Formatter – the formatter type (right now only tag-delimited messages are supported so the only option is 'tag')
- Delimiter – the delimiter string between the tag-data groups
- TagDataSeparator – the separator within the tag-data pair

```
{"name":"consumer3","type":"jms","properties":
    [
        {"name":"hostname","value":"localhost"},
        {"name":"port","value":"2809"},
        {"name":"connectionFactory","value":"jms/ProtonOutputQueueCF"},
        {"name":"destinationName","value":"eis/jms/protonOutputQueue"},
        {"name":"formatter","value":"tag"},
        {"name":"delimiter","value":";"},
        {"name":"tagDataSeparator","value":"="}
    ],
    "desription":"","createdDate":"","createdBy": "",
    "events":[{"name":"NewRoute"},{"name":"NewLocation"}]}
```

Figure 16 – consumer of type JMS formatted text message JSON definition

- **REST adapter** (see Figure 17) , is a REST web-service client, which is capable to access the web-service declared by the consumer and push Proton events into it. The relevant definitions include:

▪ URL - the fully qualified URL of the web service for event push operation.
▪ contentType - can be "text/plain", "text/xml", "application/xml", "application/json" etc. This is basically defined by the web service and have to be entered here so the client knows how to access the web service.
▪ Formatter properties - the same as in file.
▪ Properties of the tag-delimited formatter, including the delimiter ,and the tag-data separator characters
▪ A list of event types (either raw or derived) which should be delivered to this consumer

```
    {"name":"consumer3","type":"rest","properties":
          [
          {"name":"URL","value":"http://localhost:9080/RESTWeb/rest/helloworld/consumer"},
          {"name":"contentType","value":"text/plain"},
          {"name":"formatter","value":"tag"},
          {"name":"delimiter","value":";"},
          {"name":"tagDataSeparator","value":"="}
          ],
    "desription":"","createdDate":"","createdBy": "inna",
    "events":[
          {"name":"NewRoute"},{"name":"PredictedArrivalTime"},{"name":"Reroute"},
          {"name":"BeginShipment"},{"name":"TrafficAlert"},{"name":"FlightCargoUpdate"}
          ]
    }
```

Figure 17 – REST consumer

Interfaces to implement

An output adapter needs to implement a few interfaces:

3. It should extend the **AbstractOutputAdapter** abstract class, which in turn implements the **IOutputAdapter** interface , the developer should provide implementation for the following methods:
    - **IOutputAdapterConfiguration createConfiguration(ConsumerMetadata metadata)** - creates the adapter-specific configuration object after extracting the relevant properties from the consumer metadata object

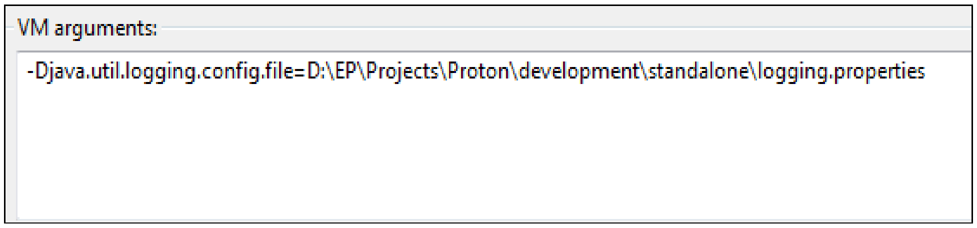
- **void initializeAdapter() -** a method existing in the abstract interface but which should be overloaded with any resource-specific initialization after the call to the parent's method (such as acquiring a handle to a file, opening a connection to datasource etc.)
- **void writeObject(IDataObject dataObject) –** which takes the Proton data object, transforms it to resource-specific format and writes it to the resource represented by this consumer.
- **void shutdownAdapter() –** a method existing in the abstract interface but which should be overloaded with any resource-specific shutdown actions after the call to the parent's method (such as closing a handle to a file, closing a connection to datasource etc.)

4. It should provide an adapter specific implementation of **IOutputAdapterConfiguration** interface for an adapter-specific configuration object. This is basically a bean with getters method carrying adapter specific information which helps the adapter to access the specific resource (such as filename for file adapter, or hostname and port name for JMS server for the JMS adapter)

Configuration

Proton can run on Apache tomcat server or as a standalone engine. The configuration of Proton when running on the Apache tomcat server is described in the "Installation and Administration Guide". The standalone configuration is described here. This configuration consists of the following files:

- **Proton.properties** – the runtime looks for this file under ./config directory , or an absolute path to the file can be specified when invoking Proton runtime as an argument. The file contains the following properties:
  - **metadataFileName –** the name of the JSON metadata file containing Proton EPN definitions
  - **inputPortNumber** – the port number for SocketServer for input adapters (this is the port number through which input adapters will communicate with Proton runtime) . Can be any free port in the system
  - **outputPortNumber –** the port number for SocketServer for output adapters (this is the port number through which output adapters communicate with Proton runtime) . Can be any free port in the system
- **logging.properties –** can be found under ./config directory. This is the properties file for java logging API. The most relevant entries within the file is the properties regarding ConsoleHandler and FileHandler. ConsoleHandler properties define how console logging is handled, specifically **java.util.logging.ConsoleHandler.level** property defines the logging level of messages which are displayed on the console, the default value is INFO. The file handler properties define where the log file is stored, what is the logging level to the file etc. Those properties can be used to control the logging of Proton runtime. In order to let Proton use this file instead of default logging properties, run the Proton jar with the following VM argument:


VM arguments:

```
-Djava.util.logging.config.file=D:\EP\Projects\Proton\development\standalone\logging.properties
```

You can specify either a relative or an absolute path to the logging.properties file.

Metadata

The Proton metadata file is JSON file created by Proton Authoring tool (see Proton User Guide for usage instructions). The JSON contains all EPN definitions, including definitions for event types, action types, EPAs, contexts, producers and consumers. A JSON schema defines the JSON format expected by Proton to allow creating Proton metadata file programmatically.
When Proton runtime jar starts running , it accesses the metadata file, loads and parses all the definitions , creates thread per each input and output adapter and starts listening for events incoming from the input adapters (aka "producers") and forwarding events to output adapters (aka "consumers") .