norther.org

Tammi Application Framework

Service API Specification

Version 6.1.13

Revision History

Date	***Version***	***Description***	***Author***
10.08.09	6.1.0	Initial version for release 6.1	imp
23.10.09	6.1.1	Updated with meta-data	imp
23.10.09	6.1.2	Attribute trait descriptions	imp
23.10.09	6.1.3	XML schemata added	imp
26.10.09	6.1.4	Data type descriptions	imp
26.10.09	6.1.5	KML integration	imp
29.10.09	6.1.6	Variable index added	imp
09.11.09	6.1.7	KML extended data revised	imp
11.11.09	6.1.8	Response schema added	imp
18.11.09	6.1.9	Parent flow added	imp
26.11.09	6.1.10	Variable validation and flow labels	imp
01.12.09	6.1.11	KML content type added	imp
12.02.10	6.1.12	Attribute name and type added	imp
23.03.10	6.1.13	More error handling added	imp

Table of Contents

1 Introduction

The Tammi Application Framework is a Java™ component based development framework and run-time container for applications supporting web browsers, mobile terminals and/or JFC based user interfaces. Tammi application components can implement independent business logic themselves or act as proxies to native libraries, remote programs and other kinds of manageable resources.

1.1 Purpose

This document contains the service API specification of Tammi for programmatic access. The technical specification of Tammi [Spec] provides a more comprehensive description of the structure and functionality of the Tammi framework itself.

1.2 Scope

The scope of this document is to describe mechanisms, structures and formats of how remote clients may programmatically access applications and services running on top of Tammi.

1.3 Definitions, Acronyms and Abbreviations

Term	***Explanation***
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JMX	Java™ Management Extensions
JRE	Java™ Runtime Environment
JSON	JavaScript Object Notation
JVM	Java™ Virtual Machine
KML	Keyhole Markup Language
MBean	Managed Bean (a specific Java class)
MVC	Model – View – Controller design pattern
REST	Representational State Transfer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VDEML	Variable Data Exchange Markup Language
WML	Wireless Markup Language
WUI	Web User Interface
XHTML	Extensible Hypertext Markup Language
XML	Extensible Markup Language

1.4 References

[Java]	Name Link	*Java™ 2 Platform, Standard Edition* http://java.sun.com/j2se/index.html

[JMX]	Name Link	*Java™ Management Extensions (JMX™)* http://java.sun.com/products/JavaManagement/index.html
[JMX-Spec]	Name Link	*JMX™ Instrumentation and Agent Specification* http://jcp.org/aboutJava/communityprocess/final/jsr003/
[JSON]	Name Link	*Introducing JSON* http://www.json.org/
[KML]	Name Link	*KML* http://code.google.com/intl/fi-FI/apis/kml/
[Multipart]	Name Link	*Returning Values from Forms: multipart/form-data* http://www.ietf.org/rfc/rfc2388.txt
[REST]	Name Link	*Representational State Transfer* http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
[Servlet]	Name Link	*Java™ Servlet Technology* http://java.sun.com/products/servlet/index.html
[Spec]	Name Link	*Tammi Technical Specification* http://tammi.sourceforge.net/pdf/tammi-spec.pdf
[URI]	Name Link	*Uniform Resource Identifier (URI): Generic Syntax* http://www.ietf.org/rfc/rfc3986.txt

1.5 Overview

The rest of the document provides a technical description of the Tammi service API including an architecture overview, control commands, data exchange formats and GIS integration.

2 Architecture

Tammi's architecture is compliant with Java™ Management Extensions [JMX-Spec]. Applications are formed by independent components plugged into Tammi, and configured dynamically during start-up and run-time. Applications follow the Model – View – Controller (MVC) design pattern separating presentation from content. The presentation is implemented as templates in some markup language interacting with content producers through context tools within template contexts. A template engine parses the templates producing either user interface pages for web browsers or data descriptions for further processing. A filter chain mechanism [Servlet] controls the process. The templates are organized into page flow steps managed by flow filters. The data content of each step is presented as a set dynamic variables with a number of named attributes of various types. The variables may be either local ones or imported from a persistent storage, such as a relational database.

2.1 JMX™ Managed Beans

In JMX™ [JMX], application components are implemented as manageable resources. The instrumentation of a given resource is provided by one or more Managed Beans, or MBeans, which are Java objects implementing a specific interface describing the attributes and operations they provide for accessing the resources behind them.

The basic functionality of the Tammi framework is implemented as MBeans. Third-party libraries has been integrated to the framework through proxy MBeans. Correspondingly, new application components may be implemented directly as MBeans, or they can be integrated through a proxy layer. A generic adapter MBean may be applied to introduce any Java class to the framework without modifying the code of the original class.

2.1.1 Foundation Services

Tammi foundation services comprises of base services required for Tammi to operate, and supporting services providing common functionality for applications. Base services include

- `Broker` supports both queries to any services registered to the MBean server and direct shortcuts to services having an explicit alias maintained in a service registry of the broker itself.
- `Converter` converts objects from different class types to others.
- `Domain` defines the default domain applied by Tammi in the MBean.
- `Factory` provides a standard interface to object factories.
- `Loader` is a dynamic class loader for both MBeans and other classes.
- `PathFinder` maintains mappings between keywords and folders.
- `Pool` adds object pool support to `Factory`.
- `ResourceFinder` is a resource finder extending the search mechanism of `java.util.ResourceBundle` to handle partial locale qualifiers.
- `Scripter` evaluates and executes scripts within the scripting framework.
- `Startup` starts, stops and restarts the server.

Supporting services provide thread management, scheduling, authentication, session management, connection handling, mail sending and other services.

2.1.2 Pipes and Filters

Applications utilize Tammi's services typically from various filters organized into filter chains. The filter chain pattern and filters are described in the Servlet API v. 2.3 specification. Tammi develops the pattern further by defining a set of configurable filters implementing the basic application behavior. In Tammi, filters are grouped into pipes, which are collections of consecutive filters executed in configured order. Pipes may form larger networks, within which branch filters control the request flow.

Filters have two variations, plain filters and key filters. Plain filters are applied to every request running through it. Key filters have a configurable key, which may appear in the request to activate the filter. The key can have an associated value specifying further the actions to take by the corresponding filter.

Plain filters include

- `HttpFilter` parses HTTP protocol requests.
- `RepositoryFilter` manages DB connections.
- `ResourceFilter` provides localized content.
- Variations of `AuthenticationFilter` authenticate users.
- `ChartFilter` for generates charts.
- `ReportFilter` for generates reports.
- `ClosingFilter` terminates the filter chain and closes the response.

Key filters provide branching control, terminal detection, styled skins, template parsing, form management, page flow control and other filtering.

2.1.3 Flows and Steps

The markup templates producing the web pages of a Tammi application are organized into page flows consisting of one or more flow steps. The flows of the application form a hierarchical structure, which can be browsed either interactively through the links generated automatically on web pages or programmatically by applying control commands of the Tammi service API.

Each step has various properties, including markup templates, executable actions, and data containers. The data containers are implemented as dynamic MBeans called variables.

The configuration of flows and steps is maintained by a flow filter, through which to control their execution, too. An application may have more than one flow filter in its separate pipes to build larger entities.

2.1.4 Variables

A `Variable` defines a dynamic container of attributes. Instances of `VariableAttributeInfo` define one attribute each. Lists of attribute instances are registered to a `VariableRegistry` with a named type.

The variable registry maintains meta-data of attributes of each registered attribute list. The registered type of each attribute list is treated as the virtual class name of the corresponding variable.

2.2 Deployment

Tammi is running on an JMX agent server connected optionally to a separate web server through appropriate protocol connectors. Clients can access Tammi either directly through an HTTP connector or via the separate web server. Stand-alone clients can connect to Tammi through a graphical user interface.

An optional database server can be accessed through database service brokers available for vendor specific database implementations.

Application components implement their manageable resources as MBeans registered to the MBean server of the JXM agent. Applications can reside in the same JVM as Tammi, as local applications on the same host, or as remote applications on a remote server. In the two latter cases, the application components are implemented as proxy MBeans for accessing remote application resources.

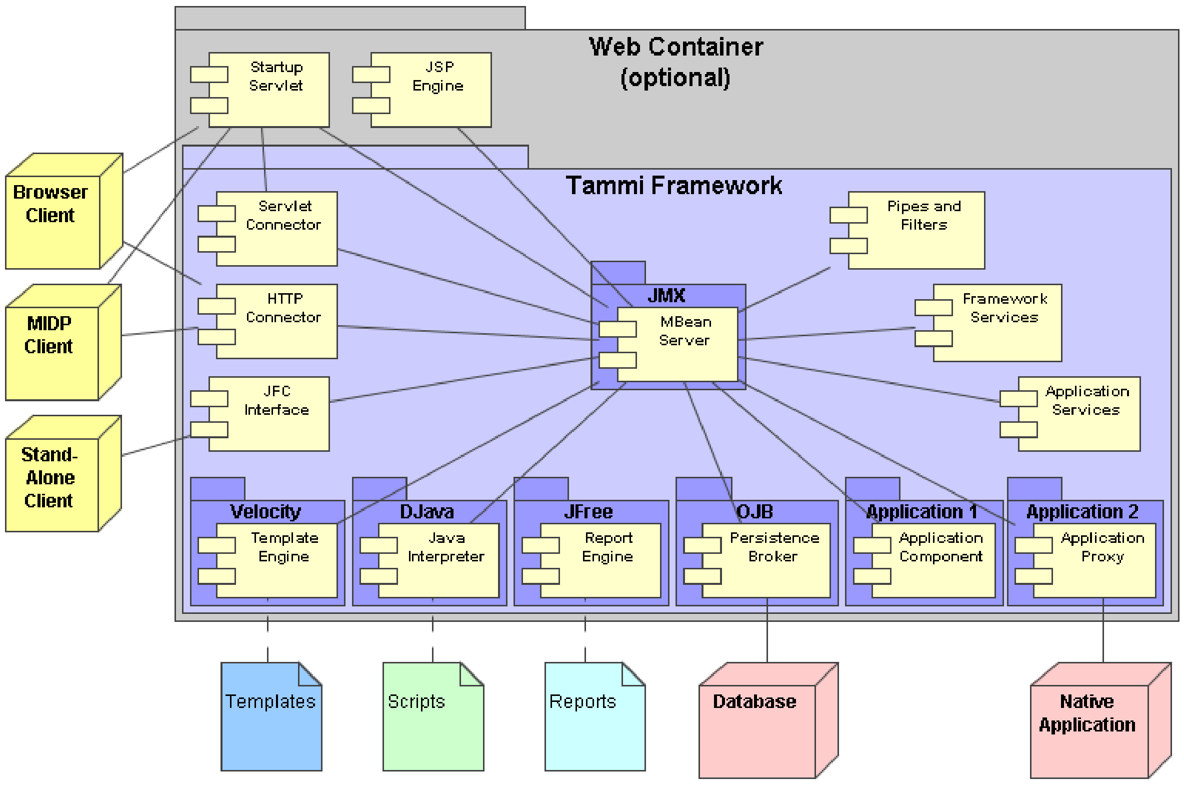

Figure 2.1 The deployment diagram

3 Control Commands

The filter chain mechanism of Tammi is controlled by the representational state transfer [REST] pattern. The key filters maintaining the state and providing the services of Tammi applications are activated by filter specific key-value pairs acting as control commands. The commands may be entered in the URL path info of a request, within the query string of a GET request or as form parameters in a POST request.

3.1 URLs

A URL represents a Uniform Resource Locator, a pointer to a "resource" on the HTTP and other Internet protocols. A URI represents a Uniform Resource Identifier, which is a super-set of a URL not limited to locations but rather identifying resources with a name or a set of attributes. The difference between the two is not remarkable [URI].

The part of the URL following the scheme, host name and port specify the URL path. A semicolon (";") separates URL object parameters from the actual URL path and the components after a question mark ("?") specify a query string.

Also the components of the actual URL path have different meanings. The division is based on the servlet specification [Servlet]. The URL path can be divided into two parts:

- `Context path` specifies the external context of servlet applications.
- `Path info` contains application specific information for pipes and filters.

3.1.1 URL Path Info

The path info contains any application specific information associated with the URL the client sent when it made a request. The path info follows the context path but precedes the object parameters and query string. By default, Tammi is configured to parse the path info as a set of key-value pairs as <key>/<value>.

If the value itself contains path separators ("/"), the key must have an indicator specifying the total number of items and slashes minus one to be included in the value as <key>-<n>/<value>.

3.1.2 Query String

A query string is optionally contained in the request after the path info and object parameters. It contains a set of key-value pairs as <key>=<value> separated by ampersands ("&"). It provides an alternate way to enter control commands.

3.1.3 Form Parameters

Control commands may also be entered in the body of the request as a set of key-value pairs as <key>=<value> separated by ampersands ("&"). The method of such a request must be POST and it its content type:
`application/x-www-form-urlencoded`

3.1.4 Object Parameters

Object parameters are used internally by the HTTP services of Tammi. E.g. the session id for clients not supporting cookies is encoded in object parameters as jsessionid=<id>.

3.2 Uploads

If the control commands contain large blocks of data to be uploaded to the server, they can be sent in the body of a POST request as a multi-part form-data [Multipart]. The content type of such a request must be `multipart/form-data`. Files in the specified data exchange format represent a typical example of such data. If the size of the file is compact, it may be uploaded in a normal POST request, too, but then its content must be encoded [URI].

3.3 Responses

The response to control commands depends on the actual commands and the specified content type. Tammi produces responses in five different markup languages: HTML, XHTML, WML, XML and KML. The format of the response is selected automatically based on content types of specified by the `accept` header of the request. The desired format may also be specified explicitly by a default command "x" for a `TerminalFilter`. The value of the command must be the file extension of the corresponding content type.

3.3.1 HTML

The HTML content type is

```
text/html
```

and the corresponding extension

```
html
```

3.3.2 XHTML

The XHTML content type is

```
application/vnd.wap.xhtml+xml
```

and the corresponding extension

```
whtml
```

3.3.3 WML

The WML content type is

```
text/vnd.wap.wml
```

and the corresponding extension

```
wml
```

3.3.4 XML

The XML content type is

```
application/xml
```

and the corresponding extension

```
vdeml
```

See chapter Data Exchange Formats for a more detailed description of XML responses.

3.3.5 KML

The KML content type is

`application/vnd.google-earth.kml+xml`

and the corresponding extension

`kml`

See chapter GIS Integration for a more detailed description of KML.

3.4 Sample Commands

A typical application is formed by one or more pipes branching at one or more levels. In addition, one server may serve one or more hosts.

3.4.1 HostFilter

The `HostFilter` is the topmost branch filter forwarding requests to specific pipes based on the requested host name. The host may also be specified by a default command "host".

3.4.2 BranchFilters

`ServiceFilter`, `PipeFilter` and `LinkFilter` are lower level branch filters forwarding requests based on default commands "at", "in" and "link" respectively. The names of the corresponding branches is application dependent.

3.4.3 FlowFilter

The `FlowFilter` maintains the page flow under work and controls the execution of the current step. The flow and step are selected by a default command "flow".

A flow is identified by its name, which is relative to its parent flow. A step is identified either by its name or by its index within its owning flow as <flow_name>-<step_index>. An absolute flow path may be applied to refer to arbitrary flows as /<flow-name_1>/<flow-name_2>/.../<flow_name_n>.

3.4.4 PageFilter

Some flow steps accept markup templates outside of their own configuration. The `PageFilter` renders templates, the names of which can be explicitly specified with a default command "page".

3.4.5 TaskFilter

In addition to step specific actions, named tasks may be activated by the `TaskFilter`. The name of the task is specified with a default command "task".

3.4.6 LocaleFilter

The locale to apply is maintained by the `LocaleFilter`. It can be explicitly specified with a default command "lang". The value of the locale is its string representation as <lang>_<country>.

3.4.7 TerminalFilter

The `TerminalFilter` defines the user terminal type based on user agent information of requests and selects an appropriate MIME type for the corresponding responses. An explicit content type is specified with a default command "x".

3.5 Error Handling

Command execution may fail at three different levels. If the requested operation is unknown, forbidden or cannot for some other reason be activated at all, the operation is canceled and the corresponding response with an applicable HTTP status code and error message is returned. See the Response Schema for details.

If the command is valid, but the corresponding flow fails to finish its execution successfully due to missing, erroneous or illegal arguments, failure exceptions are included in the flow part of the standard response. See the Flow Schema for details.

When attribute values of a variable of the command are not valid, the validity of the corresponding variable is set to `INVALID` and validation exceptions are included in the attribute part of the standard response. See the Variable Schema for details.

4 Data Exchange Formats

The XML format applied to data exchange between a programmatic client and Tammi is based on the configured page flow structure of the respective application. The client controls the application with commands to open the desired pipe, flow and step. Each step returns an XML coded page corresponding to its data content and programmed functionality.

The XML language is called Variable Data Exchange Language and its file extension is `vdeml`. The XML page describes the current flow structure and contains the variable list of the current step. The flow structure and the variable list have their own XML schema and are represented in separate namespaces within the XML page. Both are enclosed within a response element containing the status of any requested operation.

The flow structure consists of the selected flow, the name of its parent flow, the index of its current step and a a list of other steps within the flow.

The variable list contains variable instances, instance attributes and attribute values. An extended variable format includes the meta-data of the variables, too. The meta-data is included by adding a parameter `meta` with a value `true` to the corresponding request.

JavaScript Object Notation [JSON] may be supported as an alternate data exchange format.

4.1 XML

The XML format is applied by default. The response data consists of both flow structure and variable list, the upload data may contain only the variable list without meta-data.

For variable specific updates, only the values of changed attributes are submitted as form parameters of a POST request. The names of attributes are applied to parameter names **prefixed with the corresponding variable index and a colon**. The changed values are applied to parameter values.

4.1.1 Response Schema

```
<?xml version="1.0"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://tammi.sourceforge.net/pdf/tammi-api.pdf"
    xmlns="http://tammi.sourceforge.net/pdf/tammi-api.pdf">

    <xs:element
        name="Response"
        type="ResponseType">
    </xs:element>

    <xs:complexType
        name="ResponseType">

        <xs:sequence>
            <xs:element
                name="Message"
                maxOccurs="unbounded">
            </xs:element>

            <xs:element
                name="Authenticated"
                type="AuthenticatedType"
                minOccurs="0">
            </xs:element>
        </xs:sequence>
        <xs:attribute
            name="status"
            type="xs:positiveInteger"
            use="required" />

        <xs:attribute
            name="identity"
            type="xs:token" />
    </xs:complexType>

    <xs:complexType
        name="AuthenticatedType">

        <xs:sequence>
            <xs:element
                name="Attribute"
                type="AttributeType"
                maxOccurs="unbounded">
            </xs:element>
        </xs:sequence>

        <xs:attribute
            name="state"
            type="xs:boolean"
            use="required" />
    </xs:complexType>

    <xs:complexType
        name="AttributeType">

        <xs:attribute
            name="name"
            type="xs:token"
            use="required" />

        <xs:attribute
            name="type"
            type="xs:token"
            use="required" />

        <xs:attribute
            name="value"
            type="xs:string"
            use="required" />
    </xs:complexType>
</xs:schema>
```

4.1.2 Flow Schema

```
<?xml version="1.0"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://tammi.sourceforge.net/pdf/tammi-api.pdf"
    xmlns="http://tammi.sourceforge.net/pdf/tammi-api.pdf">

    <xs:element
        name="Flow"
        type="FlowType">
    </xs:element>

    <xs:complexType
        name="FlowType">

        <xs:sequence>
            <xs:element
                name="Step"
                type="StepType"
                maxOccurs="unbounded">
            </xs:element>

            <xs:element
                name="Exception"
                type="ExceptionType"
                minOccurs="0"
                maxOccurs="unbounded">
        </xs:sequence>

        <xs:attribute
            name="name"
            type="xs:token"
            use="required" />
        <xs:attribute
            name="label"
            type="xs:string" />

        <xs:attribute
            name="parentName"
            type="xs:token"
            use="required" />

        <xs:attribute
            name="parentLabel"
            type="xs:string" />

        <xs:attribute
            name="step"
            type="xs:positiveInteger"
            use="required" />
    </xs:complexType>

    <xs:complexType
        name="StepType">

        <xs:attribute
            name="name"
            type="xs:token"
            use="required" />

        <xs:attribute
            name="label"
            type="xs:token"
            use="required" />

        <xs:attribute
            name="index"
            type="xs:positiveInteger"
            use="required" />
    </xs:complexType>

    <xs:complexType
        name="ExceptionType">

        <xs:simpleContent>
            <xs:extension
                base="xs:string">

                <xs:attribute
                    name="type"
                    type="xs:string"
                    use="required" />
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:schema>
```

4.1.3 Variable Schema

```
<?xml version="1.0"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://tammi.sourceforge.net/pdf/tammi-api.pdf"
    xmlns="http://tammi.sourceforge.net/pdf/tammi-api.pdf">

    <xs:element
        name="VariableList"
        type="VariableListType">
    </xs:element>

    <xs:complexType
        name="VariableListType">

        <xs:sequence>
            <xs:element
                name="Variable"
                type="VariableType"
                minOccurs="0"
                maxOccurs="unbounded">
            </xs:element>
        </xs:sequence>
    </xs:complexType>
```

```
<xs:complexType
    name="VariableType">

    <xs:sequence>
        <xs:element
            name="Attribute"
            type="AttributeType"
            minOccurs="0"
            maxOccurs="unbounded">
        </xs:element>
    </xs:sequence>

    <xs:attribute
        name="index"
        type="xs:integer"
        use="required" />

    <xs:attribute
        name="id"
        type="xs:long"
        use="required" />

    <xs:attribute
        name="type"
        type="xs:token"
        use="required" />

    <xs:attribute
        name="validity"
        type="xs:token"
        use="required">

        <xs:simpleType>
            <xs:restriction
                base="xs:token">

                <xs:enumeration
                    value="REJECTED"/>
                <xs:enumeration
                    value="INITIAL"/>
                <xs:enumeration
                    value="INVALID"/>
                <xs:enumeration
                    value="VALID"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>

<xs:complexType
    name="AttributeType">

    <xs:sequence>
        <xs:choice>
            <xs:element
                name="Value" />

            <xs:element
                name="Values"
                type="ValuesType" />
        </xs:choice>

        <xs:choice>
            <xs:element
                name="DefaultValue"
                minOccurs="0" />

            <xs:element
                name="DefaultValues"
                minOccurs="0"
                type="ValuesType" />
        </xs:choice>

        <xs:element
            name="MinValue"
            minOccurs="0" />

        <xs:element
            name="MaxValue"
            minOccurs="0" />
```

```
          <xs:element
              name="Enumeration"
              type="EnumerationType"
              minOccurs="0" />

          <xs:element
              name="Exception"
              type="ExceptionType"
              minOccurs="0" />
      </xs:sequence>

      <xs:attribute
          name="name"
          type="xs:token"
          use="required" />

      <xs:attribute
          name="type"
          type="xs:token"
          use="required" />

      <xs:attribute
          name="group"
          type="xs:string" />

      <xs:attribute
          name="description"
          type="xs:string" />

      <xs:attribute
          name="detail"
          type="xs:string" />

      <xs:attribute
          name="qualifier"
          type="xs:string" />

      <xs:attribute
          name="pattern"
          type="xs:string" />

      <xs:attribute
          name="traits"
          type="xs:integer" />

      <xs:attribute
          name="minSize"
          type="xs:integer" />

      <xs:attribute
          name="maxSize"
          type="xs:integer" />
  </xs:complexType>

  <xs:complexType
      name="ValuesType">

      <xs:sequence>
          <xs:element
              name="Item"
              minOccurs="0"
              maxOccurs="unbounded" />
      </xs:sequence>
  </xs:complexType>

  <xs:complexType
      name="EnumerationType">

      <xs:sequence>
          <xs:element
              name="Enum"
              type="EnumType"
              minOccurs="0"
              maxOccurs="unbounded" />
      </xs:sequence>
  </xs:complexType>

  <xs:complexType
      name="EnumType">

      <xs:simpleContent>
          <xs:extension
              base="xs:string">
```

```
                <xs:attribute
                    name="name"
                    type="xs:string"
                    use="required" />

                <xs:attribute
                    name="label"
                    type="xs:string"
                    use="required" />
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

    <xs:complexType
        name="ExceptionType">

        <xs:simpleContent>
            <xs:extension
                base="xs:string">

                <xs:attribute
                    name="type"
                    type="xs:string"
                    use="required" />
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
</xs:schema>
```

4.1.4 XML Sample

```
<Response
    status="200">

<Message>OK</Message>

<flow:Flow
    xmlns:flow="http://tammi.norther.org"
    name="flow_1"
    parentName="parent_name"
    step="index">

    <var:VariableList
        xmlns:var="http://tammi.norther.org">
        <var:Variable
            index="0"
            id="id_1"
            type="type_1"
            validity="validity_1">

            <var:Attribute
                name="name_1_1"
                type="type_1_1">

                <var:Value>value_1_1</var:Value>

            <var:Attribute
                name="name_1_2"
                type="type_1_2"/>

                <var:Value>value_1_2</var:Value>

            <var:Attribute
                name="name_1_n"
                type="type_1_n"/>

                <var:Value>value_1_n</var:Value>
        </var:Variable>
        <var:Variable
            index="1"
            id="id_2"
            type="type_2"
            validity="validity_2">

            <var:Attribute
                name="name_2_1"
                type="type_2_1"/>

                <var:Value>value_2_1</var:Value>

            <var:Attribute
                name="name_2_2"
                type="type_2_2"/>

                <var:Value>value_2_2</var:Value>

            <var:Attribute
                name="name_2_n"
                type="type_2_n"/>

                <var:Value>value_2_n</var:Value>
        </var:Variable>

        <var:Variable
            index="n"
            id="id_n"
            type="type_n"
            validity="validity_n">

            <var:Attribute
                name="name_n_1"
                type="type_n_1"/>

                <var:Value>value_n_1</var:Value>

            <var:Attribute
                name="name_n_2"
                type="type_n_2"/>

                <var:Value>value_n_2</var:Value>

            <var:Attribute
                name="name_n_n"
                type="type_n_n"/>

                <var:Value>value_n_n</var:Value>
        </var:Variable>
    </var:VariableList>

    <flow:Step
        name="step_1"
        index="index_1"/>

    <flow:Step
        name="step_2"
        index="index_2"/>

    <flow:Step
        name="step_n"
        index="index_n"/>
</flow:Flow>
</Response>
```

4.2 JSON (Not Implemented)

The JSON format is specifically targeted to JavaScript programming.

4.2.1 JSON Sample

```
{ "Response": {
    "status": "200",

    "Message": "OK",

    "Flow": {
        "name": "flow_1",
        "parentName": "parent_name",
        "step": "index",

        "VariableList": {
            "Variable": [
                { "index": "0",
                "id": "id_1",
                "type": "type_1",
                "validity": validity_1",

                "Attribute": [
                    { "name": "name_1_1",
                    "type": "type_1_1",

                    "Value": "value_1_1"
                    },

                    { "name": "name_1_2",
                    "type": "type_2_2",

                    "Value": "value_2_2"
                    },

                    { "name": "name_1_n",
                    "type": "type_1_n",

                    "Value": "value_1_n"
                    }
                ]},

                { "index": "1",
                "id": "id_2",
                "type": "type_2",
                "validity": "validity_2",

                "Attribute": [
                    { "name": "name_2_1",
                    "type": "type_2_1",

                    "Value": "value_2_1"
                    },

                    { "name": "name_2_2",
                    "type": "type_2_2",

                    "Value": "value_2_2"
                    },

                    { "name": "name_2_n",
                    "type": "type_2_n",

                    "Value": "value_2_n"
                    }
                ]},
                { "index": "n",
                "id": "id_n",
                "type": "type_n",
                "validity": "validity_n",

                "Attribute": [
                    { "name": "name_n_1",
                    "type": "type_n_1",

                    "Value": "value_n_1"
                    },

                    { "name": "name_n_2",
                    "type": "type_n_2",

                    "Value": "value_n_2"
                    },


                    { "name": "name_n_n",
                    "type": "type_n_n",

                    "Value": "value_n_n"
                    }
                ]}
            ]
        },

        "Step": [
            { "name": "name_1",
            "index": "index_1" },

            { "name": "name_2",
            "index: "index_2" },

            { "name": "name_n",
            "index": "index_n" }
        ]
    }
}}
```

4.3 Meta-Data

If the meta-data of the data is requested instead of values, an extended set of optional sub-elements and attributes may be provided for `<Attribute>` elements.

4.3.1 Variable Meta-Attributes

The index of the variable within the variable list:

```
index="index"
```

The unique id of the variable:

```
id="id"
```

The class type of the variable:

```
type="type"
```

The validity of the variable after modifications:

```
validity="VALIDITY"
```

```
    REJECTED = not modifiable within the current context
    INITIAL = not yet modified within the current context
    INVALID = modified, but not all values were accepted
    VALID = modified and valid
```

4.3.2 Attribute Meta-Elements

The default value of the attribute:

```
<DefaultValue>value</DefaultValue>
```

The min value of the attribute:

```
<MinValue>value</MinValue>
```

The max value of the attribute:

```
<MaxValue>value</MaxValue>
```

The allowed (enumerated) values of the attribute:

```
<Enumeration>
    <Enum name="name_1" label="label_1">value_1</Enum>
    <Enum name="name_2" label="label_2">value_2</Enum>
    <Enum name="name_n" label="label_n">value_n</Enum>
</Enumeration>
```

An optional validation exception during modifications:

```
<Exception type="type">message</Exception>
```

4.3.3 Attribute Meta-Attributes

The name of the attribute:

```
name="name"
```

The data type of the attribute:

```
type="type"
```

The group of the attribute:

```
group="group"
```

The description of the attribute:

```
description="description"
```

The detail of the attribute:

```
detail="detail"
```

The qualifier of the attribute:

```
qualifier="qualifier"
```

The conversion pattern of the attribute:

```
pattern="pattern"
```

The traits of the attribute:

```
traits="traits"
```

The min size of the string attribute:

```
minSize="min"
```

The max size of the string attribute:

```
maxSize="max"
```

4.3.4 Attribute Traits

Attribute traits define a set of on/off like meta-features to the attribute. All traits are exchanged in one integer representing a bit mask where each bit defines the state of a particular trait:

```
READABLE trait (0x00000001)
```

Non-readable attributes are typically passwords or other classified data and their values should be shadowed when displayed.

```
WRITABLE trait (0x00000002)
```

Non-writable attributes cannot be modified by ordinary users through normal user interfaces.

`CLONEABLE trait (0x00000004)`

Cloneable attributes support cloning of their values.

`NOT MANDATORY trait (0x00000008)`

Mandatory attributes must have or must be given a value other than null. Attributes both mandatory and optional may keep their original null values.

`NOT TRANSIENT trait (0x00000010)`

Changes to values of transient attributes should not be saved by the applied persistence mechanisms.

`NOT SERIALIZED trait (0x00000020)`

Values of serialized attributes should be saved as strings by the applied persistence mechanisms.

`NOT INTERACTIVE trait (0x00000040)`

Changes to values of interactive attributes should redisplay any interactive views related to the attribute.

`TRACED trait (0x00000080)`

Traceable attributes should maintain timestamps of their last modifications accessed through the methods of the Traceable interface.

`LOCALIZED trait (0x00000100)`

Values of localized attributes should be translated before displaying.

`ORDERED trait (0x00000200)`

Values of ordered array attributes and enumerations should be sorted before assignment. The associated comparator must be applied, if any.

`SORTED trait (0x00000400)`

String values of sorted array attributes and enumerations of sorted attributes should be sorted after localization. The associated comparator must be applied, if any.

`TRIMMED trait (0x00000800)`

String representations of trimmed attributes are trimmed before assignments and conversions.

`STRIPPED trait (0x00001000)`

Null elements of stripped array attributes are stripped before assignments and conversions.

`BLANK ACCEPTED trait (0x00002000)`

Note that this option is applied to values given as string arrays.

This hack helps to solve the null problem when assigning HTML form based values.

`EMPTY ACCEPTED trait (0x00004000)`

Note that this option is applied to mandatory array attributes. If true, arrays of zero length are accepted as non-null values.

`LOCAL trait (0x00008000)`

Local attributes should not be imported or exported.

`VISIBLE trait (0x00010000)`

Visible attributes are displayed in interactive views.

`DISABLED trait (0x00020000)`

Disabled attributes are temporarily non-writable in interactive views.

`OPTIONAL trait (0x00040000)`

Optional attributes are validated only when their value is actually modified.

`RECURSIVE trait (0x00080000)`

Recursive attributes should be validated recursively.

`STREAM trait (0x00100000)`

Stream attributes may hold exceptionally large values, like mediafile contents. Once set to true, it can't be cleared.

`REFERENCE trait (0x00200000)`

Values of reference attributes may be proxies and require specific treatment when persisted. Once set to true, it can't be cleared.

`COLLECTION trait (0x00400000)`

Values of collection attributes may require specific treatment when persisted. Once set to true, it can't be cleared.

`PRIMARY trait (0x00800000)`

Values of primary attributes should uniquely identify the corresponding variable. Once set to true, it can't be cleared.

`FOREIGN trait (0x01000000)`

Values of foreign attributes should reference primary attributes of other variables. Once set to true, it can't be cleared.

`FINAL trait (0x02000000)`

Final attributes can't be replaced by inherited attributes. Once set to true, it can't be cleared.

`MARKUP trait (0x04000000)`

Markup attributes contain SGML encoded content.

`ENUMERATED trait (0x08000000)`

Enumerated attributes may have an enumerated list of allowable values.

Note that attributes of enum type have an automatically generated enumeration if their enumerated trait is NOT set. Localized attributes apply a lowercase key formed by the plain enum type followed by the enum value and a suffix (<type>.<value>.enum).

Non-localized attributes apply enumeration values with underscores replaced by spaces as keys.

`DESCRIPTOR trait (0x10000000)`

Descriptor attributes describe data of some other element. The trait may be applied to dynamically generate variables from meta variables.

`SELECTOR trait (0x20000000)`

Selector attributes select data of some other element. The trait may be applied to automatically generate nested user interfaces.

`MBEAN trait (0x40000000)`

MBean attributes contain MBean references as values. Values of MBean attributes should be registered and unregistered together with their owner variable.

`IMMUTABLE trait (0x80000000)`

Immutable attributes cannot be modified.

4.4 Data types

The data types of attributes are represented as Java class names. Also Java primitive types are represented by their corresponding object types. With array types, the Java array notation is applied, f.ex. `[Ljava.lang.Integer;`.

In principal, any Java class may be applied to attributes. In practice, most applications restrict their data types to those that can be persisted to an SQL based relational data base system.

4.4.1 SQL Types

For persistent attributes, the following mappings are used between SQL types and Java types:

```
BIT = java.lang.Boolean (boolean)

BIT VARYING = [Ljava.lang.Byte; ([B)

BINARY LARGE OBJECT = java.sql.Blob

CHARACTER = java.lang.String

CHARACTER VARYING = java.lang.String or java.lang.StringBuilder

CHARACTER LARGE OBJECT = java.sql.Clob

BOOLEAN = java.lang.Boolean (boolean)

SMALLINT = java.lang.Short (short)

INTEGER = java.lang.Integer (int)

BIGINT = java.lang.Long (long)

FLOAT = java.lang.Double (double)

DOUBLE = java.lang.Double (double)

NUMERIC = java.math.BigDecimal

DECIMAL = java.math.BigDecimal

DATE = java.util.Date
```

With location attributes, a custom type is applied:

`org.norther.tammi.acorn.lang.Location`

4.4.2 Serialization

Attribute values are serialized to strings during data exchange. Localization is not applied, but numbers and other types are represented unformatted. However, with date attributes the ISO 8601 format (`yyyy-MM-dd HH:MM:SS`) is applied.

Location attribute values are represented as pairs of decimal latitude and longitude coordinates separated by a space (`lat.dddddd lon.dddddd`).

5 GIS Integration

For applications that need to integrate to geographic information systems (GIS), the `vdeml` data may be embedded into a file encoded in Keyhole Markup Language (KML) [KML] supported in one or another form in majority of GIS software.

As KML is originally an XML file format used to display geographic data in an Google's earth and map browsers, its main focus is on geocoordinates and visual presentation.

Locations are represented as `<Placemark>` elements within a `<Document>` element defining their position on the earth's surface with geocoordinates inside a `<Point>` element. In addition to coordinates, each `<Placemark>` has a name and an optional description. Paths are represented as a list of geocoordinates inside a `<LineString>` element.

A `<Placemark>` may also have user-supplied content inside a `<description>` element that appears in the description balloon of Google browsers. For data exchange, a more structured `<ExtendedData>` element is more applicable. The contents of the element may be a set of untyped `<Data>` elements, typed custom data inside a `<SchemaData>` element, custom XML elements or a combination of the three.

5.1 Response

The response element containing the status of the requested operation is represented inside of the `<ExtendedData>` element of the `<Document>` element in its own namespace named `res`.

5.2 Location Attributes

Tammi supports KML integration for variables represented as `<Placemark>`s and having one and only one location attribute, which contains either a value represented as a `<Point>`, or an array of values represented as a `<LineString>`, or a closed array of values represented as a `<LineRing>`. In addition, the variable should contain a string attribute named "Name", the value of which is applied to the `<name>` field of the `<Placemark>`.

5.3 Feature Attributes

Other **visible** attributes of variables are represented in KML integration as `<SimpleData>` elements within the `<SchemaData>` element of the `<ExtendedData>` element. The types and names of attributes are represented as `<SimpleField>` elements within a separate `<Schema>` element. These feature attributes appear automatically in the description balloon of Google browsers.

Elements of arrays have their own <SimpleData> and <SimpleSchema> elements with name attributes suffixed with the index of the corresponding array element.

5.4 Variable List

For **all** attributes and full meta-data, the `vdeml` `<VariableList>` element may be included in the `<ExtendedData>` element in its own namespace named `var`. In KML integration, the flow structure is not included in data exchange and each variable forms its own `<VariableList>` element with only one `<Variable>` element within the list. Note that the `<Attribute>` elements of the variable include also the location attributes and feature attributes represented already inside other elements of the corresponding `<Placemark>`. The XML formatted variable list is not displayed in browsers but is provided solely for data exchange purposes.