D5.1 - Design and Specification of the Integration Architecture

Objectives

The aim of this document is to specify the integration architecture of the solutions developed in the project. Firstly the definitions of the A3I and M1i modules and their use cases in the context of the CRISTAL-iSE Kernel. The a set of use cases are presented which present a use of the CRISTAL-iSE Kernel and more precisely its client API. Finally, we specify the integration of modules. Modules being heterogeneous in term of programming language (java, windev, C# and others), due to this language heterogeneity the REST protocol [1] has been chosen to be the main integration mechanism for the project.

CIMAG-RA Module - Use cases

CIMAG-RA is a new module of CIMAG HR software its features and exact implementation was detailed previously in D2.2. Briefly, It deals with the allocation of human resources to specific tasks regarding properties such as :
- skills requirement
- shifts definition
- contracts and legal aspects
- time scales
- working patterns
- shift assignments

All these properties define a resource allocation problem to solve to provide with a schedule. It means that we have to be as exhaustive as possible to anticipate future customers needs.

Nethertheless, thanks to CRISTAL-iSE flexibility, we can easily extend the domain definition. **Application domain properties are items as defined in CRISTAL-iSE.**
First, these properties are extracted from an SQL database and then mapped to Problem and Solution items in cristal-iSE. The initial Problem item is sent to the solver to compute a solution. Following an iterative process the generated solution item become a new problem item and is computed again to provide with a better solution. In details, using a tabu search algorithm, a solution (planning) is built and then sent to a rule engine (DROOLS) to compute the score of the proposed solution. The process run until a satisfying solution is provided to the end user. Figure 1 illustrate the solving process.

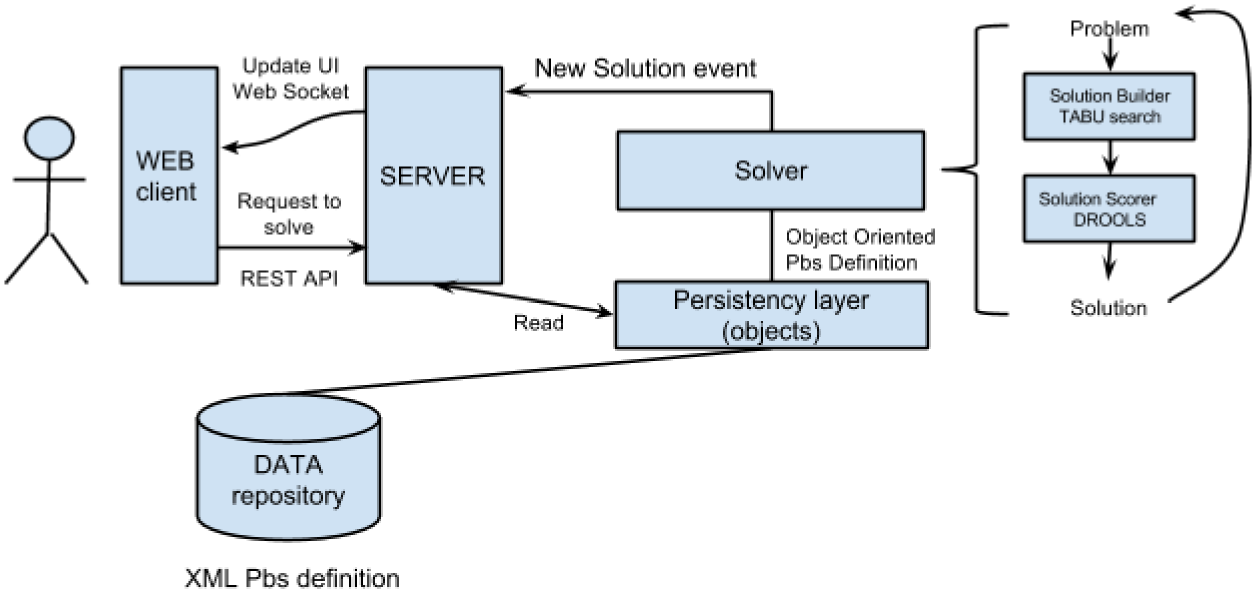

Figure 1 The resource allocation process

The solution is built on the integration of CIMAG HR solution, CRISTAL-iSE and AGILIUM NG.

- CIMAG HR is an existing solution from A3I. This solution is dedicated to human resources management, with among other:
    - time and attendance management
    - day off requests manegement
- CIMAG RA is the new module of A3I. It is dedicated to Resource Allocation management and mainly deals with a solver and rule engine.
- AGILIUM is a business process engine built on CRISTAL-iSE. It is mainly dedicated to workflow execution and in our case, HR workflow design and execution.

Figure 2 details the integration of CIMAG RA module with existing CIMAG HR solution and CRISTAL-iSE.

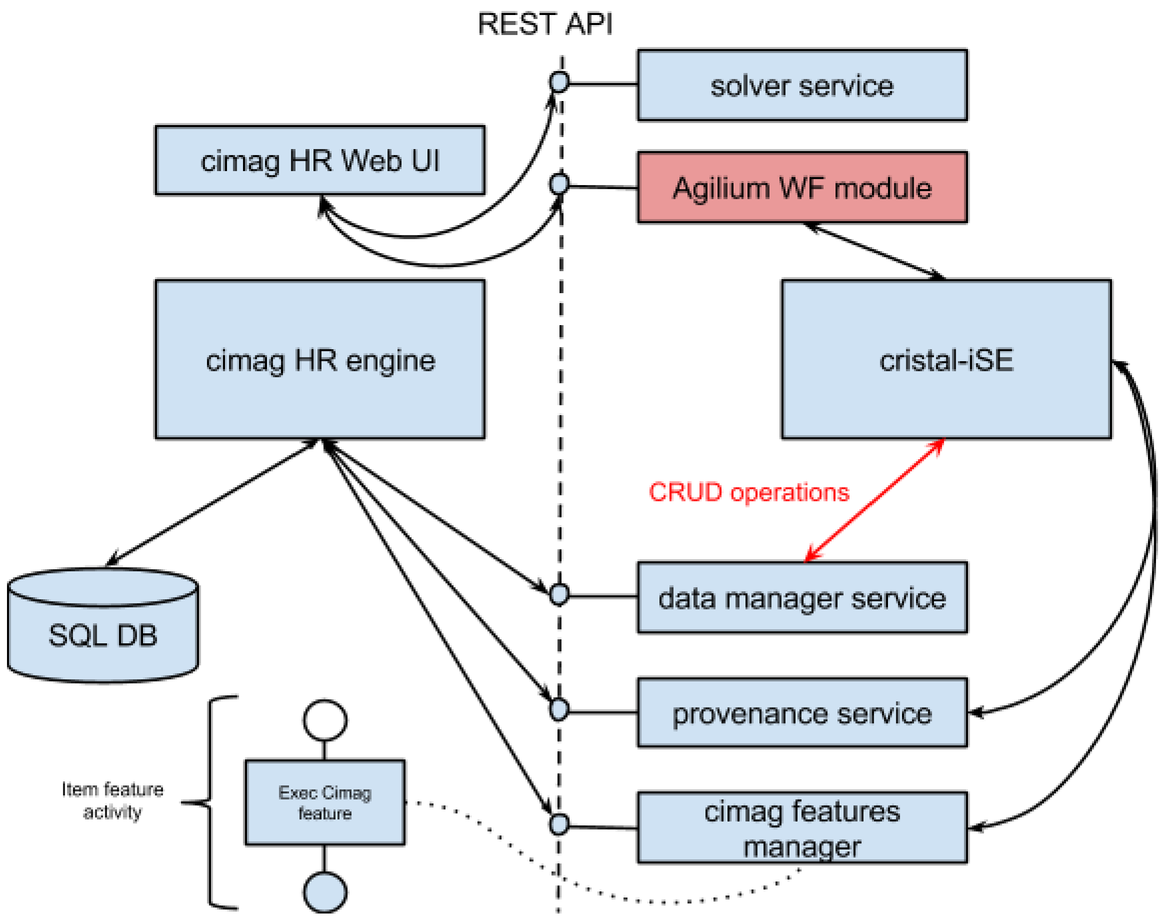

Figure 2 Integration of CIMAG HR, CIMAG RA (solver) and CRISTAL-iSE

Business data are stored in a SQL database. These business data are extracted and mapped to a Problem and Solution item in cristal-ise through the data manager service. These items are computed by the solver service and then updated in the cristal-ise repository, using the data manager service. For provenance purpose, we track execution of cimag features. Thus, we aim at managing activity items. These activity items represent resource allocation execution and more generally HR activities, such as day off request, exposed by the existing CIMAG HR software. These activities can be scheduled in workflows, using the agilium workflow module.

So based on the introduction to CIMAG HR and RA, the elementary use cases of cristal-ise in the context of resource allocation are:

- authenticate application user to server
- store item[1](s)
- get last item(s)
- update item(s)
- execute item(s)
- move to item version

[1] Items refer to elements from the application domain : employe, skills, rosters, ….+ rules and application users authorizations

These use cases are mapped to the CRISTAL-iSE methods as defined in sections CRISTAL-iSE kernel and CRISTAL-iSE methods through the data manager service.

AGILIUM NG Modules - Use Cases

Agilium NG will be used to implement validation process of the CIMAG RA proposals as business processes.
The CIMAG RA HR Web UI will call the REST API of Agilium NG :

- to create validation process instances
- to follow the advancement of these validation processes
- to allow the concerned users to execute their own part of these validation processes

CRISTAL-iSE Kernel

As Is, CRISTAL-ISE expose methods through its client API. Actual architecture is depicted in figure 1. To get data related to an object, Client API:

1- Give a path to the object reference in LDAP directory
2- Get Object UUID
3- Using this UUID, it proceed to Item resolution
4- For requested Object, It read or submit data through activity execution
5- Get data corresponding to object

Client API can also directly extract data from XML data base.

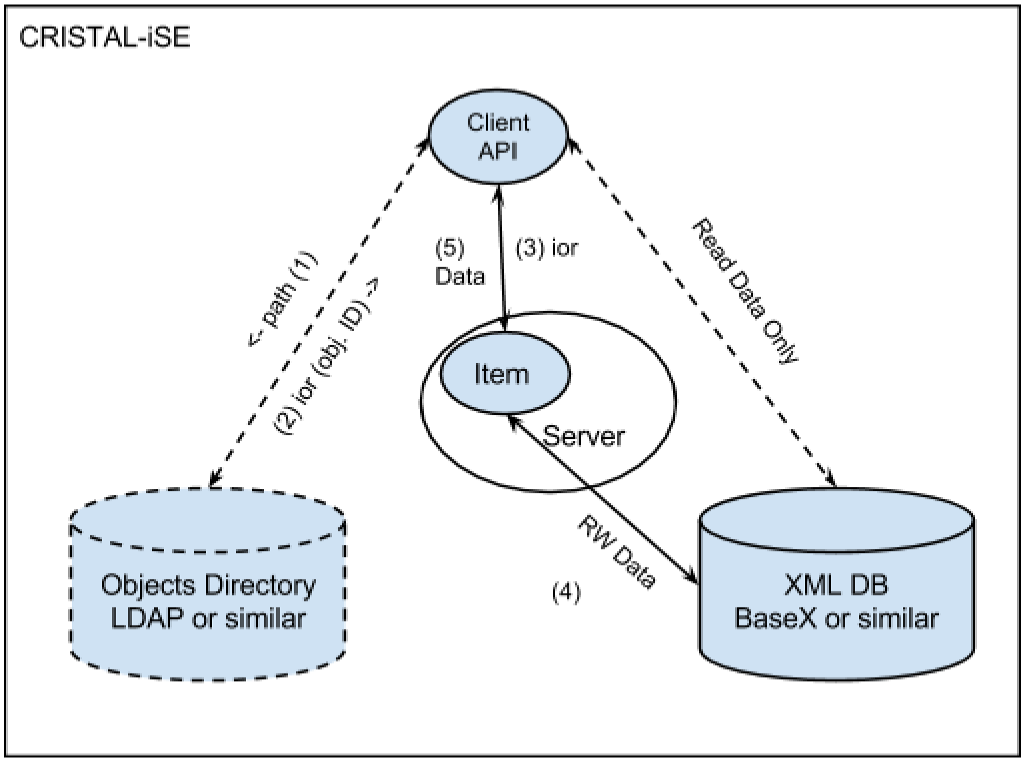

Figure 1. CRISTAL-iSE AsIS Architecture

To ease integration with third party modules, it has been decided to had a REST API on top of it, as illustrated on figure 2.

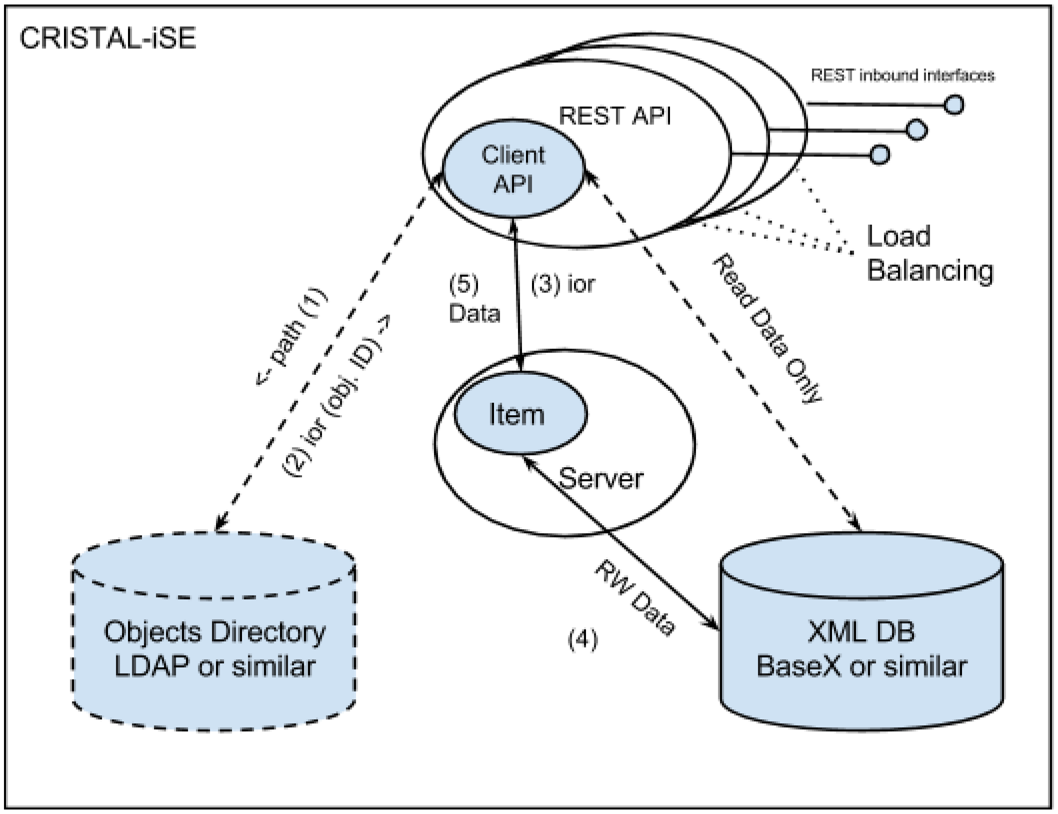

Figure 2. Integration of the REST API in CRISTAL-iSE

To expose methods from CRISTAL-iSE through a REST API, we propose to reuse existing architecture and encapsulate the client in a “REST server process”. This “REST server process” provides:

- a REST inbound interface to
- a specific outbound interface to wrappe REST requests to the existing client methods.

A key challenge here is the load balancing. A lot of validation and activity logic execution is done in client during activity execution. Thus to avoid bottlenecks, multiple REST servers can be created to share this load.

The suggested framework to use for the REST container is JSR-339 (JAX-RS 2.0) using Jersey for eventual inclusion in any servlet container and OSGi.

The REST Interface

Representational State Transfer (REST) is a software architecture style consisting of guidelines and best practices for creating scalable web services. REST is a coordinated set of constraints applied to the design of components in a distributed hypermedia system that can lead to a more performant and maintainable architecture.

REST is considered as a simpler alternative to SOAP and WSDL-based Web services (See RestFul Cookbook[2]).
RESTful systems typically, but not always, communicate over the Hypertext Transfer Protocol with the same HTTP verbs (GET, POST, PUT, DELETE, etc.).
The REST architectural style was developed by W3C Technical Architecture Group (TAG) in parallel with HTTP 1.1, based on the existing design of HTTP 1.0.
REST follow a client-server architecture. A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.
REST is a stateless protocol, which means that the client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be in transition. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition.

These 2 main characteristics of the REST protocol are essential to face the heterogeneity of the modules from A3I and M1i and later other users (cristal-ise is now an open source solution). In fact technologies range from java, .Net to windev.

REST Server

To expose CRISTAL-iSE methods, a REST server has been implemented as a module. This module also acts as a CRISTAL-iSE kernel client. The role of this server is to wrap the REST requests to CRISTAL-iSE methods. REST services act as proxys to the methods. The main constraint is the load balancing. In fact as methods execute scripts, we need to instantiate multiple CRISTAL-iSE clients in order to avoid bottleneck.

REST Client

Each one of the business module has to integrate a REST client. This will be quite easy but as services execute methods on the server, to improve performance of the applications, we suggest to develop asynchronous REST clients.

[2] RestFull Cookbook http://restcookbook.com/

CRISTAL REST API Specification

Purpose

The purpose of this API is to present an accessible, coherent and easy to understand mechanism to access CRISTAL through REST, to ease access to Items from other programming environment and guide new developers towards productive use of the description-driven method. It is not designed to replace the Java Client API, which will continue to be used by Scripts and Java clients, including the existing Swing UI. It will rely on domain applications developing or importing Item Descriptions to define the activities and outcomes available through REST, including Item Description Descriptions in the case of a domain application that provides Description editing capabilities.

The REST API presents an opportunity to simplify access to CRISTAL, presenting only the functionality that we would expect ordinary developers to need. Usage of it will neither require unmarshalling of kernel objects on the client side, nor inclusion of the kernel library on its classpath, and especially not require the client application to run in a JVM.

XML handling will not be core to the interactions defined by the API, as most regular job query and execution will use JSON as a communication medium. We will investigate the introduction of a JSON Outcome type, and automatic conversion between XML and JSON to extend this to cover execution completely. However, descriptions will remain as XML marshalled form of Java objects, so the manipulation of those will continue to be easier in the Java Client API.

Building and Running the REST Server

The Maven project of the REST implementation is available at:
https://github.com/cristal-ise/restapi
The API is implemented using JAX-RS, and so may be run in any HTTP container that supports that. It includes a small Grizzly server for testing in the Main class, which takes standard CRISTAL command-line arguments, and listens on the URI given in the property '`REST.URI`', defaulting to localhost:8081 if undefined.

Authentication

Users should be authenticated against CRISTAL Agents before interacting with the REST API, even for data reads (which may gain ACLs later). Two 'methods' are currently available, depending on security requirements

- None, requiring agentName parameter to be used for job query and execution. This method may be used when authentication is handled by the caller, and the REST server is not accessible from elsewhere. To enable this, set the property `REST.requireLoginCookie` to 'false' in config. This will also allow all other methods to be accessed anonymously.
- Encrypted HTTP Cookie
  - **GET** */login?user=<agent name>&pass=<password>*

    - Cookie consisting of agent UUID and timestamp encrypted with a runtime generated symmetric key. Cookies may be given limited duration in seconds using `REST.loginCookieLife` property.
    - This mechanism tries to use 256-bit AES encryption, but default Java installs only support 128-bit. Allow use of the weaker standard encryption with the `REST.allowWeakKey` boolean property, or include the Java "unlimited strength" crypto kit in your JRE.

Browsing Item Directory

GET */domain/<path>*

- start=<index> - batched loading start index (optional, default 0)
- batch=<batch size> - maximum size of result set (optional, default max batch size defined)
- search=(<name>:)<value>(,<name:value>) - search this sub-tree for a particular value in the named property or properties. If the property name isn't given for the first field, it defaults to 'Name'

Enumerates DomainPaths. Inside a context, returns a batched list of child nodes, listed by node name and link. Items and agents are listed by their 'Name' property, but linked to their /item/<uuid> URL (ItemPath), where they may further queried. A request to a DomainPath that is an alias to an Item redirects to the ItemPath (HTTP code 303 See Other). If the result is batched, the end of the list will give a link to next batch. Batch size is preserved in next link. Max batch size configurable by setting the CRISTAL property '`REST.Path.DefaultBatchSize`', which defaults to 75.

GET */role* - lists all roles in the system
GET */role/<role name>*
Gets information about the given role.
Response contains:

- *name*: Role name
- *hasJobList*: boolean indicating whether this role pushed jobs to Agents holding it.
- subroles: All direct child roles of this role.
- agents: List of names and URLs of Agents holding this role.

Direct Item addressing and query

GET */item/<uuid>/*<data path>
Where <data path> is:
Summary

- <Empty> - Item summary containing:
    - Name
    - Other properties
    - All collections without members, with links
    - List of all viewpoint with links

Viewpoint/Outcome/Event data

- *data/* - List of viewpoint schemas in Item, with links
- *data/<SchemaName>* - list of Viewpoints present with that schema, with links
- *data/<SchemaName>/<ViewName>* - outcome corresponding to that schema. Response header contains:
    - *Last-modified*: Event timestamp
- *data/<SchemaName>/<ViewName>/event* - All event details of that outcome
  Event data:
    - ID
    - Timestamp
    - Agent name
    - Agent Role
    - Data:
        - Schema name and version
        - View name
        - Link to outcome via /history
    - Activity
        - Name
        - Path
        - Type
    - Transition
        - Name
        - Origin State
        - Target State
        - State Machine name and version
- *data/<SchemaName>/<ViewName>/history* - (currently expensive) index of previous outcomes assigned to that viewpoint, with timestamps and event Id with links to:
- *data/<SchemaName>/<ViewName>/history/<EventId>* - Previous outcome
- *data/<SchemaName>/<ViewName>/history/<EventId>/event* - Event details of this previous outcome

- *history(&start=<startEventId>(&batch=<batchSize>))* - Events in Item with links to each one. Batched as with DomainPath browsing. Default start at 0, default end depends on configurable max batch size. Link to next batch.
- *history/last* - The latest event.
- *history/<EventId>/data* - Outcome generated with the given Event, if present. Otherwise 404 error.

Item Typing and Structure

- *name - Returns the 'Name' property of the item as plain text*
- *property* - Returns all item properties
- *property/<Name>* - Returns the given named property as plain text

- property/<Name>/details - Gives all property data
  Property data:
    - name
    - value
    - readOnly
- *collection* - Index of all collections in the item
- *collection/<Name>* - The latest version of the given collection.
  Collection data:
    - Name
    - Version
    - Type (Aggregation or Dependency)
    - isDescription
    - Item type restrictions (Dependency only)
    - Other collection properties
    - For each member:
        - ID
        - Reference to child Item (if present)
        - Item type restrictions (Aggregation only)
        - Geometry (Aggregation only)
        - Other member properties
- collection/<Name>/version/ - List with links to named collection versions
- collection/<Name>/version/<version> - The given collection version.

Agent specific data

GET */agent/<uuid>/<dataPath>*

- *job(&start=<startJobId>&batch=<batchSize>))* - batched job list
- *job/<JobId>* - load specific job
  Jobs contain same data as OPTIONS query.
  Note that job IDs may not be contiguous nor start with index 0
- *roles* - a list of roles that this Agent holds

POST /agent/<uuid>/setPassword - sets the agent's password by calling the SetAgentPassword predefined step. The new password is redacted from the stored outcome.

All the preceding verbs return the following result codes:

- 200 OK - the data was found and is included in the response.
- 400 Bad Request - when the input data was not valid (e.g. search syntax was wrong)
- 401 Not Authorized - the authenticated user does not have access to the requested data.
- 404 Not Found - the data requested does not exist in this Item
- 500 Internal Server Error - there was an exception while trying to retrieve the data. The exception type and message should be included in the result body.

Activity query and Job execution

OPTIONS (?agentName=<agent>)

Returns list of Jobs containing:

- Item link
- Activity path, name and type and properties
- State transition: origin state, target state and transition. Name, version and link to state machine.
- If defined, schema name, version, whether it is required and a link to the schema.
- Agent name and role

POST *<activityPath>(?<transitionName>&agentName=<agent>)*
- Default transition DONE if not given
- Request activity transition
- POST data contains the outcome
- Can also use URI param 'transition'

Return codes:
- *200 OK* - Successful transition
- *400 Bad Request* - Outcome not valid (including validation errors in response) or Script exception (including error message from script in response)
- *401 Unauthorized* - Agent doesn't hold correct role to perform this transition
- *404 Not Found* - Activity doesn't exist
- *409 Conflict* - Requested transition is no longer possible
- *500 Internal Server Error* - other error during execution, include exception details in response

Resource Loading
Shortcuts for quick referencing of description data. Currently implemented resources are 'schema' and 'stateMachine'.

GET */<resource>* - list all resources of the given type, with links
GET */<resource>/<name>* - list all versions of the given resource
GET */<resource>/<name>/<version>* - return the XML data for that resource. Last-modified gives the timestamp of the Event associated with that version.

Future work
- Item data pushing, change notification, possibly using WebSockets. Pubsub model with optional preloading as with ItemProxys preferable.
- Possibly administrative functions such as direct predefined step requests, though we would prefer to discourage this sort of interaction through this API.
- Additional path prefixes as shortcuts to description data and extensible by domains. Suggested domain extensions:
    - query
    - process
    - prefill
- Access the PropertyDescription of the Item's description to separate out the class identifying properties. This is not easy right now, because of the loose association

between Items and their descriptions. It may also be vague about subtypes.
It may be better to instead just supply the mutable and immutable properties.