FAST AND ADVANCED STORYBOARD TOOLS

FP7-ICT-2007-1-216048

http://fast.morfeo-project.eu

Deliverable D5.1.2
User Manual of the FAST Catalogue

Ismael Rivera, NUIG

Date: 25/01/2010

FAST is partially funded by the E.C. (grant code: FP7-ICT-2007-1-216048).

Version History

Rev. No.	Date	Author (Partner)	Change description
1.0	25/01/2010	Ismael Rivera (NUIG)	First version of the deliverable

Executive Summary

The present deliverable is intented to be the developer's manual of the prototype of the semantic catalogue. That said, whoever developing a component which will interact with the catalogue will find in this document a description of its architecture, the functionality provided together with the API (Application Programming Interface), data interchange formats, code errors and exceptions as well as several examples of usage.

Document Summary

Code	FP7-ICT-2007-1-216048	**Acronym**	FAST
Full title	Fast and Advanced Storyboard Tools		
URL	`http://fast.morfeo-project.eu`		
Project officer	Annalisa Bogliolo		

Deliverable	**Number**	D5.1.2	**Name**	User Manual of the FAST Catalogue
Work package	**Number**	5	**Name**	Semantic catalogue of screen-flow resources and back-end Web Services

Delivery data	**Due date**	27/02/2010	**Submitted**	27/02/2010
Status			final	
Dissemination Level	Public ☒ / Consortium ☐			
Short description of contents	This deliverable is the technical documentation for the prototype developed as part of the WP5. It describes the catalogue's architecture, data interchange formats, APIs and examples of how to interact with it.			
Authors	Ismael Rivera, NUIG			
Deliverable Owner (Partner)	NUIG		**email**	ismael.rivera@deri.org
			phone	+353 91 495338
Keywords	FAST, semantic catalogue, gadget catalogue, RDF store			

Table of contents

List of Tables

List of Figures

1 Introduction

This section starts establishing the goal and scope of the present document, shows how it is structured and details the relation to others documents and work packages.

1.1 Goal and Scope

This is an introductory manual for developers who want to adopt and use the FAST semantic catalogue. It explains the main functionalities implemented so far, an overview of its architecture and a detailed *Application Programming Interface* or API of the complete set of the operations offered through a REST service.

1.2 Structure of the document

The deliverable presents both the external and internal architecture in Section 3, then in Section 4 it is detailed the Catalogue API, query formats, interchange formats, error codes and so on.

1.3 Changes from previous version

This should be the first version of this deliverable since in M12 the deliverable 5.1 was not scheduled. Nevertheless, last year an extra document was written as an appendix to the FAST Catalogue in order to detail and clarify technical decisions and to have a technical reference manual. Hence, the deliverable D5.2.1 will evolve to become the D5.1.2, so it will be no longer needed by itself.

There are several new sections and major changes which need to be mentioned:

- Section 2. Installation Guide describes a set of steps to be followed in order to install and execute the FAST Catalogue. Also some configuration instructions are specified.
- Section 4.2. Content Negotiation states how the FAST Catalogue has adopted several of

the content negotiation principles.

- Section 4. RESTful Catalogue API has been intensively extended and modified in order to reflect all the new functionality and how every request or response is constructed. Some of the most important features added in this iteration are:
  - support for form elements, operators and back-end services (see Section ...),
  - convenience operations for recommendation and checking satisfaction and reachability for form elements, operators and back-end services (see Section ...),
  - and, last but not least, support for Screen planning (see Section 4.4).
- Several appendixes have been added with the JSON structure of every building block the Catalogue can deal with.
- Appendix A. Last year an evaluation to choose the best technology for the Catalogue was done. It was not included in any formal deliverable, so this year it has been documented as an appendix.

2 Installation Guide

This section provides instructions on how to manually install and configure the FAST Catalogue. The first part of this guide presents some requirements to be considered before the installation, then it gives broad general instructions, and the last part contains a more detailed installation notes for specific configurations.

2.1 System Requirements

In addition to the software itself, a standard FAST Catalogue installation has the following requirements:

- Java[1] (version 6 or later) is required to run the software.
- A Java Servlet Container that supports Java Servlet API 2.4 and Java Server Pages (JSP) 2.0, or newer. We recommend using a recent, stable version of Apache Tomcat[2]. At the time of writing, this is either version 5.5.x or 6.x.

In addition, there are various optional dependencies which are required if you want to use certain advanced features (see Section 2.3).

2.2 Obtaining the FAST catalogue

It is recommended to have a Subversion client installed before you download the code (although you can theoretically download files without Subversion, this would mean tediously downloading each individual file manually). The recommended software is the official Subversion client, available from the Subversion project page[3]. Note that this client uses a command-line interface, which the instructions below use. Alternatively, you can get subversioning software with a graphical user interface such as TortoiseSVN.

To download from the latest release (recommended), enter the following command from the

[1]http://java.sun.com/
[2]http://tomcat.apache.org/
[3]http://subversion.tigris.org/

command-line in the directory you wish to download to:

```
svn checkout https://svn.forge.morfeo-project.org/fast-fp7project/trunk/catalogue_service
```

2.3 Installation Instructions

Once you have got the source code, first step should be addressed is its compilation. You may manually compile the source yourself, however the FAST Catalogue comes along with a script to facilitate this task. This script has been made using a Java build tool called Ant [4]. Hence, we recommended to get and install such a tool in order to follow the instructions below.

Now you can compile, package and run the application via:

```
ant clean
ant prepare
ant compile
```

Or using a task which gather the previous three tasks and creates WAR file ready to distribute or deploy:

```
ant dist
```

After executing with no errors this task, a WAR file should have been created in the /dist directory. A WAR file, which stands for Web Application Archive, is just a JAR file used to distribute a collection of JavaServer Pages, servlets, Java classes, XML files, tag libraries and static Web pages (HTML and related files) that together constitute a Web application, in this case, the FAST Catalogue.

Then, last step is to deply the WAR file into the servlet container. If the server chosen was Apache Tomcat, the procedure for deploying a Web application is:

1. Stop Tomcat.
2. Delete existing deployment. If you have previously deployed "foo.war" in TOMCAT_HOME/webapps, then it has been unpacked into webapps/foo/... You must delete this directory and all its contents. On Unix, this can be done with rm -r $TOMCAT_HOME/webapps/foo

[4]http://ant.apache.org/

3. Copy WAR file to TOMCAT_HOME/webapps/.
4. Start Tomcat.

Now you have successfully built the FAST catalogue, however we highly recommend you to read Section 2.4 and tune your FAST Catalogue instance.

2.4 Configuration

The following section covers the configuration options you may have to set up before start using the Catalogue. The FAST Catalogue relies on a Sesame repository for the persistent storage. Sesame repositories can be local or remote. Remote does not mean, it has to be installed in a different machine, but in a servlet container such as Apache Tomcat. The advantage of remote repositories is the possibility of accessing it via a SPARQL endpoint. These configuration aspect can be found in the file *repository.properties* in the root directory where the Catalogue has been deployed.

Independently the type of repository, the parameter *serverURL* needs to take the value of the URL where the FAST Catalogue instance has been deployed. If you are testing the application in your machine, it may be something like `http://localhost:8080/FASTCatalogue`.

Local repositories just need the parameter *storageDir* pointing at a local directory, which has write permission in order to allow Sesame to create files in that directory or to delete any file in the directory. For instance:

```
storageDir=D:\\FAST\\catalogue\\repository
```

Nevertheless, setting up a remote repository requires a few more steps. First the OpenRDF Sesame Server needs to be installed. The installation guide of this server is detailed in the Chapter 5 and 6 of the User Guide for Sesame 2.2 [B.V., 2008]. Once the Sesame server is deployed, and a repository has been created, the parameters sesameServer and repositoryID will take the values of the Sesame server URL and the repository identifier. For instance:

```
sesameServer=http://localhost:8880/openrdf-sesame
repositoryID=c2
```

2.5 Hello World!

It is recommended to check if the FAST Catalogue has been successfully installed. To do that, a simple HTTP request, using a Web browser, can be sent to the URL the Catalogue is deployed, and specifying a type of resource such as Screen. The URL would be similar to `http://localhost:8080/FASTCatalogue/screens`. Depending on the variant chosen, the response may be different. For JSON, the response should be `[]`, but in HTML the message `No screens found.` should be shown.

3 Architecture Overview

This section presents a technical overview of the internal catalogue's architecture and which external components will interact with it. Basically three layers can be distinguished: Presentation, Business Logic and Persistence layer (see Figure 3).

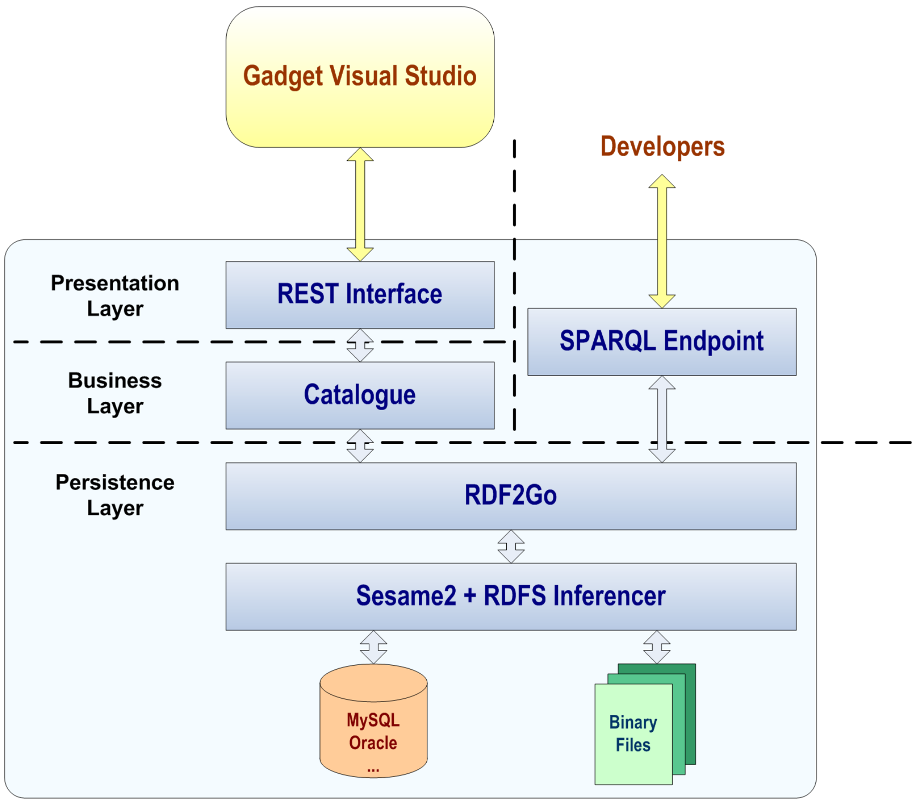

Figure 1: Catalogue's Architecture

The Presentation layer will be the public interface of the catalogue. The main purpose of this layer is to provide its functionality to other components within the Gadget Visual Studio. The service is offered in a REST (REpresentational State Transfer) style, making it easy to consume than other complex APIs. REST is an architectural style, not a toolkit or a standard. Even though, it makes use of standards like HTTP, URI, Resource Representations (XML, HTML, JSON, JPEG, etc.) or MIME types (text/xml, text/html, application/json, image/jpeg, etc.). Another characteristic of the REST style is its stateless assumption. The catalogue adopts this strategy; therefore every request needs a complete set of information in order to prepare the response.

As part of the Presentation layer, a SPARQL endpoint is offered using the SPARQL protocol service as defined in the SPROT [Grant et al., 2008] specification. The SPARQL endpoint is mostly offered to enable other developers to query directly the catalogue knowledge base using SPARQL queries. This feature is supported by the Sesame RESTful HTTP interface for SPARQL Protocol for RDF.

The Business Logic layer contains all the FAST domain-specific processing. It provides functions to interact with all the elements of the domain model specified in [Möller, 2009]. It processes the input from the Presentation layer, creating specific objects modelling the business logic, and vice versa. In addition it interacts with the Persistence layer in order to persist the model.

The Persistence layer provides an API which allows the Business layer to work with a standard set of objects that read and save their state to the triple store. For this reason, on top of this layer it is used RDF2Go, an abstraction over triple (and quad) stores. The RDF2Go API allows interacting with the semantic representation of the model (triples) in a generic manner, and also brings the flexibility of choosing different triple stores to persist them. The selected implementation for the triple stored is Sesame 2. Moreover, this framework is completely extensible and configurable regarding to storage mechanisms, inferencers, RDF file formats, query result formats and query languages. For the actual catalogue prototype, the storage mechanism used is the native file storage system of Sesame, the inferencer used is a subset of the RDFS entailment rules [Hayes, 2004] following a forward-chaining policy.

4 RESTful Catalogue API

This section provides a high-level overview of the Catalogue API. It describes the calls or operations supported, specific parameters and responses for each operation, supported interchange formats and some examples to facilitate the understanding of the API.

4.1 JSON Interchange Format

The format supported, and in which they must be constructed, by every HTTP request is JSON[5]. JSON is a lightweight data interchange format whose simplicity has resulted in widespread use among web developers, easy to read and write and able to using any programming language because its structures map directly to data structures used in most programming languages.

Every HTTP request should be encoded using the MIME type `application/json` and the charset `UTF-8`.

The following is an example of a Find & Check request:

```
{
  "canvas": [
    { "uri": "http://localhost:8080/catalogue/screens/238" },
    { "uri": "http://localhost:8080/catalogue/screens/323" }
  ],
  "elements": [
    { "uri": "http://localhost:8080/catalogue/screens/636" }
  ],
  "domainContext": {
    "tags": [
      {
        "label": { "en-GB": "Amazon" },
        "means": "http://dbpedia.org/page/Amazon.com"
      }
    ],
    "user": "irivera",
  },
  "criterion": "reachability"
}
```

[5]http://www.json.org/

4.1.1 Internationalisation I18n

In order to offers an adaptable solution to various languages and regions without major engineering changes, internationalisation is considered from an early stage in the catalogue development. Underlying representation technologies used to develop the catalogue (RDF/XML and JSON) allow to implement this feature. This feature is implemented by the addition of a language tag to every 'string' desired. The specification of this language tag is composed by the language code and the country code, following the ISO 639[6] for languages and the ISO 3166[7] for countries. The following example illustrates how to use it properly in both formats.

```
{
  "label": {
    "en-GB": "Simple Example",
    "es-ES": "Ejemplo Sencillo",
    "de-DE": "Einfaches Beispiel"
  },
  "description": {
    "en-GB": "This is a simple example of a screen",
    "es-ES": "Esto es un ejemplo sencillo de pantalla",
    "de-DE": "Dies ist ein einfaches Beispiel für einen Bildschirm"
  },
}
```

```
<rdf:Description rdf:about="http://www.morfeoproject.eu/fast/fco#ExampleScreen">
  <rdf:type rdf:resource="http://www.morfeoproject.eu/fast/fco#Screen"/>
  <rdfs:label xml:lang="en-GB">Simple Example</rdfs:label>
  <rdfs:label xml:lang="es-ES">Ejemplo Sencillo</rdfs:label>
  <rdfs:label xml:lang="de-DE">Einfaches Beispiel</rdfs:label>
  <description
    xmlns="http://purl.org/dc/terms/"
    xml:lang="en-GB">This is a simple example of a screen</description>
  <description
    xmlns="http://purl.org/dc/terms/"
    xml:lang="es-ES">Esto es un ejemplo sencillo de pantalla</description>
  <description
    xmlns="http://purl.org/dc/terms/"
    xml:lang="de-DE">Dies ist ein einfaches Beispiel für einen Bildschirm</description>
</rdf:Description>
```

Until now, using internationalisation is allowed for the label and the description of a screen, but in the near future, more attributes will benefit from this feature.

[6] http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt
[7] http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

4.2 Content Negotiation

Before any major details about how content negotiation is addressed by the catalogue, a few concepts need to be understood. First of all, say a URI is not a file name. It is common to see a URI as the location for a file, but a URI or Universal Resource Identifier is not a URL: it is not a file name or location, it is an identifier (or a reference) to a resource. Then by dereferencing a URI the actual resource can be retrieved. Let's see a request to retrieve the resource "Screen 123" is sent to the server. The resource is just a piece of information, however the server will answer an HTML document with a text describing the screen or an image representing it. On the Web, these equivalent representations of a resource are called variants, and content negotiation is the mechanism used to determine which of the representations of most appropiate for a given request.

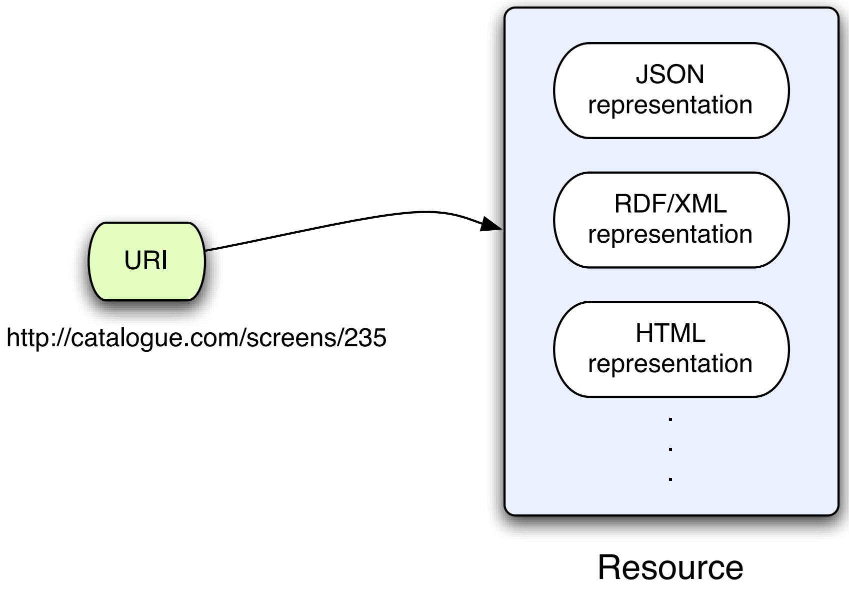

Figure 2: Content Negotiation

In summary, the basic idea of content negotiation, as stated in the HTTP 1.1 specification [Fielding et al., 1999], is to serve the best variant for a resource, taking into account what variants are available, what variants the server may prefer to serve, what the client can accept, and with which preferences: in HTTP, this is done by the client which may send, in its request, Accept headers (Accept, Accept-Language and Accept-Encoding), to communicate its capabilities and preferences in Format, Language and Encoding, respectively.

In fact, what the catalogue really does is "format negotiation" since the alternate representationos are just based on the selection of the media type, through the Accept header, but does not consider different languages or encoding types. The formats supported are JSON, RDF+XML and HTML.

Lastly, content negotiation needs to identify which player is going to take the lead on it. There are two kinds of content negotiation which are possible in HTTP: server-driven and agent-driven negotiation. The approach followed by the catalogue is agent-driven negotiation, hence selecting a specific representation for a resource is responsibility of the user agent. If none is specified, by default, the server will choose the JSON representation.

4.3 API Calls

This section contains detailed descriptions of the interface provided by the catalogue via REST services, detailing request parameters, response elements, any special errors and examples of requests and responses. The URL format is also specified for each operation, where 'catalogueURL' has to be replaced for the real URL the service is installed (e.g. http://demo.fast.morfeo-project.org/catalogue).

4.3.1 CRUD operations

Any building block within the Catalogue is a resource in terms of REST philosophy, hence it can be created, retrieved, modified or deleted, using a certaing URL and a specific HTTP method for every operation. Two concepts have to be defined: *collection*, as a set of resources which access URI is http://catalogueURL/<type>/ where catalogueURL is the specific URL where the Catalogue server is installed and <type> is the plural of the name of the building block (e.g. screens, services), and *member* of the collection, in other words, the building block itself, which access URI is http://catalogueURL/<type>/<id> where the <id> has to be replaced by the identifier of a specific building block. The details of the operations and which HTTP verb has to be used can be found in the Table 1.

The JSON structure for create any building bock can be found in the appendixes. All of them are

Table 1: CRUD operations

Resource	HTTP method	HTTP body	Description
Collection URI	GET	N/A	**List** the members of the collection.
Collection URI	PUT	N/A	Not used.
Collection URI	POST	JSON representation of the building block	**Create** a new entry in the collection where the URI is assigned automatically by the collection. The URI created is returned by this operation.
Collection URI	DELETE	N/A	Not used.
Member URI	GET	N/A	**Retrieve** the addressed member of the collection.
Member URI	PUT	JSON representation of the building block	**Update** the addressed member of the collection or create it with a defined URI.
Member URI	POST	N/A	Not used.
Member URI	DELETE	N/A	**Delete** the addressed member of the collection.

composed of a common structure which can be found in Section Appendix B, and some specific information depending on the type. Screen-flows are defined in Appendix C, screens in Appendix D and forms, operators and back-end services share the same structure detailed in Appendix E. However, a few examples of request and responses are shown in order to clarify how the request are constructed and sent, and how the responses look like.

To create a new screen, a POST request is send to the catalogue server, including the JSON representation of the screen as the body of the request:

```
{
  "code": "http://demo.fast.morfeo-project.org/code/amazonList.js",
  "creationDate": "2010-01-26T17:01:13+0000",
  "creator": "irivera",
  "description": {"en-gb": "Please fill the description..."},
  "homepage": "http://fast.morfeo-project.eu/",
  "icon": "http://demo.fast.morfeo-project.org/images/amazonList.png",
  "id": "28",
  "label": {"en-gb": "Amazon Product List"},
  "name": "ProductList",
  "postconditions": [
    [
      {
        "id": "item",
        "label": { "en-gb": "An item" },
        "pattern": "?I
                    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                    http://aws.amazon.com/AWSECommerceService#Item",
        "positive": "true"
      }
    ]
  ],
  "preconditions": [
    [
      {
        "id": "filter",
        "label": { "en-gb": "A search criteria" },
        "pattern": "?F
                    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                    http://aws.amazon.com/AWSECommerceService#SearchCriteria",
        "positive": "true"
      }
    ]
  ],
  "rights": "http://creativecommons.org/",
  "screenshot": "http://demo.fast.morfeo-project.org/images/amazonProductList.png",
  "tags": [{"label": {"en-gb": "amazon"}}],
  "version": "0.1"
}
```

The URI of the screen is created using the id specified in the JSON request. So, the response for this operation is:

```
{
  "code": "http://demo.fast.morfeo-project.org/code/amazonList.js",
  "creationDate": "2010-01-26T17:01:13+0000",
  "creator": "irivera",
  "description": {"en-gb": "Please fill the description..."},
  "homepage": "http://fast.morfeo-project.eu/",
  "icon": "http://demo.fast.morfeo-project.org/images/amazonList.png",
  "id": "28",
  "label": {"en-gb": "Prueba1"},
  "name": "ProductList",
  "postconditions": [
    [
      {
        "id": "item",
        "label": { "en-gb": "An item" },
        "pattern": "?I
                    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                    http://aws.amazon.com/AWSECommerceService#Item",
        "positive": "true"
      }
    ]
  ],
  "preconditions": [
    [
      {
        "id": "filter",
        "label": { "en-gb": "A search criteria" },
        "pattern": "?F
                    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                    http://aws.amazon.com/AWSECommerceService#SearchCriteria",
        "positive": "true"
      }
    ]
  ],
  "rights": "http://creativecommons.org/",
  "screenshot": "http://demo.fast.morfeo-project.org/images/amazonProductList.png",
  "tags": [{"label": {"en-gb": "amazon"}}],
  "type": "screen",
  "uri": "http://localhost:8080/catalogue/screens/28",
  "version": "0.1"
}
```

To obtain all the screens stored in the catalogue a HTTP GET request is sent to the Collection URI and the response may be something like this:

```
[
  {
    "code": "http://demo.fast.morfeo-project.org/code/amazonList.js",
    "creationDate": "2010-01-26T17:01:13+0000",
    "creator": "irivera",
    "description": {"en-gb": "Please fill the description..."},
    "homepage": "http://fast.morfeo-project.eu/",
    "icon": "http://demo.fast.morfeo-project.org/images/amazonList.png",
    "id": "28",
    "label": {"en-gb": "Prueba1"},
    "name": "ProductList",
    "postconditions": [
      [
        {
          "id": "item",
          "label": { "en-gb": "An item" },
          "pattern": "?I
                      http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                      http://aws.amazon.com/AWSECommerceService#Item",
          "positive": "true"
        }
      ]
    ],
    "preconditions": [
      [
        {
          "id": "filter",
          "label": { "en-gb": "A search criteria" },
          "pattern": "?F
                      http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                      http://aws.amazon.com/AWSECommerceService#SearchCriteria",
          "positive": "true"
        }
      ]
    ],
    "rights": "http://creativecommons.org/",
    "screenshot": "http://demo.fast.morfeo-project.org/images/amazonProductList.png",
    "tags": [{"label": {"en-gb": "amazon"}}],
    "type": "screen",
    "uri": "http://localhost:8080/catalogue/screens/28",
    "version": "0.1"
  },
  {
    "creationDate": "2009-02-07T09:59:52+0000",
    "creator": "fabio",
    ...,
  "uri": "http://localhost:8080/catalogue/screens/14",
    "version": "1.0"
  },
  {
    "creationDate": "2009-02-07T09:59:52+0000",
    "creator": "javier",
    ...,
  "uri": "http://localhost:8080/catalogue/screens/564",
    "version": "1.0"
  }
]
```

4.3.2 Screen Find

The find operation searches inside the catalogue for any screen which could be somehow related to the gadget the user is creating. It provides a recommended set of screens depending on the domain context, the canvas, and so on.

The specific URL to access this operation is http://catalogueURL/screens/find using HTTP POST method. From now on, the method 'find' only considers the domain context and the pre/postconditions of the screens. It will try to satisfy all the unsatisfied preconditions of a given list of screens also known as canvas. The results will be all the screens stored in the catalogue which fulfil some of these preconditions, and are tagged with the tags specified in the domain context. The request parameters are shown in Table 2 and the response parameters are shown in Table 3.

Following is shown an example of usage of this operation considering a canvas with the screen `http://localhost:8080/catalogue/screens/654`.

```
{
  canvas: [
    { uri: "http://localhost:8080/catalogue/screens/654" }
  ],
```

Table 2: Screen Find Request Parameters

Name	Description	Type
canvas	The canvas is composed by a list of screens. Only the URI is needed.	Optional
domainContext	The domain context contains a list of tags and a user.	Optional
elements	It is a list of resources, previously stored in the catalogue. For example, the list of screens recommended last execution. Only accepts screens.	Optional

Table 3: Screen Find Response Parameters

Name	Description	Type
N/A	A list of recommended screens URIs is returned.	Required

```
  domainContext: {
    tags: [],
  user: null
  },
  elements: []
}
```

After the execution of the recommendation algorithm, the response given by the catalogue is:

```
[
  "http://localhost:8080/catalogue/screens/371",
  "http://localhost:8080/catalogue/screens/24",
  "http://localhost:8080/catalogue/screens/253",
  "http://localhost:8080/catalogue/screens/12"
]
```

4.3.3 Screen Check

This operation verifies the state of certain list of screens depending on a specific criterion. The only criterion accepted in this stage of the prototype is 'reachability'. A screen will be reachable if it all its preconditions are satisfied by postconditions of the screens the canvas contain. The specific URL to access this operation is http://catalogueURL/screens/check using HTTP POST method.

The scenario for the following example is composed by a screen in the canvas which precondition

Table 4: Screen Check Request Parameters

Name	Description	Type
canvas	The canvas is composed by a list of screens. Only the URI is needed.	Optional
domainContext	The domain context contains a list of tags and a user.	Optional
elements	It is a list of resources, previously stored in the catalogue. For example, the list of screens recommended last execution. Only accepts screens.	Optional
criterion	The criterion specifies what it has to be check. The only possible supported now is 'reachability' in order to check if the preconditions of a screen are satisfied.	Required

Table 5: Screen Check Response Parameters

Name	Description	Type
Canvas	The canvas indicating which screens satisfy the critetion. More aditional information may be included, for example, a satisfaction attribute for every precondition for reachability.	Optional
Domain Context	The domain context contains a list of tags and a user.	Optional
Elements	The elements indicating which satisfy the critetion. More aditional information may be included as in the Canvas.	Optional

is a foaf:Person which foaf:workplaceHomepage is `http://www.deri.ie/`, and a screen in the elements list which precondition is a foaf:Person. The state to check in this case is the reachability and satisfaction of the screens and preconditions.

```
{
  "canvas": [
    { "uri": "http://localhost:8080/catalogue/screens/238" }
  ],
  "elements": [
    { "uri": "http://localhost:8080/catalogue/screens/636" }
  ],
  "domainContext": {
    "tags": [
      {
        "label": { "en-GB": "Amazon" },
        "means": "http://dbpedia.org/page/Amazon.com"
      }
    ],
    "user": "irivera",
  },
  criterion: "reachability"
}
```

The response shows that there are not reachable screens at the moment, and preconditions are not satisfied.

```
{
  "canvas": [{
    "preconditions": [{
      "label": {"en-gb": "A person working in DERI"},
      "pattern": "?person rdf:type foaf:Person .
                  ?person foaf:workplaceHomepage http://www.deri.ie/",
      "positive": true,
      "satisfied": false
    }],
    "reachability": false,
    "uri": "http://localhost:8080/catalogue/screens/238"
  }],
  "elements": [{
    "preconditions": [{
      "label": {"en-gb": "A person"},
      "pattern": "?person rdf:type foaf:Person",
      "satisfied": false
    }],
    "reachability": false,
    "uri": "http://localhost:8080/catalogue/screens/636"
  }]
}
```

4.3.4 Screen Find & Check

For convenience and to minimise the number of request, the Find operation may be combined with the Check operation. This operation will look for new screens based on the given information, and will perform the Check to both the screens in the canvas and the screens in the *elements* attribute. The format of the request is the same as the one used in the Check operation:

```
{
  "canvas": [
    { "uri": "http://localhost:8080/catalogue/screens/238" }
  ],
  "elements": [
    { "uri": "http://localhost:8080/catalogue/screens/636" }
  ],
  "domainContext": {
    "tags": [
      {
        "label": { "en-GB": "Amazon" },
        "means": "http://dbpedia.org/page/Amazon.com"
      }
    ],
    "user": "irivera",
  },
  criterion: "reachability"
}
```

The response for the above request is:

```
{
  "canvas": [{
    "preconditions": [{
      "label": {"en-gb": "A person working in DERI"},
      "pattern": "?person rdf:type foaf:Person .
                  ?person foaf:workplaceHomepage http://www.deri.ie/",
      "positive": true,
      "satisfied": false
    }],
    "reachability": false,
    "uri": "http://localhost:8080/catalogue/screens/238"
  }],
  "elements": [
    {
      "preconditions": [{
        "label": {"en-gb": "A person"},
        "pattern": "?person rdf:type foaf:Person",
        "satisfied": false
      }],
      "reachability": false,
      "uri": "http://localhost:8080/catalogue/screens/636"
    },
    {
      "preconditions": [],
      "reachability": true,
      "uri": "http://localhost:8080/catalogue/screens/132"
    }
  ]
}
```

4.3.5 GetMetadata

Although the information about a resource can be obtained by its specific retrieval operation supported by the CRUD interface, sometimes is needed to retrieve information which would imply a request per resource, hence this operation allow to get the metadata of a list of resources in only one request. This prototype supports this operation for screen-flows and screens. The specific URL to access this operation is http://catalogueURL/getmetadata using HTTP POST method. Table 6 shows the parameters needed for the invocation and Table 7 details the different parameters may contain a response.

Table 6: GetMetadata Request Parameters

Name	Description	Type
N/A	A list of URIs.	Required

Table 7: GetMetadata Response Parameters

Name	Description	Type
screen-flows	A set of screen-flows with all the metadata associated to them..	Optional
screens	A set of screens with all the metadata associated to them.	Optional
forms	A set of forms with all the metadata associated to them.	Optional
operators	A set of operators with all the metadata associated to them.	Optional
backendservices	A set of back-end services with all the metadata associated to them.	Optional

The following example send two URIs of screens and a URI of a screen-flow to the GetMetadata operation:

```
[
  "http://localhost:8080/catalogue/screens/12",
  "http://localhost:8080/catalogue/screens/48",
  "http://localhost:8080/catalogue/forms/9",
]
```

The response obtained are two lists, one containing all the metadata regarding to the screen-flows and another one with the information about the screens:

```
{
  "screenflows": [],
  "screens": [
    {
      "creationDate": "2009-02-07T09:59:52+0000",
      "creator": "irivera",
      ...,
      "uri": "http://localhost:8080/catalogue/screens/12",
      "version": "1.0"
    },
    {
      "creationDate": "2009-02-07T09:59:52+0000",
      "creator": "irivera",
      ...,
      "uri": "http://localhost:8080/catalogue/screens/12",
      "version": "1.0"
    }
  ],
  "forms": [
    {
      "actions": [
        {
          "name": "init",
          "preconditions": [],
          "uses": []
        },
        {
          "name": "showTable",
          "preconditions": [{
            "id": "list",
            "label": {"en-gb": "A product list"},
            "pattern": "?PList
                        http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                        http://aws.amazon.com/AWSECommerceService#ProductList",
            "positive": true
          }],
          "uses": []
        }
      ],
      ...,
      "uri": "http://localhost:8080/FASTCatalogue/forms/9",
      "version": "1.0"
    },
  ],
  "operators": [],
  "backendservices": []
}
```

4.3.6 Screen Component Find & Check

Similarly to the Screen Find & Check described in Section 4.3.4, this operation search for screen components (forms, operators and back-end services) through the catalogue in order to satisfy a given request, and it will attach information about satisfaction and reachability as well.

A request is formed of the following parameters:

preconditions Preconditions of the screen.

postconditions Postcondition of the screen.

canvas The canvas is composed by a list of screens. Only the URI is needed.

forms It is a list of forms, such as the list of forms recommended last execution.

operators It is a list of operators, such as the list of operators recommended last execution.

backendservices It is a list of back-end services, such as the list of back-end services recommended last execution.

domainContext The domain context contains a list of tags and a user.

pipes List of pipes for the connection between screen components.

selectedItem URI of a screen component from the canvas.

The screen components recommended are based on the given domain context and the pre/postconditions of the any of the resources in the canvas or the conditions of the parameters preconditions and postconditions. Hence, any screen component from the catalogue which has any of these conditions may be suitable for the screen, so it is included in the response, in the list corresponding to the type of screen component.

An example of request is the following:

```
{
  "preconditions": [{
    "id": "Searchcriteria_1",
    "label": {"en-gb": "A search criteria"},
    "pattern": "?F
                http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                http://aws.amazon.com/AWSECommerceService#SearchCriteria",
    "positive": true,
  }],
  "postconditions": [],
  "canvas": [
    { "uri": "http://localhost:8080/catalogue/forms/1" },
    { "uri": "http://localhost:8080/catalogue/services/1" }
  ],
  "forms": [],
  "operators": [],
  "backendservices": [],
  "domainContext": {
    "tags": [],
    "user": null
  },
  "pipes": [
    {
      "from": {
        "buildingblock": "http://localhost:8080/catalogue/services/1",
        "condition": "list"
      },
      "to": {
        "action": "showTable",
        "buildingblock": "http://localhost:8080/catalogue/forms/1",
        "condition": "list"
      }
    }
  ],
  "selectedItem": "http://localhost:8080/catalogue/forms/1"
}
```

And the response according to the above request is:

```
{
  "backendservices": [],
  "canvas": [
    {
      "actions": [{
        "name": "search",
        "preconditions": [{
          "id": "filter",
          "label": {"en-gb": "A search criteria"},
          "pattern": "?F
                      http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                      http://aws.amazon.com/AWSECommerceService#SearchCriteria",
          "positive": true,
          "satisfied": false
        }],
        "satisfied": false,
        "uses": []
      }],
      "reachability": false,
      "uri": "http://localhost:8080/catalogue/services/1"
    },
    {
      "actions": [
        {
          "name": "init",
          "preconditions": [],
          "satisfied": true,
          "uses": []
        },
        {
          "name": "showTable",
          "preconditions": [{
            "id": "list",
            "label": {"en-gb": "A product list"},
            "pattern": "?PList
                        http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                        http://aws.amazon.com/AWSECommerceService#ProductList",
            "positive": true,
            "satisfied": false
          }],
          "satisfied": false,
          "uses": []
        }
      ],
      "reachability": true,
      "uri": "http://localhost:8080/catalogue/forms/1"
    }
  ],
  "connections": [
    {
      "from": {
        "buildingblock": null,
        "condition": "Searchcriteria_1"
      },
      "to": {
        "action": "search",
        "buildingblock": "http://localhost:8080/catalogue/services/1",
        "condition": "filter"
      }
    },
  ],
  "forms": [],
  "operators": [],
  "pipes": [
    {
      "from": {
        "buildingblock": "http://localhost:8080/catalogue/services/1",
        "condition": "list"
      },
      "satisfied": true,
      "to": {
        "action": "showTable",
        "buildingblock": "http://localhost:8080/catalogue/forms/1",
        "condition": "list"
      }
    }
  ],
  "postconditions": []
}
```

For a screen component to be reachable, its preconditions need to be connected by a pipe to any other screen component which is already reachable, or to any of the screen preconditions. Moreover, the pipe needs to be satisfied, in other words, conditions in both edges are compatible.

The response includes an attribute called "connections". This attribute will be retrieved when "selectedItem" is given, and it will contain a list of potential pipes to connect to that item.

4.4 Planning

While creating a screen-flow, the user need to find proper screens which will satisfy her necessity. Once the user has selected one or several screens to be included in her screen-flow, she needs to find screens which satisfy the preconditions of these screens, in order to create an executable screen-flow. The operation Find, detailed in Section 4.3.2, helps the user to find these screens, however the response is just a set of screens which somehow may help to accomplish her goal, but not in a single and straight-forward step.

This functionality offers to the user the possibility to receive a whole set of screens, or plan, which will make a given screen, or goal, reachable. These plans take into account the screen which are already in the canvas, in order to minimise the number of screens.

The specific URL to access this operation is `http://catalogueURL/planner`, and to execute the operation a POST request needs to be done. The request body is composed of:

goal Required. URI of the goal screen.

canvas Required. List of screens URIs.

page Optional. Number of the page to be retrieved.

per_page Optional. Number of plans to be included per page in the response.

Let's see the following example:
Request

```
{
  "goal": "http://localhost:8080/catalogue/screens/35",
  "canvas": [
    { "uri": "http://localhost:8080/catalogue/screens/56" },
    { "uri": "http://localhost:8080/catalogue/screens/85" }
  ],
  "page": 1,
  "per_page": 10
}
```

Response

```
[
  [
    "http://localhost:8080/catalogue/screens/13",
    "http://localhost:8080/catalogue/screens/23"
  ],
  [
    "http://localhost:8080/catalogue/screens/79",
    "http://localhost:8080/catalogue/screens/37",
    "http://localhost:8080/catalogue/screens/14"
  ],
  [
    "http://localhost:8080/catalogue/screens/16",
    "http://localhost:8080/catalogue/screens/44",
    "http://localhost:8080/catalogue/screens/32"
  ]
]
```

The response is ranked based on adding the minimum number of new screens into the canvas, and giving preference to the screens the user has previously selected (canvas).

In the above example, let's assume the first plan is using the two screens from the canvas to accomplish the goal, and the second plan is able to make the goal reachable by itself. That said, a screen-flow with the goal and the second plan could be executable, and it would be composed of just four screens, however, taking the first plan, the resulting screen-flow will be composed of five screens. It could make sense to place the second plan in first position, but the plan with just two screens reuse the screens from the canvas which has been added by the user.

4.5 API Error Codes

There are two types of error codes, client and server.

- Client error codes are generally caused by the client and might be an invalid domain or an invalid request parameter. These errors are accompanied by a 4xx HTTP response code. For example: ResourceNotFound.
- Server error codes are generally caused by a server-side issue and should be reported. These errors are accompanied by a 5xx HTTP response code. For example: ServerUnavailable.

The following table lists all the error codes.

Table 8: API Error Codes

Error	Description	HTTP Status Code
ResourceNotFound	The resource <resourceURI> has not been found.	404 Not Found
AccessFailure	Access to the resource <resourceName> is denied.	403 Forbidden
InternalError	Request could not be executed due to an internal service error.	500 Internal Server Error
InvalidAction	The action <actionName> is not valid for this web service.	404 Bad Request
InvalidHttpRequest	The HTTP request is invalid. Reason: <reason>.	400 Bad Request
InvalidLiteral	Illegal literal in the filter expression.	400 Bad Request
InvalidParameterValue	The specified parameter value is not valid.	400 Bad Request
InvalidURI	The URI <requestURI> is not valid.	400 Bad Request
MissingAction	No action was supplied with this request.	400 Bad Request
MissingParameter	The request must contain the specified missing parameter.	400 Bad Request
NotYetImplemented	Feature <feature> is not yet available.	401 Unauthorized
ServiceUnavailable	The service is currently unavailable. Please try again later.	
UnsupportedHttpVerb	The requested HTTP verb is not supported: <verb>.	400 Bad Request
URITooLong	The URI exceeded the maximum limit of <maxLength>.	400 Bad Request

References

[B.V., 2008] B.V., A. (2008). User guide for sesame 2.2.

[Fielding et al., 1999] Fielding, R. T., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext transfer protocol – http/1.1.

[Grant et al., 2008] Grant, K., Feigenbaum, L., and Torres, E. (2008). Sparql protocol for rdf. Recommendation, World Wide Web Consortium (W3C).

[Hayes, 2004] Hayes, P. (2004). Rdf semantics. Recommendation, World Wide Web Consortium (W3C).

[Möller, 2009] Möller, K. (2009). Ontology and conceptual model for the semantic characterisation of complex gadgets. Deliverable 2.2.2, FAST Project (FP7-ICT-2007-1-216048).

[Urmetzer et al., 2010] Urmetzer, F., Delchev, I., Hoyer, V., Janner, T., Rivera, I., Möller, K., Aschenbrenner, N., Fradinho, M., and Lizcano, D. (2010). State of the art in gadgets, semantics, visual design, SWS and catalogs. Deliverable D2.1.2, FAST Project (FP7-ICT-2007-1-216048).

Appendix A. Semantic Web Technologies Evaluation

In [Urmetzer et al., 2010] were presented different technologies used in the Semantic Web community regarding Web services and catalogues or repositories. These are potential technologies to serve as the basis for the Catalogue implementation. Said that, an evaluation was done in order to choose the most accurate technology with the most advantages for it. Of particular importance in this evaluation is real-time performance, because the GVS (Gadget Visual Storyboard) actually relies on the Catalogue in parts of its user interface.

Basically, two general solutions were considered. One was to build the Catalogue, and the internal components or building block of the gadgets, based on a dedicated SWS platform. The other was to build a light-weight solution from scratch based on RDF and RDFS.

For both solutions, a few ideas about their potential benefits and disadvantages were in mind beforehand. SWS platforms were very tempting to use, because the modelling approach for screens in FAST is quite similar to SWS, in how they define their inputs and outputs. E.g., both FAST and WSMO have pre- and post-conditions, OWL-S has inputs/outputs. Also, SWS platforms come with ready-made implementations for service composition, which could have been used for automatically combining screens to screen-flows. On the downside, it was expected that SWS platforms a bit too slow for some FAST requirements, as the need of real-time discovery. On the other hand, building a light-weight implementation directly on RDFS promised to be faster. Moreover, there is a broader, more mature and probably more active tool support. However, going in this direction would imply to invest a lot more work in modelling and implementation efforts, since all the features that SWS would bring straight away are could not be reused.

To perform the evaluation of the different approaches, an abstract random ontology of 40 concepts was created. About 70% of those were randomly assigned to be sub-concepts of other concepts. The abstract ontology was instantiated in WSMO, OWLS and RDFS. Then, 3 sample sets of random screens of three different sizes (100, 1000 and 10000) were created, such that each screen has 0..3 random pre- and post-conditions. For each set in each technology, every screen was taken to perform a match with any other screen, and the average response time was measured. Figure 4.5 shows the results of the evaluation (RDFS in blue, WSMO in yellow and OWL-S in green). Be aware that the scale of the time is logarithmic.

It can be seen that RDFS performs rather well, even in the 10.000 size set, whereas the other two are pretty far behind. E.g., average response times for WSMO is > 1.5 sec already for the smallest 100 size set. Hence, based on this evaluation the approach to follow for the implementation was

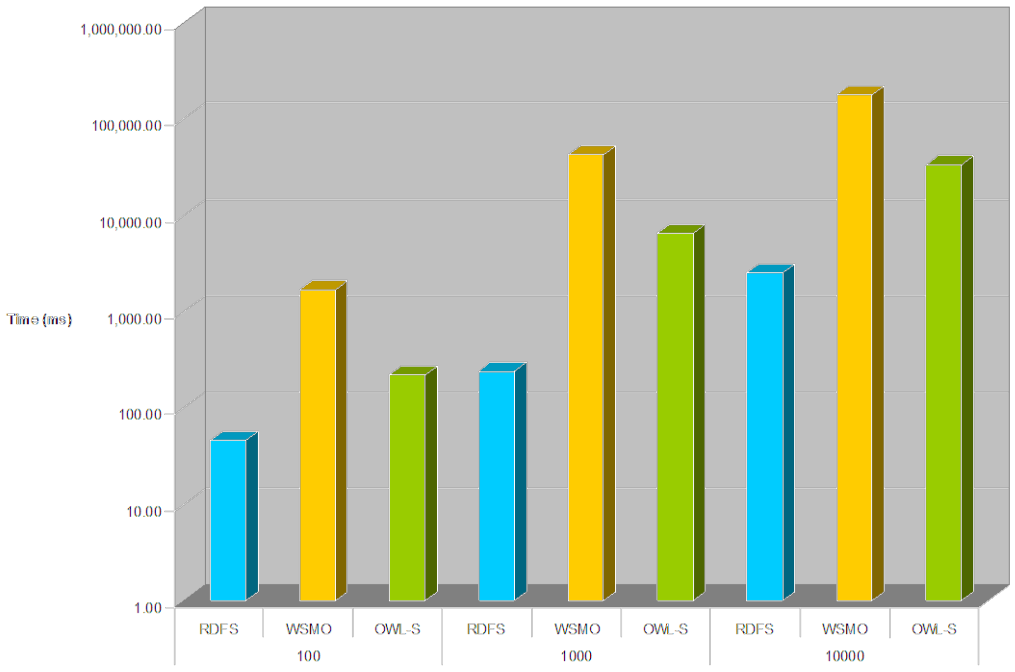

Figure 3: Semantic Web Technologies Performance Comparison

RDFS, since WSMO and OWL-S simply could not give the performance required for the real-time discovery.

Appendix B. Generic Building Block JSON Structure

```
{
  "code": "http://url.com/.../code.js",
  "creationDate": "2009-04-20T17:00:00+0100",
  "creator": "http://www.fabio.es/",
  "description": {"en-gb": "This is a description of the building block"},
  "tags": {
    "tags": [
      "ebay",
      "amazon",
    ],
    "user": <String> // not defined yet
  },
  "homepage": "http://www.homepage.ie/",
  "icon": "http://url.com/images/icon.png",
  "id": "Id of the resource",
  "label": {"en-gb": "Label or title of the building block"},
  "name": "Amazon Item to Ebay Filter",
  "rights": "http://creativecommons.org/",
  "screenshot": "http://www.deri.ie/eBayList-screenshot.jpg",
  "type": "type of the building block", // screenflow, screen, form, operator y resource
  "uri": "http://localhost:8080/catalogue/screens/525",
  "version": "1.0"
}
```

Appendix C. Screen-flow JSON Structure

```
{
  <GENERIC BUILDING BLOCK JSON>,
  "contains": [
    "http://purl.oclc.org/fast/ontology/gadget#Screen1257860772723",
    "http://purl.oclc.org/fast/ontology/gadget#Screen1257860720035",
  ],
}
```

Appendix D. Screen JSON Structure

```
{
  <GENERIC BUILDING BLOCK JSON>,
  "postconditions": [[{
    "id": <identifier:String>,
    "label": {"en-gb": "Purchase URL"},
    "pattern": "?P
                http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                http://aws.amazon.com/AWSECommerceService#PurchaseURL",
    "positive": true
  }]],
  "preconditions": [[{
    "id": "cart",
    "label": {"en-gb": "A shopping cart"},
    "pattern": "?C
                http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                http://aws.amazon.com/AWSECommerceService#ShoppingCart",
    "positive": true
  }]],
  "code": "URL of the code",
  "definition": {
    "buildingblocks: [
      { // Form
        "id": "form1", // unique for the container screen
        "uri": "http://purl.oclc.org/fast/ontology/gadget#Form670238"
      },
      { // Operators
        "id": "op1",
        "uri": "http://purl.oclc.org/fast/ontology/gadget#Operator213487"
      },
      {  // Backend services
        "id": "bs1",
        "uri": "http://purl.oclc.org/fast/ontology/gadget#Backendservice38399"
      },
      {...}
    ]
    "pipes": [
      {
        "from": {
          "buildingblock": "bs1",
          "condition": "cA"
        },
        "to": {
          "buildingblock": "op1",
          "condition": "cA", // can be the same id, unique for the building block
          "action": "filter"
        }
      },
    "triggers": [
      {
        "from": {
          "buildingblock": "form1",
          "name": "refresh"
        },
        "to": {
          "buildingblock": "bs1",
          "action": "list"
        }
      }
    ]
  }
}
```

Appendix E. Screen Component JSON Structure

```
{
  <GENERIC BUILDING BLOCK JSON>,
  "actions": [{
    "name": "filter",
    "preconditions": [[{
      "id": "item",
      "label": {"en-gb": "Ebay List"},
      "pattern": "?Item
                  http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                  http://aws.amazon.com/AWSECommerceService#Item",
      "positive": true
    }]],
    "uses": []
  }],
  "libraries": [],
  "postconditions": [[{
    "id": "filterEbay",
    "label": {"en-gb": "Ebay List"},
    "pattern": "?eFilter
                http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                http://developer.ebay.com/.../FindItemsAdvanced.html#Request",
    "positive": true
  }]],
  "triggers": ["itemAmazon"]
}
```