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

Ciprian Palaghita, Cyntelix
Ismael Rivera, NUIG

Date: 25/02/2011

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	19/02/2010	Ciprian Palaghita (Cyntelix) Ismael Rivera (NUIG)	Release for M24
2.0	25/02/2011	Ciprian Palaghita (Cyntelix) Ismael Rivera (NUIG)	Release for M36

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**	28/02/2011	**Submitted**	25/02/2011
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	Ciprian Palaghita, Cyntelix Ismael Rivera, NUIG			
Deliverable Owner	Cyntelix		**email**	cpalaghita@cyntelix.com
(Partner)			**phone**	+353 858494258
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 section enumerates and gives a brief overview of the changes made in the deliverable with regards to the version 1.0 released for the milestone M24.

- The installation guide has been modified to include some minor changes in the configuration, and a better example for the “Hello World!”.
- Two operations have been removed: ’Find’ and ’Check’ for screens, used at screen-flow design, since the same functionaly was offered by the operation Find&Check, so no functionality is duplicated or overlapped. However, using this unique operation does not mean that every request does a *find* and a *check*, these actions are optional and can be specified

in the request.

- There have been some substantial changes in the API methods, Find&Check, Planner and Metadata reflect these changes made in the Catalogue prototype.
- There have been created new operations in the API to support the prototype-based semantics for RDF as defined in D2.2.3 [Möller et al., 2011].
- The Catalogue Architecture has been updated to reflect the new functionality and other changes.
- There have been implemented new mechanisms for managing and importing ontologies. These mechanism are explained in 3.2.
- A new appendix has been added in order to explain the decisions and logic behind the main methods of the API, how the building block recommendation works, algorithm supporting the building block planning, matching and inferencing in pre-/post-conditions, and the integration with iServe[1], enhancing the web service discovery.
- There are two new resources, and their CRUD operations: concepts (classes) and samples (instances).

[1] http://iserve.kmi.open.ac.uk/

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[2] (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[3]. 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[4]. 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.

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

To download from the latest release (recommended), enter the following command from the 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 that 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 [5]. 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.

Note that it is also possible to get the compiled WAR file directly from the SVN server. This file can be located at `https://svn.forge.morfeo-project.org/fast-fp7project/distribution/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:

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

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
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 configuration is done in the file *repository.properties* in the root directory where the Catalogue has been deployed.

The resources (building blocks) created in the Catalogue have an unique identifier or URI. This URI is constructed relatively to the Catalogue URL which is configured by the parameter *serverURL*. If you are testing the application on your machine, it may look like `http://localhost:8080/FASTCatalogue`.

The FAST Catalogue relies on a Sesame repository for the persistent storage. Sesame repositories can be *local* or *remote*. Local repositories just need the parameter *storageDir* pointing at a local directory, with write permission in order to allow Sesame to create and delete the necessary binary files to store and retrieve the data.

```
storageDir=/opt/fast-catalogue/repository
```

Setting up a remote repository requires a few more steps. *Remote* means the repository is installed in a web server and accessed via HTTP, independently if it is on the same machine or on a different dedicated machine. 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.

```
sesameServer=http://sesameServerURL/openrdf-sesame
repositoryID=repository-id
```

Note: If you want to provide the data in the repository via a SPARQL endpoint, for flexible data access to third-party applications, the easiest way it is to configure the Catalogue using a remote repository, as in this case Sesame provides an HTTP accessible SPARQL endpoint by default.

2.5 Hello World!

It is recommended to check if the FAST Catalogue has been successfully installed. The simpliest manner is to send a HTTP GET request to retrieve concepts (classes) and attributes (properties) of the predefined ontologies to `http://catalogueURL/FASTCatalogue/concepts` (e.g. typing the address in a Web browser). The response is a list of all the concepts and their associated meta-data, as in the following example:


```
[
  {
    "attributes": [],
    "label": {"en": "A request to search for an item"},
    "tags": [{
      "label": {"en": "amazon"},
      "means": "http://www.amazon.com"
    }],
    "uri": "http://fast.morfeo-project.org/ontologies/amazon#SearchRequest"
  },
  {
    "attributes": [
      {
        "type": "http://xmlns.com/foaf/0.1/Document",
        "uri": "http://fast.morfeo-project.org/ontologies/amazon#smallImage"
      },
      {
        "type": "http://www.w3.org/2001/XMLSchema#double",
        "uri": "http://fast.morfeo-project.org/ontologies/amazon#price"
      }
    ],
    "label": {"en": "An item handled by the Amazon Service"},
    "tags": [{
      "label": {"en": "amazon"},
      "means": "http://www.amazon.com"
    }],
    "uri": "http://fast.morfeo-project.org/ontologies/amazon#Item"
  },
  ...
```

```
]
```

By default, the Catalogue does not include any building block, therefore any GET request for any type of building block, as for instance to `http://catalogueURL/FASTCatalogue/screens` should return an empty list `[]` (if JSON requested) or `No screens found` (if HTML requested).

3 Architecture Overview

This sections presents a technical overview of the Catalogue's architecture (see Figure 3), and how the interaction with external applications or API is done. The Catalogue platform has a modular design, which are spread in basically three layers, depending on the nature of each component:

- *Presentation tier* is the top layer which represents the public API for the Catalogue. It brings together all the underlying functionality in a simple but flexible manner to be used by other IDEs or applications. In FAST, this is the unifying interface with other components such as the Gadget Visual Storyboard or the Service Wrapper.
- *Logic tier* is the core layer which implements the business logic for the different processes involved in the Catalogue, and communicates with third party libraries or APIs.
- *Data tier* is the bottom layer. It offers storage capabilities in triple stores, relational databases and binary files, and acts as an abstraction to RDF model logic.

3.1 Presentation tier / Public API

The Presentation tier is the public interface or API of the Catalogue. The main purpose of this layer is to provide its functionality to the Gadget Visual Storyboard, or any other third party IDE or application. The API 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 or MIME types. 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. The API provided is explained in detail in Section 4.

As part of the Presentation tier, a SPARQL endpoint has been implemented 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.

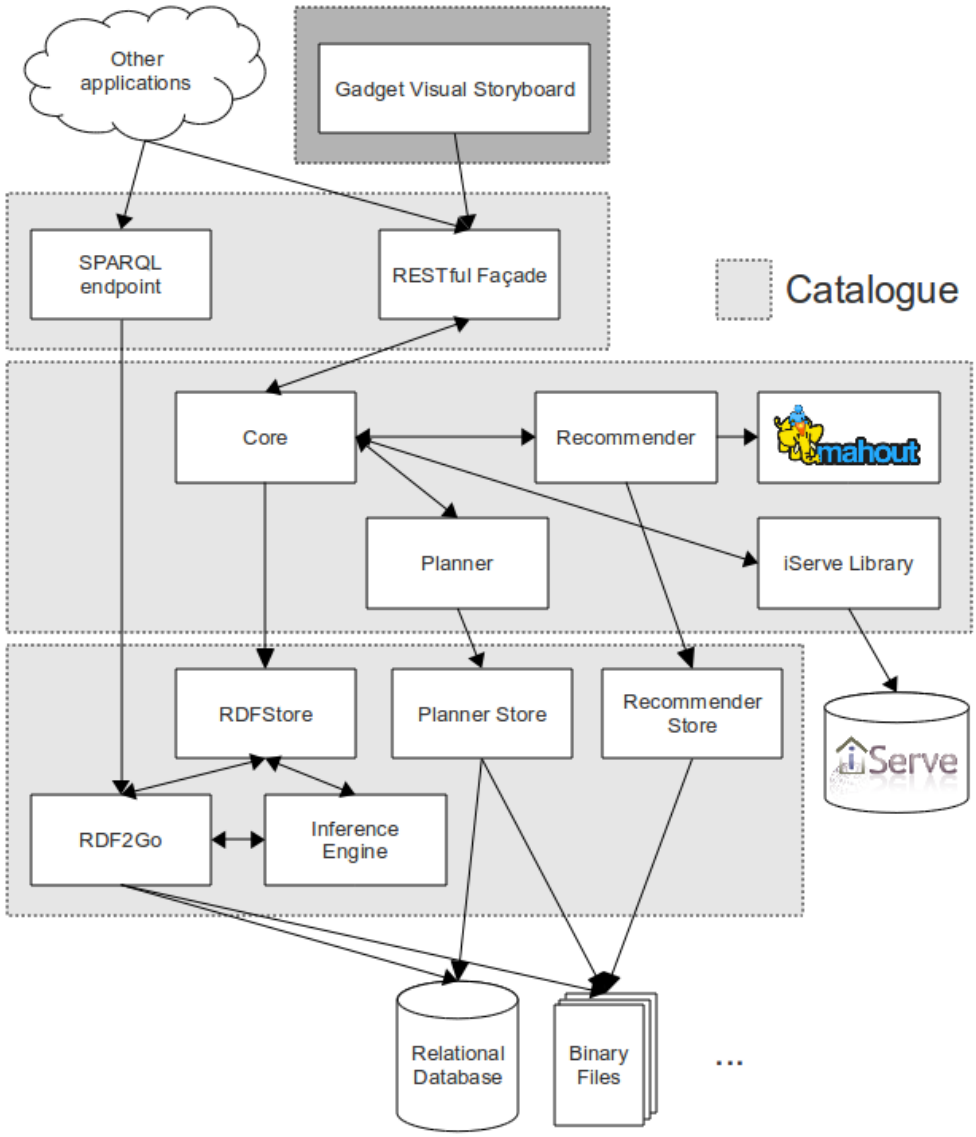

Figure 1: Catalogue's Architecture

3.2 Logic tier

The Logic tier or business logic layer contains all the FAST domain-specific processing. The resources and relations between each other, which define the domain model, are extracted from the FAST ontology [Möller et al., 2011]. This layer provides support to the functions defined in the API, and interact with the data tier to retrieve and store any information of the data model.

This layer provides several mechanisms to search and browse these different resources. To know more about these capabilities, please read Sections 4.6.3, 4.6.4 and 4.7 where the API calls are

defined, and Appendix F for more details about the implementation.

At the bottom level of the Gadget Architecture [Reyes et al., 2011] are the back-end services. These are RESTful services, SOAP-based web services, or similar. In FAST, these web services are wrapped, extracting their definition and encapsulating their behaviour. The creation of these service wrappers is on demand, and the input for this process is a particular definition of a web service. However, the Catalogue does not crawl the Web looking for Web services. Instead, and to enrich the collection of available web services, the Catalogue integrates and queries iServe, a platform for publishing Semantic Web Services as linked data, no matter their original format. Therefore, thousands of web services defined or annotated in a semantic way, are also served from the Catalogue, in a uniform manner.

The Catalogue is a semantic application. As a consequence, it must handle and work with ontologies and vocabularies. The business model itself is described in the FAST ontology, which is included in the Catalogue along with a few established ontologies integrated in its core (see [Möller et al., 2011]). However, the definition of the pre-/post-conditions of the building blocks are open to any ontology or vocabulary, and they may reference classes and properties from any of them. To have a better understanding of these classes, properties, and so on, the Catalogue needs to import into its own knowledge base the ontology in which they are described along with the relations with other classes and properties. In order to acquire these vocabularies, it follows the basic recipe dictated in [Berrueta et al., 2008] which says that deferencing a vocabulary URI should provide a machine-processable (RDF) content back. Nonetheless, this is just a good practice and not every ontology published on the Internet follows it. Hence, going a step further, whenever an ontology cannot be retrieved by deferencing its URI, the Catalogue queries Sindice[6], a semantic search engine, which crawls and contains thousands of vocabularies and metadata as their provenance, which the Catalogue can take advantage of.

3.3 Data tier

The Data tier abstracts the interaction with triple stores, RDF models and physical persistence to disk, database, etc. It provides different mechanisms to allow the Catalogue the storage of domain objects into a triple store in its underlying representation of RDF.

[6] http://www.sindice.com/

There are many triple stores solutions, all of them with their own advantages and disadvantages. To make the Catalogue independent of the solution adopted, it makes uses of 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 current implementation of the Catalogue prototype at M36 uses Sesame 2 as triple store. This solution has a wide community behind, and it has attractive advantages such as being completely extensible and configurable regarding to storage mechanisms, inferencers, RDF file formats, query result formats and query languages.

4 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 body of the requests to the API must be in JSON[7]. JSON is a lightweight data interchange format which simplicity has resulted in widespread use among web developers, being easy to read, write, be processed and parsed 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://catalogueURL/screens/clones/2338" },
    { "uri": "http://catalogueURL/screens/clones/1323" }
  ],
  "palette": [
    { "uri": "http://catalogueURL/screens/636" }
  ],
  "domainContext": {
    "tags": [
      {
        "label": { "en-GB": "Amazon" },
        "means": "http://dbpedia.org/page/Amazon.com"
      }
    ],
    "user": "http://ismaelrivera.es/#me",
  },
  "criterion": "reachability"
}
```

[7] 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[8] for languages and the ISO 3166[9] for countries. The following example illustrates how to use it properly in both formats.

```
{
  "label": {
    "en-GB": "Product Search",
    "es-ES": "Búsqueda de productos",
    "de-DE": "Produktsuche"
  },
  "description": {
    "en-GB": "On this screen, the user can enter a search term.",
    "es-ES": "En esta ventana, el usuario puede introducir el criterio de búsqueda",
    "de-DE": "Auf diesem Screen kann der Benutzer einen Suchbegriff eingeben."
  }
}
```

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:dc="http://purl.org/dc/terms/"
    xmlns:fgo="http://purl.oclc.org/fast/ontology/gadget#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:ctag="http://commontag.org/ns#">
  <rdf:Description rdf:about="http://catalogueURL/screens/1">
    <rdf:type rdf:resource="http://purl.oclc.org/fast/ontology/gadget#Screen"/>
    <rdfs:label xml:lang="en-gb">Product Search</rdfs:label>
    <rdfs:label xml:lang="en-es">Búsqueda de productos</rdfs:label>
    <rdfs:label xml:lang="en-de">Produktsuche</rdfs:label>
    <dc:description xml:lang="en-gb">On this screen, the user can enter a search
    term.</dc:description>
    <dc:description xml:lang="en-gb">En esta ventana, el usuario puede introducir
    el criterio de búsqueda.</dc:description>
    <dc:description xml:lang="en-gb">Auf diesem Screen kann der Benutzer einen
    Suchbegriff eingeben.</dc:description>
  </rdf:Description>
</rdf:RDF>
```

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

Internationalisation is being used by the attributes *label* and *description* of any building block, *label* of the pre-/post-conditions and *label* of the tags.

4.2 Linked Data

Apart from serving as the back-end system for the Gadget Visual Storyboard (GVS), the catalogue publishes all FAST building blocks as linked data, following the principles as originally defined in [Berners-Lee, 2006], in order to make them available to arbitrary third party applications. Each building block is identified by an HTTP URI and hosted through the catalogue so that it can be dereferenced through the same URI. For each building block, data is available in representations in different standard formats such as JSON (for communication with the GVS), RDF/XML, Turtle, or even HTML+RDFa as a human-readable version. Following the principles and best practices proposed in [Berrueta et al., 2008] and [Sauermann et al., 2008], these representations are served based on the request issued by the requesting agent, using a technique called *content negotiation*. As required by the forth rule of linked data, individual building blocks link to other data on the Web, thereby preventing so-called isolated "data islands".

In order to understand how content negotiation is implemented in the catalogue, a few concepts need to be understood. The URI of a particular building block must be understood as the identifier of the building block such, as opposed to a particular representation (JSON, HTML, etc.) of it. In fact, each such representation has its own URI, under which it can be retrieved. Using the terminology established in [Jacobs and Walsh, 2004], the building block is a *non-information resource*, whereas each representation is an *information resource*. In the process of content negotiation, the requesting agent will first request the URI of the building block as such (e.g., `http://catalogueURL/screens/235`), as well as the required representation format (e.g., JSON). The catalogue will then inspect the request and redirect the requester to the appropriate representation (e.g., `http://catalogueURL/screens/235.json`), as illustrated in Fig. 4.2. On the Web, different representations for the same resource are called *variants*, and content negotiation is the mechanism used to determine which of the representations is most appropriate for a given request.

In summary, the basic idea of content negotiation, as stated in the [Fielding et al., 1999], is to serve the best variant for a resource, taking into account what variants are available, what vari-

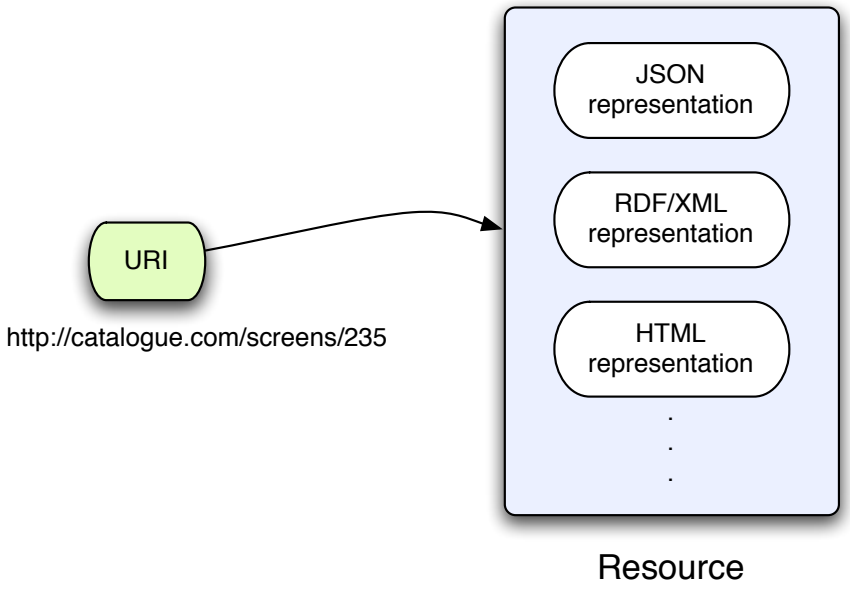

Figure 2: Content Negotiation

ants 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 representations 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, Turtle and HTML+RDFa. Even though the accept header is desired, the different representations can also be retrieved directly by dereferencing their own URI. The convention used in the catalogue for deriving a representation URI from a building block URI is to simply attach a matching suffix, such as `.rdf`, `.ttl`, `.json` or `.html` for RDF/XML, Turle, JSON or HTML respectively.

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

The Catalogue API is a programmatic interface into many of the Catalogue's features. It includes not only control over creation, modification and deletion of building blocks, but it has been extended to support advanced search and recommendation capabilities.

This section contains detailed descriptions of the operations permitted by the API, 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).

4.4 Managing building blocks

Any building block within the Catalogue is a resource in terms of REST philosophy, therefore it can be created, retrieved, modified or deleted, using a certaing URL and a specific HTTP method for every operation. Two concepts should be clarified: *collection*, as a set of resources are accessed at the URL `http://catalogueURL/[type]` where *catalogueURL* is where the Catalogue server is installed and `[type]` is the plural of the name of the building block (e.g. screens, operators), and *member* of the collection, in other words, the building block itself, being accessed at the URL `http://catalogueURL/[type]/[id]` where `[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 creating any building block can be found in the appendices. All of them are 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 requests are constructed and sent, and how the responses look like.

To create a new screen, a POST request is sent to the catalogue server, concretely to `http://catalogueURL/screens`, including the JSON representation of the screen in the body of the request:

```
{
  "code": "http://demo.fast.morfeo-project.org/code/amazonList.js",
```

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.

```
"creationDate": "2010-01-26T17:01:13+0000",
"creator": "http://ismaelrivera.es/#me",
"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"},
"postconditions": [
  {
    "id": "item",
    "label": { "en-gb": "An item" },
    "pattern": "?item
               http://www.w3.org/1999/02/22-rdf-syntax-ns#type
               http://fast.morfeo-project.org/ontologies/amazon#Item",
    "positive": "true"
  }
],
"preconditions": [
  {
    "id": "filter",
    "label": { "en-gb": "A search request" },
    "pattern": "?search
               http://www.w3.org/1999/02/22-rdf-syntax-ns#type
```

```
                http://fast.morfeo-project.org/ontologies/amazon#SearchRequest",
      "positive": "true"
    }
  ],
  "rights": "http://creativecommons.org/",
  "screenshot": "http://demo.fast.morfeo-project.org/images/amazonProductList.png",
  "tags": [{"label": {"en-gb": "amazon"}}],
  "version": "1.0"
}
```

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": "http://ismaelrivera.es/#me",
  "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"},
  "postconditions": [
      {
        "id": "item",
        "label": { "en-gb": "An item" },
        "pattern": "?item
                    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                    http://fast.morfeo-project.org/ontologies/amazon#Item",
        "positive": "true"
      }
    ]
  ],
  "preconditions": [
    [
      {
        "id": "filter",
        "label": { "en-gb": "A search request" },
        "pattern": "?search
                    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                    http://fast.morfeo-project.org/ontologies/amazon#SearchRequest",
        "positive": "true"
      }
    ]
  ],
  "rights": "http://creativecommons.org/",
  "screenshot": "http://demo.fast.morfeo-project.org/images/amazonProductList.png",
  "tags": [{"label": {"en-gb": "amazon"}}],
  "uri": "http://catalogueURL/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": "http://ismaelrivera.es/#me",
    "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"},
    "postconditions": [
      [
        {
          "id": "item",
          "label": { "en-gb": "An item" },
          "pattern": "?item
                     http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                     http://fast.morfeo-project.org/ontologies/amazon#Item",
          "positive": "true"
        }
      ]
    ],
    "preconditions": [
      [
        {
          "id": "filter",
          "label": { "en-gb": "A search request" },
          "pattern": "?search
                     http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                     http://fast.morfeo-project.org/ontologies/amazon#SearchRequest",
          "positive": "true"
        }
      ]
    ],
    "rights": "http://creativecommons.org/",
    "screenshot": "http://demo.fast.morfeo-project.org/images/amazonProductList.png",
    "tags": [{"label": {"en-gb": "amazon"}}],
    "uri": "http://catalogueURL/screens/28",
    "version": "0.1"
  },
  {
    "creationDate": "2011-02-07T09:59:52+0000",
    "creator": "fabio",
    ...,
    "uri": "http://catalogueURL/screens/14",
    "version": "1.0"
  },
  {
    "creationDate": "2011-02-07T09:59:52+0000",
    "creator": "javier",
    ...,
    "uri": "http://catalogueURL/screens/564",
```

```
    "version": "1.0"
  }
]
```

4.5 Managing concepts (classes) and attributes (properties)

This method allows the creation of new classes and properties which do not belong to any existing ontology or vocabulary. The URI of this new class has the Catalogue URL as prefix, in order to be dereferenceable, followed by a domain, which groups these classes, and a name (http://catalogueURL/concepts/[domain]/[name]).

The URL for this operation is `http://catalogueURL/concepts`.

The concepts are another type of resource, hence the common CRUD operations (GET to retrieve, POST to create, PUT to update and DELETE to remove) are supported for them.

Example of concept:

```
{
  "description": {"en-gb": "Represents a school with its coordinates."},
  "label": {"en-gb": "School"},
  "tags":[{
    "label": {"en-gb": "school"}
   }],
  "subclassOf": "http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing",
  "attributes": [
    {
      "uri": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
      "type": "http://www.w3.org/2001/XMLSchema#double"
    },
    {
      "uri": "http://www.w3.org/2003/01/geo/wgs84_pos#long",
      "type": "http://www.w3.org/2001/XMLSchema#double"
    }
  "name": "school",
  "domain": "institutions"
}
```

4.6 Managing samples/instances of concepts

It is allowed to create instances of classes, or so-called samples of concepts. The samples are another type of resource, hence the common CRUD operations (GET to retrieve, POST to create, PUT to update and DELETE to remove) are supported for them.

Example of sample:

```
{
  "concept": "http://catalogueURL/concepts/institutions/school",
  "attributes": [
    {
      "uri": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
      "value": "60.025"
    },
    {
      "uri": "http://www.w3.org/2003/01/geo/wgs84_pos#long",
      "value": "34.224"
    }
}
```

4.6.1 Prototype cloning

The resources created by the requests described in Section 4.4 are prototypes. To create a clone of a prototype is as simple as sending a HTTP POST request to `http://catalogueURL/cloning` indicating the URI of the building block to be cloned. It is not the aim of this deliverable to explain the logic behind the prototypes and clones. For a better understanding on this topic, [Möller et al., 2011] contains an extend section about prototype-based semantics for RDF.

Request example:

```
{
  "uri": "http://catalogueURL/forms/7"
}
```

Example of response:

```
{
  "clone": "http://catalogueURL/forms/clones/745"
  "uri": "http://catalogueURL/forms/7"
}
```

As clones are building blocks as well, the operations explained in Section 4.4 are valid to retrieve

and delete them, being `http://catalogueURL/[type]/clones/[id]` the URI to send the HTTP request.

4.6.2 Metadata

Retrieves all the metadata about a list of building blocks or classes (concepts) stored in the catalogue. This functionality is already provided by 4.4 if one specific building block, or the entire collection, is to be retrieved. However, this operation provides a way to get the metadata of an arbitrary number of resources, specified by their URIs, with no disctintion of their type.

The URL to access this operation is `http://catalogueURL/metadata`. The operation is invoked sending a HTTP POST request, Table 2 shows the parameters needed for the invocation and Table 3 details the different parameters of the response.

Table 2: Metadata Request Parameters

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

Table 3: Metadata Response Parameters

Name	Description
screen-flows	A list of screen-flows with all the metadata associated to them.
screens	A list of screens with all the metadata associated to them.
forms	A list of forms with all the metadata associated to them.
operators	A list of operators with all the metadata associated to them.
backendservices	A list of back-end services with all the metadata associated to them.
classes	A list of classes or concepts, with metadata such as label, properties, subclasses, etc.

Example request:

```
[
  "http://catalogueURL/screens/48",
  "http://catalogueURL/forms/clones/339",
```

```
  "http://fast.morfeo-project.org/ontologies/amazon#Item",
]
```

Example response:

```
{
  "screenflows": [],
  "screens": [
    {
      "creationDate": "2011-02-07T09:59:52+0000",
      "creator": "http://ismaelrivera.es/#me",
      ...,
      "uri": "http://catalogueURL/screens/48",
      "version": "1.0"
    }
  ],
  "forms": [
    {
      "actions": [
        {
          "name": "init",
          "preconditions": [],
          "uses": []
        },
        {
          "name": "showTable",
          "preconditions": [{
            "id": "list",
            "label": {"en-gb": "A product list"},
            "pattern": "?urlList
                       http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                       http://fast.morfeo-project.org/ontologies/amazon#ProductList",
            "positive": true
          }],
          "uses": []
        }
      ],
      ...,
      "uri": "http://catalogueURL/FASTCatalogue/forms/clones/249",
      "version": "1.0"
    },
  ],
  "operators": [],
  "backendservices": []
  "classes": [
    {
      "attributes": [
        {
          "type": "http://xmlns.com/foaf/0.1/Document",
          "uri": "http://fast.morfeo-project.org/ontologies/amazon#smallImage"
        },
        {
          "type": "http://xmlns.com/foaf/0.1/Document",
          "uri": "http://fast.morfeo-project.org/ontologies/amazon#mediumImage"
```

```
        },
        {
          "type": "http://xmlns.com/foaf/0.1/Document",
          "uri": "http://fast.morfeo-project.org/ontologies/amazon#largeImage"
        },
        {
          "type": "http://www.w3.org/2001/XMLSchema#double",
          "uri": "http://fast.morfeo-project.org/ontologies/amazon#price"
        }
      ],
      "label": {"en": "An item handled by the Amazon Service"},
      "tags": [{
        "label": {"en": "amazon"},
        "means": "http://www.amazon.com"
      }],
      "uri": "http://fast.morfeo-project.org/ontologies/amazon#Item"
    }
  ]
}
```

4.6.3 Screen Find & Check

Searches for new screens and checks the satisfaction/reachability of screens in a given screenflow. The response will contain all the elements pass on passed in the request, plus information regarding the conditions' satisfaction, and therefore, whether the screens are reachable. Moreover, if screens are not reachable (their pre-conditions are not satisfied), it searchs for new screens which would make these screens reachable.

The URL for this operation is `http://catalogueURL/screens/findcheck`.

Example of the request:

```
{
  "canvas": [
    { "uri": "http://catalogueURL/screens/clones/338" }
  ],
  "palette": [
    { "uri": "http://catalogueURL/screens/636" }
  ],
  "domainContext": {
    "tags": [
      {
        "label": { "en-GB": "Amazon" },
        "means": "http://dbpedia.org/page/Amazon.com"
      }
    ],
    "user": "http://ismaelrivera.es/#me",
```

Table 4: Screen Find & Check Request Parameters

Name	Description	Type
canvas	The canvas is divided in three lists: **screens** List of the screens of the screen-flow. Only the URI is needed. **preconditions** List of the pre-conditions of the screen-flow. **postconditions** List of the post-conditions of the screen-flow.	Required
palette	It is a list of screens (prototypes), which are not in the canvas but need to be checked based on the specified criterion.	Optional
domainContext	The domain context contains a list of tags and the user who makes the request.	Optional
criterion	The criterion specifies what has to be checked. At the moment, only 'reachability' (checks conditions satisfaction and screen reachability) is supported, but this parameter may support other values.	Required

Table 5: Screen Find & Check Response Parameters

Name	Description
canvas	List of screens in the canvas, with extra information indicating satisfaction of the criterion, for the screens themselves and their pre-conditions.
palette	A list of screens obtained by the search, attaching information about the satisfaction of the criterion.
domainContext	It contains the same value as in the request.
limit	Integer. It is used to limit the number of results returned. Default value is -1 (no limit).
offset	Integer. It is used to jump or skip a number of results to be returned. Default value is 0.

```
  },
  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://catalogueURL/screens/clones/338"
  }],
  "palette": [
    {
      "preconditions": [{
        "label": {"en-gb": "A person"},
        "pattern": "?person rdf:type foaf:Person",
        "satisfied": false
      }],
      "reachability": false,
      "uri": "http://catalogueURL/screens/636"
    },
    {
      "preconditions": [],
      "reachability": true,
      "uri": "http://catalogueURL/screens/132"
    }
  ]
}
```


A more detailed explanation regarding the search, recommendation, and ranking algorithms used by this operation can be found in Appendix F.

4.6.4 Screen Component Find & Check

Similarly to the Screen Find & Check described in Section 4.6.3, this operation searches for screen components (forms, operators and back-end services) through the catalogue in order to satisfy a given request, and attaches information regarding building block satisfaction and reachability as well.

The URL for this operation is `http://catalogueURL/components/findcheck`.

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 clones of forms, operators and back-end services. Only the URI is needed.

forms It is a list of forms (prototypes), not in the canvas, but related to the request (may be the list of forms retrieved in previous searches).

operators It is a list of operators (prototypes), not in the canvas, but related to the request (may be the list of operators retrieved in previous searches).

backendservices It is a list of back-end services (prototypes), not in the canvas, but related to the request (may be the list of back-end services retrieved in previous searches).

domainContext The domain context contains a list of tags and the user who makes the request.

pipes List of pipes for the connection between screen components and conditions. If the building block in a pipe is the screen itself, the value must be set to *null*.

selectedItem URI of a screen component from the canvas, or id of a pre-/post-condition.

actions List of actions to be executed in the operation. Three actions can be specified: *find*, *check* and *iserve*. The action *find* indicates that it must search for forms/operators/services. The action *check* checks whether the components/conditions/pipes are satisfied/reachable. *iserve* is set to look for web services descriptions in the iServe platform.

limit Integer. It is used to limit the number of results returned. Default value is -1 (no limit).

offset Integer. It is used to jump or skip a number of results to be returned. Default value is 0.

Example request:

```
{
  "preconditions": [{
    "id": "Searchcriteria_1",
    "label": {"en-gb": "A search request"},
    "pattern": "?search
               http://www.w3.org/1999/02/22-rdf-syntax-ns#type
               http://fast.morfeo-project.org/ontologies/amazon#SearchRequest",
    "positive": true,
  }],
  "postconditions": [],
  "canvas": [
    { "uri": "http://catalogueURL/forms/clones/112" },
    { "uri": "http://catalogueURL/services/clones/312" }
```

```
  ],
  "forms": [],
  "operators": [],
  "backendservices": [],
  "domainContext": {
    "tags": [],
    "user": "http://ismaelrivera.es#me"
  },
  "pipes": [
    {
      "from": {
        "buildingblock": "http://catalogueURL/services/clones/312",
        "condition": "list"
      },
      "to": {
        "action": "showTable",
        "buildingblock": "http://catalogueURL/forms/clones/112",
        "condition": "list"
      }
    }
  ],
  "selectedItem": "http://catalogueURL/forms/clones/112",
  "actions": ["find", "check", "iserve"]
}
```


Example response:


```
{
  "backendservices": [],
  "canvas": [
    {
      "actions": [{
        "name": "search",
        "preconditions": [{
          "id": "filter",
          "label": {"en-gb": "A search request"},
          "pattern": "?search
                      http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                      http://fast.morfeo-project.org/ontologies/amazon#SearchRequest",
          "positive": true,
          "satisfied": false
        }],
        "satisfied": false,
        "uses": []
      }],
      "reachability": false,
      "uri": "http://catalogueURL/services/clones/312"
    },
    {
      "actions": [
        {
          "name": "init",
          "preconditions": [],
          "satisfied": true,
```

```
        "uses": []
      },
      {
        "name": "showTable",
        "preconditions": [{
          "id": "list",
          "label": {"en-gb": "A product list"},
          "pattern": "?urlList
                      http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                      http://fast.morfeo-project.org/ontologies/amazon#ProductList",
          "positive": true,
          "satisfied": false
        }],
        "satisfied": false,
        "uses": []
      }
    ],
    "reachability": true,
    "uri": "http://catalogueURL/forms/clones/112"
  }
],
"connections": [
  {
    "from": {
      "buildingblock": null,
      "condition": "Searchcriteria_1"
    },
    "to": {
      "action": "search",
      "buildingblock": "http://catalogueURL/services/clones/312",
      "condition": "filter"
    }
  },
],
"forms": [],
"operators": [],
"pipes": [
  {
    "from": {
      "buildingblock": "http://catalogueURL/services/clones/312",
      "condition": "list"
    },
    "satisfied": true,
    "to": {
      "action": "showTable",
      "buildingblock": "http://catalogueURL/forms/clones/112",
      "condition": "list"
    }
  }
],
"postconditions": [],
"iserve": [
  {
    "address": "http://ws.audioscrobbler.com/2.0/?method=user.getfriends
```

```
                &user={http://xmlns.com/foaf/0.1/Person}&api_key={APIKey}",
      "classes": ["http://xmlns.com/foaf/0.1/Person"],
      "doc": "http://iserve.kmi.open.ac.uk/resource/documents/[id]/hrest.html",
      "label": "LastFm Friends",
      "service": "http://iserve.kmi.open.ac.uk/resource/services/[id]#LastFmFriends"
    },
    ...
  ]
}
```

For a screen component to be reachable, its pre-conditions need to be connected by a pipe to any other screen component which is already reachable, or to any of the screen pre-conditions. 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.

A more detailed explanation regarding the search, recommendation, and ranking algorithms used by this operation can be found in Appendix F.

4.7 Planning

Search for plans (sets of screens) which accomplish a goal. A goal is a screen or set of screens which must be made reachable.

The URL to access this operation is `http://catalogueURL/planner`. This operation is invoked sending an HTTP POST request with the following parameters:

goal Required. URI or list of URIs (clones) conforming the goal.

canvas Required. List of screens URIs (clones).

page Optional. Number of the page to be retrieved.

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

Example request (single goal):

```
{
  "goal": "http://catalogueURL/screens/clones/135",
  "canvas": [
    { "uri": "http://catalogueURL/screens/clones/135" },
```

```
    { "uri": "http://catalogueURL/screens/clones/656" },
    { "uri": "http://catalogueURL/screens/clones/385" }
  ],
  "page": 1,
  "per_page": 10
}
```

Example request (multi goal):

```
{
  "goal": [
    "http://catalogueURL/screens/clones/135",
    "http://catalogueURL/screens/clones/537"
  ]
  "canvas": [
    { "uri": "http://catalogueURL/screens/clones/135" },
    { "uri": "http://catalogueURL/screens/clones/537" },
    { "uri": "http://catalogueURL/screens/clones/656" },
    { "uri": "http://catalogueURL/screens/clones/385" }
  ],
  "page": 1,
  "per_page": 10
}
```

Example response:

```
[
  [
    "http://catalogueURL/screens/clones/113",
    "http://catalogueURL/screens/clones/423"
  ],
  [
    "http://catalogueURL/screens/clones/669",
    "http://catalogueURL/screens/clones/37",
    "http://catalogueURL/screens/clones/114"
  ],
  [
    "http://catalogueURL/screens/clones/711",
    "http://catalogueURL/screens/clones/464",
    "http://catalogueURL/screens/clones/322"
  ]
]
```

A more detailed explanation regarding the algorithms used for the planning can be found in Appendix F.

4.8 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 possible error codes are listed in Table 4.8.

Table 6: 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
InvalidHttpRequest	The HTTP request is invalid. Reason: <reason>.	400 Bad Request
InvalidURI	The URI <requestURI> is not valid.	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

[Berners-Lee, 2006] Berners-Lee, T. (2006). Linked data. http://www.w3.org/DesignIssues/LinkedData.html.

[Berrueta et al., 2008] Berrueta, D., Phipps, J., and Swick, R. (2008). Best practice recipes for publishing rdf vocabularies. Technical report, W3C.

[B.V., 2008] B.V., A. (2008). User guide for sesame 2.2. http://www.openrdf.org/doc/sesame2/2.3-pr1/users/index.html.

[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).

[Han et al., 2000] Han, J., Pei, J., and Yin, Y. (2000). Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29:1–12.

[Hayes, 2006] Hayes, P. (2006). Rdf semantics. http://www.w3.org/TR/rdf-mt/#RDFSRules.

[Jacobs and Walsh, 2004] Jacobs, I. and Walsh, N. (2004). Architecture of the World Wide Web, Volume One. W3C Recommendation. Recommendation, W3C.

[Li et al., 2008] Li, H., Wang, Y., Zhang, D., Zhang, M., and Chang, E. Y. (2008). Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems*, RecSys '08, pages 107–114, New York, NY, USA. ACM.

[Möller et al., 2011] Möller, K., Rivera, I., and Palaghita, C. A. (2011). Ontology and conceptual model for the semantic characterisation of complex gadgets. Deliverable 2.2.3, FAST Project (FP7-ICT-2007-1-216048).

[Pedrinaci et al., 2010] Pedrinaci, C., Liu, D., Maleshkova, M., Lambert, D., Kopecky, J., and Domingue, J. (2010). iserve: a linked services publishing platform. In *Ontology Repositories and Editors for the Semantic Web Workshop at The 7th Extended Semantic Web*, volume 596.

[Reyes et al., 2011] Reyes, M., Garcia, J. F., Lopez, J., and Fuchsloch, A. (2011). FAST complex gadget architecture. Deliverable D3.1.3, FAST Project (FP7-ICT-2007-1-216048).

[Sauermann et al., 2008] Sauermann, L., Cyganiak, R., Ayers, D., and Völkel, M. (2008). Cool URIs for the Semantic Web. Interest group note, W3C. http://www.w3.org/TR/cooluris/ 05/05/2009.

[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 concluded that SWS platforms are 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 4.8 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

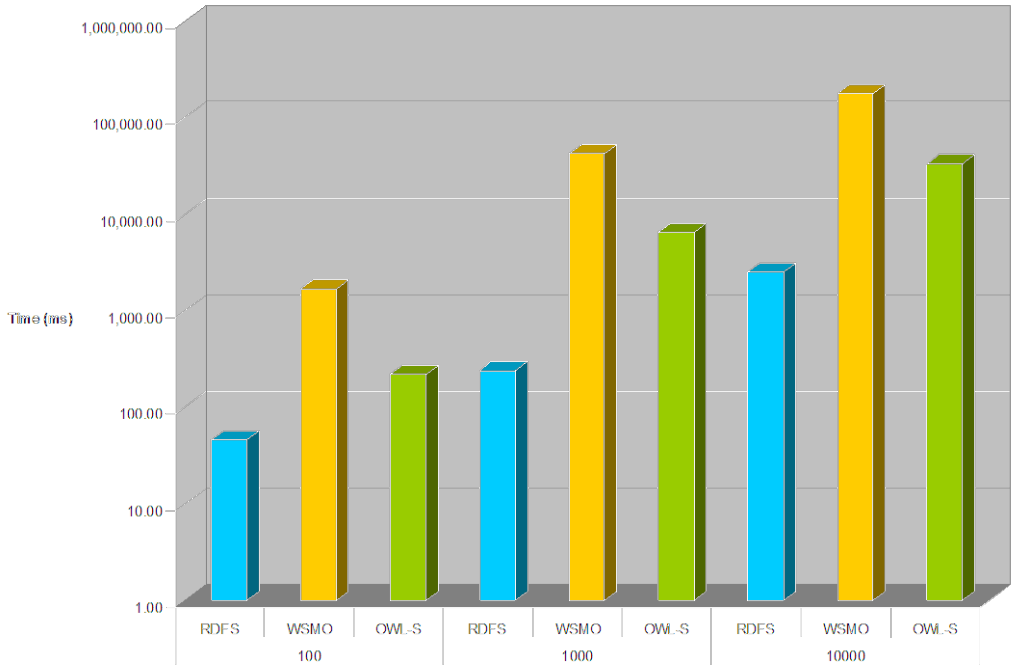

Figure 3: Semantic Web Technologies Performance Comparison

100 size set. Hence, based on this evaluation the approach to follow for the implementation was 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": "2011-04-20T17:00:00+0100",
  "creator": "username",
  "description": {"en-gb": "This is a description of the building block"},
    "tags":[
      {
        "label":{"en-gb":"Amazon"},
        "means":"http://dbpedia.org/page/Amazon.com"
      },
      {
        "label":{"en-gb":"eBay"},
        "means":"http://dbpedia.org/page/Ebay"
      }
    ],
  "homepage": "http://url.com/.../homepage",
  "icon": "http://url.com/images/.../icon.png",
  "id": "Id of the building block", // i.e. 3545
  "label": {"en-gb": "Label or title of the building block"},
  "rights": "http://creativecommons.org/",
  "screenshot": "http://url.com/.../screenshot.jpg",
  "uri": "http://url.com/screens/525",
  "version": "1.0"
}
```

Appendix C. Screen-flow JSON Structure

```
{
  <GENERIC BUILDING BLOCK JSON>,
  "contains": [
    "http://url.com/screens/clones/72723",
    "http://url.com/screens/clones/20035",
  ],
}
```

Appendix D. Screen JSON structure

```
{
  <GENERIC BUILDING BLOCK JSON>,
  "postconditions": [[{
    "id": "Id of the building block", // i.e. 3545
    "label": {"en-gb": "Purchase URL"},
    "pattern": "?url
                http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                http://fast.morfeo-project.org/ontologies/amazon#PurchaseURL",
    "positive": true
  }]],
  "preconditions": [[{
    "id": "cart",
    "label": {"en-gb": "A shopping cart"},
    "pattern": "?cart
                http://www.w3.org/1999/02/22-rdf-syntax-ns#type
                http://fast.morfeo-project.org/ontologies/amazon#ShoppingCart",
    "positive": true
  }]],
  "code": "URL of the code",
  "definition": {
    "buildingblocks: [
      {
        "uri": "http://catalogueURL/forms/clones/670238"
        "uri": "http://catalogueURL/operators/clones/213487"
        "uri": "http://catalogueURL/services/clones/38399"
      },
    ]
    "pipes": [
      {
        "from": {
          "buildingblock": "http://catalogueURL/services/clones/38399",
          "condition": "cA"
        },
        "to": {
          "buildingblock": "http://catalogueURL/operators/clones/213487",
          "action": "filter"
          "condition": "cA",
        }
      },
    "triggers": [
      {
        "from": {
          "buildingblock": "http://catalogueURL/forms/clones/670238",
          "name": "refresh"
        },
        "to": {
          "buildingblock": "http://catalogueURL/services/clones/38399",
          "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://fast.morfeo-project.org/ontologies/amazon#Item",
      "positive": true
    }]],
    "uses": [{
      "id":"cart",
      "uri":"http://fast.morfeo-project.org/ontologies/amazon#ShoppingCart"
    }]
  }],
  "libraries": [{
    "language":"JavaScript",
    "source":"http://url.com/libcode.js"
  }],
  "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"]
}
```

Appendix F. Algorithms for building block recommendation

The Catalogue serves as a metadata repository for the different kind of resources used by the Gadgets in the FAST platform, or so-called building blocks. The number of building blocks stored in the catalogue is arbitrary, and a simple browsing or keyword-based search through the whole catalogue would be not usable if the number is relatively high.

There are two scenarios where the user may be assisted: screen-flow design, and screen design. At screen-flow design, a user needs to find the proper screens to fulfil her goal. At screen design, the scenario is a bit more complex, where different types of building blocks are involved, making screen creation process more challenging. Therefore, the Catalogue includes several mechanisms in order to help in these situations.

F.1 Pre-/post-conditions matching search

The building blocks are connected (implicitly or explicitaly) to each other via their pre- and post-conditions. These connections are guaranteed by a matching method which determine when a pre-condition is satisfied by a post-condition.

At screen-flow or screen design level, the user needs to find new building blocks which may satisfy elements from the canvas. There is a high probability that building blocks which have compatible pre-/post-conditions might be used to accomplish her goal. The goal for the algorithm is to satisfy a set of pre-conditions, which are still not satisfied, of the elements in the canvas. Therefore, this algorithm searches the Catalogue for building blocks which have post-conditions that satisfy, in a certain way, some or all of these pre-conditions.

The matching algorithm used to determine the compatibility of two conditions is a SPARQL query which compares whether two condition patterns are compatible. How the pre- and post-conditions are defined is explained in [Möller et al., 2011], but as an example:

```
?user a sioc:User ;
      foaf:accountName ?account_name .
```

It describes a hypothetical condition requiring the existence of a `sioc:User` which must have a certain `foaf:accountName`.

Defining the condition patterns in this manner allows the usage of inference or reasoning mechanisms, which would relax the restrictions to connect building blocks between each other. The Catalogue supports RDFS Entailment Rules, as defined in [Hayes, 2006].

As an example, enabling the inferencing capabilities would allow building blocks pre-conditions to be satisfied by a more specialised condition, in terms of classes and subclasses. In the following scenario:

```
sioc:User a rdfs:Class .
ex:Professor rdfs:subClassOf sioc:User .
```

A screen A which pre-condition pattern is `?user a sioc:User` would be satisfied by a screen B which post-condition pattern is `?professor a ex:Professor`, as the pre-condition requires the existence of a more generic class than the one defined in the post-condition.

F.2 Planning

One of the limitations of the pre-/post-conditions matching algorithm is that its output is limited to a set of building blocks which satisfy a given set of conditions, but they may include new pre-conditions which are not yet satisfied. The consequence is that the user has to add building blocks iteratively until her goal is accomplished (screen-flow is executable).

The approach followed by this algorithm is to find sets of screens, or so-called *plans*, which fully accomplish a certain goal in a one-step action.

From the point of view of this algorithm, the screen knowledge base is seen as a graph, as represented in Figure 4.8, where the vertices (or nodes) are the screens, and the edges represent whether a screen satisfies or is satisfied by another screen (the arrow points towards the screen which satisfies). There is a special node connected to all the screens with no pre-conditions (those are reachable by default).

Therefore, the aim of the algorithm is to find the paths or plans from a given screen to the special node or root. This is a very well-known problem in graph theory, and therefore there are many graph and tree search algorithms dealing with it, such as Floyd's, Dijkstra, A* or depth-first search, among others.

Floyd's, Dijkstra and A* are focused on finding shortest paths for a given problem. In the case

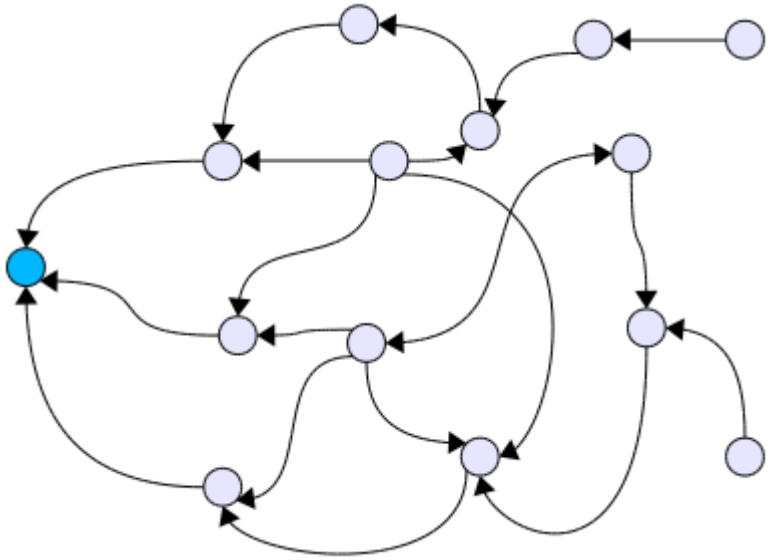

Figure 4: Graph of compatible screens by their pre-/post-conditions

of A*, the advantage is that a heuristic function may be define to prioritise a set of screens (i.e. screens in the current canvas). However, finding the shortest path is not the intention of this algorithm. In fact, it is preferred to find a small set of possible solutions, thus depth-first search is a good fit as it finds all the paths from a given node to another, avoiding cycles. Then, the paths returned by depth-first are then ranked in order to minimise the path lenght, and to maximise the number of co-occurrences of elements in the path and the canvas.

The depth-first search algorithm has not been re-implemented for the catalogue. Instead, it relies on Neo4j[10], a flexible graph database, which apart from providing the aforesaid algorithm, it provides the necessary mechanism to store, access and query the graph. The advantages of using Neo4j are that it is highly optimized for storing graph structures for maximum performance and scalability, and supports several search algorithms, including as well Dijkstra and A*, which provides a great flexibility to the catalogue.

[10] http://neo4j.org/

F.3 Association Rule Mining

Association Rule Mining is a well known method for the discovery of relations between variables in large databases. Given a set of transactions, find rules that will predict the occurrence of an item based on the occurrences of other items in the transaction. For example, the rule `{onions, potatoes} = {burger}` found in the sales data of a supermarket would indicate that if a customer buys onions and potatoes together, he or she is likely to also buy burger.

This method is divided in two steps: frequent itemset mining and rule generation. There are many algorithms for frequent itemset mining as Apriori, Eclat and FP-Growth. This is still a computationally expensive task, and when a mined dataset has a considerable size it is resource intensive [Han et al., 2000]. However, FP-Growth has demostrated to execute faster than the rest, which lead the open source project called Mahout[11] to create its parallel implementation of the algorithm [Li et al., 2008].

We are not reinventing the wheel doing our own implementation of any algorithm for frequent pattern mining. We opted for integrating the Catalogue with Mahout to make uses of its power, and its scalability advantages.

This algorithm is used both in screen-flow design and screen-design. The approach is similar in both cases, finding building blocks which co-ocurrence is high.

As an example, given the following set of screen-flows:

Screen-flow	Screens
screenflow-A	screen-1, screen-2
screenflow-B	screen-1, screen-3, screen-4
screenflow-C	screen-1, screen-4
screenflow-D	screen-4, screen-5, screen-7, screen-8
screenflow-E	screen-3, screen-4, screen-9
screenflow-F	screen-5, screen-7, screen-2
screenflow-G	screen-10, screen-11, screen-5, screen-7
screenflow-H	screen-1, screen-2, screen-3

The algorithm will search for the screens with the highest co-ocurrence for a given input, i.e. a

[11] http://mahout.apache.org/

screen-flow during its design phase. For example, for a screen-flow which contains the screen-5, the algorithm will recommend screen-7 as its co-ocurrence is high.

However, the algorithm used for the operations Find & Check at screen-flow and screen design, is a combination of the pre-/post-conditions matching and frequent itemset mining algorithms. The output items of the frequent itemset mining algorithm are ranked depending on their co-ocurrence, adding the output of the pre-/post-conditions matching search algorithm as if their co-ocurrence was one.

F.4 iServe platform integration

iServe is a platform for publishing Semantic Web Services as linked data [Pedrinaci et al., 2010]. In a nutshell, it crawls the Web for web services descriptions and imports them into the platform, using the Minimal Service Model (MSM), which provides a common vocabulary for service annotations of different formalisms such as OWL-S, SAWSDL and WSMO. To facilitate the consumption and manipulation of the data, iServe provides three interfaces: a Web-based application (iServe Browser), a RESTful API and a SPARQL endpoint.

In FAST, the definition of the building blocks, and therefore of the web services have many similarities to the MSM. Without going into much detail, both have operations (or actions) which have inputs and outputs. These inputs and outputs are linked to classes in an ontology. In FAST, an input/output is defined by a set of condition or triple patterns, which says the class of the condition and adds some restrictions to it (values for certain properties, etc.). iServe references the classes or types of the inputs and outputs using the property *modelReference* from the SAWSDL vocabulary.

The Catalogue queries the iServe platform in the Find & Check for screen components operation. The goal of this operation is to find forms, operators or web services wrappers which may fit into a given screen (canvas). The integration is done sending SPARQL queries to its SPARQL endpoint[12], looking for web services which operations, inputs, or outputs reference (sawsdl:modelReference) the classes used by a set of conditions (e.g. unsatisfied conditions in the canvas).

[12] http://iserve.kmi.open.ac.uk/data/execute-query