%Versus: A Framework for Content-Based File Comparison %Luigi Marini; Smruti Padhy

Versus: A Framework for Content-Based File Comparison

Versus is a framework for content-based file comparison. The primary goals of the Versus framework are to support comparison of digital objects and comparisons of file containers with multiple types of digital objects contained in them. The framework tries to accomplish these goals by providing facilities for bringing together metrics and algorithms for content-based comparison of files of any format and by providing prototypes of scalable and extensible environments to execute such comparisons.

The framework provides several ways in which it supports comparison of files: an application programming interfaces (API), basic implementations of such APIs, a set of web services to support remote submission and execution of comparisons, and a variety of clients for the end-user such as a command-line interface (CLI), a simple desktop GUI based application and a full fledged web application. It also provides support for indexing service for content-based retrieval and descision support service.

This document is organized in three major parts. Core Package And API (Part I) describes the core package of the Versus framework, with an overview of the building blocks present in the application programming interface as well as a few facilities to make working with the metrics a little easier such as engines and registries. Part II describes versus service REST API and how to interact with versus service using various clients/interfaces. We name it as User's Manual. Part III describes the details to add new implementations for the Java APIs to the versus framework. We call it as Developer's manual.

This document is written with primarily four types of audiences in mind:

- an user that uses the versus service web interface or versus-service REST endpoints. A user can be a researcher or archivist who just wants to use the versus-service.
- an administrator that deploys the versus-service and configure the versus.properties file
- a developer that implements the Java API of versus-core and add new methods
- an advanced developer that modifies the framework design and creates new Java APIs and REST APIs

Part I and Part II is for all types of audiences while Part III specifically written for developers and advanced developers.

Core Package And API

The Versus core package includes the Java APIs, a multithreaded engines, a registry and a command line interface. We will also show an example of versus desktop GUI application build on versus-core package.

Application Programming Interface

The Application Programming Interface (API) is the set of specifications and rules that method provides can follow to integrate within the overall framework. Versus' API is written in Java due to the popularity of the language, the portable abstraction layer over the operating system provided by the Java Virtual Machine (JVM) and the good performance of the JVM.

The Versus API provides simple interfaces that a developer can write against to add new methods to the framework. By writing against such interfaces, researchers can contribute new methods to the Versus ecosystem. This promotes reuse of new and existing methods. Also if the Versus APIs need to be changed to incorporate new design, the developers responsible to do so provides new APIs and will be part of the versus-core.

There are four main components that make up the Versus Java API. They can be viewed together as one linear workflow where each component feeds into the next, starting at the raw files and ending with a similarity value. We describe each one in turn.

Adapters

Adapters encapsulate the code used to load the raw information from the input files. The information could be the raw bytes from any file, pixel values from images, text from documents, sound waves from audio files. Adapters take care of understanding the different underlying formats, abstracting such formats and providing the extractors with file type agnostic versions of the content of the file.

Feature Extractors

Extractors provide code needed to extract specific feature representations from the elementary content of the file. The raw information stored in a file might be too large or redundant. Feature extraction (some times called feature selection) is a form of dimensionality reduction to help alleviate the effect of curse of dimensionality by shrinking the original content to a more manageable set relevant to specific comparison methods.

Feature Descriptors

Feature descriptors encapsulate ways of representing the original digital content in more manageable ways to avoid the curse of dimensionality typical of digital files. Each feature representation holds only the attributes relevant for a particular comparison method.

Comparison Measures

Comparison measures encapsulate the algorithms in charge of establishing how similar or dissimilar two files are. Given two feature representations of two digital. objects, a comparison measure returns a proximity value representing how similar or dissimilar two digital objects are.

Proximity

Proximity represents how similar or dissimilar two digital objects are. These can be one or more numbers along with descriptions of what the minimum/maximums are and what their meaning is.

Putting it all together, the overall flow of a content-based comparison in Versus looks like Figure 1.

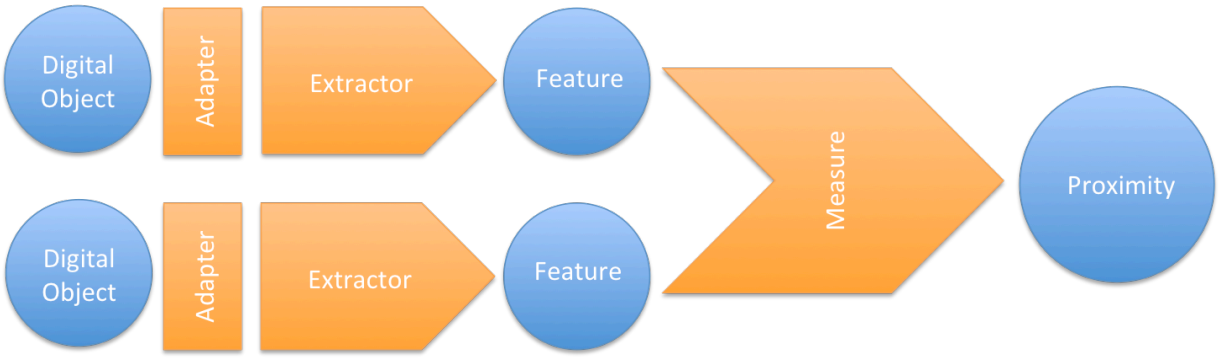


Figure 1 API abstraction: given two files, extract a feature descriptor per file and then compute the proximity between descriptors using a specific measure.

Indexing and Search Result

Besides content-based comparison in Versus, Versus framework supports content-based retrieval. It indexes descriptors obtained from the extractor applied to the digital object (loaded by an adapter). When query object is sent to Versus, it again obtained the descriptors from the query object and query against the index and returns all matches.

The overall flow of a content-based retrieval in Versus looks is shown in Figure 2.

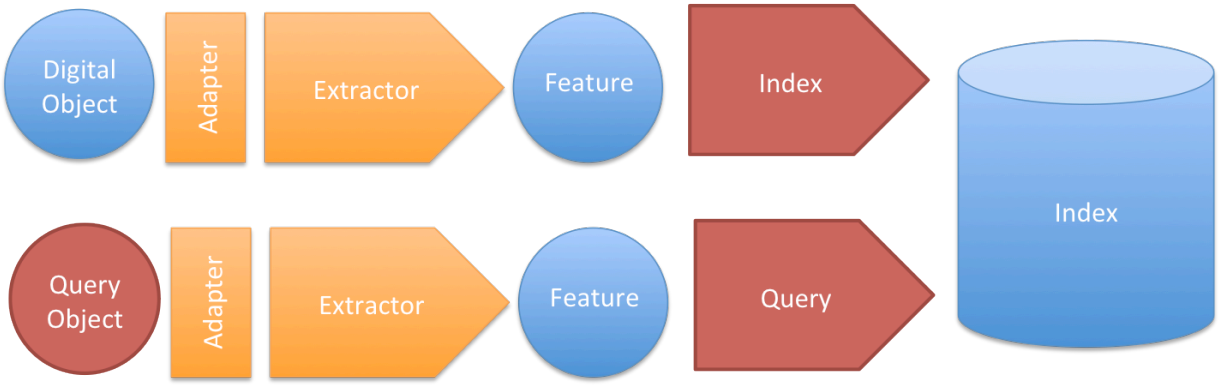


Figure 2 Content-based Retrieval in Versus Framework

Engines

An engine interface is included in the coke package. It supports different engine implementations, each performing different computation with different components of versus framework.

A multithreaded **execution engine** is included in the core package. The execution engine is for developers who want to develop clients for the Versus framework to use comparison resources. This same engine is used in all the clients currently developed as part of the framework, but it could also be used in new clients.

New submissions to the engine are in the form of Jobs . A job is a set of pairwise comparisons and a unique ID. A comparison is a tuple of (Adapter, Extractor, Measure) and two input files. The engine

creates a thread for each comparison and submits to a thread pool that is bound to a predefined number of concurrent threads at one time. The developer can submit a status handler along with the job to respond to specific events in meaningful ways based on the application being developed. A job can be in one of the following states: started, done, failed and aborted.

There are three more implementations of engine interface: Indexing Engine, Extraction Engine and Comprehensive engine.

Comprehensive Engine extends the execution engine. It computes the similarity value for two files for different combination of adapter, extractor and measure pair and finally do a linear combination of these results with some weights to each result.

Indexing Engine is also a multithreaded engine that is being used for indexing and retrieval. When a file is submitted to Versus for indexing, indexing engine extract the content descriptors or signatures from file and index the file based on the descriptors. When an query file is submitted to index resource, it uses indexing engine to query and return the matched search results.

Extraction Engine computes the content descriptors of a file for a given adapter and extractor pair. Each such extraction job is performed by an individual separate thread.

Registry

The registry can be queried for a list of adapter, extractors and measures available on the Java classpath. This creates a simple plugin mechanism that allows methods' developers to register individual implementations of adapters, extractors and measures with the Versus framework.

The registry uses Java service providers to register all possible implementations on the classpath so that if a new Java archive (jar) that includes implementations of Versus method is added to the classpath the registry will find them.

Command Line Interface

A simple command line interface (CLI) to execute comparisons is included in the core package. The headless CLI is much simpler to adopt for certain use cases, for example, when the goal is to script local execution or run comparison in windowless terminals. The interface requires the specification of the location of the two files being compared and the adapter/extractor/measure triple to use for the comparison as seen in Figure 3.

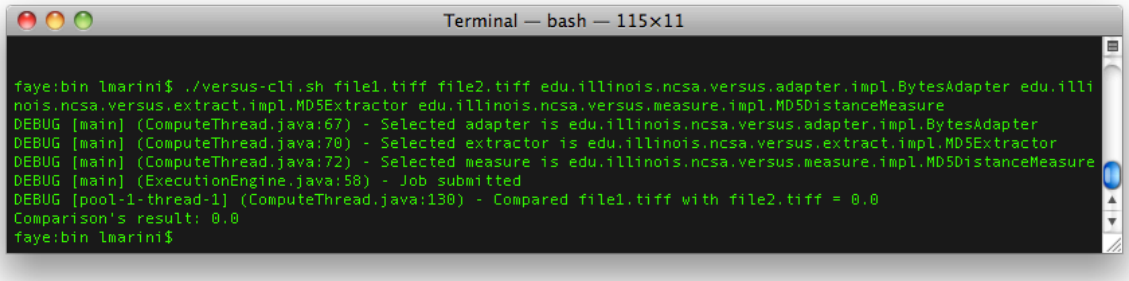


Figure 3. Command Line Interface Example. In this example we executed an MD5 comparison of two TIFF images. The result was 0, meaning the two files are different.

Simple Versus GUI

The simple Versus GUI available in the versus-desktop package provide a simple desktop application for the end user to launch local comparisons. The application was developed to make testing of the API and implemented comparison measure and extractors easier. It is also a very simple example of how all the different components of the core packages can be used to write an application.

The application allows the user to select a directory on disk and load its contents. The user can then select a data representation, feature extractor and similarity measure to launch pair wise comparisons on all the files selected. Pairwise comparison results are presented down below in a table format.

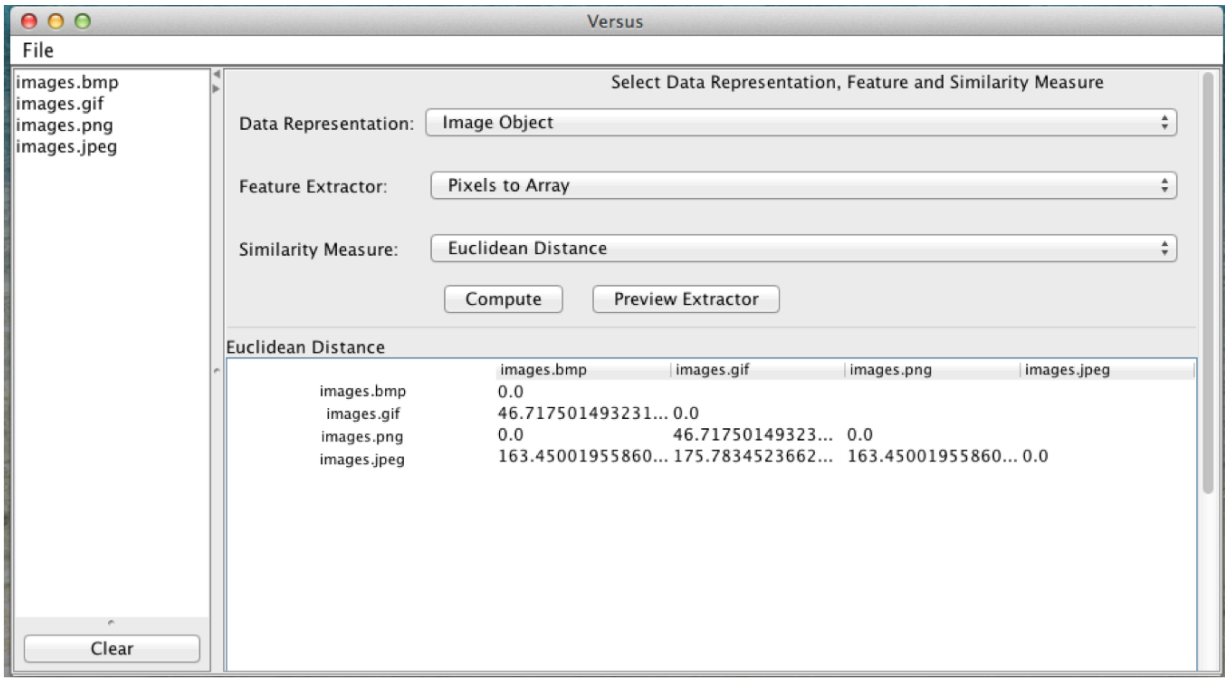


Figure 4. The list on the left shows all the files loaded. On the right pull down menus allow the selection of a data representation (adapter), feature extractor and measure to be used in the next run. The table at the bottom shows the results of the pairwise comparisons on all possible combinations of files.

User's Manual

This section of the manual is relevant to all types of users. This explains how to use Versus web service and web application.

Getting the Binary and Versus Service Deployment

- Download latest build from http://isda.ncsa.illinois.edu/versus/latest/versus-service-0.6.0-SNAPSHOT-bin.zip
- unzip it
- cd versus-service/bin
- Configure differnt parameters in `versus.properties` in bin/src/main/config as explained below based on your requirements:
    - **master**: specify the URL of the master. If this is not set or commented out, by default, the current instance of versus web server becomes the master.
    - **repository** : Currently there are three implementations for storing comparisons in the repository, i.e, memory, MySQL and mongoDB. By default, the value is set to `mem`. The other options are `myql` and `mongo`.
    - **extract**: The field is used to specify extraction data storage implementation. By default, it is set to `mem`.
    - **files**: This specifies the storage type for the uploaded file. By default, files are stored in `disk`. The other implementation is `mongo`.
    - **index**: By default, the index is stored in disk and it is set to `diskIndex`. The other option is `jdbcIndex` in which case it is stored in MySQl.
    - **maps**: By default, it is set to `disk` indicating that the map objects that store versus id corresponding to their external urls for each file in the index are stored in the disk. The other option is `mqsql` that let Versus stores in MySQL.
    - **numThreads**: By default, it is set to 2. It denotes the number of threads for the ExecutionEngine and ComphrensiveEngine
    - **OS**: This denotes the operating system in which you would be running versus-service. The options are `windows` and `Linux`
    - To use the default on-disk implementations, Set paths to data directory where you want to store the data. For example:

      ```
      file.folder=/Users/smruti/Data/versus/index1.txt
      file.indexerfolder=/Users/smruti/Data/versus/indexer
      file.map=/Users/smruti/Data/versus/map1.txt
      file.directory=/Users/smruti/Data/versus/upload
      ```

- Go into the bin folder and execute the appropriate startup script. There are two optional parameters that can be specified:

1. the port number (defaults to 8080)
2. the url of the master (if not specified the instance will be the master)

For example:

```
> ./startup.sh
> ./startup.sh 8184 http://localhost:8182/api/v1
```

Versus Web Service and RESTful APIs

The Versus web service allows distributed applications to make use of the Versus comparison algorithms and hardware resources remotely.

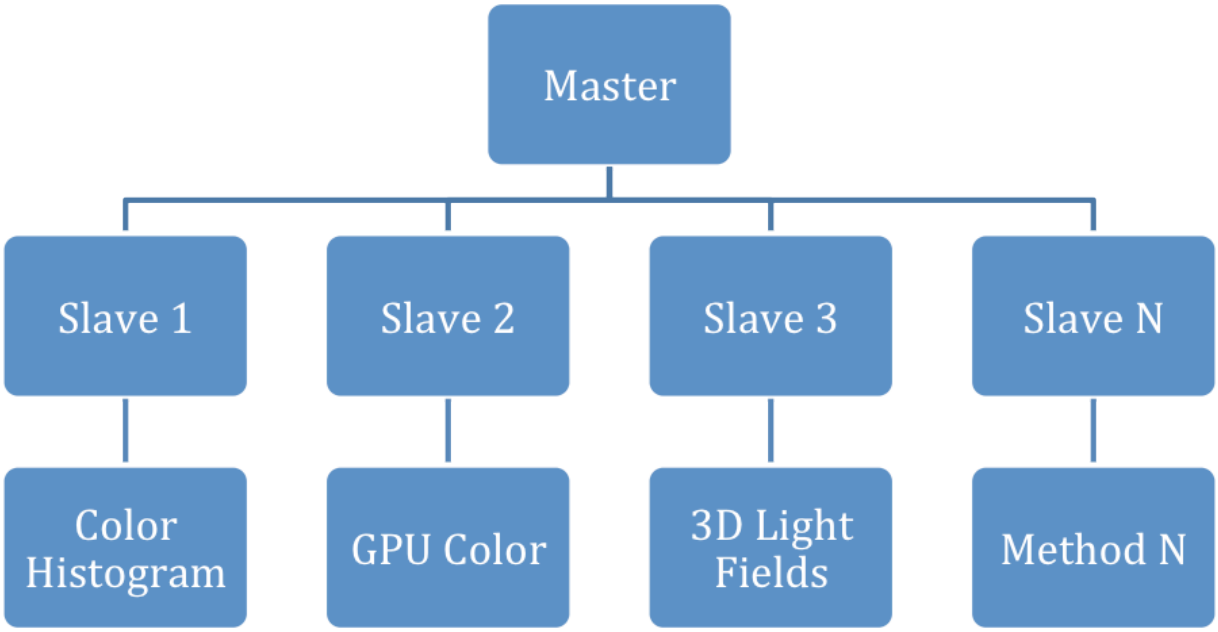
For those application developers who might not have the most advanced computational resources and can scale horizontally their computational resources by adding more hardware of the same specifications.

The service interface exposes methods written using the RESTful software architecture over the Hypertext Transfer Protocol (HTTP). Any client who can speak HTTP, can easily communicate with the service after understanding the specific endpoints available in the Versus web service. Because most programming languages and systems support HTTP, and the RESTful architecture is easy to understand, writing clients for the service is easy.

The goals of this service layer are 1) to enable horizontal scaling by allowing the addition of computational resources in cloud-like way 2) to allow multi-OS executions of comparisons (i.e., the composition of multiple Versus services running on different OS platforms), and 3) to provide comparison web services for orchestrating complex services in workflows. We describe these three goals in more detail.

The design of the service layers supports a master-slave architecture. A slave service can subscribe itself to a master service. New slaves can be added to the master at runtime so that available resources can shrink and grow on demand. This will help optimize resource use when dealing with very large collections of documents.

When a master service becomes aware of a new slave, it can query the slave using appropriate service interfaces to ask what methods are available on it. This is particularly useful for methods that require specific resources. For example, certain methods might only work on a specific operating system. A method implemented on a FPGA board might only be available on specific hardware resources. By being part of the cloud, all the services provide a consistent interface to the client, so that the client does not have to worry about implementation details and can focus on which methods to execute based on their needs.



Last but not least, by providing a service interface to comparison resources, we enable loose coupling of potential clients and comparison services. This means that access to the comparison services can be easily scripted in any application without requiring building against the Versus libraries. With a service architecture, any researcher or system that wants to make use of the comparison service has a simple way to do so.

The service API provides REST endpoints in line with underlying Java API. The current implementation includes the following REST endpoints for submitting different requests:

Adapters

An user can query for a list of current known adapters available on that particular instance of the versus service via HTTP GET request to `/adapters` .

For example :

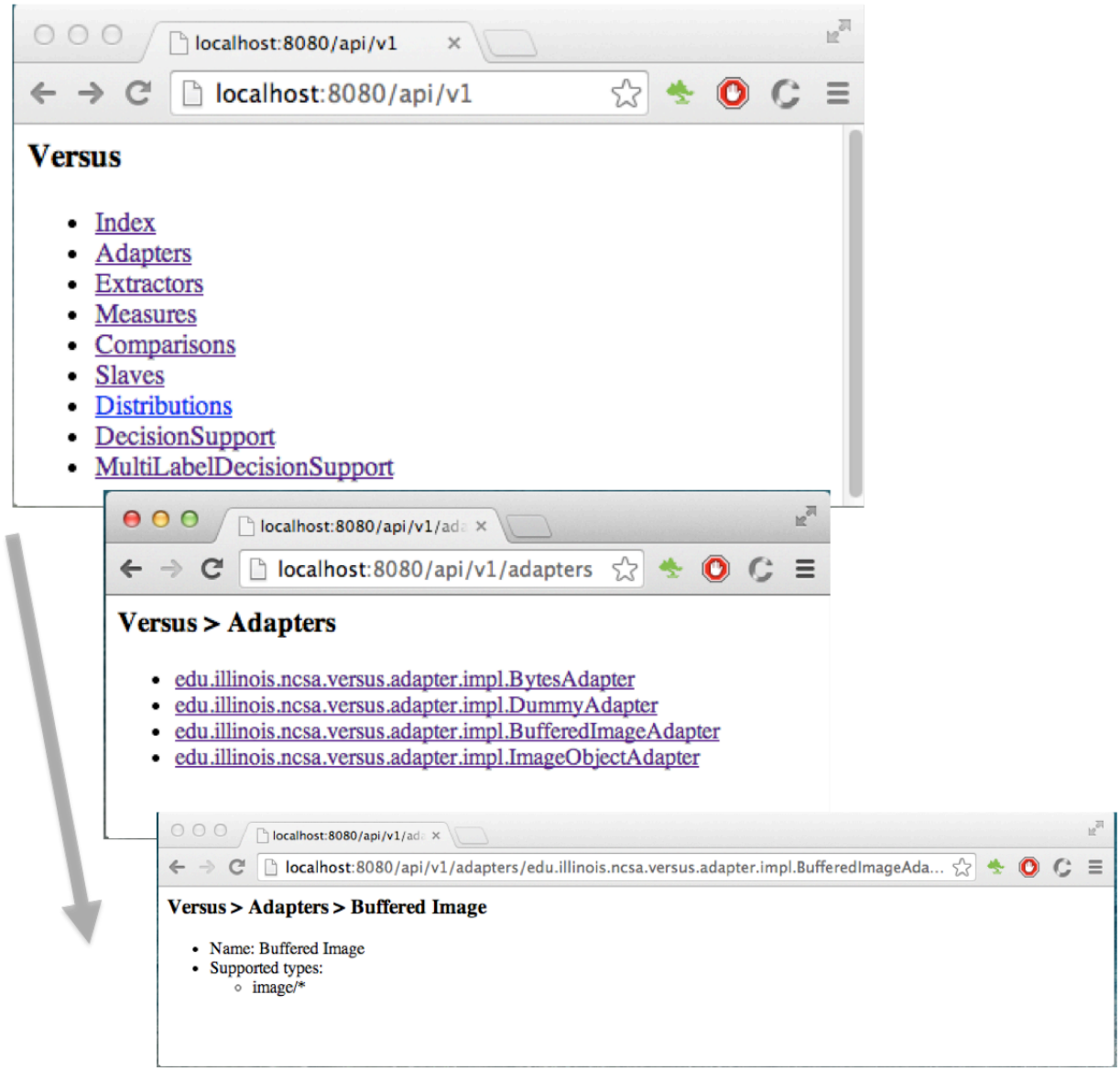
```
GET http://host:8080/api/v1/adapters
```

Each adapter available has its own endpoint describing information about that particular module via GET request to `/adapters/{adapter_id}`.

```
GET http://host:8080/api/v1/adapters/{adapter_id}
```

`{adapter_id}` is the specific identifier of the adapter that we are interested in. By default, the service uses the fully qualified class name of the Java class as the identifier of the adapter.

An example of an web interface to query `/adapters` REST endpoints is shown below:



Notice that in the last URL the last part of the URL is the ID of that particular adapter inside Versus. The important thing to keep in mind is that the client does not need to know this ahead of time because that particular URL is just one of the many returned by the previous GET. The above figure shows an example of this when interacting with the service from a browser.

Extractors

To get the list of currently available extractors on that particular instance of the versus service, an

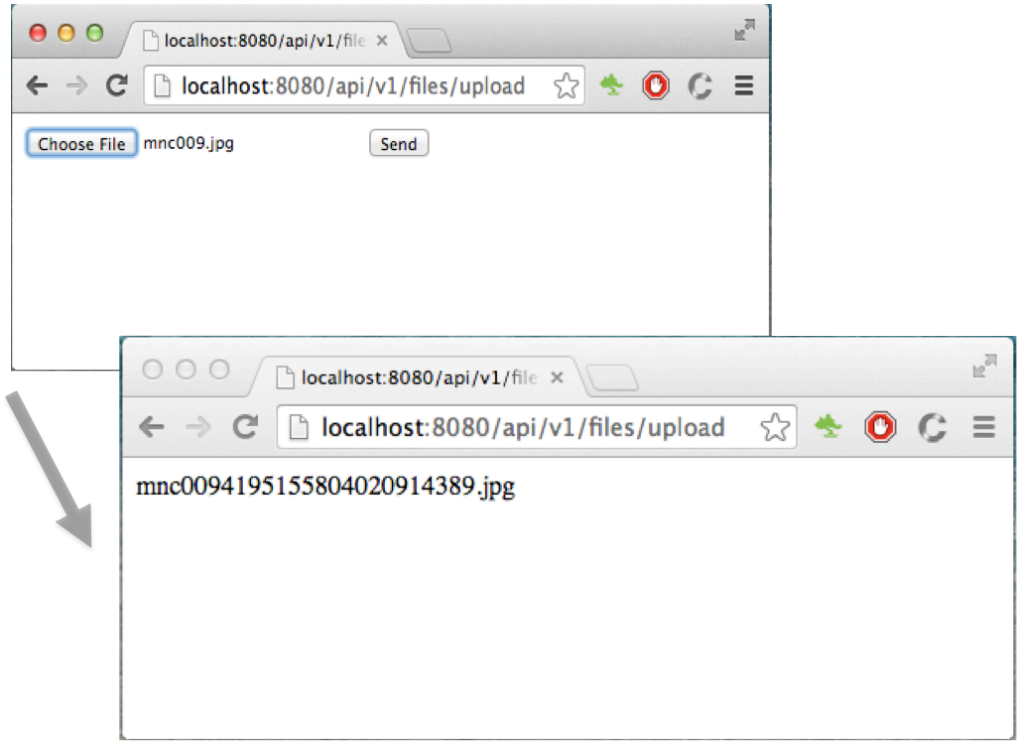
user can query REST endpoint `/extractors`. To get details about specific extractor module, the user can query `/extractors/{extractor_id}`. Like for adapters, Versus web server can be queried for extractors using web interface.

Measures

To get the list of currently available measures on that particular instance of the versus service, an user can query REST endpoint `/measures`. To get details about specific measure module, the user can query via GET request to `/measures/{measure_id}`. Like for adapters and extractors, Versus web server can be queried for measures using a web browser.

File Upload

In order to perform pairwise comparison between two files, the two files need to be uploaded. A file can be uploaded via GET request to `/files/upload` followed by a POST request. With a GET request to the */files/upload*, the server sends a webform that allow you to select and upload a file. After one press *send*, it does a POST request and upload the file to the server and sends back a `file id` to the client. This flow of action is shown below.



A previously uploaded file can be downloaded via GET request to the endpoint `/files/{file id}`.

Comparison REST API

To submit a new comparison request to the versus web server, one can send an HTTP POST request to the URL:

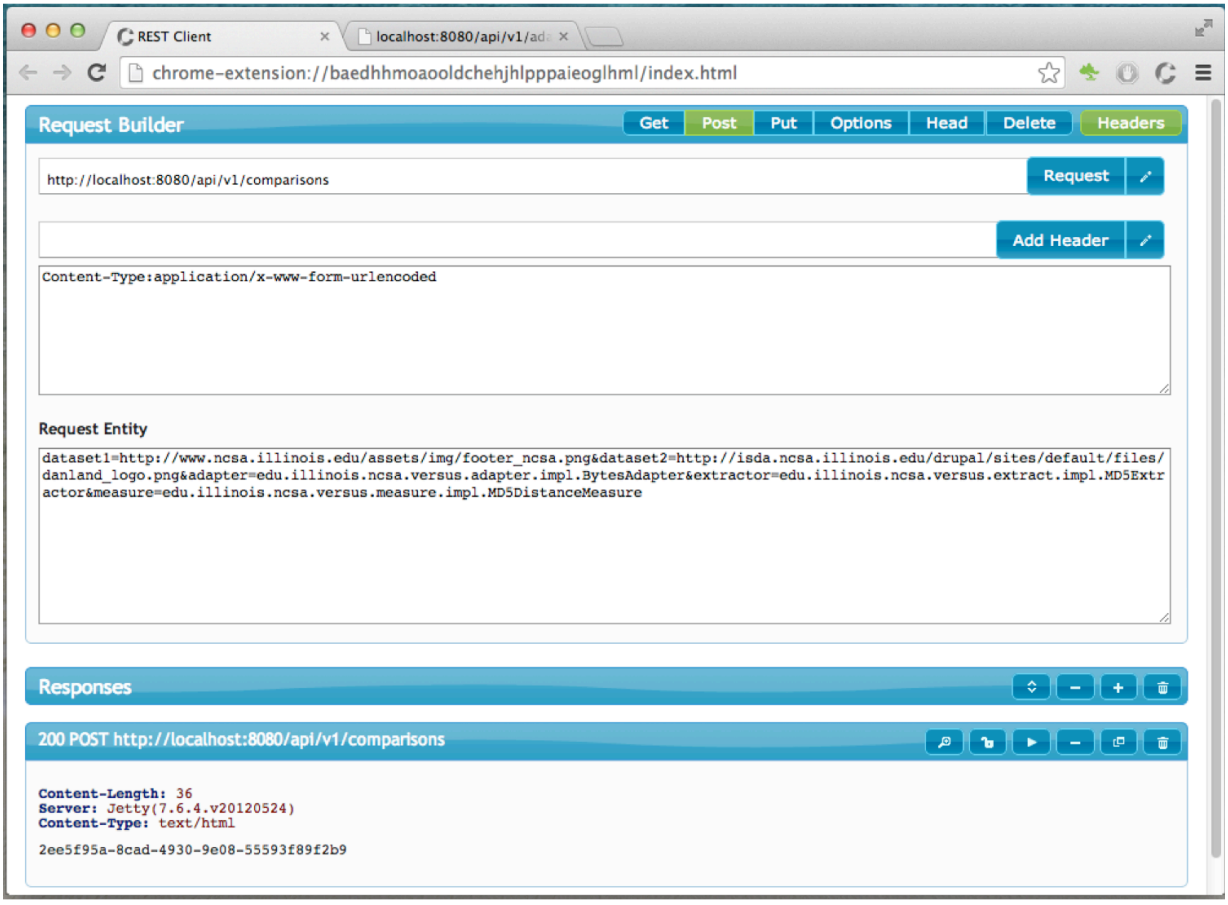
```
http://host:8080/api/v1/comparisons
```

Included in the body of the POST are URLs of the two files being compared and the identifiers of the adapter, the extractor, and the measure to use.

For example, the arguments are specified as shown below (key, value) pair appended to form the body of the request:

```
dataset1=file1Url&dataset2=file2Url&adapter=AdapterId&extractor=ExtractorId&measure=
measureId
```

A web browser with GUI extensions to specify HTTP methods and body can be used to make a comparison request to the Versus Web Server. For example for Chrome you can try cREST client. An example of using cREST client with chrome to make HTTP POST request to `/comparisons` endpoint is shown below.



In the response to the request, the server returns a comparison id and URL to access the

comparison.

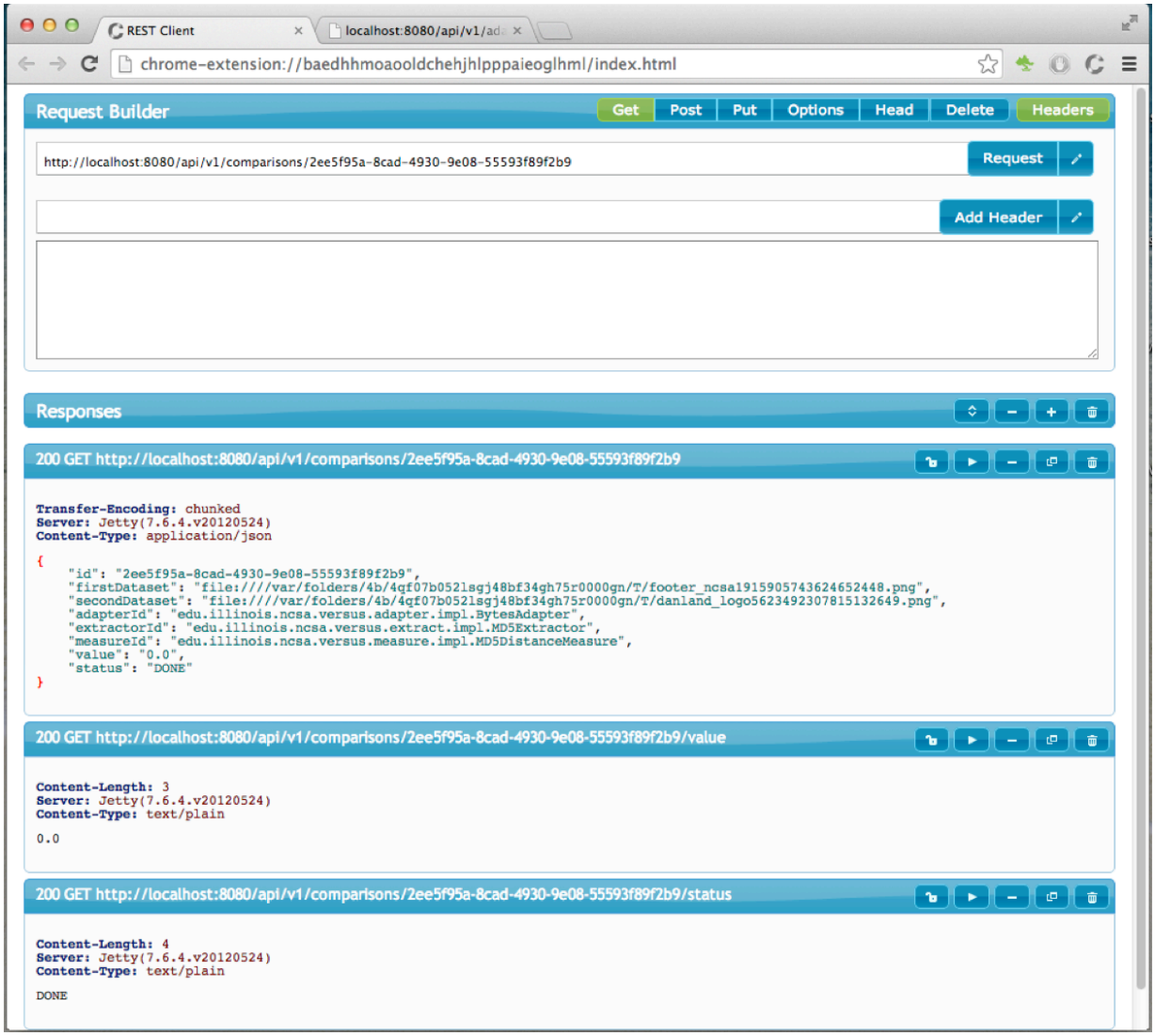
The client can access the information about the comparison using the following URL:

```
http://host:8080/api/v1/comparisons/{id}
```

The client can query the status and and value of the specified comparison using the following URLs :

```
http://host:8080/api/v1/comparisons/{id}/status
http://host:8080/api/v1/comparisons/{id}/value
```

An example of comparison service endpoint with various status and value request is shown below.



Decision Support Service REST API

Decision support web services that utilize the Versus core Framework and Library have been implemented. The implementation involved developing decision support web services for getting statistics and optimal combination of adapter, extractor and measure for binary labeled and multi labeled sampled files.

To submit a decision support service request for binary labeled files (similar or dissimilar files) to the versus web server, send an HTTP POST request to the URL

```
http://host:8080/api/v1/decisionSupport
```

Included in the body of POST are URLs of the set of similar and dissimilar files, an adapter to use.

For example: The arguments are appended with '&' and provided as request body as shown below.

```
adapter=edu.illinois.ncsa.versus.adapter.impl.BufferedImageAdapter&
similarFiles=http://localhost:8080/api/v1/files/mno0105947062984081429305.jpg&
similarFiles=http://localhost:8080/api/v1/files/mno011279764175837685 8352.jpg&
similarFiles=http://localhost:8080/api/v1/files/mno0128915359200313575286.jpg&
similarFiles=http://localhost:8080/api/v1/files/mno0135147501424480553896.jpg&
similarFiles=http://localhost:8080/api/v1/files/mno0144276238537340112360.jpg&
similarFiles=http://localhost:8080/api/v1/files/mno0151971531815575185168.jpg&
dissimilarFiles=http://localhost:8080/api/v1/files/Image07117236837207586707.jpg&
dissimilarFiles=http://localhost:8080/api/v1/files/Image084565197057939613645.jpg&
dissimilarFiles=http://localhost:8080/api/v1/files/Image096244790252667283114.jpg&
dissimilarFiles=http://localhost:8080/api/v1/files/Image101369081828034857026.jpg&
dissimilarFiles=http://localhost:8080/api/v1/files/Image114742913348975720071.jpg&
dissimilarFiles=http://localhost:8080/api/v1/files/Image126000615581631743708.jpg
```

Note that the format of input is same as in case of comparison service.

Once the decision support request is submitted, it returns an id for the request submitted.

You can access the results of the request via HTTP GET method to

```
http://host:8080/api/v1/decisionSupport/{id}
```

You can get all decision support requests results via HTTP GET to the endpoint `\decisionSupport`.

Similarly, to submit a request for decision support service for multi labeled files, send a HTTP POST request to

```
http://host:8080/api/v1/multiLabelDecisionSupport
```

Included in the POST body urls of the multilabeled files, number of labels, method to perform decision support(i.e probabilitics or inverse K-mean) and an adapter to use.

Shown below is an example of the body of the POST request.

```
adapter=edu.illinois.ncsa.versus.adapter.impl.BufferedImageAdapter&
method=probabilistic&
k:3&
data0=http://localhost:8080/api/v1/files/owl0095898639013296543251.jpg&
data0=http://localhost:8080/api/v1/files/owl0107580951319826054753.jpg&
data0=http://localhost:8080/api/v1/files/owl0116858822357862217962.jpg&
data1=http://localhost:8080/api/v1/files/owl0126252339603498263706.jpg&
data1=http://localhost:8080/api/v1/files/owl0138941707136462733705.jpg&
data1=http://localhost:8080/api/v1/files/owl0144347999815809592024.jpg&
data2=http://localhost:8080/api/v1/files/owl0152520539069110679269.jpg&
data2=http://localhost:8080/api/v1/files/owl0164203616588807869187.jpg&
data2=http://localhost:8080/api/v1/files/owl0177472309747319811844.jpg&
data0=http://localhost:8080/api/v1/files/owl0187953897869039444700.jpg&
data1=http://localhost:8080/api/v1/files/owl0194097604817385207009.jpg&
data2=http://localhost:8080/api/v1/files/owl0206552970288709601801.jpg&
```

Once the request is submitted, an id is returned and the results can be accessed via HTTP GET to

```
http://host:8080/api/v1/multiLabelDecisionSupport/{id}
```

Master Slave REST API

This master slave API is mostly used by administrator to horizontally scale the versus web service.

A versus web server instance becomes a slave if in `versus.properties` file , a master url has been specified that is different than the instance's own url or as command line argument, master url is specified., otherwise by default it becomes a master.

A versus web server automatically register itself to the master as slave using REST endpoint `/slaves/add` at instantiation of the service. To query list of slaves, the client can send a GET request to endpoint `/slaves` .

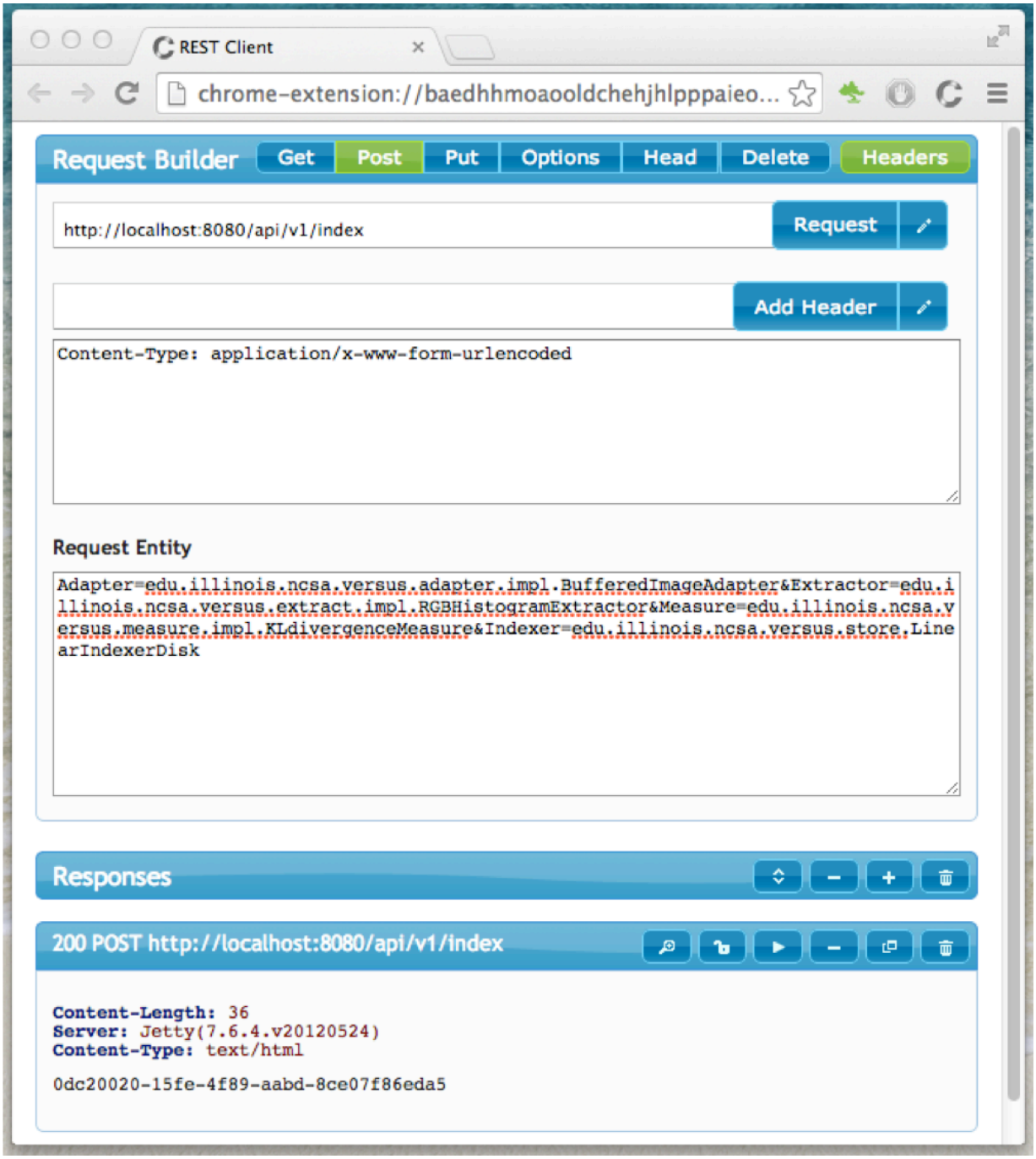
Indexing Service REST API

Versus indexing service can be accessed using the endpoint `/index` and URL of the form:

```
http://host:8080/api/v1/index
```

where host is the IP address of the host running Versus web server. An HTTP GET method to this URL would return all indexes ids stored in Versus web server. To create a new index, we send a HTTP POST request with (Adapter Id, Extractor Id, Measure Id, Indexer Id) tuple to the above endpoint (i.e /api/v1/index). All the indexes can be deleted via HTTP DELETE method to the same URL.

An example usage of the endpoints is as shown below:



A specific index with id {index_id} can be accessed via HTTP GET method to the following URL :

```
http://host:8080/api/v1/index/{index_id}
```

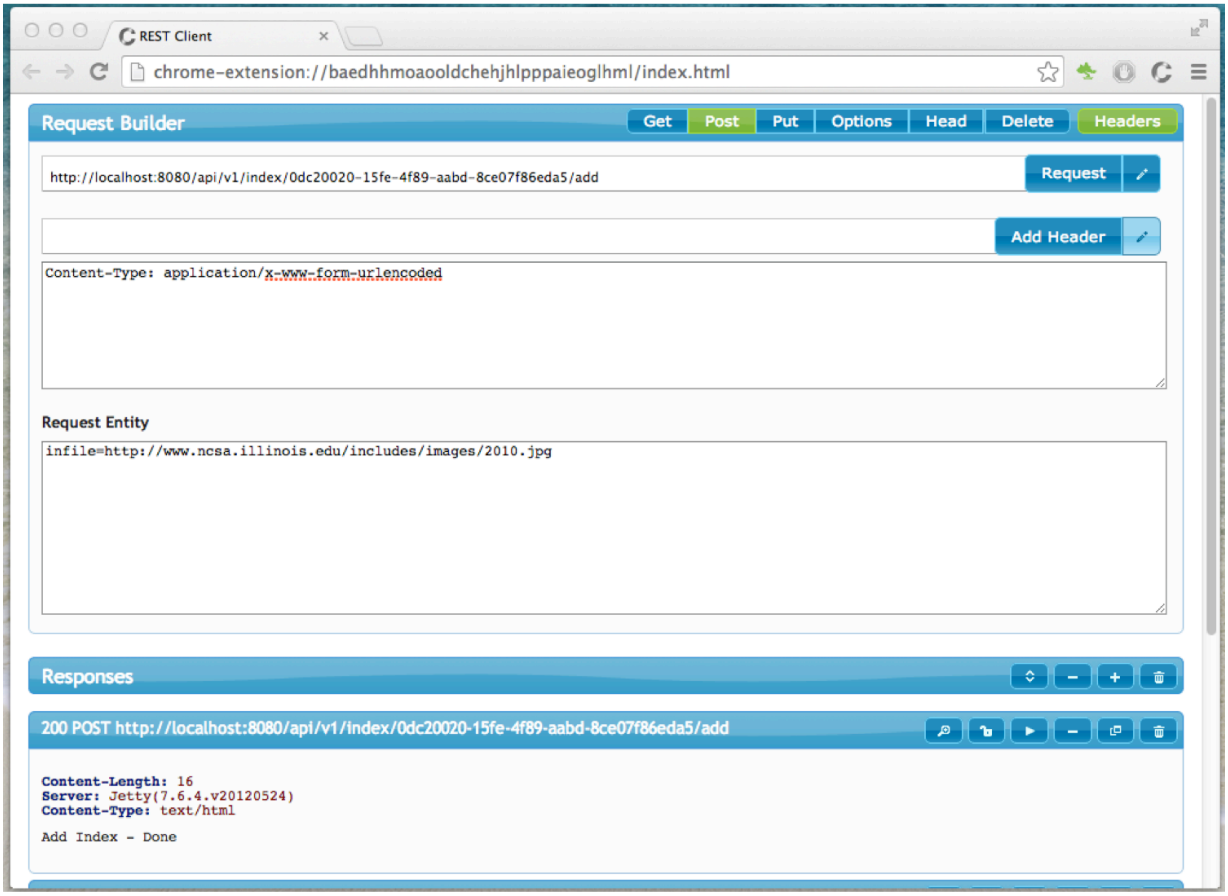
It would return Adapter Id, Extractor Id, Measure Id, Indexer Id and List of File Ids in the specified

index. An index can be deleted with the same URL via HTTP DELETE request.

If we add more files to the database and consequently need to add their content descriptors to a specific index, we send an HTTP POST request with file's URL to the following endpoint:

```
http://host:8080/api/v1/index/{index_id}/add
```

An example of the usage of the `/index/{indexid}/add` endpoint is shown below:



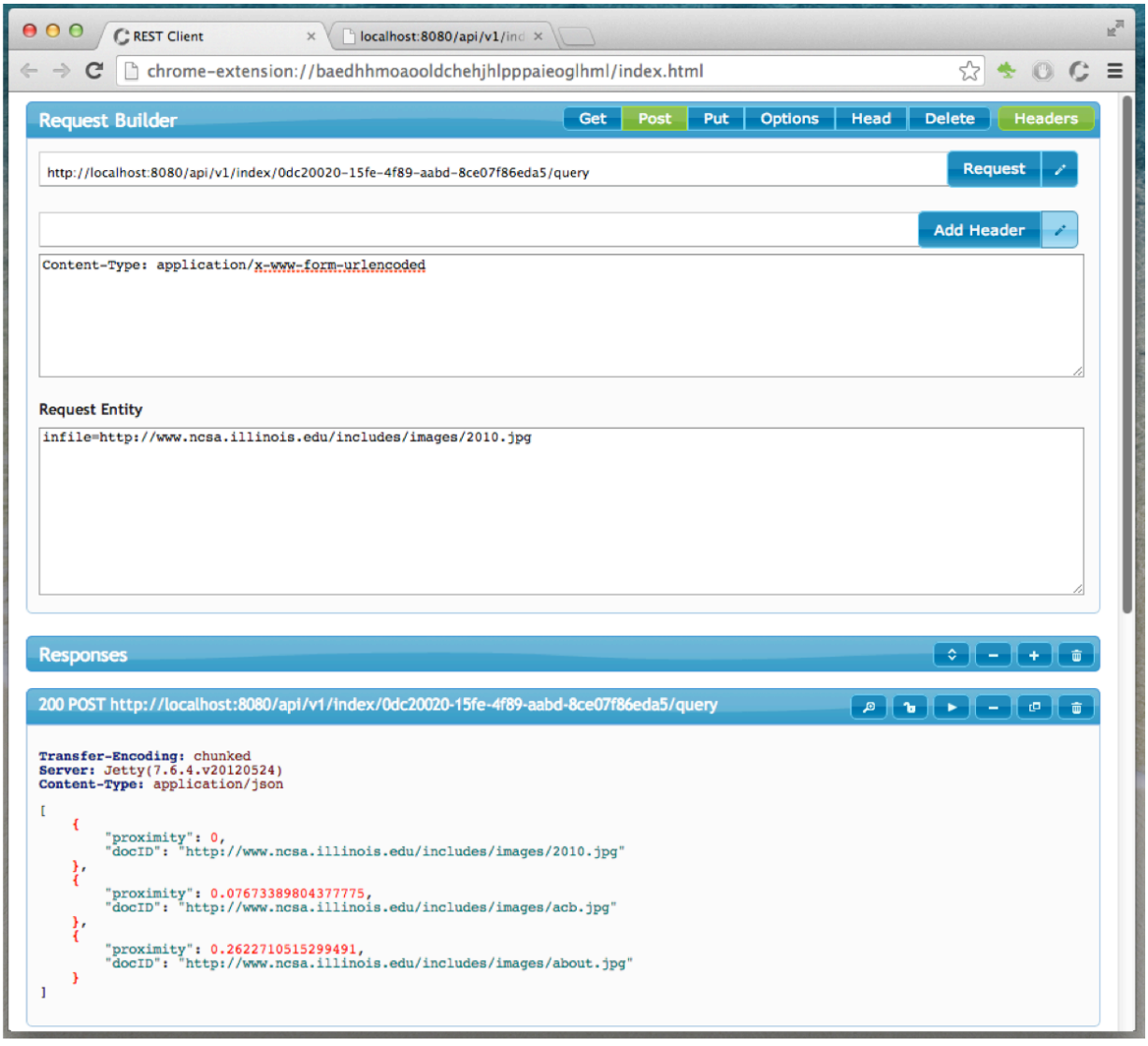
After several additions of files' content descriptors and corresponding identifiers to respective lists of an indexer, we build the specified index via an HTTP POST request to the following URL with no request body specified:

```
http://host:8080/api/v1/index/{index_id}/build
```

A specific index can be queried by sending request via an HTTP POST with query's file URL to the following URL:

```
http://host:8080/api/v1/index/{index_id}/query
```

This would display the search results for the query file. An example usage of this endpoint is shown below.



Clients

You can write different clients to talk to the service RESTful API.

cURL

An user can use curl to test the service. If you are on Linux or MacOSX you should have it already. Try typing `curl` on the command prompt. If you are on windows, you can download a build at

http://curl.haxx.se/. If you want a more rich GUI experience most web browsers have extensions that can be used instead. For example for Chrome you can try cREST client.

To start testing versus-service without deploying it, you can use the service available at http://versus.ncsa.illinois.edu:8182/api/v1 as in the example but if you are using a different instance (for example `localhost`) replace accordingly.

To list methods Extractors available at that location:

```
curl -H "Accept: application/json"
http://versus.ncsa.illinois.edu:8182/api/v1/extractors
```

We asked for json. The service actually default to json. But if we go there with the browser we see formatted html. Try requesting html:

```
curl -H "Accept: text/html" http://versus.ncsa.illinois.edu:8182/api/v1/extractors
```

Now let's make it a bit more interesting by submitting a comparison between two files.

```
curl -H "Content-type: application/x-www-form-urlencoded" -X POST -d
"dataset1=http://www.ncsa.illinois.edu/assets/img/footer_ncsa.png&dataset2=http://is
da.ncsa.illinois.edu/drupal/sites/default/files/danland_logo.png&adapter=edu.illinoi
s.ncsa.versus.adapter.impl.BytesAdapter&extractor=edu.illinois.ncsa.versus.extract.i
mpl.MD5Extractor&measure=edu.illinois.ncsa.versus.measure.impl.MD5DistanceMeasure"
http://versus.ncsa.illinois.edu:8182/api/v1/comparisons
```

Please pay close attention to the payload of our submission. It is the same format as we submitted through the browser using CREST client. Take a look in a more readable form:

```
dataset1  = http://www.ncsa.illinois.edu/assets/img/footer_ncsa.png
dataset2  =
http://isda.ncsa.illinois.edu/drupal/sites/default/files/danland_logo.png
adapter   = edu.illinois.ncsa.versus.adapter.impl.BytesAdapter
extractor = edu.illinois.ncsa.versus.extract.impl.MD5Extractor
measure   = edu.illinois.ncsa.versus.measure.impl.MD5DistanceMeasure
```

It's hard to see if you are on the command line, but the call should have returned a long string of characters, for example `f6eda3ee-400e-475a-9703-426811dc4633`. That's the id of our new comparison. Let use that id to see the status of the comparison:

```
curl http://versus.ncsa.illinois.edu:8182/api/v1/comparisons/f6eda3ee-400e-475a-
9703-426811dc4633
```

Should return something like this:

```
{"id":"f6eda3ee-400e-475a-9703-
426811dc4633","firstDataset":"http://www.ncsa.illinois.edu/assets/img/footer_ncsa.pn
g","secondDataset":"http://isda.ncsa.illinois.edu/drupal/sites/default/files/danland
_logo.png","adapterId":"edu.illinois.ncsa.versus.adapter.impl.BytesAdapter","extract
orId":"edu.illinois.ncsa.versus.extract.impl.MD5Extractor","measureId":"edu.illinois
.ncsa.versus.measure.impl.MD5DistanceMeasure","value":"0.0","status":"DONE"}
```

For this particular measure a value of 0 means the MD5 hashes did not match, while with 1 they did. Try comparing the same file:

```
curl -H "Content-type: application/x-www-form-urlencoded" -X POST -d
"dataset1=http://www.ncsa.illinois.edu/assets/img/footer_ncsa.png&dataset2=http://ww
w.ncsa.illinois.edu/assets/img/footer_ncsa.png&adapter=edu.illinois.ncsa.versus.adap
ter.impl.BytesAdapter&extractor=edu.illinois.ncsa.versus.extract.impl.MD5Extractor&m
easure=edu.illinois.ncsa.versus.measure.impl.MD5DistanceMeasure"
http://versus.ncsa.illinois.edu:8182/api/v1/comparisons
```

Let's try a different set of measures. Something image related like the distance between binned RGB histograms:

```
curl -H "Content-type: application/x-www-form-urlencoded" -X POST -d
"dataset1=http://www.ncsa.illinois.edu/assets/img/footer_ncsa.png&dataset2=http://is
da.ncsa.illinois.edu/drupal/sites/default/files/danland_logo.png&adapter=edu.illinoi
s.ncsa.versus.adapter.impl.BufferedImageAdapter&extractor=edu.illinois.ncsa.versus.e
xtract.impl.PixelHistogramExtractor&measure=edu.illinois.ncsa.versus.measure.impl.Hi
stogramDistanceMeasure" http://versus.ncsa.illinois.edu:8182/api/v1/comparisons
```

Let's switch our attention to indexing. Let's create a new index by specifying what implementations to use.

```
curl -H "Content-type: application/x-www-form-urlencoded" -X POST -d
"Adapter=edu.illinois.ncsa.versus.adapter.impl.BufferedImageAdapter&Extractor=edu.il
linois.ncsa.versus.extract.impl.RGBHistogramExtractor&Measure=edu.illinois.ncsa.vers
us.measure.impl.KLdivergenceMeasure&Indexer=edu.illinois.ncsa.versus.store.LinearInd
exerDisk" http://versus.ncsa.illinois.edu:8182/api/v1/index
```

Let's add a few files to the index:

```
curl -H "Content-type: application/x-www-form-urlencoded" -X POST -d
"infile=http://www.ncsa.illinois.edu/includes/images/about.jpg"
http://versus.ncsa.illinois.edu:8182/api/v1/index/54d55e93-f268-428f-860b-
6d80280f76a7/add
```

After adding at least three files we can query by posting to the `/query` endpoint:

```
curl -H "Content-type: application/x-www-form-urlencoded" -X POST -d
"infile=http://www.ncsa.illinois.edu/includes/images/about.jpg"
http://versus.ncsa.illinois.edu:8182/api/v1/index/54d55e93-f268-428f-860b-
6d80280f76a7/query
```

Java Client

One can also write a java client for versus-service comparisons easily.

For example:

```
VersusComparisonClient client=new VersusComparisonClient("localhost",8080);

String dataset1="http://www.ncsa.illinois.edu/includes/images/about.jpg";
String dataset2="http://www.ncsa.illinois.edu/includes/images/acb.jpg";
String adapter="edu.illinois.ncsa.versus.adapter.impl.BufferedImageAdapter";
String extractor="edu.illinois.ncsa.versus.extract.impl.RGBHistogramExtractor";
String measure="edu.illinois.ncsa.versus.measure.impl.HistogramDistanceMeasure";

client.setData(dataset1, dataset2, adapter, extractor, measure);
String comparisonid=client.compare();
String status=client.getStatus(comparisonid);
if(status.equals("DONE"))
{
String value=client.getValue(comparisonid);
log.debug("Comparison ID: "+ comparisonid+"    Status="+ status+ "  Value="+value);
}
```

VersusComparisonClient is nothing but a wrapper around a HttpClient that establishes a http connection between client and Versus web server and does all the http requests. This example has been included in the versus-service package.

Versus Web Application

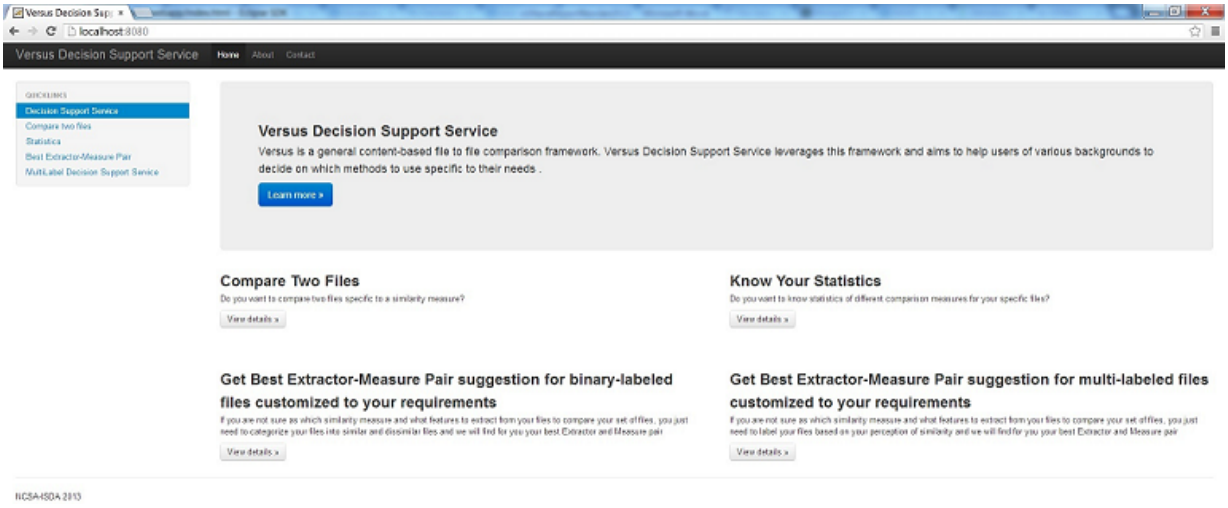
We designed Versus web application focusing on the end-user needs specifically for comparison

and decision support services.

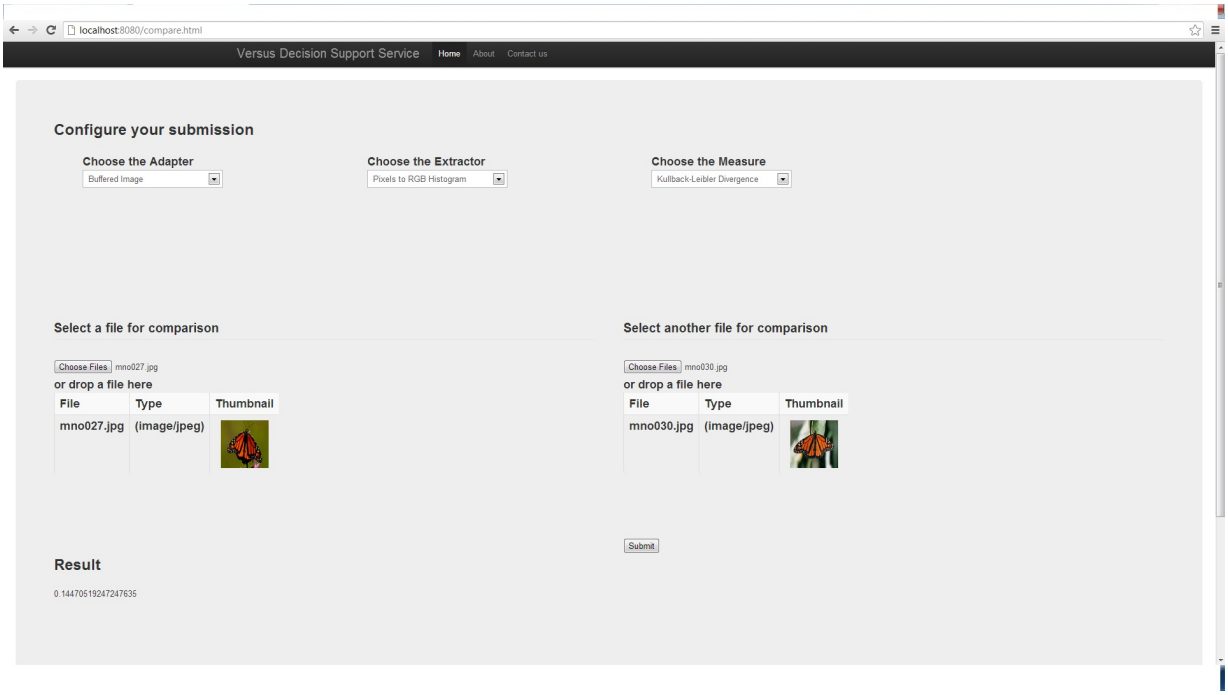
To start Versus web application, you need to start the JettyServer.java in ds-web project. To use Versus web application, type in

```
http://localhost:8080
```

into your browser. It will display the page shown below:

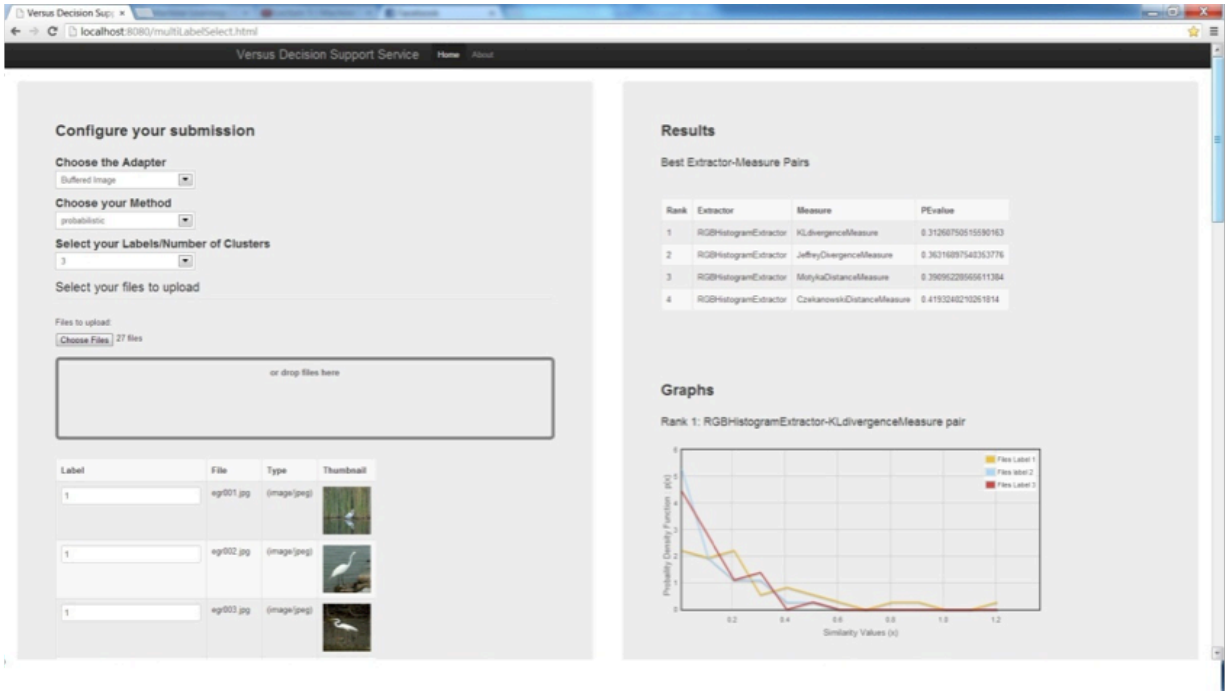


To compare two files, click on the corresponding button and it will go to the compare webpage as shown below:



Compare two files web interface lets the user to choose an adapter, an extractor and a measure and upload two files to perform the comparison and returns the similarity value.

If you want to decide on the best extractor-measure pair for a multilabled files, then you need to click on the corresponding button, it will go to decision support web page for multi-labeled files. The webpage is shown below:



In this web interface, the user chooses an adapter, the decision support method, the number of labels, and uploads a set of files and label them, and submit the request. The results and corresponding graphs are displayed on the webpage for the user.

Developer's Manual

A developer is a person who either contributes new methods to the existing framework using existing Java APIs or a person who contribute to the design of the framework.

Software Requirements

1. Eclipse IDE for Java Developers (the recent version has maven in it)
2. Java JDK 6
3. MySQL (if you configure versus.properties for mysql for storage of all comptutations and/or index)

Checking Out Code

Versus Project source code can be obtained from from our git repository at

https://opensource.ncsa.illinois.edu/stash/scm/vs/

Versus Project consist of several sub projects. versus-core is the standalone buildable unit and that does not depend on any another sub projects. It can be checked out using the following link:

https://opensource.ncsa.illinois.edu/stash/scm/vs/versus-core.git

versus-core consists of all basic interfaces and their implementations. It can be used as a library/framework to other service.

For other services and support, one can checkout other projects such as versus-service, versus-image, etc similar to versus-core. Note that all other sub projects depends on versus-core. Following are the list of projects and their purposes:

1. versus-image: It consists of different implementation of adapter, extractor and measure interface specific to images. It depends on versus-core.
2. versus-service: It provides REST API for various services using versus framework and depends on versus-core, versus-image and versus-census. It provides simple comparison service, decision support service and indexing services.
3. versus-ds-web: It provides the web interface to decision support service. It depends on versus-core,versus-service, versus-image.
4. versus-census: It provides specific implementation of extractor interface for handwritten scanned document.

Execution

Before execution, one has to set various configuration parameters in the *versus.properties* file as specified in User's manual.

Go to *edu.illinois.versus.service* and run JettyServer.java as a java application

Building code

To build the source code and create a jar file, the developer has to do

- Run as->maven clean
- Run as-> maven build
- goal: assembly:assembly

Versus Java API

The versus-core package consists of core interfaces required in the Versus framework. In the overview section we gave an overview of those interfaces. In this section, we will go in more details of Java API and classes and methods responsible for it.

Adapter

The interface that defines the Adapter is *edu.illinois.ncsa.versus.Adapter*. It enforces two methods: getName() and getSupportedMediaTypes(). getName() method returns the name of the adapter and getSupportedMediaTypes() lists the supported mime types by the adapter.There are also interfaces that extends this Adapter interface such as Fileloader (to load a file from the disk to the data structure specific to the adapter), HasPixels (get pixels from the document), HasBytes (get bytes from the document), etc.

The package that contains various implementation of the Adapter Interfaces is *edu.illinois.ncsa.versus.adapter.impl*. Currently there are four implementations of Adapters: BytesAdapter (versus-core), DummyAdapter (versus-core), BufferedImageAdapter (versus-image) and ImageObjectAdapter (versus-image).

Extractor

The Extractor interface extracts a particular feature from an adapter. It enforces the method extract() that takes an adapter as input and returns a specific descriptor. Other methods that it defines are: getName(), getFeatureType(),hasPreview(),and previewName(). Currently there are several implementations of Extractor interface and can be found in the package *edu.illinois.ncsa.versus.extract.impl* inside versus-core package as well inside versus-image for image extractors.

Measure

The Measure interface compares two features and returns a similarity value indicating how similar the two features are. It enforces compare() method that takes as input two descriptors obtained from an extractor and returns a similarity value. Three other methods defined in the interface are

getFeatureType(), getName() and getType(). Currently there are several implementations of Measure interface that can be found inside package *edu.illinois.ncsa.versus.measure.impl* inside versus-core project and in versus-image project.

Engine

The Engine interface enforces methods to implement methods that do the comparison tasks. There are two implementations for this interface: ExecutionEngine and ComphrensiveEngine.

There are also other engines in the *edu.illinois.ncsa.versus.engine.impl* such as IndexingEngine and ExtractionEngine but they do not extend Engine interface currently. *edu.illinois.ncsa.versus.engine.impl.IndexingEngine* provides a way to index a document. The user can add new documents by passing in files to the engine. The engine keeps a thread pool of workers to handle however many jobs at one time depending on the number of threads configured. *edu.illinois.ncsa.versus.engine.impl.ExtractionEngine* enables user to pass in file for extraction of features and returns descriptors for it.

Indexer

To help developers for rapid development of indexing service with different indexers, we defined an Indexer interface. It enforces three important methods: addDescriptor() to add descriptor of the document to the indexer; build() to build the index; and query() to query for a digital object into the existing index. The various implementation of Indexers can be found in edu.illoinois.ncsa.versus.store package inside versus-service project. Note that for any implementation of Indexer interface has both disk and JDBC implementation of it. That is a linear indexer can serializes it content and store it in disk or the indexer can use MySQL to do the same. So you will find two implementation of LinearIndexerDisk and LinearIndexerJDBC. This is true for other indexers as well.

Adding New Implementations

Developers who want to add new methods to the framework can easily do so by extending Java APIs in versus-core project. To add new implementations of Adapter, Extractor, and Measure need to extend their respective interface and implement the methods. To be able to retrieve all the implementations of Adapter, Extractor amd Measure avaliable on the classpath, ServiceLoader functionality has been introduced. The CompareRegistry class uses ServiceLoader to get all available implementations on classpath and load them to a list. It also implemented some helper methods such as getAdapters(), getExtractors(),getMeasures(), getAvailableAdaptersIds(), getAvailableExtractorIds(), getMeasureIds(), etc.

To use SeviceLoader functionality, one needs to add the class name to file for the corresponding Interface under *services* folder in *src/main/resources/META_INF* . For example, if you have implemented an Extractor, say *edu.illinois.ncsa.extract.impl.PdfExtract*, then add *edu.illinois.ncsa.extract.impl.PdfExtract* to edu.illinois.ncsa.versus.extract.Extractor file in *src/main/resources/META_INF/services*.

Versus-Service

Versus web service is designed to support RESTful API. This has been implemented using JBoss Resteasy(which follows JAX-RS specifications). Several services have been implemented and can be accessed through their REST endpoint as explained in the user manual section. Inthis section we look into the details of versus web service development and configuration.

Storage Implementation

Individual services have the option of storing comparison results in memory, disk, MySQL or in mogoDB. This is also true for other service such as indexing service etc. Files are stored in disk. From an end-user perspective, switching between storage implementations is enabled via the versus.properteis file by setting the desired storage choice. From developer perspective, implementing a new storage implementation is enabled by providing implementations of a simple Java interface. To link the interface to the implementations, we use dependency injection and rely on the Guice Java library to make it easier for new storage implementations to be added to framework. The dependency injection design pattern decouples the different implementations to the framework.

For example: For comparison service, *edu.illinois.ncsa.versus.store.ComparisonProcessor* interface defines methods to manipulate storage of comparisons in repository. There are three implementation of this interface: *InMemoryComparisonProcessor*, *JDBCComparisonProcessor* and *MongoComparisonProcessor*. There is another interface edu.illinois.ncsa.versus.store.ComparisonService that handles dependency injection of storage implementation in repository. There is a class *edu.illinois.ncsa.versus.store.RepositoryModule* that binds the ComparisonService interface to its implementation class,i.e., *ComparisonServiceImpl*, and binds the ComparisonProcessor class to a specific storage implementation class. Based on the storage option configured in the `versus.properties`, the ComparisonProcessor binds that to the implementation, say *JDBCComparisonProcessor* , and ComparisonServiceImpl does the dependency injection of the implementation chosen and uses this implementation every time there is manipulation of comparisons and store comparisons in MySQL.

Similarly, for indexing service, *edu.illinois.ncsa.versus.store.IndexProcessor* interface defines methods to manipulate storage of indexes in repository. There are three implementations of this interface: *InMemoryIndex*, *JDBCIndexProcessor* and *DiskIndexProcessor*. The interface *edu.illinois.ncsa.versus.store.IndexService* handles the indexes in the repository and does the dependency injection. Again in *edu.illinois.ncsa.versus.store.RepositoryModule*, all bindings are done.

Note that though an indexer has a disk and JDBC implementation, no specific implementation is binded in the RepositoryModule. The reason behind it is that an indexer is instantiated on user request for creation of index or query or add to index. So it is not known at the compile time about the implementation.

Servlet Configuration

When the Jetty Server starts, the servlet context needs to be initialized before the web application starts. For that, *edu.illinois.ncsa.versus.serviceVersusServletConfig* implements Java ServletContextListener interface. It enforces contextInitialized() and contextDestroyed(). The contextInitialized method loads versus registry to the servlet context, instantiates execution, indexing

and extraction engine, initialized all data structures that are shared across different requests. contextDestroyed() methods removes the context that were initialized by contextInitialized().

Guice dependency injector is also added to the servlet context implementing GuiceServletConfig class that extends GuiceServletContextListener.

All these listener classes and Application(service resource) are specified in src/main/webapp/WEB-INF/web.xml file whose path is specified in JettyServer for setting up WebApp context.

REST Service Resource

Various Versus web services are implemented and packaged in edu.illinois.ncsa.versus.rest in versus-service project. For example, for comparison service, we implemented a class *edu.illinois.ncsa.versus.rest.ComparisonResource* . To register the service with the web server so that REST endpoint, for example /comparisons, is available for users, the class is added to the versus REST service application in *edu.illonois.ncsa.versus.rest.VersusRestApplications* class.

Testing

To test various versus service, several unit tests have been included in the package *edu.illinois.ncsa.versus.rest* in the folder src/test/java. For example, to test comparison resource, a unit test is included with @Test annotation with testSubmit method. A developer can easily include their own unit test for different service resources.

Acknowledgments

The software development was partially supported by the National Archives and Records Administration (NARA).