***Heliophysics Integrated Observatory***

**Project No.: 238969**

**Call: FP7-INFRA-2008-2**

# HELIO API
# User Manual
*Version 2.1*

| | |
|---|---|
| *Title:* | **Client API – User Manual** |
| *Document No.:* | HELIO_FHNW_S5_005_UM |
| *Date:* | 25 June 2012 |
| *Editor:* | **Marco Soldati**, FHNW |
| *Contributors:* | |
| *Distribution:* | Project |


## Revision History

| Version | Date | Released by | Detail |
|---|---|---|---|
| 2.0 | 25/06/2012 | Marco Soldati | Initial release, based on deliverable HELIO-S5-002-d2_API_110831 |
| 2.1 | 25/09/2012 | Marco Soldati | Rework |
| | | | |
| | | | |

Note: Any notes here.

# Introduction

HELIO can be accessed in three different ways. First, each service implements a SOAP or REST style interface that can be directly used. Second, a Java library called HELIO Client API can be used for simplified programmatic access to HELIO. Third, HELIO can be used in an interactive way from a scripting environment.

This document focuses on the HELIO Client API, the second alternative. The first alternative is covered by the (ref Interface doc). The third alternative is based on IDL and further described in [1].

Although named "User Manual" this document is not intended for end users, but rather for developers wanting to use the HELIO system in their applications or scripts. Complementary to this document is the HELIO API developer guide [2] which puts its focus on the implementation details of the HELIO Client API.

## *Behind the 'HELIO Client API'*

The HELIO Client API sits on top of several other HELIO components. Figure 1 is adapted from the HELIO Architecture document and shows the structural view of the components involved in the HELIO infrastructure. It shows how the client applications, located in the service user layer, are connected to the HELIO Client and Service API. The HELIO Service API is not explicitly noted but implicitly described by the interface of the listed infrastructure components and service providers. Actually, the three arrows that are connected to the Service User Layer box can be seen as the three main access routes to the system.

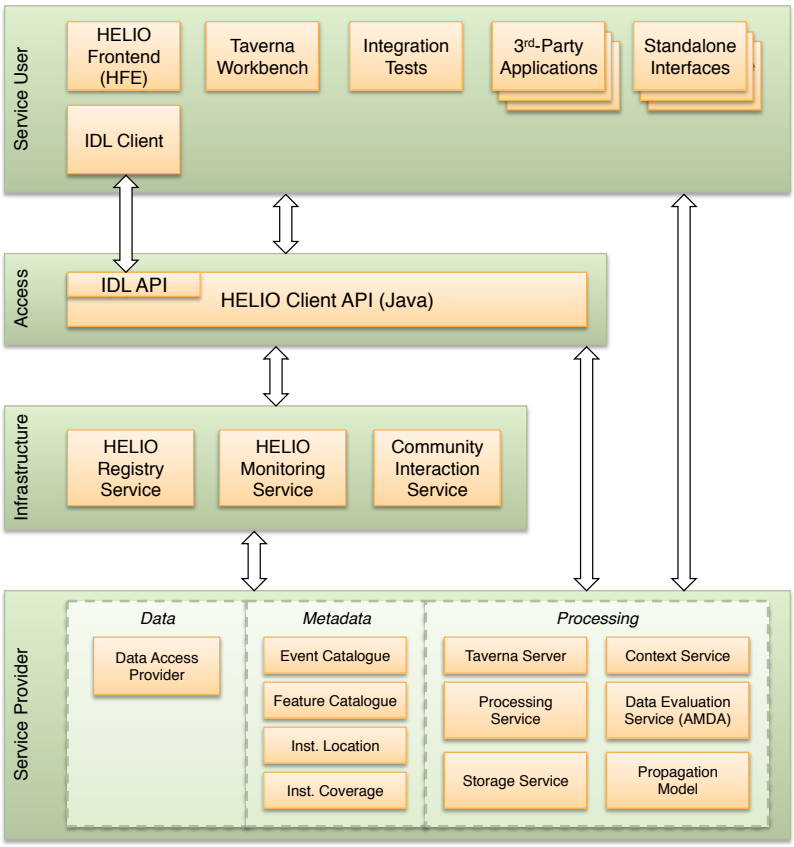


Figure 1 Structural view of the main components of the HELIO Architecture. Services User components can access the underlying services either directly through the individual components Service API, through the HELIO Client API or through the IDL API.

Service providers include infrastructure, data, meta-data and processing services. These services form the backend of the HELIO infrastructure and provide all information and facilities required to answer scientific questions. They are described in more detail in the HELIO Architecture [3] and in dedicated documents.

Service users consist of all client applications. Some client applications are developed and maintained by the HELIO consortium: the HELIO frontend, the Interactive Data Language (IDL) client, the Taverna Workbench and the HELIO testing infrastructure. Externally developed applications are other virtual observatories (VO), IDL scripts or other third party software. They all access the service providers in one of the three ways described above.

In the context of this document only the HELIO Client API is of interest. It is implemented as a Java library. It provides unified access methods to different underlying services. It transparently handles service discovery, failover and security and supports the user in calling asynchronous services. In addition the HELIO Client API enhances some of the services by adding additional functionality or by combining two services. Another task of the HELIO Client API is to locally cache service results in order to improve the overall performance of the infrastructure.

## *About the HELIO Client API*

As stated in the introduction, the HELIO Client API is a downloadable Java library that hides the complexity of the HELIO infrastructure. It is designed for use by a broad set of

client applications. This section describes what types of client applications can be distinguished and what type of users we expect in our system.

## Client application types

The client applications of the HELIO Client Layer can be grouped into statically and dynamically bound clients.

- Statically bound clients are programs that are linked to the API at compile time. They know that a method has to be called in a certain way in order to return a certain result. Statically bound clients are not able to change the way they access the system at runtime. Typical representations of statically linked programs are graphical user interfaces like the HELIO Frontend or third party tools that programmatically access HELIO.
- Dynamically bound clients access the system through scripting languages such as the Interactive Data Language (IDL). In this case a human being will directly interact with the API. Users can adapt their way of working with the data depending on the response from a method call, e.g. if a query returns too many results the statically bound clients may not know how to restrict the number of results unless a developer has implemented this functionality at some place. The user of a dynamically bound client is able to find out how to perform this restriction by first consulting the API documentation and then adapting his or her query.

The HELIO Client API is designed for both types of usage by providing access to lots of metadata at runtime.

## User types

Within HELIO we distinguish three categories of end users: beginners, advanced and power users.

- Beginners will use the predefined functionality that is available through the HELIO Frontend. They will not be able to solve any tasks that go beyond this core functionality. As they do not use the HELIO Client or Server API directly they will not be covered any further in this document.
- Advanced users have specific tasks they want to solve in an interactive way. They do not want to learn the internal concepts of our system such as the data model. Therefore they need an API that tells them in an intuitive way how to use the system. Advanced users will access the system through an interactive data analysis environment. In HELIO we provide direct support for IDL SolarSoft. But the API should work in a similar fashion for different environments such as pure IDL, Matlab, Python, Groovy, etc. A high level introduction to this concept is given in the next chapter.
- Power users are willing to study our internal design and our data model. Therefore they will be able to execute sophisticated tasks and to perform queries in a real query language such as SQL or PQL. Developers that want to integrate HELIO in their system may be seen as a particular sub group of power users.

The HELIO Client API needs to serve two types of users, the advanced and the power users. Thus it needs to provide support for statically bound as well as dynamically bound clients. The definition of the API for statically bound clients is straightforward. Basically it consists

of a set of interfaces that reflects the underlying web services and is complemented by a set of client specific utilities. Support of dynamically bound clients is harder to be achieved. The following section shows provides an example based introduction to the API.

# How to Get Hold of the HELIO Client API

The HELIO Client API is released as JAR file.

The most current build can be found at http://helio-dev.cs.technik.fhnw.ch/jenkins/job/helio-clientapi/lastBuild/eu.heliovo$helio-clientapi/

Alternatively you can setup a Maven installation to pick it up from

```
<pluginRepository>
    <id>heliovo-snapshots</id>
    <name>Helio-VO Archiva Snapshots Mirror</name>
    <url>http://helio-dev.cs.technik.fhnw.ch/archiva/repository/snapshots</url>
</pluginRepository>
```

Specify the following Maven dependency in your pom.xml:

```
<dependency>
    <groupId>eu.heliovo</groupId>
    <artifactId>helio-clientapi</artifactId>
    <version>5.0-SNAPSHOT</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>
```

The source code can be found at http://helio-vo.svn.sourceforge.net/.

You may want to look into the JUnit tests and in particular into class http://helio-vo.svn.sourceforge.net/viewvc/helio-vo/trunk/helio-client/helio-clientapi/src/test/java/eu/heliovo/clientapi/HelioClientDemo.java?view=markup

# Programmatically Access the HELIO Client API

## *Class 'HelioClient'*

The HELIO Client API provides a single class to access all its functionality:

`eu.heliovo.clientapi.HelioClient`

Use the following Spring-code to get hold of the singleton instance of the HelioClient.

```
import eu.heliovo.clientapi.HelioClient;
import org.springframework.context.support.GenericXmlApplicationContext;
…
GenericXmlApplicationContext context = new
    GenericXmlApplicationContext("classpath:spring/clientapi-main.xml");
HelioClient helioClient = (HelioClient) context.getBean("helioClient");
```

This will initialize the system, which includes:

- Query the HELIO registry for registered services.
- Load catalogue descriptors from catalogue services such as HELO event catalogue (HEC).
- Load the list of available instruments from the ICS and DPAS.

### Automatic Caching

During the initialisation phase of the HelioClient all configuration data is cached in a local directory at: `${USER_TEMP_DIR}/.helio/${helio.appId}`. The property '`helio.appId`' is specified in a File called '`helio.properties`' which has to be located in the classpath root. The current default value is '`helio-dev`'. To overwrite this you can provide a File called `helio.properties` in your classpath and add property '`helio.appId = helio-boom'.`

If any of the remote resource is not available the system will try to use the locally cached correspondents. This improves the overall stability of the system. You may see several Warning messages at start-up, though.

Once the system is initialized you can start using HELIO. The HELIO Client API divides the HELIO services in different groups with slightly different access interfaces. These interfaces are further described here.

## *Get registered Services*

The following method allows listing all services that are currently available through the HELIO client API.

```
SortedSet<ServiceVariant> variants = helioClient.getRegisteredServiceVariants();
for (ServiceVariant serviceVariant : variants) {
    System.out.println(serviceVariant);
}

// sample output
// serviceName=HPS, serviceVariant=ivo://helio-vo.eu/hps/pm_cme_fw
```

In the following examples it should become clear what you could do with this information.

## *Metadata service (QueryService)*

In HELIO all metadata services implement the HELIO Query Interface (HQI). The HQI is mapped to the interface `eu.heliovo.clientapi.query.QueryService`.

### Conveniences methods

The following example shows how to access the ILS service through the convenience query-method.

```
// Get hold of a proxy to the query service.
QueryService service = (QueryService)helioClient.getServiceInstance(
        HelioServiceName.ILS);
// HelioServiceName is a dynamic enumeration of all registered HELIO services.

// Start the query in a background thread. The method terminates immediately.
HelioQueryResult result = service.query(
    "2009-01-01T00:00:00", 2009-01-02T00:00:00", // start and end time.
    "trajectories",                              // catalog name
    0, 0);                                       // maxRecords, startIndex

// The result object allows to get the result in different flavors
// All methods will block until the result is there or a timeout occurred.
URL url1 = result.asURL();                    // Result URL after default timeout
URL url2 = result.asURL(10, TimeUnit.Seconds); // Result URL after custom timeout
String str1 = result.asString();              // Plain string representation
```

```
VOTABLE votable = result.asVOTable();          // A VOTABLE object

// In addition you can query for the phase
Phase phase = result.getPhase();   // PENDING, EXECUTING, COMPLETED, ERROR, …
// and get the user logs which include logging messages from the called
// service, if available
java.util.logging.LogRecord[] userLogs = result.getUserLogs();
```

## Bean-style access

A more generic and flexible access method, which leads to the same result, is to use JavaBean properties.

```
// Get hold of a proxy to the query service.
QueryService service = (QueryService)helioClient.getServiceInstance(
        HelioServiceName.ILS);
// Use setters to define the properties.
// startTime, endTime and From are mandatory properties.
service.setStartTime(Arrays.asList("2009-01-01T00:00:00"));
service.setEndTime(Arrays.asList("2009-01-01T00:00:00"));
service.setFrom(Arrays.asList("trajectories"));

// now execute the query
HelioQueryResult result = service.execute();

// and do whatever you want with your result.
```

The JavaBean-like access has some advantages as dynamic clients can retrieve metadata about the values allowed for certain properties. This is achieved through the interface `eu.heliovo.clientapi.config.AnnotatedBean`, which provides the following method to get a BeanInfo instance.

```
public java.beans.BeanInfo getBeanInfo();
```

The following example shows how to further use the BeanInfo.

```
QueryService service = (QueryService)helioClient.getServiceInstance(
        HelioServiceName.ILS);
if (service instanceof AnnotatedBean) {
    BeanInfo beanInfo = ((AnnotatedBean)service).getBeanInfo();
    PropertyDescriptor[] propertyDescriptors = beanInfo.getPropertyDescriptors();
    for (PropertyDescriptor propDesc : propertyDescriptors) {
        System.out.println("Property: " + propDesc.getName() +
                           ", type: " + propDesc.getPropertyType());
        Collection<? extends DomainValueDescriptor<Object>> domain =
           getValueDomain(propDesc);
        if (domain != null) {
             System.out.println(" - " + domain);
        }
    }
}

private Collection<? extends DomainValueDescriptor<Object>>
    getValueDomain(PropertyDescriptor propDesc) {
   if (propDesc instanceof ConfigurablePropertyDescriptor<?>) {
```

```
        ConfigurablePropertyDescriptor<?> confPropDesc =
            (ConfigurablePropertyDescriptor<?>) propDesc;
        return confPropDesc.getValueDomain();
    }
    return null;
}
```

The output of the script above will look as follows. Please note the currently only the 'from'-property defines a value domain.

```
Property: from, type: interface java.util.List
 -   [[IlsCatalog: id=trajectories, label=Trajectories],
     [IlsCatalog: id=keyevents, label=Key Events],
     [IlsCatalog: id=obs_hbo, label=Observatory HBO]]
Property: startTime, type: interface java.util.List
Property: endTime, type: interface java.util.List
Property: where, type: class java.lang.String
Property: join, type: class java.lang.String
Property: maxRecords, type: class java.lang.Integer
Property: startIndex, type: class java.lang.Integer
```

### Sync vs. Async queries

The HQI allows running queries in a synchronous or asynchronous mode. In the sync mode the request is sent to the HQI server, the query is processed there and the result is wrapped into the SOAP result. In the async mode the request is sent to the server, which immediately returns a job ID. The ID can then be used to poll for the result and once it is there to retrieve an URL which points to the result. The Sync mode is usually much faster; the async is better suited for long running queries or for large result sets. Use the following bean property to switch between the two states

```
service.setQueryType(QueryType.ASYNC_QUERY);
// or
service.setQueryType(QueryType.SYNC_QUERY);
```

## *Processing Service*

All HELIO processing services implement the interface `eu.heliovo.clientapi.processing.ProcessingService`, which provides the method:

```
public ProcessingResult<T> execute() throws JobExecutionException;
```

Like the QueryServices the ProcessingServices are implemented as JavaBean. Thus a user can use JavaBean introspection to populate the properties. This works similar to the QueryService example above.

Alternatively most Processing services provide convenience methods to programmatically access them. An example is the CME forward propagation Model. The CME forward propagation model runs on the HELIO Processing Service (HPS) and uses a collection of IDL routines to compute the propagation of a CME through the heliosphere. It returns a VOTable with the arrival time of the CME at some points of interest such as planets or satellites and a collection of images to visualise the propagation.

```
// Get the service instance, here you need to tell the system which variant of a
// HPS service you want to use.

CmePropagationModel processingService = (CmePropagationModel)
    helioClient.getServiceInstance(HelioServiceName.HPS,
    CmePropagationModelImpl.SERVICE_VARIANT, null);
// Define input parameters
Date startTime = newUTCDate(2003, Calendar.JANUARY, 1, 0, 0, 0);
float longitude = 0;
float width = 45;
float speed = 100;
float speedError = 0;

// Execute the query
ProcessingResult<CmeProcessingResultObject> result =
    processingService.execute(startTime, longitude, width, speed, speedError);

// Use the result object to wait for the result.
// Internally the result object polls the HPS until the result is here.
CmeProcessingResultObject resultObject =
    result.asResultObject(60, TimeUnit.SECONDS);
// The result object is again a JavaBean with some convenience methods to get the
// result. Unlike the QueryService it does not allow to convert the results to
// different formats.
System.out.println(resultObject.getVoTableUrl());
System.out.println(resultObject.getInnerPlotUrl());
System.out.println(resultObject.getOuterPlotUrl());
```

Access to Taverna, to the DES plotting service and to the Context services work in the very same fashion. You will find examples for all of them in the Unit tests of the HELIO client API code.

# Bibliography

[1] Marco Soldati, André Csillaghy, David Guevara, and Hans-Peter Wyss, "Report on the implementation of the HELIO user interface," Technical Note 2011.

[2] Marco Soldati et al., "HELIO_FHNW_S1_005_TN_Architecture: HELIO Architecture," Technical Note 2012.

[3] Gabriele Pierantoni, "Placeholder for HPS doc," 2012.

[4] Robert D Bentley and Christian Jacquey, "HELIO-UCL-S2-002-RQ: Context Service," 2010.

[5] Robert D Bentley, "HELIO-UCL-N1-004-TN: Helio Concepts Document," 2010.

[6] Marco Soldati, "HELIO_FHNW_S5_006_TN_API_Developers_Guide: HELIO API Developers Guide," Technical Note 2012.

[7] Marco Soldati, "HELIO_FHNW_S5_008_UM_IDLAPI_User_Guide: HELIO IDL API Deverlopers Guide," FHNW, Windisch, Switzerland, Technical Note 2012.

[8] Kevin Besson, "Service Interface Specification," Technical Note 2012.

[9] Anja LeBlanc, "HELIO_UNIMAN_R1_004_TN_DataModel: HELIO Data Model," Technical Note 2012.

[10] Marco Soldati, "HELIO_FHNW_S1_003_TN_HMS: HELIO Monitor Developers Guide," FHNW, Switzerland, Technical Note 2012.

[11] Marco Soldati and Matthias Meyer, "HELIO_FHNW_S5_005_UM: IDL API User Manual," FHNW, Technial Note 2012.