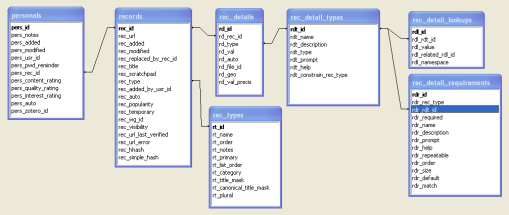
Programming Reference

# Application development tools

The main programming toolkit consists of the Heurist API (HAPI), the Geographic objects Interface (GOI) and Temporal Objects Interface (TOI), and these are documented for use by programmers who intend to carry out applciation development rather than modifications to the source code of Heurist.



**Development directions and tools**

In this section we also provide some general indications of development directions for Heurist and the and the related tools we have developed. However, we do not attempt to document internal structures, which are adequately documented within the code itself.

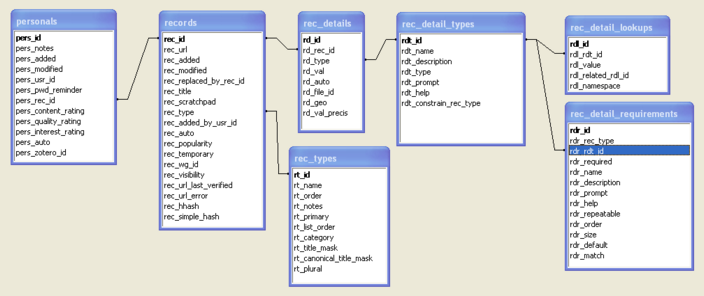
# Technical Overview

TODO: This will need some updating, but is basically a dry technical description which may be useful to keep here for the techeads who are lookign to understand what heursit does. It could ahve some extra diagrams.

Heurist is a system for storing descriptive information about, and connections between, a heterogeneous set of entities, and delivering this information on-the-fly in a variety of programmable formats including XML feeds, html, RDF, data exchange formats, interactive maps and timelines. A relationship browser framework allows multiple linked views (record view, linked list, map, timeline) to be navigated together.

The information stored for entities can include a dynamically defined range of textual attributes and pointer variables, geographic and temporal information, extended rich text, internal annotations and digital representations. Heurist also manages differential access based on user and workgroup profiles, personal bookmarks, personal, shared and workgroup notes, tagging and keywording, with social functions such as tag-following, collaborative rich text and blogging.

Heurist is implemented as a server-based MySQL/PHP/Javascript application. The [HAPI](http://heuristscholar.org/help/HAPI.html) (Heurist API, fully developed), GOI (Geographic Object Interface, fully developed) and TOI (Temporal Object Interface, in planning stage) programming APIs allow development of applications that use Heurist without needing direct access to the underlying data structures.



Heurist implements a number of original underlying data structuring methodologies. Entities are represented by a record in a primary database table. Attributes of entities are represented by records in an instance table linked by a foreign key to the primary table. Record types in the primary table and the metadata elements used to store attributes are defined in a series of lookup tables which define the use of both unique and shared attributes in different record types – definitions are thus soft-coded and can be modified without rebuilding the database.

User bookmarks are represented by records in a bookmarks table linked by a foreign key to the primary table.[Search](http://heuristscholar.org/help/Search1.html) parameters can be stored in a saved search table. Rich text segments are stored as binary objects which can be assembled into documents on the fly. Digital objects are stored as files on disk indexed by the database. Bibliographic relationships, as well as other relationships between primary records, are stored through internal pointers implemented as attributes, and internal pointers also allow embedding of other resource records within rich text fields.

All primary, bookmark, saved search and rich text segment records store the identity of their creator and workgroup ownership as foreign keys to the user and workgroup tables, allowing application of read and modification policies on the fly. Rich text segments in addition store specific read/write permissions down to individual user granularity. Users belong to one or more workgroups defined through workgroup lookup tables and also have the ability to define their own colleague groups.

Multiple Heurist instances can reference the record type and metadata definitions in the primary instance, while providing customised views of these definitions for a particular application or user group. Heurist stores creation and modification date of all main record types. The database stores a granular audit trail of all database changes using database triggers, allowing selective rollback of the data, although rollback routines have not yet been implemented.

Since the database structure of Heurist is relatively simple, the data management and permissions-controlled access methodologies are implemented through a complex backend implemented in MySQL and PHP. A Javascript API is provided for use by all custom applications which are built on a Heurist backend database. A general-purpose interface provides generic methods of defining, importing, editing, managing, searching and viewing data in Heurist.

Import methodologies include bookmarking of web pages, harvesting of hyperlinks from web pages, text file parsing, XML, GML and KML import including geographic objects and import of bibliographic exchange formats including heuristic record identification, disaggregation of hierarchical relations and creation of record relationships. Output methodologies include a method of generating comprehensive XML data dumps with controllable nesting of related records, XSLT processing for the generation of formatted output, RSS feeds and web services to specific applications.

Specific research has been carried out leading to a methodology for representing networks of relationships between entities, modelled using Heurist relationships and visualised through a relationship browser. [Relationships](http://heuristscholar.org/help/Relationships.html)are directional, record a relationship type based on an ontology, can be geographically and temporally located, and inherit all the other functionality of Heurist records (attributes, ownership, notes and annotation, tagging, searchability etc.). The relationship browser uses multiple linked views to visualise a record and its related records, and navigate all the views together. The relationship browser provides a record view (with attributes, rich text and embedded annotations), a linked list (with summary data including thumbnails and grouping by relationships), an interactive map (with spatial search and filtering, multiple map layers, symbols, connecting lines, labelling and rollover/popup information) and an interactive timeline (with time filtering, symbols, spans, labelling and rollovers/popups).

*Created with the Personal Edition of HelpNDoc:* [*Create iPhone web-based documentation*](http://www.helpndoc.com/feature-tour/iphone-website-generation)

# HAPI

The Heurist API is a Javascript library that provides full access to the data in Heurist.

[Documentation for HAPI](http://hapi.heuristscholar.org/) is available online for programmers interested in developing applications using the Heurist API.

# HAPI Developer's Guide

Note: for a full guide, visit the website:  [hapi.heuristscholar.org](file:///C:\Documents%20and%20Settings\ACL\My%20Documents\HelpNDoc\Output\Build%20word%20documentation\hapi.heuristscholar.org)

**Overview**

The Heurist Application Programming Interface (HAPI) is a Javascript library that provides access to the Heurist database, on which the Heurist web application is built.  Because of the flexible nature of the Heurist DB structure, it is able to support a wide range of applications.  HAPI makes development of such AJAX applications quick and easy.

If there is not an an example of how to do a certain thing in the Useful snippets section of this document, the API reference will most likely have what you are looking for.  The naming of functions and classes is clear enough that usually one can easily discover how to do something.

Hello, world

This is as near as one might come to a "Hello world" application - a basic search page.

    <html>

     <head>

      <title>HAPI Search sample application</title>

      <script src=http://hapi.heuristscholar.org/load?v=02&key=ce1c992c139cfd5a6fdc77c5c71444925f229942></script>

Loads the HAPI library.  The v parameter specifies the version.  If omitted it will default to the latest stable version.  If a value of d is given, the latest development version of HAPI will be loaded.  The key parameter is the API key for your site.

      <script>

        function doSearch() {

        var mysearch = new HSearch(document.getElementById("search-string").value);

            var loader = new HLoader(

                function(s,r) {

                    displayResults(r);

                },

                function(s,e) {

                    alert("load failed: " + e);

                });

            HeuristScholarDB.loadRecords(mysearch, loader);

        }

        function displayResults(r) {

            var div = document.getElementById("results");

            div.innerHTML = "";

            for (var i = 0; i < r.length; i++) {

                div.innerHTML += "<a href=\"http://heuristscholar.org/resource/" + r[i].getID() + "\">" + r[i].getTitle() + "</a><br/>";

           }

        }

      </script>

     </head>

The HSearch object specifies a Heurist search.  The syntax is the same as that used in the Heurist web application search.

An HLoader contains an onLoad callback and an onError callback.

HeuristScholarDB, an instance of an HStorageManager, provides access to the main instance of the Heurist DB.

     <body>

      <h3>Search Heurist database</h3>

      <input type="text" id="search-string"></input>

      <input type="button" value="search" onclick="doSearch();"></input>

      <div id="results"></div>

     </body>

    </html>

Click here to try this sample application.

Useful snippets

Create a new record

    var record = new HRecord();

Set record type

    record.setRecordType(HRecordTypeManager.getRecordTypeById(1));

Get detail types for record

    var detailTypes = HDetailManager.getDetailTypesForRecordType(record.getRecordType());

Add a detail to a record

    var TitleType = HDetailManager.getDetailTypeById(160);

    record.addDetail(TitleType, "HAPI test record");

Save a record

    var saver = new HSaver(

        function(r) { alert( ":-)" ); },

        function(r,e) { alert( ":-(" ); }

    );

    HeuristScholarDB.saveRecord(record, saver);

# Geographic Object Interface (GOI) for Heurist API

Note: for a full guide, visit the website:  [hapi.heuristscholar.org](file:///C:\Documents%20and%20Settings\ACL\My%20Documents\HelpNDoc\Output\Build%20word%20documentation\hapi.heuristscholar.org)

As in the Heurist API, all classes and objects in the GOI are accessible through the global namespace, and also the GOI namespace, and also the HAPI.GOI namespace.  For example, HFooBar is also accessible as GOI.FooBar and HAPI.GOI.FooBar; internally all names use the former naming scheme.

HPointValue, HBoundsValue,  HCircleValue, HPolygonValue and HPathValue objects represent bibliographic details of the geographic variety.

class HPointValue extends HGeographicValue

Constructor

    HPointValue( wkt-string )

        Create a new point value, using the given raw geographic data

    HPointValue( x, y )

        Create a new point value with the specified coordinates

Methods

    getX()  =>  float

    getY()  =>  float

        Return the coordinate values of the point

class HBoundsValue extends HGeographicValue

Constructor

    HBoundsValue( wkt-string )

        Create a new bounds (rectangle) value, using the given raw geographic data

    HBoundsValue( xMin, xMax, yMin, yMax )

        Create a new bounds (rectangle) value with the specified coordinates

Methods

    getX0()  =>  float

    getX1()  =>  float

    getY0()  =>  float

    getY1()  =>  float

        Return the coordinate values of the rectangle

class HCircleValue extends HGeographicValue

Constructor

    HCircleValue( wkt-string )

        Create a new circle value, using the given raw geographic data

    HCircleValue( x, y, radius )

        Create a new circle value with centre at (x,y) and given radius

Methods

    getX()  =>  float

    getY()  =>  float

        Return the coordinates of the centre of the circle

    getRadius()  =>  float

        Return the radius of the circle

class HPolygonValue extends HGeographicValue

Constructor

    HPolygonValue( wkt-string )

        Create a new polygon value, using the given raw geographic data

    HPolygonValue( points )

        Create a new polygon value - points is a list of { x: X, y: Y } objects.  The last point must be the same as the first point.

Methods

    getPoints()  =>  Array[ [x,y] ]

        Return the polygon's points.  Return value is an array of arrays of x,y values

class HPathValue extends HGeographicValue

Constructor

    HPathValue( wkt-string )

        Create a new path (poly-line) value, using the given raw geographic data

    HPathValue( points )

        Create a new path value - points is a list of { x: X, y: Y } objects

Methods

    getPoints()  =>  Array[ [x,y] ]

        Return the polygon's points.  Return value is an array of arrays of x,y values

class Digitiser

Constructor

    Digitiser( map )

Methods

    digitisePoint()

    digitiseBounds()

    digitiseCircle()

    digitisePolygon()

    digitisePath()

        These functions start the digitisation process on the given Google Map.  Once invoked, the user double-clicks the map to start digitising.

    getShape()  =>  HGeographicValue

        Return the currently digitised shape, or null if there is no shape being digitised.

object google

This object provides convenience methods for creating Google Maps objects from HGeographicValue objects.

Methods

    getLatLng( value )  =>  GLatLng  throws HInvalidGeographicValueException

        Returns a new GLatLng object representing the given HGeographicValue.

        Behaviour for different types is as follows:

            HPointValue:  does the obvious thing.

            HBoundsValue: uses the bottom-left corner.

            HCircleValue: uses the centre point.

            HPolygonValue / HPathValue: uses the first point.

    getLatLngs( value )  =>  Array[GLatLng]  throws HInvalidGeographicValueException

        Returns an array of new GLatLng objects representing the given HGeographicValue.

        Note: for HCircleValues, 40 points around the circumference of the circle will be generated.

    makeMarker( pointValue, opts ? )  => GMarker  throws HInvalidGeographicValueException

        Returns new GMarker object representing the given HPointValue.

        The opts argument is as specified at http://code.google.com/apis/maps/documentation/reference.html#GMarkerOptions

    makePolyline( pathValue, color ?, weight ?, opacity ?, opts ? )  => GPolyline  throws HInvalidGeographicValueException

        Returns new GPolyline object representing the given HPathValue.

        The color, weight and opacity arguments are as specified at http://code.google.com/apis/maps/documentation/reference.html#GPolyline

        The opts argument is as specified at http://code.google.com/apis/maps/documentation/reference.html#GPolylineOptions

    makePolygon( value, strokeColor ?, strokeWeight ?, strokeOpacity ?, fillColor ?, fillOpacity ?, opts ? )  => GPolygon  throws HInvalidGeographicValueException

        Returns new GPolygon object representing the given HGeographicValue.

        The strokeColor, strokeWeight, strokeOpacity, fillColor and fillOpacity arguments are as specified at

        http://code.google.com/apis/maps/documentation/reference.html#GPolygon

        The opts argument is as specified at http://code.google.com/apis/maps/documentation/reference.html#GPolygonOptions

    getBounds( overlays, pad ? )  => GLatLngBounds

        Returns a new GLatLngBounds object that contains all the given overlays (GMarkers, GPolylines and GPolygons).

        If the optional pad argument is true, the bounds will be given some extra padding in all directions.

# T1000 Database

The T1000 database wizard permits the building of stable, well-structured, easily modified web database application in minutes without any knowledge of database programming.

Databases generated with the T1000 wizard use portable, open-source software (MySQL and PHP) and automatically support passworded access, validated input, file upload, coordinates and web mapping, free text searching, pre-defined lookups, one-to-many and many-to-many relationships, delivery of data as a web service, RSS feeds - all without any extra work.

The databases can be run on the Heurist server, or be downloaded and copied to another server.  T1000 provides the core functionality of Heurist and all theother online database applications.

More information: [T1000 Wiki, installation wizard and documentation](http://heuristscholar.org/tmwiki/index.php/T1000)

# TimeMap Java Webmap

## SourceForge

Heurist has been developed to interoperate with TimeMap (another tool developed by ACL).

[TimeMap](http://sourceforge.net/projects/timemap/) is an open source mapping/animation application and it is available on SourceForge.

# Database definition templates

**Specification developed by Ian Johnson 18 Jan 2011**

When a new database is set up, it is automatically configured with a set of required Record Types, Detail Types (Fields), Ontologies, Vocabularies, Terms and Constraints. This set is initially created from populateBlankDB.sql but it may be better to download it from the Heurist Reference database using getDBStructure.php and buildCrosswalks.php (this does however mean databases can only be created when online). In any case, buildCrosswalks will be used against the Reference and other Heurist databases to acquire additional definitions.

**Reserved** definitions are set by xxx\_Status = Reserved. Reserved definitions are those that MUST exist in the database, primarily bibliographic data types. Reserved record types can be hidden in the interface by setting rty\_ShowInPulldowns = 0. Reserved record types are a very limited set and can only be set by the Heurist Reference server. Unlike other definitions, Reserved codes are invariant and have specific meaning eg. 160 = Title; reserved codes all lie within a specific range starting at 1 (1 - 1000 for record type etc.).

**Approved** record types are the next level down. These are typically record types which have been refined and are 'authoritative' within a particular community. They can be set on any server, implying that the approval mechanism is internal to a community (ideally of many people, but potentially one person can set themselves up as a 'community'). Approved records will typically have a number of fields, vocabularies etc. associated with them which are marked as xxx\_MayModify = 'Locked' or 'Discouraged'.

Whenever record types are downloaded, Heurist also downloads dependant data in defRecStructures, defDetailTypes, defOntologies, defVocabularies and defTerms (maybe also defRelationshipConstraints).

When a definition is downloaded from another database, xxx\_OriginatingDB, xxx\_NameInOriginatingDB and xxx\_IDInOriginatingDB are set. The Name is effectively a code identifying the concept in the original database (although the ID is more inherently stable) and is also the label by which the concept is identified within the interface or stylesheets of that database. It is important that this name should NOT be changed for any databases which are likely to be used as a source for definitions in other databases.

At a later time we may provide multilingual translation tables to render alternative labels for names in the definition talbes.

If the definition is then borrowed along a chain, the originating database information is NOT changed - that is, definitions are always identified back to their original source and code. However we should have a mechanism to reset the originating database information back to zero so that the current database becomes the origin for a new definition (presumably heavily modified from the original). This provides a simple copying mechanism rather than obliging people to manually create new record types and other definitions.

**Open** record types, and their dependant definitions, are freely editable.

When a sysadmin consults another Heurist database to download definitions:

1. Heurist finds the database through the Heurist Index database, so the source database MUST be registered;
2. The Reserved and Approved definitions are displayed by default (if the database is the Heurist Reference database, the Reserved definitions are downloaded automatically if not already in the target database).

**Downloading XSLT**

When an XSLT is downloaded from a (or the) reference server, the names used are immediately translated from the names used in the source to the names used in the target, allowing the file to be modified using local terminology. The process is reversible, in that the original source can then download the XSLT and in the process it will be translated back into the source servers names.

<<we need a methodology of coping with definitions which are purely local or came from another source>>

# Enum values, targets, relationship constraints

**Enum values, pointer targets and relationship constraints**

Heurist implements a fairly complex system of record pointers and relationship types. Relationship types are set through an enumerated values field which may use terms from one or more vocabularies. This page describes the mechanisms behind enumerated values, relationships and relationship constraints.

**Ontologies, vocabularies and terms**

Terms (defTerms) belong to a vocabulary (defVocabularies) which in turn belong to an ontology (defOntologies). Each of these are identified internally by immutable codes, which allows editing of the names/labels eg. for localisation purposes. At the present time (Jan 2011) we make little use of the ontologies table.

**Enumerated values**

Enumerated value details (fields) are defined in the defDetailTypes table as dty\_Type = enum. Ordinary enum details are used for classifications such as raw material, gender, condition or country (Ordinal and Nominal variables). Relationship types, althoguh defined as an enumerated detail, are handled rather differently (see below).

The set of vocabularies and terms which can be used with an enumerated detail type may vary according to the context of use (record type). So, for example, the detail type Techniques might use one set of vocabularies when applied to Buildings and a different set of vocabularies when applied to Paintings. The applicable vocabularies (and terms within them) are defined by a table:

***defEnumVocabs [ env\_ID, env\_DetailTypeID, env\_RecTypeID, env\_VocabID, env\_VocabSubset ]***

For any one detail type/record type combination, multiple records in this table can define multiple vocabularies. These will be displayed as a single list of terms with headings for each vocabulary. env\_VocabSubset defines the specific tems to be dispalyed through a comma-separated list of term IDs chosen from within the specified vocabulary; any term IDs not in the vocabulary will be ignored.

**Relationship type enumerated values**

Relationships between records are implemented as a standard Heurist record of type Relationship. By default, any record type can be related with any other record type using any of the standard relationship vocabulary terms supplied with Heurist (vocabulary 200 <check>).

The most important details in the relationship record are:

***Source Record*** *(pointer to a record ID)*

***Target Record*** *(pointer to a record ID)*

***Relationship type*** *(enumerated value, detail code 200, defaulting to vocabulary 200)*

Since we wish to be able to constrain relationship records to particular sets of record types (for instance, Person might be related to Person or to Organisation but not to Artefact or Species) as well as defining specific allowable relationship types (for instance a Person can be the child of another Person, but not of an Organisation or Building), we have developed a more complex mechanism than the defEnumVocabs mechanism described above, which allows us to define the behaviour of relationship records.

This is implemented through a table:

***defRelationshipConstraints [rcs\_ID, rcs\_SourceRectypeID, rcs\_TargetRectypeID, rcs\_DetailtypeID, rcs\_VocabID, rcs\_VocabSubSet, ...]***

This table defines an additive set of constraints on relationships. So for example (200 is the code for the relationship type detail):

1 | 47 | 36 | 200 | 5 | 345, 234, 367, 823

1 | 47 | 36 | 200 | 6 | 172, 123

1 | 47 | 36 | 200 | 8 | 1187, 1188, 1189

means that record types 47 and 36 can be related using relationship types in vocabularies 5, 6 and 8 (and the specific terms listed after the vocabulary - 345, 234, etc.). Furthermore:

* 1. record types 47 and 36 can ONLY be related with these relationship types;
  2. if no other record specifies a source record type 47, record type 47 can ONLY be related to record type 36 (the existence of a specific relationship constraint overrides the default any-with-any relationship capability for the source record type).

The relationship constraints table can also use the 'wildcards' Null or 0 for the source and target record types and the detail type, in which case the constraint applies to all values, in the absence of any more specific constraint which override the general ones. << I am having trouble deciding exactly where this is useful/usable >>

**Relationship Markers**

Record relationships are normally entered via the Relationships tab on the data entry form (they can also be set for multiple records through the Relate action on the search interface). However, one often wishes to tightly specify the required set of relationships in the data entry interface and embed them at a particular point in the data editing form (eg. parents before siblings and after details relating to the person's name and address).

For instance, a Person record might require the following relationships:

* + School (relationships to an Organisation, 1 (required) relationship, valid terms: attendsSchool)
  + Siblings (relationships to a Person, 0 - n relationships, valid terms: hasBrother, hasSister, has HalfBrother, etc.)
  + Parents (relationships to a Person, 2 values, different relationship types, valid terms: hasFather, hasMother - each must be present)

In order to embed these fields at specific points in the record editing form, to label them and to set these rather precise requirements, we have a special relationship marker detail type which can be used like any other detail type (field) in constructing the record editing form. Note that these marker details are not used in the recDetails table to store any data; they simply trigger the display of specific relationship records at particular points on the data entry screen.

When relationship marker details are used to add a relationship, the relationship record is created as normal in the database with relationship type detail 200; there is no way of telling that the relationship record was created through a relationship marker rather than through the Relationships tab. When the record is reopened in the record editing form, the form identifies the existing relationships and positions them according to the relationship markers which they satisfy; any additional relationships are displayed on the Relationships tab.

Relationship marker details use the defRelationshipConstraints table to define the source and target record types and the vocabularies/terms available. This means that the same relationship marker may offer different terms and/or requirements between different record type pairs. For example, a relationship marker detail in the Unpublished Report editing form might allow the creation of relationship records between the report and either a Person (with relationship types isAuthor, isEditor, isCompiler) or an Organisation (with relationship types isCorporateAuthor, isCommissioner).

The relationship marker may be 'multi value' (that is, multiple relationships can be entered through it), with one and only one author, editor or compiler (Person) required, and an optional Organisation relationship. In practice, to avoid confusion, these relationships would probably be handled by two separate relationship markers.