Micello WorkingSet Data Framework

TODO

- The service definitions include the service name. I can do this if I start using annotations to define the service rather than webconfig entries.
- I don't have a way of specifying a snapshot version or a single snapshot
- need a reload config service
- I should put some threading protection on reloading the config file. Now there can be a problem if the config is reloaded and another request happens.
- I set null values by sending null values into a prepared statement. I should investigate using setNull().

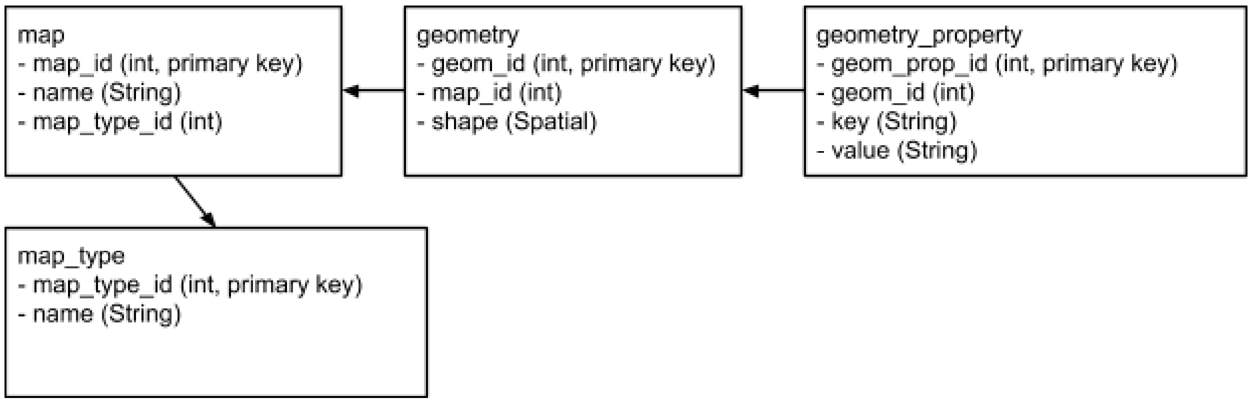
Overview

The WorkingSet data framework is a database access and versioning framework designed for the scenario where clients work with regular working sets of data from a relational database.

What is a Working Set?

A working set is a collection of database records defined by a database search together with a collection of database records cascaded from the search result records.

We will use this simple map database as an example to illustrate a working set. This database consists of several small maps, such as maps of parks.



An example of a working set with this database is the data for a single map with id 221, which is data that would be retrieved by someone wanting to view or edit the map. This corresponds to the given data.

- Main Records - Records the are in the principle DB search

    - "map" record with map_id = 221
- Cascaded Records - Records that are cascaded from other records in the data set.
    - all "geometry" objects which have the map_id returned in the main search (221)
    - all "geometry_property" records which have a geom_id returned with the collection.
    - the "map_type" record which is referenced from the "map" object in the main search.

Another example of a data set is the list of map types.
- Main Records - Records the are in the principle DB search
    - all "map_type" records

Note that the two working sets given are overlapping. An arbitrary number of overlapping working sets is supported by the framework.

WorkingSet Framework Features

Database Configuration

The WorkingSet framework allows you to:
- Easily configuration of your database tables and generate of Java objects of each table.
- Easily define working sets of data. There are two types of working sets:
    - Versioned - These working sets will be versioned. Versioning is discussed below.
    - Non-versioned - These working sets are used to simplify requests and have no versioning.

Client Requests

A client can request a pre-defined working set of data or construct a working set on the fly.

Snapshots and Versioning

It is possible to make a snapshot for any versioned working set. The snapshot is stored as a copy of the working set, as a file in the same format that is received from a client request. There is a version number the accompanies the snapshot. This is intended as a versioned release process.

Client Commits

A client can submit changes to the database, called a *commit.* A commit consists of a list of *actions*. Each action contains a number of *instructions*. A single instruction is an operation on a single database record.

Each commit is logged on the server. Additionally, the versioned working sets that are modified by the commit are also logged and marked as dirty. This dirty flag can be used to see when a new snapshot is needed.

Also, the previous commits can be looked up. These can also be used to rollback changes to individual working sets rather than the entire database.

During a commit, a check is made for database collisions. An error is returned if one user tries to update data that has been updated by another user since the first user downloaded the data.

Security

The framework contains authorization system that allows authorization at the record level, meaning individual records can have permission set on them, allowing separate read and edit access.

Client Library

The WorkingSet framework includes a client side java library to accompany the server side code for easier data access. The client side has error checking so illegal database operations can be detected and automatically cleaned up preventing data corruption.

The client side code uses the same instruction formalism as the database allowing it to have an "undo/redo" feature built into it

Implementing a WorkingSet Project

To implement the WorkingSet server, the libraries listed below must be included in a java web service project. The servlets in the library should be exposed through the service. A context parameter must also be added pointing to a configuration file for the service.

These are the needed libraries:

- MiceDataLib - Use for all applications
- MiceIloUtil - Needed for all applications
- MiceDataServerLib - Used for the server applications
- MiceDataClientLib - Used for client applications
- Additional JAR files:
    - Base64.jar
    - mysql-connector-java-5.1.10-bin.jar
    - json.jar

The server should also have a context parameter specifying the file path for the config file.

- Context parameter name - "config_file"
- Context parameter value - config file location

The config file is a json file that should have the following fields:

Config file fields:

- datasource_url - the url for access to a conection pool for the database
- auth_server - The url for the authentication server. See notes on this below.
- release_base_path - The file path where the snapshots are stored
- commit_base_path - The file path where the commit data is stored.
- search_map_file - A json file that specifies the working set definitions
- data_manager_factory_class - The class name for the factory to create the data manager. This is used to link to the implementation data model.

Here is an example config file:

```
{
        "datasource_url":"jdbc/MiceMaps41DS",
        "auth_server":"http://localhost:8080/TestAuthorization",
        "release_base_path":"/opt/snapshot",
        "data_manager_factory_class":"com.micello.miceditor.data.MiceditorDataManagerFactory",
        "search_map_file":"SearchDefFile.json",
        "commit_base_path":"/opt/commit"
}
```

In addition an Authentication service must be defined. See the section on the authentication server.

Database Setup

Database Definition

The database should be a MYSQL database with a connection pool set up for the server.

The database contains a few standard tables:

- active_snapshot - holds the active snapshot for each versioned working set
- commit - holds a reference to the commit data
- commit_release - holds the versioned working sets modified by a given commit.
- snapshot - holds a reference to the snapshot data

Additional tables are added for the desired data model. Each table should have the following format:

- *ID_FIELD* (int or bigint) - This is the primary key for the table. It can be set to auto increment or not. The name can be set arbitrarily.
- *DATA_FIELDS* - These are arbitrary fields defined by the data model
- read_perm (int) - This is a permission value for reading this record.
- edit_perm (int) - This is a permission value for writing this record.

See the section on authorization for more information on permissions.

Setting Up Data Objects

To initialize the database, a series of POJ data objects are created with the following format.

Database - DataManager Object

The DataManager object is used to represent the complete data set. It is needed for initialization. the DataManager defines a name and a version for the database. Clients accessing the service must list the proper version. The DataManager must include a DataSetInfo object.

DataSetInfo - This defines the data set

- Database Name - "EntityStaging"
- Database Version - "4.1.2"

The following code is an example DataManager definition.

```
public class EntityStagingDataManager extends DataManager {

        //Create a DataSetInfo object with the dataset name
        //and version.
```

```
	public final static DataSetInfo DB_INFO = new
		DataSetInfo("EntityStaging","4.1.2");

	//Pass the Data setInfo object into the base class
	//in the constructor.
	public EntityStagingDataManager() {
		super(DB_INFO);
	}

	//the remaining code is added because before the dataset is
	//used each object must be referenced so the data defintions
	//are properly made. There needs to be a better way of
	//making sure this initialization happens.
	public static boolean initialized = false;
	public final static Object lock = new Object();

	/** this method explicitly references each data object so
	 * the static initializers of each is called. */
	public static void initializeDataObjectInteractions() {
		synchronized(lock) {
			if(initialized) return;

			//call everything to make sure it is loaded
			RecordInfo ri;
			ri = DataCollection.COLLECTION_RECORD;
			ri = EntityAddress.ENTITY_ADDRESS_RECORD;

			initialized = true;
		}
	}

}
```

Tables - DbObjects

DbObjects are POJ objects used to represent the individual tables (or records) in the data model. The java object is defined with a RecordInfo object to link the object to a table and a FieldInfo object to link properties to the database columns. In addition, ForeignReferenceList fields can also be defined. These are lists of objects in remote tables that reference a record in this table.

Defining the database using java objects provides convenient data objects when using the java library for server or client software while making it easy to define and update the database configuration.

Once the links to the tables and field are defined, the getters for the fields are automatically generated.

RecordInfo - This links the record to a database table. It includes the following fields:

- Table name

- Reference to DataSetInfo object holding this record
- Java class for this object
- Name of the ID column for the table

FieldInfo - These define the fields in the table
- Column name
- Data type for the column
- Null OK flag
- Default value for the column

ForeignReferenceList - This is a list of remote references to the local record:
- name
- data type
- isUnique flag - If the object is a unique reference
- sort by field - results in records in list being sorted by this field
- restrict delete - If this is true the record can not be deleted if there are objects that reference it. This defaults to true.

Notes:
- The foreign reference lists are not returned with an individual record. However, they are automatically constructed if fields are cascaded in the data set. The foreign reference list must be defined to define a cascade.
- If the sort flag is used, the sort field must appear in the remote record BEFORE the foreign reference field or the sort will not work properly.
- I believe there are cases in the client library where the local sort is not updated on editing. Check this out.

The code below shows an example definition of a DbObject.

```
public class EntityAddress extends DbObject<EntityAddress> {

       //=======================
       // RecordInfo
       //=======================

       //A RecordInfo object must be defined for the table
       public final static RecordInfo<EntityAddress>
              ENTITY_ADDRESS_RECORD = new RecordInfo<EntityAddress>(
              "entity_address", //table name
              EntityStagingDataManager.DB_INFO, //DatasetInfoObject for record
              EntityAddress.class, //class type for this object
              "entity_address_id" //name of id column
       );

       //=======================
       // FieldInfo
       //=======================
```

```java
//For each column a FieldInfo object is defined
//This field is a foreign key reference to another DbObject
//called DataCollection. The DB value is an integer but the
//field class here isa DataCollection object.
public final static FieldInfo<DataCollection,EntityAddress>
       COLLECTION_FIELD = new FieldInfo<DataCollection,EntityAddress>(
       "collection_id", //column name
       ENTITY_ADDRESS_RECORD, //RecordInfoObject for field
       DataCollection.class, //java class for field value
       false,//null ok - false
       null //default value
);

//Another FieldInfo object
//This one is a Long data type.
public final static FieldInfo<Long,EntityAddress>
       ENTITY_FIELD = new FieldInfo<Long,EntityAddress>(
       "collection_time", //column name
       ENTITY_ADDRESS_RECORD, //RecordInfoObject for field
       Long.class, //java class for field value
       false, //null ok - false
       null //default value
);

public final static FieldInfo<String,EntityAddress>
       ADDRESS_KEY_FIELD = new FieldInfo<String,EntityAddress>(
       "address_key",
       ENTITY_ADDRESS_RECORD,
       String.class,
       true, //null ok - true
       null
);


public final static FieldInfo<String,EntityAddress>
       ADDRESS_VALUE_FIELD = new FieldInfo<String,EntityAddress>(
       "test_count",
       ENTITY_ADDRESS_RECORD,
       Integer.class,
       false,
       4 //default value for table set here
);

//=======================
// ForeignReferenceList
//=======================

//It is also possible to add a field that gives a list of objects that
//reference this record using a foreign reference list.
//When the accessor is used for this field, a List object is returned.
//For cascades defined in the WorkingSet, a ForeignReferenceList must exist.

//This is a foreign reference list, of objects referencing this
//record. This is the basic version of the FieldInfo definition.
public final static ForeignReferenceList<OtherTableClass,EntityAddress>
       ADDRESS_LIST = new ForeignReferenceList<OtherTableClass,EntityAddress>(
                "addresses", //a name by which to reference this list
                ENTITY_ADDRESS_RECORD, //The RecordInfoObject
                OtherTableClass.EntityAddress //the fieldInfo in the remote object
        );

        //This is a foreign reference list with an alternate constructor
        //with additional options, for database relation constraints.
        //These constraints are checked in code, particularly on the
        //client side, to see if an operation is legal.
        // - unique - allows unique constraint on db relation
        // - order by - allows list ot be ordered
        // - restrict delete - set to true if db restricts delete
        public final static ForeignReferenceList<EAProp,EntityAddress> EA_PROP_LIST = new
        ForeignReferenceList<GeometryProp,EntityAddress>(
                "ea_prop", //a name by which to reference this list
                ENTITY_ADDRESS_RECORD, //The RecordInfoObject
                EAProp.GEOMETRY_FIELD, //the fieldInfo in the remote object
                false, //is unique (true means one entry in list)
                GeometryProp.ORDER_FIELD, //if present, ordering field for list
                true //restrict delete if object in db referenced
        );



        //The constructor should pas the record object to the super
        //class
        public EntityAddress() {
                super(ENTITY_ADDRESS_RECORD);
        }

        //There is a getter that is automatically created for each
        //field. See below.

        //There are no setters. To set a field when using for example
        //on a client, an instruction obejct must be
        //created to change the value of a field. Example code
        //demonstrates this.

        //Optionally, for convenience, the automatic getter can be used
        //to create a more simple getter not requiring the field
        // argument.
        public String getAddressKey() {
                return getField(ADDRESS_KEY_FIELD);
        }
}
```

The following code is the format of a getter for the DbObject. The static FieldInfo object instance is passed into retreive the field. (Note that the

```
        /**
         * Returns the field specified by the FieldInfo object.
         *
         * @param <TF>    The type of the field object.
         * @param field   The FieldInfo object that specifies the field.
         * @return        The field object
         */
```

```
public <TF> TF getField(FieldInfo<TF,TR> field);
```

The sample code demonstrates examples of setting up the database.

Working Sets

Working sets are groups of data that are used together. They are defined in a configuration file. A single working set is defined by a database search. It consists of a main record search and a cascade of additional records.

Main Record(s)

There are three ways to define the main records for a search:

- Table - A table is specified. The entire table is returned.
- Record - A table and ID is specified. The record with that ID from that table is returned.
- Field - A table, a field and a value are specified. All records with that value in that field is returned. In the configuration file there are two ways to specify the field value:
    - As a fixed value
    - As a variable value, to be determined at search time

Cascade Records

A cascade of records can be defined for the search. This is a list of records linked, directly or indirectly, to the main records returned for the search.

Here is an example from the Miceditor database. The main search may be a specific community. The cascade might be all drawing records that are in that community. Also the cascade may include all levels that are in the drawings that are in the community.

There are two types of cascades.

- Local Reference Cascade - In this cascade a field from table A is specified. The field must correspond to a reference (the ID) to another table, B, in the database. In this case, when a record from table A is loaded in the search, the references record from table B is also loaded. This is true whether A is in the main search or one of the cascade records.
- Remote Reference Cascade - In this cascade a foreign reference field from table A is specified, referring to records in table B. Then when a record from table A is loaded, all records in table B referencing the loaded record in table A are also loaded.

Working Set Configuration File

The working set configuration file specifies a list of searches. Each one is given a name along with the main search and the cascade fields. Additionally, each search has a flag specifying if this search should be used for a snapshot.

These named searches can then be used in the API to retrieve the desired working set of data.

Also if the snapshot flag is set it is possible to take a snapshot of the working set and a version number is given to it. The snapshot is in the same format as the data request. It is stored on the server and is available by name and version number.

For any search that has a snapshot enabled flag, the service tracks when any commit of data modifies the return result for that search. When it does, a dirty flag is set in a table that identifies the snapshot. This can be used to check if a new snapshot should be taken.

Search Definition File Fields:

- "searchType" - This has the following options:
    - "table" - return all records from a table
    - "record" - return a specific record, by ID
    - "fixed_field" - return all records with the specified value in the specified field
    - "field" - same as fixed field except the field value is specified at search time
- "isSnapshot" - If this value is true the search is a versioned search
- "search" - This object defines the search and is different for the different search types.
    - Table search:
        - "record" - Theh table to return
    - Record search:
        - "record" - The table being searched
        - "id" - This will be set (overwritten) at search time
    - Fixed Field search
        - "record" - The table being searched
        - "field" - The specified field
        - *field_name* - The value for the field. The key *field_name* should be the name of the field.
    - Field search - This is the same as a fixed field search except the field value will be overwritten at search time.
- "cascade" - This is an array of cascaded fields. The name can be either a name from the FieldInfo object (a column name) or the name from a ForeignReferenceList object. Each entry in the array contains the following data:
    - "record" - the record which holds the field
    - "field" - the name of the field or foreign reference to cascade

The following is an example working set configuration file:

```
{
        "collection":{
                "searchType":"record",
                "isSnapshot":false,
                "search":{
                        "record":"collection",
                        "id":0
                },
                "cascade":[
                        {"record":"collection", "field":"addresses"},
                ]
        },
        "community":{
                "searchType":"field",
                "isSnapshot":true,
                "search":{
                        "record":"collection",
                        "field":"community_id",
                        "community_id":"0"
                },
                "cascade":[
                        {"record":"collection", "field":"addresses"}
                ]
        },
        "community_collection":{
                "searchType":"field",
                "isSnapshot":true,
                "search":{
                        "record":"collection",
                        "field":"community_id",
                        "community_id":"0"
                },
                "cascade":[
                ]
        }
}
```

Server API

The WorkingSet framework has the following standard requests for access to the data. It is easy to include additional custom services in the web service.

The following are the standard services:

- **Request** - This is a search where the search definition, similar to what appears in the working set configuration file, appears in the request body. The return value is the search result.
- **CannedSearch** - This returns the working set of data associated with a named search from the configuration file. In the case of a record search or a variable field search a value must also be passed to specify the search.
- **Commit** - This request contains data to be committed to the database. The commit is logged and any snapshots that are affected by the commit are flagged as dirty.
- **SnapshotRelease** - This request creates a snapshot. It stores the results of a CannedSearch (which is marked as snapshot true) in the snapshot repository, assigning it a version number.
- **ReleaseInfo** - This service looks up information on a given release.
- **SnapshotLookup** - This returns a snapshot by name and version. If no version is specified the latest version is returned.
- **CommitList** - This returns a list of commit objects (including metadata but not the commit data) for a commit. This can in one of two ways:
    - user - This returns the commits for a given user
    - release - This returns the commits for a given working set.
- **CommitLookup** - This returns the body of a commit based on the commit ID.
- **CommitUndoRedo** - This method will undo or redo a commit, as specified by commit ID. The commit does not have to be the most recent commit. However, a check is made to make sure the commit does not violate the integrity of the database. If it does the undo/redo will fail.
- **ReloadConfig** - This method reloads the config file for the service.
- **SchemaLookup** - This returns a JSON containing the specifying info on the data set, thre records in the data set and the fields and foreign reference objects defined for the records.
- **SearchDefinitions** - This returns the search definition map.

Each of these services is called using HTTP and a standard WorkingSet protocol on top of that.

The WorkingSet framework requires an Authentication service which is specified in the Authentication section.

WorkingSet Protocol

The web service requests are in HTTP format and include HTTP header values and a JSON body.

HTTP Request Header Fields

The request header contains the following values:

- "proto-version" - The string version of the protocol. Currently "4.0"
- "db-version" - The string version of the database schema. Currently "4.0"
- "comp" - Compression parameters, bitwise flags. Indicates if the request body and response body are zipped.
  - request compressed flag = 0x01
  - response compressed flag = 0x02
- "sessionKey" - A string that denotes user session. This identifies the user for authentication and the database and service level. Use the authentication service to get a session key.

Additional fields may be required for particular services.

HTTP Request Body

The request body is a JSON object whose format depends on the particular service.

HTTP Response Headers

The response header contains the following values:

- "comp" - A flag indicating if the response body id compressed. The flga used to indicate the response is compressed is the same flag used for response compressed in the request.

HTTP Response Body

The response body is a JSON object with the following format:

```
{
      "success": boolean success value,
      "data":response json data, if applicable, on success,
      "errorCod":integer error code, on failure,
      "msg":text message, on failure
}
```

Response Error Codes

```
public final static int ERROR_CODE_UNAUTHORIZED = -1;
public final static int ERROR_CODE_SERVER_ERROR = -2;
public final static int ERROR_CODE_SESSION_EXPIRED = 0;
```

Service Request and Response Formats

Request Service

`http://[`*`host`*`]/[`*`service name`*`]/request`

This is a database search where the search definition is defined in the search body, rather than using a search defined in the working set configuration file. The search can be one of three types:

- Table Search - All records from a given table are returned
- ID Search - A record from a table with a given ID is returned
- Field Search - Any record from a table with a given field value is returned.

These results are returned in two fields. The records resulting from the primary search are returned in a list with the the name "Response". The cascaded fields are returned in a list named "Records".

Request Format

The request has the following format:

```
Request:
{
	"search":search object,
	"cascade":array of cascaded objects
}

Search Types:
Table Search:
{
	"record":search table name,
}
ID Search:
{
	"record":search table name,
	"id":id value
}
Value Search:
{
	"record":search table name,
	"field":search field name,
	search field name:search field value,
}
Cascade Entry:
{
	"record":cascade table,
	"field":cascade field
}
```

Response Format

The request response has the following as the data entry in the response:

```
{
        "Time":java timestamp,
        "Response":array of primary records,
        "Records":array of cascaded records
}
```

An individual record has the following format:

```
{
        "Type":tableName,
        "Id":record ID,
        "read_perm":read permission integer,
        "edit_perm":edit permission integer,
        "edit_ok":boolean, user has edit permission
        "Fields":{
                fieldName:fieldValue,
                ...
        }
}
```

Canned Search Service

`http://[host]/[service name]/cannedsearch`

The canned search service is similar to the request service except the search is not defined in the body of the request. Rather, a named search defined in the working set configuration file is used. Additional parameters may be needed to fully specify the search.

The following fields are included in the body:

- "search" - This should be the name of the search as defined in the working set definition file.
- Additional fields are needed for some search types:
    - Record Search
        - "id" - this specifies the ID value for the desired record.
    - Non-fixed Field Search
        - *field_name* - The key for this value is the name of the field. The value is the desired value that should be matched in the search.

The response is identical to the response from the Request Service.

Example JSON

Example JSON for loading a named record search "community" with ID = 327.

```
{
        "search":"community",
        "id":327
}
```

Commit Service

`http://[host]/[service name]/commit`

A commit is how data on the server is updated. It consists of three components. The main object is the History, which is a list of all the work done for the commit. The history contains a list of Actions, which is a single logical database change. An action contains a list of instructions, which are a single row operation on the database.

The return for the commit message is a mapping of any temporary IDs passed in the commit to the resulting actual IDs.

History

```
{
	"Name":"Micello", //the data set
	"Message": text message, //a human readable commit message
	"Actions":[array of actions]
}
```

Action

```
{
	"Title": text message, //brief description of the action
	"Instr":[array of instructions]
}
```

Instruction

```
{
	"Type": "Create" | "Update" | "Delete" //the type of instruction
	"Record":table name, //the table to act on
	"Id":record ID, //the record id or, for a create, a temp id

	//initial avlues in db, valid for delete or update
	//for delete, all fields should be set
	//for update, all fields that are changed should be included
	"initialReadPerm":permission integer
	"initialEditPerm":permission integer
	"Initial":{
		//field mapping for target values
		field name:target value,
		...
	}

	//fields to be set, valid for create or update
	//for create, all fields to be written should be set
	//for update, all fields that are changed should be included
	"targetReadPerm":permission integer
```

```
        "targetEditPerm":permission integer
        "Target":{
                //field mapping for target values
                field name:target value,
                ...
        }
}
```

Instruction types

There are three instruction types:

- Create - Creates a record. In a create record, a temporary ID should be assigned to the object. This should be a negative value that is unique to the commit. The database will create a real ID for the record when it is inserted in the database. The returner data from the commit will be a mapping of the negative temporary IDs to the actual IDs for the created records. The create record should include all fields that the database will write. If a field is not included the default value will be applied, if applicable. The read and edit permissions must be set.
- Update - Updates a record. It should include initial and final values for all fields that are being updated. Read and edit permissions should be included only if they are being updated.
- Delete - Deletes a record. It should include all initial fields, including the read and edit permissions.

Sample Commit Message

This is the body to submit to create an entity record with two property records, name and phone.

```
{
         "Name":"Micello",
        "Message":"Test commit",
        "Actions":[
                {
                         "Title":"Create entity",
                        "Instr":[
                                {
                                        "Type":"Create",
                                        "Record":"entity",
                                        "Id":-1,
                                        "targetReadPerm":1,
                                        "targetEditPerm":2,
                                        "Target":{
                                                 "community_id":1,
                                                "flags":0
                                        }
                                },
                                {
                                         "Type":"Create",
                                        "Record":"entity_prop",
                                        "Id":-2,
                                        "targetReadPerm":1,
                                        "targetEditPerm":2,
                                        "Target":{
                                                  "entity_id":-1,
                                                 "key_name":"name",
                                                 "key_value":"Test Entity",
                                                 "lang":"en"
                                        }
                               },
                              {
                                        "Type":"Create",
                                        "Record":"entity_prop",
                                        "Id":-3,
                                         "targetReadPerm":1,
                                        "targetEditPerm":2,
                                         "Target":{
                                                  "entity_id":-1,
                                                 "key_name":"phone",
                                                 "key_value":"8675309"
                                        }
                               }
                     ]
          }
     ]
}
```

SnapshotRelease

`http://[host]/[service name]/snapshotrelease`

The snapshot release request triggers the release of a new snapshot. The snapshot file created is identical in format to the data returned by a CannedSearch request. A release number is associated with the snapshot and it is stored in the snapshot directory. It can re retrieve with a call to snapshot lookup.

Just as the snapshot file is identical to the return from CannedSearch, the body of the request is also of the same format.

ReleaseInfo

`http://[host]/[service name]/releaseinfo`

The release info request is used to request the latest snapshot for a release and whether or not it is dirty. It also gives the commit time for the latest commit that modified this release.

The request has the following format:

```
{
          {
                    "search": search name,
                    "id": record id //used only for record search snapshot
```

```
        field name: field value //used only for field search
    }
}
```

The response has the following format:

```
{
    "releaseName": release name,
    "releaseId": release id
    "version": latest version number,
    "dirty": true/false,
    "timestamp":java timestamp (units of msec, from Unix time start date)
    "inactive": true/false,
}
```

SnapshotLookup

`http://[host]/[service name]/snapshotlookup`

The snapshot lookup request is used to request a snapshot. The snapshot is in the same format as the canned search file. The snapshot lookup response however is different from the canned search response in that the snapshot lookup request looks up and returns a list of snapshots. Likewise the snapshot request has a list of search entries rather than a single search entry.

The request has the following format, for a list of snapshots:

```
{
    "list":[
        {
            "search": search name,
            "id": record id //used only for record search snapshot
            field name: field value //used only for field search
            "version": snapshot version //optional - if none specified, latest is returned
        },
        … //one entry for each desired snapshot
    ]
}
```

The request has the following format for a single snapshot request:

```
{
    "search": search name,
    "id": record id //used only for record search snapshot
    field name: field value //used only for field search
    "version": snapshot version //optional - if none specified, latest is returned
}
```

There should be a way to request a single snapshot and to specify the specific version desired.

CommitList

`http://[host]/[service name]/commitlist`

This request retrieves a list of the commits associated either with a specific username of a specific release name.

Request Body

Username request:

```
{
      "user":username
}
```

Release request

```
{
      "release":release name
}
```

Response Body

Username Response:

```
{
      "user":username,
      "commits":[
            {
                  "commit_id":commit_id,
                  "releases":[
                        release name,...
                  ]
            },
            ….
      ]
}
```

CommitLookup

`http://[host]/[service name]/commitlookup`

The commit lookup request returns a commit file. The commit file returned is in the same format as a commit request body.

The commit request is in the format:

{

"commit_id": *commit id*
}

CommitUndoRedo

`http://[host]/[service name]/commitundoredo`

A commit undo/redo request undoes or re-does a commit. The undo/redo should be successful if there have been no successive commits for the affected snapshots. However, it is possiblee to try to undo/redo any commit. The undo/redo will fail if there is a collision with other changes to the data.

The request has the following format:
{
"commit_id": *commit id,*
"redo": *redo flag //OPTIONAL - defaults to false, or undo*
}

An alternate way of doing an undo or redo is to explicitly request the commit object of interest and construct a new commit.

When a commit is undone the version number of the snapshot does not decrease. Any new snapshot version is strictly increasing.

ReloadConfig

`http://[host]/[service name]/reloadconfig`

This method reloads the config file. This method does not follow the standard protocol format. The URL can simply be typed in the a browser to reload the config.

SchemaLookup

`http://[host]/[service name]/schemalookup`

This method returns the schema information in the form of a JSON as the data payload of a response. The request takes no arguments but requires the standard headers.

This JSON has the following format:
- DataSetInfo
  - name - The name of the dataset.
  - version - The version of the data set.
  - records - A JSON map of the records (RecordInfo) in the data set.
- RecordInfo
  - name - The name of the record.

    - idField - The ID field name for the record.
    - recordClass - The java class corresponding to this record.
    - fields - A JSON array of fields (FieldInfo) in this record.
    - foreignList - A JSON array of foreign refernces (ForeignReferenceList) in this record.
- FieldInfo
    - name - The name of this field.
    - fieldClass - The java class for this field. If the field is a record in the data set, the class will be the corresponding record class but the value received in the download will be a long corresponding to the id of the record.
    - nullOk - Tells if null values are OK.
    - defaultValue - The default value for the field, if not null.
- ForeignReferenceList
    - name - The name of the foreign reference list.
    - remoteRecord - The remote record that refers to the local record..
    - remoteField - The remote record field that refers to the local record.
    - orderField - The field used to order the list.
    - isUnique - If true there is a unique constraint on this list.
    - restrictDelete - If true this record can not be deleted if there are any remote records corresponding to this foreign reference list.

SearchDefinitions

```
http://[host]/[service name]/searchdefinitions
```

This method returns the search definition JSON file as the data payload of a response. The request takes no arguments but requires the standard headers.

Authentication

The WorkingSet server requires access to an Authentication service of the format given below.

There are three types of authorization:

- Service authorization - The gives yes/no authorization for a service name. It si required to call a service.
- Table authorization - This gives yes/no authorization for a table name. It is required to create or delete or entry from a table or to modify the permission on a record in that table.
- Permission authorization - This gives yes/no authorization for an integer value. For any record, read and edit permission are stored as an integer value, which is checked in this call. This permission integer is used in the following ways:
    - Update or Delete Record - To update or delete a record, the user must have authorization for the initial value of the record permission.
    - Modify Permission - To modify a read or edit permission, the user must have authorization for the target permission.
    - Foreign Reference List - Some records are defined foreign reference lists for another record, such as the geometry in a given level. If a user modifies such a foreign reference list, the user must have authorization for the modified record (both the initial and target values) in addition to having permission for the record which contains the foreign reference list.

The Authentication Service has the following service calls:

- Login - This logs in a user, creating a session.
- VerifyPermission - This is used to verify a permission.
- VerifyService - This is used to verify a service.
- VerifyTable - This is user to verify a table.
- BulkVerify - This is used to verify a list of permissions, services and tables.

Request Format

Login

```
[auth server url]/login
```

A user must login to create a session. Once this is done, the session is active for a fixed amount of time. When the session expires, the user must login again. The login returns a session key which is used as a token to identify the user's session.

Request Header Parameters

- "userName" - user name
- "password" - password

Response Header Parameters

- "responseCode"
    - 1 = success
    - -1 = unauthorized
    - -2 = internal error
- "sessionKey" - the token for the session, on success
- "message" - error message, on an internal error

Verify Permission

`[auth server url]/verifypermission`

This service verifies an integer permission, used to control access on individual records in the database.

Request Header Parameters

- "permission" - permission integer
- "sessionKey" - session key string

Response Header Parameters

- "responseCode"
    - 1 = success
    - -1 = unauthorized
    - -2 = internal error
- "message" - error message, on an internal error

Verify Table

`[auth server url]/verifytable`

This service verifies access to a table.

Request Header Parameters

- "table" - table name
- "sessionKey" - session key string

Response Header Parameters

- "responseCode"
    - 1 = success
    - -1 = unauthorized
    - -2 = internal error
- "message" - error message, on an internal error

Verify Service

`[auth server url]/verifyservice`

Request Header Parameters

- "service" - service name
- "sessionKey" - session key string

Response Header Parameters

- "responseCode"
    - 1 = success
    - -1 = unauthorized
    - -2 = internal error
- "message" - error message, on an internal error

Bulk Verify

`[auth server url]/bulkverify`

This service verifies a list of permissions, tables and services. There is an option to quit on failure or to verify all requested values.

Request Header Parameters

- "sessionKey" - session key string

Request Body

```
{
	"services":[
		service name,...
	],
	"terminateServices":terminate flag, //OPTIONAL defaults to true
	"tables":[
		table name,...
	],
	"terminateTables":terminate flag, //OPTIONAL defaults to true
	"permissions":[
		permission integer,...
	],
	"terminatePermissions":terminate flag, //OPTIONAL defaults to true
}
```

Response Header Parameters

- "responseCode"
    - 1 = success
    - -1 = unauthorized
    - -2 = internal error
- "message" - error message, on an internal error

Response Body

```
{
	"services":{
		service name : is ok (boolean),
		...
	},
	"tables":{
		table name : is ok (boolean),
		...
	},
	"permissions":{
		permission integer(as string) : is ok (boolean),
		...
	}
}
```