Software project ideas

Contact: Stephen M. Richard, Arizona Geological Survey, U.S. Geoscience Information Network
steve.richard@azgs.az.gov,
520-869-8545 cel 520-209-4127 office

1. Catalog scanning stuff

These components are all part of a framework that revolves around accessing a catalog service, getting metadata records for resources, parsing them to look for various kinds of links and taking some action with the links. USGIN is currently using the OGC Catalog Service for the Web (CSW) for catalog services, and the USGIN ISO metadata profile for metadata content and XML encoding conventions (based on ISO 19115 and ISO19139).
CSW spec (not easy reading….) http://portal.opengeospatial.org/files/?artifact_id=20555
Tutorial for CSW http://www.ogcnetwork.net/node/630
USGIN ISO profiles: http://usgin.github.io/usginspecs/USGIN_ISO_Metadata.htm
A CSW catalog instance: http://catalog.usgin.org/geoportal/csw?request=GetCapabilities&service=CSW

The NGDS data-archiver is a project that uses this approach to 'scrape' a catalog, creating a directory containing copies of all the file-based resources listed in the catalog. It's a JavaScript application, built to run under Node.js. Because of all the http GETs that are happening, running as an asynchronous process is the approach taken. Various throttling tricks had to be introduced to keep from overloading the server or getting behind on disk writes.
https://github.com/ngds/data-archiver

This app is run from command line, and configuration stuff like where the downloaded files get stored is not obvious. A user interface to make it easier to use would be the first thing to do.

URL checker:

This is a simple subset of the more generic Web service monitor (below) that tests Web Locators (URLs) for simple HTTP document retrieval. Parse a given document, find URLs, run GET (or HEAD) on each URL to see if it returns a 200. Log created in database (could be just a text file) contains these fields: document ID, URL (decompose into host and path), HTTP response codes, error messages (if present), time of request, and response time on each request. Configuration: 1. Document type to parse (csv; JSON; XML attribute values, element values); 2. time-out time on GET request, log time outs as errors as well. Reporting: reporting API resource is a list of log items. Provide filter by http response code, request time (start, end, ISO8601 simple), URL, DocumentID, response time, URL, and host. Consider using HTTP HEAD instead of GET to save bandwidth.

Web service monitor (heartbeat) and reporting

scan catalog for service-based distribution (use protocol on CI_OnlineResource or ServiceType), ping services with a simple request to see if they are live. Report metadata recordID, URL (decompose host and path), response code, error message (see URL checker above). Modularize in two pieces:

1) access the catalog through a service (CSW, openSearch, OAI-PMH etc.), gets metadata records. Configuration for catalog type and endpoint location. (different modules for different catalog service types?)
2) Given a metadata record, parse to locate service distributions (e.g. CI_Online resource in ISO metadata) that are service affordances, and test the service. This requires modularization for
    a. How to identify service affordances in the metadata. E.g. regular expressions looking for 'service=WMS' in a URL, or testing gmd:protocol = 'OGC:WMS' in ISO metadata…
    b. How to test a given service. E.g. for OGC services, a GetCapabilities requires, for OpenSearch request the openSearch description document, for OpenDAP request ??, custom RESTful services might have various conventions for a self-description document or basic ping test (e.g. DataOne API)

Repository dump

Builds on URL checker and web service checker. Component to scrape a catalog and archive all listed resources. The component has been built in JavaScript by Adrian Sonnenschien for USGIN system scraping (https://github.com/ngds/data-archiver) – it needs a UI to make it simpler to use, and the archive objects should be loaded into one of the already specified data packages. The program is asynchronous, node-js based.
The process is

1. scan catalog

2. find resources based on filter criteria
3. get each resource
4. package it in data package container that includes 1. Manifest; 2. Metadata; 3. The resource object(s).
5. save package in file directory. Structure of directory needs to be considered. Adrian's app creates a directory for each resource host end point. Various other approaches to directory structure might be considered. Currently we write to a local directory and then have a separate process to push that to a cloud (Amazon) archive. Might provide option to write packages to any web location directly. I think the problem we ran into is Amazon's write rate couldn't keep up with the rate we were trying to push packages.

For background on datapackages, see Bagit (http://tools.ietf.org/html/draft-kunze-bagit-10), OKFN data package (http://data.okfn.org/doc/data-package), MaxOgden data package (https://github.com/maxogden/datapackage-json), also OASIS OpenDocument format (http://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2.html)

2. Edit and registry components

Linked data registry.

Deploy a generic registry component e.g. the UK Gov't Linked data registry project (https://github.com/UKGovLD); also have UN FAO VocBench (http://aims.fao.org/vest-registry/tools/vocbench (vocabulary focused, but it's the same problem); SISSVoc on Sesame (https://www.seegrid.csiro.au/wiki/Siss/SISSVoc); Drupal Semantic web modules for SKOS; probably lots of other implementations (REST API on a SparQL endpoint). Part of the project is evaluating the options and picking a good one—collecting requirements, use cases, testing the various available software. Has to do content negotiation; different representations could be cached in a document type database (xml blobs, json blobs, various rdf encoding blobs), or the tool could do on the fly transformation from the internal storage representation to the response representation. This would be a core component of USGIN (and hopefully EarthCube), not a single central registry but a network of registries. The centralized component would be the registry of registries…

Use to implement the NGDC metadata components (https://geo-ide.noaa.gov/wiki/index.php?title=Docucomp_Component_Management_System). Should also be able to implement a generic URI redirect as well—the representation is a 303 redirect to another URI; redirect has to pass through all the header parameters in the request. See https://github.com/usgin/uriredirect.

Content editor:

A configurable editor that works something like the Protégé UI or other self-configuring form. Ideally this editor would be used in the Strabo system for structural geology field data (https://github.com/StraboSpot/strabo-mobile), but the concept is generic. An information exchange specifies a content model and interchange format for the content model, and optionally, a service protocol for requesting data using the specified interchange format. Given an information exchange URI and the URI for a resource that conforms to the exchange specification.

1) Read the information exchange specification (representations in xml, JSON, etc. should be accessible by dereferencing the URI with the correct MIME type, see registry above, e.g. http://schemas.usgin.org/contentmodel/activefault.xml). The specification specifies an information model (OWL, XML schema, JSON schema, see e.g. http://schemas.usgin.org/models, https://github.com/usgin/usginmodels, https://github.com/usgin/modelmanager), and an interchange format. The current information model content might be enhanced (phase 2) with information that associates information elements with UI widgets, pick lists, constraints, etc. to enable spinning up a UI. For now make simple mapping to text fields. Work on the model manager is a registry issue—separate project.
2) Spin up a browser-based UI using the information exchange representation.
3) Load data item identified in initial request (no item provided means make a new one; nice feature would be to have templates that could be specified to use for new records—if template is provided, make new record based on the template)
4) User edits record
5) Can save result locally, or with authentication, push the update or new record to a target host, along with provenance information (who did edit, version of content editor used, when, change log note optional from user).

Start with simple UI widgets like text boxes, single pick combo box, and extend functionality from there. Combo boxes would use the 'Combo box list from registry' component described below. Start by hard configuring the list, then dynamically populate from registry with fixed source URI, then read URI for list from UI information in Information Exchange spec.

Combo box list from registry

Ajax widget that will requests an item collection from a registry (GET collection by URI), gets a list of labels and identifiers, presents user with picklist showing labels. When user picks, GET the representation for that item using its URI (from the Linked data registry, below…). Content negotiation for representation of collection and representation of items. Configuration of widget requires registry endpoint, MIME types for collection and item representations to request. This widget would be used in user interfaces for metadata editor or the generic feature editor.

Use components in content editor

Extend ‘content editor’ above with ‘Combo box list from registry’ ‘ using the URI associated with a pick list item to get a content component (e.g. NGDC metadata components https://geo-ide.noaa.gov/wiki/index.php?title=Docucomp_Component_Management_System) and populate appropriate other fields in the generated output from the form. Simplest case is when component is represented in the same info model/encoding scheme as the form output so component items map directly to the form output (e.g. a gmd:CI_ResponsibleParty component fed to a form that is generating an ISO19139 XML metadata output).