Healthcare Starter Kit Cookbook

The Healthcare Starter Kit (HSK) is a self-contained project that both illustrates how to build a Data Hub, Lake or Fabric using healthcare data, and accelerates the job by providing models, templates, libraries, configurations, and techniques.

This “cookbook” is organized by task so that the existing project can be altered or extended in the typical ways needed to build out a healthcare data hub. The data and tasks focus on the Healthcare Payer domain, with particular emphasis on public-sector, Medicaid processing, but the project is useful in commercial and even non-healthcare contexts.

# Feature areas

Before covering how to extend the HSK, we first briefly review the features of the HSK. Most of the features included in the HSK have a cookbook entry explaining how to extend or modify the feature.

* Data model for Claims, Locations, Providers, members
  + Based on FHIR
    - One FHIR resource = one Entity Model
  + Flattened and simplified
    - Link to philosophy blog
* Sample mappings from MITRE generated CSV data
  + Not all sample data is mapped; more for reference or convenience
  + Illustrations of mappings to typical healthcare structures, loops, nested data
  + Joins during mappings to pull together many CSV rows for one persistent Entity
* Transform librarry to convert persistent model (ES-FHIR) to true FHIR using minimal config data
  + Uses configurations rather than code
* MDM rule set for Patient (member) records
  + Conservative to avoid linking wrong records
  + Notifications capture close matches for metadata or later merging
  + Includes custom step to set up given- and family-name dictionaries for double-metaphone match
* Mapping functions and techniques
  + Illustrate mostly CSV mappings
  + Sample custom functions to join csv records into larger records (function looks up related info)
  + Shows use of range indexes to allow fast joins during mapping (avoid prop-val-q)
  + NOTE: should use .xqy for speed
  + Reusable utility classes
    - valueSetLookup: parse and process FHIR codes from (**url**) and use for lookup
  + 1:M and M:1 mapping approaches (as noted elsewhere with joins and multiple steps per file)
* Ontology-based query
  + Using a claim TDE to produce a row so the claim data and SNOMED data can be queried as a unit via Optic
* Unit testing
  + Using the ml-unit-test library (gradle include as an mlBundle)
    - See claimSearchSuite for ontology-driven broader/narrower queries
    - See claimSuite for tests that invoke data hub steps (mapping) in memory to verify output
  + Java tests that verify the fully processed sample data is mapped and mastered properly
* Data Services
  + An claim search service with ability to query using ontology terms to enhance query
  + See FHIR-Mapper for insight into how to build FHIR-oriented data services
* DevOps
  + Gradle patterns to run flows for both ingest and curation
  + Pattern to set up smaller data set to allow faster reload/reprocessing if needed
  + Use of “net.saliman” plugin to use environment specific overrides as needed
* Security
  + Security configuration to restrict access to PII data
  + Security configuration also uses reference data to restrict access to certain diagnoses and employee’s health records
  + Sample users (DrSmith, MrJones, MrNoPII) who are granted certain roles
* De-identification / redaction configuration
  + A set of rules to remove PII from Member (aka Patient) and Claim data
  + An export configuration to produce a redacted data set using the rules

# Extension and Customization Tasks

Below are specific tasks with instructions on how to do each. The instructions are intended to cover all steps needed, but we do not plan to actively “test” each extension task on every version of the HSK, so there may be slight gaps and a general knowledge of MarkLogic Server and MarkLogic Data Hub are still required to understand the overall process.

## Adding additional data sources

The HSK ships with sample data including CSV formatted data from the MITRE Synthea project. This sample data is included to allow the HSK to be run and used as a sample application right away. Obviously, real projects will have to use real production data instead.

Typically, developers will add a small data set to the /data directory and build an Ingest Step to load the data, a target Entity to represent the canonical model (for at least some data fields; not all of the data must be canonicalized) and then a Mapping step to convert the raw data to the Entity format.

### Adding new raw data in batches

To load new data via batch, place the sample data in the /data directory (by convention; this location is not required) and create an Ingest step to load it. The Data Hub will natively handle JSON, XML or delimited (e.g. CSV) files. CSV will be re-formatted as JSON by using the column headings as JSON properties. See: <https://docs.marklogic.com/datahub/5.6/flows/about-steps.html> for Ingest step details. (5.6 is the latest version at this writing. Please check for new versions.)

Consider what data you need to actually use, to be agile about data mapping and ELT/ETL processing. That is, model, map, master, clean or otherwise process data you see a need for, which may not be the entire incoming record. See “Extending data models…” below to update the Entity model to hold new data from the new source that is not yet processed.

Once the new fields in the model are added, build a new Mapping Step that converts the raw data to the desired Entity Services model. First, load some data so the data mapping GUI will have a record to drive interactive mappings via the “test” button.

## Extending data models to hold new data

If new data is needed in an API, export or other functionality, the data is often best modeled in Entity Services as a canonical model. Data can then be mapped into the new Entity model.

This is not unlike modeling in any other tool. Entities and sub-entities are created, with various fields and types. The difference is that data can be left un-modeled by either

1. Leaving it in the STAGING database with a link to the FINAL data captured in a StagingURI: or similar Entity property
2. Putting non-modeled data in the $attachments property of an Entity. This property is intended to hold raw, un-modeled data.

Using $attachments pushes complexity and work from the Mapping Step to later query or data services. Because data complexity is not addressed or canonicalized during mapping, it must be dealt with at query time.

## Altering mastering rules

### Matching rules

Mastering rules are often driven by use cases. For instance a “Twins” use case can be captured in a spreadsheet or a text document of business policy where two Person records with the same last name, DOB, and address, both lacking a SSN, will **not match** because they may be biological twins. A “moved locally” use case may capture that two Person records with the same last name, first names similar, different addresses, different (wired) phone numbers, same DOB, same city and state, **are a match**.

To update rules, consider the desired change in these pair-wise matches and non-matches, and adjust rules to ensure the proper sets of matching and non-matching fields do what is expected. The Matching Step GUI will help with this. However, if you are using post-step processing, or if the match rules GUI gets too crowded due to a high numbers of rules, consider using a spreadsheet with one row per use case, and spreadsheet formulas to add points for various match situations, and comparing the overall score to the score threshold.

MarkLogic Consulting has sample spreadsheets and can help with this process.

See: <https://docs.marklogic.com/datahub/5.6/flows/about-mastering.html> for details on configuring Matching steps.

Merging rules

As documented at <https://docs.marklogic.com/datahub/5.6/flows/about-mastering.html> per-field merge rules can be configured to combine documents that are identified as a match.

In addition, a post-step interceptor, or if need be a separate step, can be written to update merged records and ensure more sophisticated rules are applied. In particular, rules involving multiple fields may be implemented in code this way. E.g. if a record has all of “givenName,” “familyName,” and “fullName” where fullName is intended to be the concatenation of the other two, you may need to build fullName after the field-specific rules have computed the givenName and familyName fields. To force ordering in this way, a later step to do a few fix ups is recommended.

## Creating new redaction rule sets for de-identification

The HSK includes custom “redaction” rules useful for Medicaid and healthcare processing. To de-identify data further, add new rules to the src/main/ml-schemas/redaction area of the project. This “rules” area holds configurations only, but references custom code found in src/main/ml-modules/root/lib/redaction/redactionFunctions.sjs as well as built-in standard redaction functions.

Update collections.properties and permissions.properties to ensure the user (system login) that will invoke redaction will “see” and be able to read the redaction rules. Note these rules are stored in the Schemas database, as documented.

Some custom functions are included in HSK are:

* redactMappedValue()
  + This allows a pre-computed mapping to be used to replace real values. It expects a range index (a form of lexicon) to be set up with values of the form oldvalue:newvalue. This allows complex mappings to be pre-computed.
  + Example: all addressesLine1 values can be computed using a list of known, valid addresses. These address mappings would be stored in JSON documents including a field of the form “123 Main St:99 Elm Ave” which is range indexed. This indicates that the real records with “123 Main st” as AddressLine1 should consistently get “99 Elm Ave” as the new (redacted) AddressLine1.
* redactDate() and related functions (deterministic)
  + Allows a date to be changed within a specified envelope of time, such as +/- 100 days
* redactStreetAddress() (deterministic, dictionary-based)
  + Uses a standard MarkLogic redaction dictionary to redact the number and street name of an address, by applying some basic parsing to the address string.
* redactUUID(), redactText(), redactReference(), redactZipCode()
  + These and other functions use the “cipher” class to provide fast, deterministic and non-colliding redaction. That is, UUIDs, IDs and so on will be altered in a way that won’t be easily reversible, but also will not clash.
  + Particularly useful for IDs where multiple records are linked or must use the same IDs, and pure randomization would break that link in the redacted output. E.g. if a Patient and Claim record both reference a Patient ID to indicate a link (the Claim is for the Patient) we want to have the IDs still be the same in the redacted output.
  + This is rather secure, however not as secure as totally-random, pre-computed IDs that are then referenced during redaction by redactMappedValue() above.

This last group are all based on the Ciper utility class in HSK (see: src/main/ml-modules/root/lib/redaction/redactionUtils.sjs). The Cipher class can be subclassed to use different sets of character. It will only alter characters in the group, allowing the dashes in an SSN or UUID to be left alone, for instance. By respecting the sets of characters, it also ensures that a hex UUID (using only letters A-F) keeps in that range.

The Cipher works by “advancing” each character forward in the set of characters in a predictable way. So the 9th character might always move 20 positions forward in the array. If you think about it, this allows the result to be random, but avoids collisions. Collisions would make a redaction algorithm inappropriate for use on IDs or join keys.

See <https://docs.marklogic.com/guide/app-dev/redaction> for redaction information.

## Data Mapping

### Joining input data into one persistent record (many to one mapping)

When input data is normalized (split into many records as in typical relational modeling) you should join it into a more “denormalized” model in MarkLogic. MarkLogic’s ideal record size for JSON or XML is about 10-100K, and individual row-based documents are typically much smaller. Per-record overhead will reduce your maximum data size. Systems with billions of tiny row records have notably higher costs than systems that join records into fewer, larger records.

To join data, you can use a few approaches

First, it is probably best and ideal to join data in the pre-step interceptor of a Mapping step. This provides the joined record as a “raw” record in the Mapping GUI (note DHF 5.6.0. is the first version to show the record after pre-step interceptor processing).

For additional performance, you may want to make the Step a “batch” step by setting the acceptsBatch Step configuration property to “true.” The pre-step interceptor can now gather the join keys (linking IDs) from an entire batch of 10 or 100 or perhaps 1,000 records and find all matching records with one query. (vs per-record query which is simpler)

The HSK includes sample configuration and functions such as **claimGetLines()** (see: src/main/ml-modules/root/custom-modules/mapping-functions/claimGetLines.sjs) to do such joins. This function is a custom mapping function which is not ideal in that you cannot see the claim lines in the mapping GUI, but it is a simple way to get additional, related data. This function runs as a mapping expression in the Mapping Step for the Claim entity, to allow the final Claim record to include claim line info even though the raw input document is the master Claim row, and the ClaimLine rows require run-time fetching.

Again, it is better to optimize this work in the pre-step interceptor, but this approach is still useful and illustrative.

### Splitting input records: creating many persistent records (one to many mapping)

The Data Hub Central tool includes a 1:M mapping capability for when a larger input record comes in as raw (STAGING) data and multiple output records are needed.

*(Note: this is for cases where a single conceptual incoming business entity maps to a few related canonical business entities. It is not for cases where a list or table of incoming data is mapped to a list of canonical, persistent records. You should have already split your incoming data using other tools, such as mlcp, Glue, NiFi, Kafka, or the data hub loading functions)*

See: “Mapping to Multiple Entities” here : <https://docs.marklogic.com/datahub/5.6/flows/about-mapping.html>. This is limited to situations where the model captures entity relations, and you are mapping a single input record (typically a larger JSON or XML record) into multiple related outputs.

## Data Access

Once data is in the MarkLogic Server database, there are many ways to access it. A core strength of MarkLogic is to process and secure data once, but expose it in many ways. Data access approaches include Data Services from Java or Node.js, REST endpoints, bulk exports, HTTP calls out for notifications alerts and messaging, SQL via ODBC, and SPARQL via the 1.1 endpoint specification.

### Data Services

The newest and most optimized access pattern is to create a Data Service and access it via Java or Node.js. Data Services stand up servlet-like listeners (on certain app server ports) in MarkLogic and also generate Java and JavaScript proxy classes that allow access to the data from those languages.

Data Services can run arbitrary logic and transforms on the data to provide it in any desired format.

### Exports

### FHIR-compliant data services and FHIR server integration

Particularly note that FHIR data can be served easily. The data modeling pattern used by HSK is FHIR based, and a FHIR transform is provided. See “Connecting to a compliant FHIR server” below for details.

## Using Ontologies for Subsumation Searching

MarkLogic includes RDF and ontology query and processing, making it easier to use complex reference data, including taxonomies and ontologies, to enhance search and query operations. This “ontology-based query expansion” uses the ontologies to find other terms to query on, based on an input query.

To do this for health data, we have provided ontologies for ICD-10-CM and ICD-10-PCS because those are freely available without license and describe procedure code taxonomies. If you are using ICD-10 codes in your data you can simply use the loadIcd10Ontologies task available in the gradle build file to get that reference data into the database (enabling use as below).

If you are using SNOMED-CT then you will need to download the ontology yourself as it requires a license for use and distribution. Once downloaded you will need to run the ZIP file through the snomed-owl-toolkit and then run the resulting ontology-<time-run>.owl through ROBOT in order to transform the data into a format that will be understood by the MarkLogic Database for ingestion. Once transformed place the ingestable file in src/main/ml-data/ontologies/SNOMED-CT.ttl and you will be able to run loadSnomedCTOntology (or if you want to use all 3 ontologies you can use loadOntologies).

If you are using SNOMED-CT and would like to use the standardized queries that we have provided in this project you will also need to run the provided task that adds SKOS Core Notation triples to the SNOMED-CT ontology. This is currently set up as part of the import task but if you do not want to use this process you can separate the two steps and add the SKOS Core Notations separately by running the normalizeSnomedCTOntology task.

Keep in mind that if you use a different data model you will need to tweak your code to work with it.

### Claims search example with ontology-enhanced querying

Once reference ontolog(ies) are loaded (as above), you can use services such as the provided claim search at src/main/ml-modules/root/data-services/claim/search.sjs to find data using the ontologies. This Data Service queries for claims using diagnosis and procedure codes and expands these codes by “joining” to a reference ontology.

This service joins two (row-structured) views. The first view is the claims view, which is a TDE-constructed tabular view for Claims records. The TDE template that projects this table structure from Claims JSON records is at: src/main/ml-schemas/claimExtract.tdej.

The other view is a row-result from a SPARQL query which contains only the broader or narrower concepts from the relevant ontology. That is, all super-concepts or sub-concepts for the query term are returned in a small row-set that is then joined to the claims schema view.

This results in a “join” among all relevant procedures or diagnoses, and all Claims that have them.

# Connecting to a compliant FHIR server for healthcare interoperability

The CMS Interoperability and Patient Access final rule requires that patient data be exposed as FHIR for Medicaid and other payers. FHIR is also a standard, widely-understood format, so FHIR-compliant APIs are useful.

## FHIR Mapping Project

To see how we connect a HAPI FHIR server to a generic MarkLogic database, check out the MarkLogic FHIR Mapper project. The FHIR Mapping project includes HAPI IResourceProvider implementations and query logic that will be useful to integrate HAPI to this project as well.

The FHIR Mapping project is different in that it uses data services which invoke a DHF mapping step to convert a storage model into FHIR in memory. The step mapping is used because the persistent model is no similar to FHIR and a full mapping is required. Here, we simplify, and avoid writing a full mapping from persistent to FHIR by leveraging the similarity of our storage model to FHIR.

While the .sjs data service implementations in the FHIR Mapping project will not work here, the Java integration code and data service .api specifications will work as-is, if a new search and transform data service (search.sjs module) is written for this project. The intent is to use the generic FHIR transform described below in a new data service. If the data service needs additional query parameters or logic, that can then be added.

## Adding FHIR data services to this project

This project's persistent data model (the Entity Services model) is a simplified version of FHIR, with extensions. These simplifications to the data model are minimal and regular, meaning that one can derive conformant FHIR easily from the persistent data model using mechanical transforms. These mechanical transforms use information encoded in field names, along with some FHIR mapping metadata to drive the transform. A library is included at src\main\ml-modules\root\lib\fhirTransforms\templateTransform.sjs that performs the transform from persistent models to FHIR.

*TODO: add path to a sample metadata file used in tests.*

The main difference between the HSK persistent model and FHIR is that the persistent model "flattens" CodableConcept fields and Identifier fields, replacing them with simpler structures. This reduces the nesting level and complexity of our data models, simplifies and speeds queries, and avoids confusion about how to persist records by fixing one "system" and storing all values in that same system. The library noted above uses a FHIR rewriting template to specify how a persisted document is converted to FHIR. The template specifies the CodableConcept and Identifier structures that have been simplified in our persistent model. That is, the templates specify the flattened fields that need to be changed and replaced. In this way, we have a configuration-driven process to map persistent data to FHIR, which effectively reverses the mechanical "flattening" that was done to the FHIR schemas to yield our persistence-optimized model.

## Steps for FHIR integration

To build a data service that will work with the Java calling code in the FHIR Mapping project, use a similar data service to those such as data-services\patient\search.sjs but instead of calling egress.transformMultiple() (which invokes a data hub step to transform a persistent record), call templateTransofrm() in the library above.

The templateTransform.sjs file also has utility functions for easily converting string fields in the storage model into standard FHIR formats. For example, to convert a flattened string code into a CodeableConcept, you can use the utility template like this:

{

"codeableConcept": [

{

"path": "claim.payee.code\_\_type",

"system": "http://hl7.org/fhir/ValueSet/payeetype",

"lookupValueSet": "payeetype"

}

]

}

This will replace the code\_\_type field in the claim.payee object with a CodeableConcept named type with a coding value of the value in code\_\_type, a system of http://hl7.org/fhir/ValueSet/payeetype, and will look up the display value from the lookup file in src\main\ml-data\referenceData\ValueSets\payeetype.json.

Note that we are assuming that all data is stored in one CodableConcept.system. This makes the persistent value uniform and queryable. Because of this fundamental difference between flexible message formats and uniform persistent formats, we are able to almost always flatten a CodabelConcept into an underscore\_\_delimited field without the complexity and nesting of a CodableConcept, and similarly flatten many Identifiers.

## Adding data elements or records that have no FHIR counterpart

The section above describes how the HSK data model is based on FHIR, but somewhat simplified for resources and types that are covered in the FHIR spec. However, some fields, or entire resources, are not covered by FHIR at all. In that case, use FHIR structures such as Identifier, Address, Money, Period and others found at <https://www.hl7.org/fhir/STU3/datatypes.html> to keep the overall form and feeling of the model the same as true FHIR models.

If non-FHIR data is embedded in persistent Entity definitions, add a configuration to the mapping templates to remove them in the transforms to true FHIR. That is, just as a flattened or simplified CodableConcept must be mapped to a compliant FHIR CodableConcept to create compliant FHIR, an added, non-FHIR data structure must be mapped to undefined (removed) to create compliant FHIR.

Note that custom mappings or complex templates can also be used to create FHIR Extensions. We find the FHIR Extension mechanism to be complex and something to avoid if your business requirements do not absolutely demand it.

## Subsumation searches and FHIR

Adding :above and :below modifiers to searchs can be achieved by using the "Ontologies for Subsumation Searching" described earlier in this cookbook. This will change the way you query for documents, but will not affect the conversion from your storage model into a FHIR record.

## Unit Testing

A number of tests are provided in HSK. The goals of these unit tests are not to test everything, but to illustrate unit testing techniques. Many unit tests (but not all) should be removed when you build your own system, because they address specific mappings and flow logic related to the sample input records. Your data will be different, so your tests should not include these unit tests. If you have similar data (e.g. Patients/Members or Claims) you may want to alter rather than remove some data-dependent unit tests.

### Java testing at the service level

The src/test/java directory contains Junit tests that check that the provided sample data set was properly ingested and mapped. This pattern can be used to test other processes such as Mastering, validation or other processes as well as mapping.

These tests assume you have run the ingest and mapping gradle tasks for all data, and they merely retrieve certain records from the data and verify they look as they should. Failures in ingest or mapping can be caught by these tests.

### Unit testing of internal functions and modules

The src/test/ml-modules directory contains marklogic-unit-test tests, which are JavaScript (or XQuery) tests routines. See <https://github.com/marklogic-community/marklogic-unit-test> for more information on this testing framework.

**Step tests:** Many of these tests run a Step on a particular input to verify the Step works as expected. These Steps are run in memory so do not rely on particular documents being loaded previously into the database. These are more properly integration tests as they test full flows and data services from outside MarkLogic (from Java).

**Other tests:** other functions are also included, such as tests for redaction and for ontology-based claims searching. These are more properly unit tests as they test specific JavaScript functions within MarkLogic.

All “interesting” functions built in your own project should ideally be tested. MarkLogic Unit Test has convenience functions to load data from the …/test-data directories (see the ClaimSuite directory for an example) so that a test can rely on data in the database(s) and still be self-contained.