RH Use and Extension Guide

Table of Contents

[What’s a RH 3](#_Toc34029927)

[Key Concepts 3](#_Toc34029928)

[JSON and XML Documents 3](#_Toc34029929)

[Knowledge Graph and RDF 4](#_Toc34029930)

[Entities and Concepts 4](#_Toc34029931)

[Relevant guides and documentation 5](#_Toc34029932)

[Entity Services have a slightly different notion of Entity 5](#_Toc34029933)

[Base Objects 5](#_Toc34029934)

[Data Flows and Steps 6](#_Toc34029935)

[Link detection via queries 6](#_Toc34029936)

[Ontologies and Taxonomies as reference data 7](#_Toc34029937)

[Ontologies and Taxonomies as business data 7](#_Toc34029938)

[Facets and Uber-facets 7](#_Toc34029939)

[Workspaces 8](#_Toc34029940)

[Boost queries for Search 8](#_Toc34029941)

[Graph-based Recommendations 8](#_Toc34029942)

[Quick Start, Gradle and DevOps 8](#_Toc34029943)

[Unit Tests 9](#_Toc34029944)

[Example Projects –install and uninstall 9](#_Toc34029945)

[Adding or sharing your own example 9](#_Toc34029946)

[DHS (Data Hub Service) 10](#_Toc34029947)

[Development approach uses local machines 10](#_Toc34029948)

[Deployment is always to DHS 10](#_Toc34029949)

[Differences between local DHF operation and production DHS operation 10](#_Toc34029950)

[System Architecture 10](#_Toc34029951)

[Research Hub Foundation = Data Hub Framework + Grove 11](#_Toc34029952)

[No middle tier customization needed 11](#_Toc34029953)

[Quick Start Guide – Basic Extension Steps 12](#_Toc34029954)

[Get started with the stubs example 12](#_Toc34029955)

[Model the desired Entities using Entity Services 12](#_Toc34029956)

[Obtain sample data or synthetic data 12](#_Toc34029957)

[Ingest and explore the sample data as “raw” data 12](#_Toc34029958)

[Map the raw data to Entity Services “final” data formats 13](#_Toc34029959)

[Write two JavaScript classes for each Entity, which extend the appropriate Base Classes 13](#_Toc34029960)

[Adjust the order of the Entity classes 13](#_Toc34029961)

[Adjust the order of the Uber Facets on the GUI 13](#_Toc34029962)

[Write a JavaScript class for each Concept, which extends the Concept Base Class 13](#_Toc34029963)

[Define link queries that will cause Entities to be linked in RDF 13](#_Toc34029964)

[Extend the Mappings with custom hooks that populate headers 14](#_Toc34029965)

[Create TDE Templates 15](#_Toc34029966)

[Use IRIs for Entities and Concepts, Strings for labels 15](#_Toc34029967)

[Define Range Indexes for faceted fields of Entities 15](#_Toc34029968)

[Define facets for each Entity 15](#_Toc34029969)

[Modify the workspace relevance query logic 15](#_Toc34029970)

[Define GUI Views and Cards 16](#_Toc34029971)

[DetailView 16](#_Toc34029972)

[Result View 16](#_Toc34029973)

[Card View 16](#_Toc34029974)

[Graph View Customization 16](#_Toc34029975)

[Data Integration Steps using ML DHF 16](#_Toc34029976)

[UI Development using Grove 16](#_Toc34029977)

[Details on Extension Steps 17](#_Toc34029978)

[Create TDE Templates 17](#_Toc34029979)

[Data Modeling Best Practices 18](#_Toc34029980)

[Core MarkLogic data modeling practices still apply 18](#_Toc34029981)

[Major Business Entities are Entities 18](#_Toc34029982)

[Contextual and minor Business Entities are Concepts 18](#_Toc34029983)

[Concepts can be converted to Business Entities later 18](#_Toc34029984)

[Relationships among Entities and Concepts are RDF 19](#_Toc34029985)

[Extensions and Augmentation 19](#_Toc34029986)

[Ontology-based query expansion 19](#_Toc34029987)

[Business Intelligence and Reporting 19](#_Toc34029988)

[Data Exports 19](#_Toc34029989)

[Data Science 20](#_Toc34029990)

[Operational and Transactional Applications atop a RH Foundation 20](#_Toc34029991)

[MarkLogic Training and Knowledge Transfer 20](#_Toc34029992)

[Training 20](#_Toc34029993)

[Knowledge Transfer Recommendation 20](#_Toc34029994)

[Customer support in getting started 21](#_Toc34029995)

[Partner support in getting started 21](#_Toc34029996)

# What’s a RH

* A Framework to allow a combined team of business analysts and developers to create Data Hubs, together with a Search and Discovery application to expose the data
* Data Integration
  + Integrates many data sets together – like any MarkLogic Data Hub
  + Supports structured and unstructured (or semi-structured) data sources
  + Combines an appropriate subset of the data into a knowledge graph using RDF and semantic technology
  + Masters data where appropriate (in the MDM sense)
  + Provides APIs for search, discovery, updates, reporting, business intelligence
* Search, Discovery and Knowledge Sharing GUI
  + Search for any type of data
    - See the data as results and details pages
    - See a graph view to provide context around the data and discover new items
  + Support users gathering and saving items into Workspaces for sharing and Knowledge Transfer
  + Support users gathering and saving items into Workspaces for long-running discovery processes
* AI
  + Recommend relevant content based on current and past activities
  + Adding to Workspaces guides search results, by default
  + Including items on the graph view guides graph-based Recommendations (using link walking and analysis)
* Extensible to any feature set that MarkLogic generally supports, including compliance, 360-degree views of key business entities, analytics, powering other applications in an enterprise via REST or SOAP calls

# Key Concepts

## JSON and XML Documents

Most data in a Research Hub (or any MarkLogic Data Hub) is stored in JSON or XML format. These documents group data together into manageable information chunks which represent business entities.

## Knowledge Graph and RDF

Research Hub data often also exists in RDF form, forming a network structure (mathematically called a “graph” structure). Generally, the graph provides users context for their searches. Manually clicking through to review 10 or 20 search results, one by one, is a slow and cumbersome way understand the overall information landscape, and the Knowledge Graph solves that problem.

The Knowledge Graph is a set of Entities and Concepts, together with links among them, expressed as RDF triples. The overall Knowledge graph has all Entities and Concepts in the system as nodes in the graph, and an RDF links between any two related Concepts or Entities, forming a graph structure of nodes and edges. As usual with RDF, each Entity and Concept has a unique IRI (like a URI) to identify it within the graph.

The Knowledge Graph also contains a set of facts about each Entity and Concept (in addition to the relationships) including the display name (preferredName) and data type of each. Other data may optionally be included in the Knowledge Graph, but name and type are required.

The Knowledge graph fully powers the Research Hub graph views, enabling a user to visualize the Knowledge Graph, expand it, and remove items from a view. That is, no graph information other than the tooltips are taken from the JSON and XML Documents: all information is taken from the RDF via SPARQL queries.

The RDF links in the Knowledge Graph are also referred to variously as “triples,” “ontology” data, “taxonomic” data, OWL data, or “semantic data.” In this guide we will typically call this the “Knowledge Graph” but know that the underlying form is essentially RDF.

Note that the information in the Knowledge Graph is not separate from or in addition to the information in the JSON and XML Documents – rather the graph is a projection out of the documents, accomplished using TDE as described below.

## Entities and Concepts

An Entity is a structured data record that typically matches a Business Entity. In the RH system it is stored as a JSON or XML Document. (see *JSON and XML Documents* for more information).

A Concept is a smaller, structured record that typically represents some aspect of a Business Entity.

In the RH system, Entities are modeled using the MarkLogic “Entity Services” specification and are persisted as JSON or XML Documents. Entities each have their own search context on the GUI, and an “uber facet” selector at the top of the GUI to enter that search context; that is, to show the Entity-specific facets for that Entity, and to constrain searches to only query that type of data.

Concepts are modeled as RDF, and are persisted as RDF triples. Concepts generally only appear on the graph view. Entities also appear on the graph view, so both appear on the graph view, but only Entities are used in the search contexts. There is no need to explicitly model or build an ontology for a Concept, but optionally one may do that. It is only important to add RDF triples and query them appropriately.

For example, a Person may be an Entity, and may be stored as a Person JSON Document which includes nested, structured data such as one or more Addresses, and Skills or Competencies. The Skills, competencies, and Postal Codes stored as part of the Person Addresses might be projected into RDF to form Concepts. The Concepts do not have their own JSON or XML Documents, but are instead derived from data inside the main Entities, and projected into the RDF graph data structures.

### Relevant guides and documentation

* See <https://developer.marklogic.com/features/flexible-data-model/> for more information on storing both JSON/XML Document and RDF data as parts of an overall combined data model.
* See <https://docs.marklogic.com/10.0/guide/app-dev/TDE> for information on projecting data from Documents to RDF using Templates.
* See <https://docs.marklogic.com/guide/entity-services/intro> for more information on Entity Services.

### Entity Services have a slightly different notion of Entity

MarkLogic has a feature called Entity Services where each JSON or XML Document type or sub-type is defined as an “entity.” The “Entities” in a Research Hub system are only the top-level Entities.

For example, a Research Hub may have Person Entities with Address information in them. The Addresses would likely be a sub-object comprising the city, state, zip, street etc. of an address. In Entity Services, the Address would be modeled as its own entity, who’s only purpose is to be embedded as a sub-object in the Person or other Entities. In Research Hub, we only consier the “main” entity that defines and entire Document to be an “Entity.” The Address is not an Entity in the Research Hub sense, even though the Entity Services model will have a separate config file for Address.

## Base Objects

Much of the configuration required for a new set of Entities will be encapsulated in a set of Objects, which subclass provided Base Objects using the (new-ish) ECMA-6 “extends” keyword.

Each Entity must have both a front-end and back-end class representing certain logic for the Entity. On the back end, a JavaScript class must be written for each entity that extends EntityConfig (see: See *src/main/ml-modules/root/entities/entityconfig.sjs* in the repository). This back end code will run in MarkLogic Server. On the front end, a JavaScript class must be written for each entity that extends Entity (see: *ui/ui/src/entity/Entity.js* in the code repository). This front end code will run in the users’ web browsers.

Each Concept must have a front-end class, which must be written and must extend Concept (see: *ui/ui/src/concept/Concept.js* in the repository). There is no back-end base class for Concepts.

The Base Objects have default implementations or methods to override for behaviors like returning the text name of the entity, or returning the color of the entities icon, or searching for related Entities in the database to link incoming Entities to other, related entities (thus building the knowledge graph).

The base object also gets the code to create mouse-over tooltips (hover display titles) for nodes on the graph view, which goes in the method getGraphTitle(). This returns HTML to allow multi-line and formatted hover titles.

## Data Flows and Steps

The Data Hub Framework (DHF) is the underlying toolkit that provides for data integration. Incoming Entity and Concept data, in whatever form, is ingested as “raw” data into the DHF, and then “curated” by running a number of “steps” that map, master, or otherwise prepare and transform the data.

As explained in DHF documentation, a typical flow will have two steps: an Input Step that defines an endpoint which will accept incoming data to be stored “as-is” as “raw” data; and a Mapping Step which will convert the raw record(s) into the form defined in the Entity Services models.

A third “Mastering” step is also fairly common, for data that needs merging and de-duplication. Mastering Steps in DHF 5.1 and beyond, are more efficiently run as separate “Matching” steps and then a “Merging” step. That is, Mastering = Matching + Merging.

## Link detection via queries

In Research Hub applications, we extend the DHF steps and ensure there is a process in one of the steps to detect relationships (aka “links”) among the Entities and Concepts. Generally this has two parts: extract some simplified match criteria to the “headers” section of the Entity envelopes, and build a query to be used by one Entity to find or query the other entity.

For Example, if Person records are ingested and mapped, and Department records are also ingested and mapped, some Person DHF step would build a query describing how to find the Person’s Department, so that an RDF link (a triple) can be added as part of the Person mapping process.

The triple should not be directly added, but rather the Department document IRI will be put into the Person record in the “header” section of the envelope, and then TDE (see below) will be used to project that relationship into RDF triples automatically.

## Ontologies and Taxonomies as reference data

An ontology represents an arbitrary network of facts, in a highly structured manner. A Taxonomy is a simpler version of an ontology (but is still an ontology) where the structure is limited to super- and sub- relationships among the facts or items.

Sometimes Ontologies (and by extension Taxonomies) are best considered “reference” data and are used in lookups. E.g. a Taxonomy of places may be good reference data. Such a taxonomy is also called a gazetteer. Such a taxonomy could be used to tag missing “county” data for a Person’s address, if the zip code and city is provided, but not the county. In that example, the Ontology is used to perform lookups during mapping, and is referenced data.

Alternatively, Ontologies can be used in the “final” database to augment queries. One such approach is “ontology-based query expansion” as described below.

## Ontologies and Taxonomies as business data

In other examples, an Ontology is best considered as a small database. E.g. an ontology of People and Competencies for a company could be ingested as raw data, which captures every person in the organization and what they know how to do. This Ontology would be considered business data for a skills- and job-matching hub. In that case, a flow should be built and run that “maps” every ontology concept into an Entity (either a new Entity or by joining and merging the Ontology data into an Entity with many sources).

## Facets and Uber-facets

MarkLogic natively supports faceted search (note: facet links are sometimes called navigation links or drilldown links).

At the top of any Research Hub GUI is a set of top-level selectors called “uber facets.” These select the type of information that will be searched. These links correspond exactly to the Entities defined in the Research Hub.

When an uber facet is selected, the GUI will enter a mode where it will search that type of data only (documents whose type is the selected uber facet). Regular facets will appear on the left-hand-side of the screen. These regular facets are typical drill-down navigational links to narrow the scope of data being searched.

Note that facets are particular to an Entity type. So a Person uber facet selection may cause relevant facets such as postalCode, age, gender to appear. Most facets do not make sense across many or all Entities, so when no uber facet is selected, no regular facets are available either.

When there are no uber-facets selected, the behavior of the GUI will be to search all data.

## Workspaces

A Workspaces is like a topic of investigation or a search session. During a potentially long-running process to find an answer or gather matching and relevant data on some topic, a user can add many things to a Workspace.

Workspaces can be referred to later, and are shared with others implicitly because workspaces can be discovered when they show up on other users’ knowledge graphs.

Research Hub systems allow Entities to be gathered and saved in Workspaces. Any Entity can be saved in a workspace. By default, the ways an Entity related to a workspace is that it is “related” “likely to be related” or “not related.”

A Workspace is a special kind of Entity, which does not have a data ingest or mapping flow or steps. Users directly author Workspace inclusions.

## Boost queries for Search

Workspaces contribute to search relevance. As items are added to a workspace with “definite” or “likely” modifiers, that is considered positive evidence that the user is looking for that sort of information. Each Entity in a workspace then contributes a “boost” to the search queries being performed. Entities added with the “not” modifier contribute a negative boost.

The nature of how each Entity affects the boost query is alterd by changing the code ni the file: *src/main/ml-modules/root/lib/searchModel.xqy*. (Note this is a somewhat unusual XQuery file, which is callable from the main JavaScript code.)

## Graph-based Recommendations

When looking at a Graph, the user can choose to “include recommendations” which loads a large subset of Entities and Concepts from the Knowledge Graph, and computes which are mostly tightly linked (by link count) to the items already on the Graph.

This is useful for knowledge discovery, because items closely linked to existing, relevant data, are often themselves relevant.

## Quick Start, Gradle and DevOps

The MarkLogic Data Hub Framework includes gradle build scripts, and a QuickStart GUI. These must be used in the typical Data Hub way to integrate and map data for a Research Hub. See the MarkLogic Data Hub documentation for more details.

Briefly, QuickStart allows visualization of the steps and flows defined for a Data Hub, runs them, and browses both system data and step execution metadata or history. Gradle automated all these tasks from the command line, and for the developer audience this guide is written for, gradle will be preferable after comfort is established with the Data Hub Framework, because it is command-line driven and scriptable.

Gradle is a popular build scripting tool, and has a MarkLogic extension called ml-gradle, which ships as part of the Research Hub. Gradle is oriented around “tasks” that can be executed in order. Extending the Research Hub with new Entities requires new configuration entries are added as Tasks in the build.gradle file. (this may move to a separate build file in time).

## Unit Tests

Unit tests should be added for all interesting or important aspects of a Research Hub system. The ml-unit-test framework is included in the Research Hub, and allows database-oriented tests to run to ensure that steps and other operations work properly. These tests often read and write from and to the database, and then retrieve the data to ensure it is correct.

## Example Projects –install and uninstall

Example projects ship with the Research Hub for both a pharmaceutical research domain and Person/Resume/Job domain. A script will install or un-install these sample projects by copying all the required classes and configuration objects from the /examples area into the various locations in the Research Hub directory structure where they belong. The custom code and configuration includes: Entity definitions, base classes, view code, TDE templates, index configurations and other information. Data files are also copied to support raw data ingest and ontology load.

The command to do this is

./gradlew examples:install -Pexample=person-position-matching

Or

./gradlew examples:install -Pexample=pharma-research

Because the directory names of the two provided examples are “person-position-matching” and “pharma-research” respectively, those are the two possible “example:” parameters.

To clear it all out (remove customized code and config files), run “uninstall.”

### Adding or sharing your own example

Once your new Research Hub is configured and coded (assuming proper practices are followed) the domain-specific files can be copied out into a new Example directory using the “createExample” task, which is in build.gradle in the provide example directories. This output can be zipped up or shared as needed. Others can then “install” the code. This will also ease upgrades in the future, when you pull down a new Research Hub framework, and install your custom code from an example sub-directory.

Overall, “createExample” and “install” tasks allow you to disentangle your code from core framework code. For this reason, it is important to only add new files in the recommended locations, and not edit or change the core framework files.

## DHS (Data Hub Service)

The Data Hub Service (DHS) is a cloud-based service available in AWS or Azure (as of this writing) which provisions and executes a scalable, robust Data Hub Framework service.

### Development approach uses local machines

To develop a new Research Hub, download the Research Hub files and develop on a local machine (e.g. a laptop).

### Deployment is always to DHS

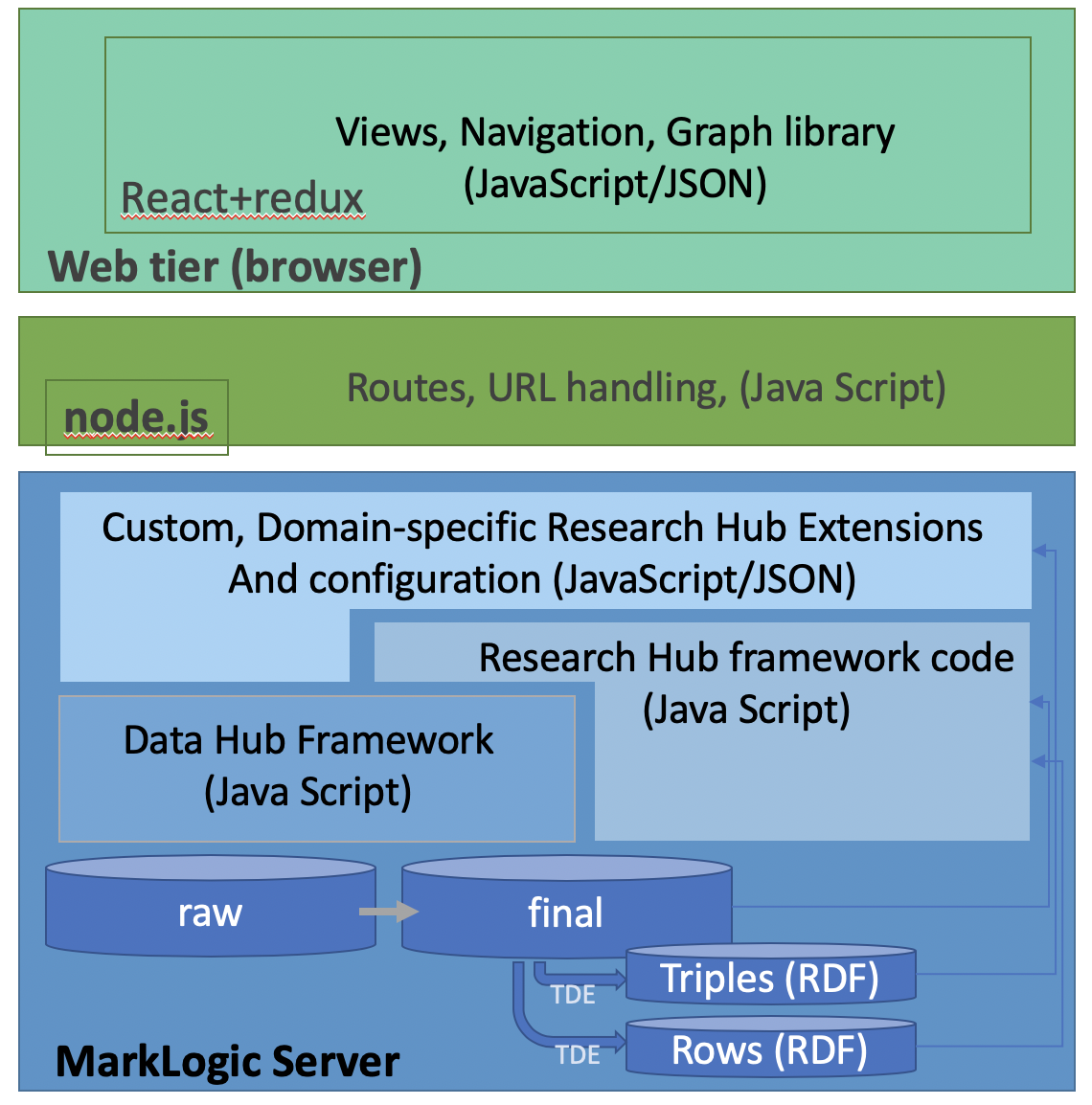
Once complete, or when scalable data loads are required, use the typical Data Hub Framework to DSH deploy process to “push” the local build into the cloud environment.

### Differences between local DHF operation and production DHS operation

Almost all functions available on a local MarkLogic server DHF instance are available in DHS as well. The full stack for Resarch Hub also requires a node.js server and a data loader machine. A non-cloud laptop can be used to load data, but it is recommended to use a separate data loader EC2 or Azure instance, or load data from the node.js middle tier server.

# System Architecture

A Research Hub system is a three-tier application, with a rather thin middle tier. Most of the logic is in the web tier (executing in the web browser) or in MarkLogic Server.



The back end is a MarkLogic Data Hub Framework application, which in turn runs on the proprietary MarkLogic Server data product.

The front end uses the popular React framework together with the Redux library, and is built on a higher-level GUI framework specific to MarkLogic called Grove.

The Graph component of the Research Hub (node/edge graph display) uses vis.js. However there is out of the box configurations that should hide the vis.js use in many cases.

## Research Hub Foundation = Data Hub Framework + Grove

It is important to understand, from the perspective of documentation, that the back-end of a Research Hub application is a Data Hub built using the MarkLogic Data Hub Framework, and the front end is a Grove application built using the Grove toolkit.

It is not necessary to understand every part of all underlying technologies to build and extend a Research Hub, however basic familiarity is required with each of:

* Data Hub Framework: <https://docs.marklogic.com/datahub/>
* MarkLogic Server: <http://docs.marklogic.com/>
* Grove GUI toolkit: <https://developer.marklogic.com/code/grove/>
* React: <https://reactjs.org/>

The main skills needed from each of these (which will not be described in detail here, deferring to the more complete documentation for each) are:

* Creating and running Steps and Flows for Input Steps and Mapping Steps, and sometimes running Mastering Steps in the Data Hub Framework (DHF)
* Extending and automating flows and deployment to DHS for DHF
* Modeling data using Entity Services per the MarkLogic documentation
* Building new compoents and views in React (e.g. detail pages, and search results tiles)
* Extracting RDF data from Entities using Template Driven Extraction in MarkLogic
* Building queries in MarkLogic
* Lightweight query using SPARQL to access the RDF triple store in MarkLogic
* JavaScript, and the particular MarkLogic objects and types provided

To repeat: these topics are extensively documented elsewhere, and this guide will refer to that documentation, therefore developers and analysts building a Research Hub must take training and familiarize themselves with these underlying concepts at some level.

We recommend that a new research hub developer or analyst partner with MarkLogic to ramp up efficiently on the various topics, and quickly learn the specific topics required while avoiding having to sift through the full scope of MarkLogic Server functionality, which extends well-beyond the features required for Research Hub operations.

## No middle tier customization needed

The node.js middle tier has dynamically-created routes that will connect the front end and back-end code. By properly adding back-end Entities per this guide, the middle tier will automatically create routes named after the entities.

Note that the addition of search.search “options nodes” for faceting for each Entity is what drives the middle tier routes.

# Quick Start Guide – Basic Extension Steps

Perform these steps to build a working Research Hub. For the most part, extending the Research Hub is done by adding a set of Entities and Concepts to the out-of-the-box Research Hub code base. Each Entity and Concept corresponds to a number of files, including code and configuration files.

The steps below generally tell what functionality to configure or code in which file, as well as the patterns to use and libraries to call to do so.

## Scaffolding configuration and code for new Entities and Concepts

### Scaffolding a new Entity

The Research Hub framework is entity- and concept-based. Every Entity (as you can see elsewhere in this guide) is configured and coded by adding and extending a variety of files, including extensions of Base JavaScript classes, EntityServices models, search options, GUI rendering code files, and others.

To create these automatically for an Entity, with reasonable defaults for the various files, run:

./gradlew createEntity -PentityName=YourEntityName

The task will create the necessary files for the system to run with the new Entity.

The task will also print out the generated files’ locations so you know which files to then edit and update with more complete and correct behavior.

### Deleting a scaffolded entity

If for some reason you need to delete an entity that you scaffolded, simply run this command.

run

./gradlew deleteEntity -PentityName=YourEntityName

### Scaffolding a new Concept

You can create code and configuration files for a new Concept by running this gradle command:

./gradlew createConcept -PconceptName=YourConceptName

The task will create the necessary files and print out their locations for you to then edit.

### Deleting a scaffolded concept

To delete a Concept (perhaps one you incorrectly scaffolded) run this command:

./gradlew deleteConcept -PconceptName=YourConceptName

## Model the desired Entities using Entity Services

Entity Services models can be graphically created in the QuickStart GUI for the Data Hub Framework, or edited by hand as a JSON descriptor, which is similar to JSON Schema.

## Obtain sample data or synthetic data

A Research Hub is built to expose a set of disparate data. You’ll need some to test with early on.

## Ingest and explore the sample data as “raw” data

Create Input steps per the DHF instructions to load the sample data.

## Map the raw data to Entity Services “final” data formats

The DHF mapping screen allows an analyst to view the incoming raw records and graphically map them to the defined Entity Services models.

## Write two JavaScript classes for each Entity, which extend the appropriate Base Classes

Subclass the appropriate base class to create both server-side and web-tier classes for each Entity. Various methods must be overridden for these classes to be complete. Some of these methods will be static (what is the text name of the entity class) and some instance specific.

One file on the client and server must import and list all Entities and Concepts. These are

* src/main/ml-modules/root/entities/allEntities.sjs (for the back end)
* ui/ui/src/entity/entityConfig.js (for the front end Entity classes)

### Adjust the order of the Entity classes

The order of the “uber-facets” up top will be determined by the order of the entities in

* ui/ui/src/entity/entityConfig.js

### Adjust the order of the Uber Facets on the GUI

The order of the properties in the exported object in entityConfig.js (in the GUI code) determines the order of the top-level selectors (uber facets) below the search bar.

## Write a JavaScript class for each Concept, which extends the Concept Base Class

Define the additional Concepts to cause new nodes and edges to appear in the graph view, and to enhance the underling RDF Knowledge Graph. See example above where a Person Entity may have a “postalCode” concept to cause the graph view to cluster people by US zip code.

Import all Concept classes into the file:

* ui/ui/src/conceptConfig.js (for the front end Concept classes)

## Define link queries that will cause Entities to be linked in RDF

For text content, use the entity-enrichment.xqy library to find text terms matching the various structured records in the database. In this pattern, structured records are ingested before text or semi-structured records (e.g. before PDFs or XML representing text articles). Each structured Entity includes a reverse query as the link query.

Once these are included, a call to

entityEnrichment.enrich(uriOfTextEntity, enrichmentOptions)

will return an enrichment structure with all related content, which can be included in the entity to drive links. This includes checking the reverse query conditions and also performing a text search as a relevance check (ensuring the cts.confidence of the match is over a threshold). See the Pharma Hub example in the PubMedXML flow to see how this works.

## Extend the Mappings with custom hooks that populate headers

Entity Services defines an “envelope” structure with a “headers” portion. The Research Hub project requires that graph-related data or metadata is populated into this headers section. This is a development task, utilizing JavaScript code to extract and normalize the data. Typically, headers data is added to easily extract triples from the headers via Template Driven Extraction (TDE).

Use a custom hook in the Data Hub Framework steps to set the headers. This should be a “post step” hook meaning that the runBefore: flag is set to false. Obtain and modify the headers section via the code pattern shown below:

var uris; // an array of URIs (may only be one) being processed

var content; // an array of objects for each document being processed

[ . . . ]

content = content.map(c => {

let value = c.value.toObject();

let headers = value.envelope.headers;

[ . . . manipulate the content and headers sub-objects in

any way needed . . . ]

c.value = value;

c.value.envelope.attachments = null; // generally do not keep the original

return c;

});

Key points about this code pattern for hooks

1. content is passed in as an array. Modify them all.
2. Convert the marklogic node object, c, to a regular JavaScript object via c.value.toObject()
3. Manipulate that object. Search for other data, write related records to the database, log items or other actions needed.
4. Set the object back to the “value” of c (no need to convert back to a database node. This will happen when the data is stored to the DB later)
5. Null out the “attachments” which otherwise will store the raw data (input) document. Our pattern is to leave the raw data in the Staging database, and not replicate to the final documents.

You can then add any required facet or link information (such as IRIs of related items, or values to appear in facets) to the headers subobject. The overall, modified envelope will be saved. This allows GUI-based steps such as a Mapping Step to be used (which do not set headers) and then modified by a minimal chunk of code that only does what the Mapping Step cannot easily do such as set reverse-queries.

As an alternative, put similar code into a new reverseQuery construction function and register it with the DHF 5.1 mapper, and add the reverse query directly to the Instance data of the entity. This requires that the Instance model includes a field for the reverse query.

## Create TDE Templates

Templates are declarative configuration objects (in JSON) which specify how document data is projected into a RDF triples materialized view. They must be added to the *src/main/ml-schemas/tde* directory.

Note that there are templates for both triples and rows (SQL projections). Research hub requires you to add only triple projections.

These TDE templates must also create some standard triples, such as PRH:preferredName and PRH:entityType. (PRH can be remembered as standing for: “Prefix Research Hub”)

Every Concept or Entity that will show on the graph must have an IRI. This IRI will be the URI of the Entity for an Entity, and can be something else for a Concept (since there is no document for a Concept). Two specific triples must be created for every Entity and Concept, as above, capturing the name and type, respectively.

### Use IRIs for Entities and Concepts, Strings for labels

Note that the standard used and recommended is to use IRIs for Entities and Concepts, and Strings for labels, types and other information. E.g. the entityType should always be a string.

## Define Range Indexes for faceted fields of Entities

For a facet to exist for some field or property in an Entity, a supporting Range Index structure must be defined. The Range Index allows fast access to the values and counts needed by a facet. These are defined per the Entity Services specification, by configuring the index in the DHF Quick Start modeling GUI, or directly editing the …/entity/<entity\_name>.entity.json file (which is also written by the GUI if QuickStart is used). Be careful that the indexes written manually or with QuickStart have the same collation and the indexes described in the options file range constraint entries.

## Define facets for each Entity

Create an “options node” per the search API specification (<https://docs.marklogic.com/guide/search-dev/search-api> ) and add that configuration object to: *src/main/entity-config/exp-final-entity-options.xml* (JSON can be used also).

Each “facet” defined in the options configuration must have a Range Index configured.

## Modify the workspace relevance query logic

The Research Hub has a built-in facility to show more relevant content in searches based on what is in the currently-selected workspace. To do this, each included item in the workspace contributes a weighted “boost” query (see <https://docs.marklogic.com/cts.boostQuery> for details).

To configure this, override the method getBoostQuery(entity, weight) in each Entity base class in the server code. The inputs will be

* Entity – the XML or JSON node representing the entity in the workspace
* Weight – how much boost weighting is passed in from the framework code

The return type should be a cts.query, and is expected

## Define GUI Views and Cards

Each Entity must be displayed in two or three ways, requiring two or three React components per Entity.

### DetailView

Add a JavaScript class called <entityName>DetailView.js for each Entity. This view displays the full details of an Entity instance using a React component. See React documentation for how to define views and add that as a JavaScript file.

For an Entity named FooBar call this *entity/FooBar/FooBarDetailView.js.* Note there is a separate directory in the GUI code area for each entity, sharing the name of that Entity.

This view can be rather large, and typically includes a “Card” (see below) for some or all related Entities of various types.

### Result View

Add a JavaScript class called <entiityName>/<entity>Result.js which shows a short summary of an Entity and will be used in lists of results. This view should not be very large so 10 or 20 of them can be included on one screen, and reasonably scrolled through by the user.

Often the Result view is a small wrapper around the Card view.

### Card View

Add a JavaScript class called <entiityName>/<entity>Card.js which shows a short summary of an Entity and will be used as an embedded preview in larger views.

### Graph View Customization

The Concept and Entity objects have code to customize the graph view, including icons, colors of nodes, and “title” html that shows when the mouse pointer hovers over a node.

Behavior of the graph, such as expansion behavior, is customized in the SPARQL query snippets per Entity and Concept.

## Data Integration Steps using ML DHF

Where possible use the DHF 5.1 mapper GUI to perform raw to Entity transforms. The GUI is capable but not as flexible as running arbitrary code, so do what can be done in the GUI, and then code the rest using a customHook defined in the mapping step (with runBefore set to false so it runs after the mapping). Like all steps, this is configfured in the flow.json file for the Entity.

## UI Development using Grove

The Grove toolkit is found at <https://developer.marklogic.com/code/grove/> , and the Research Hub GUI is a Grove app, using the React front end option and the node.js middle tier option.

Various pre-built GUI files and components are included for

* Overall look and feel, css etc.
* Graph vs list views, and toggling between them
* Facet placement and control (normal facets as well as “uber facets”)
* Workspace functionality, including bookmarking
* Graph view via vis.js
* Custom “or” facets, where every facet allows multiple selections and every uber-facet enables a different per-Entity facet
* Boost query toggling
* Breadcrumbs for facet selections

The individual views are fairly typical React code.

Redux is also used, to manage the overall application state, allows caching of information pulled in from the back-end, and facilitates easier sharing of information among de-coupled components.

# Details on Extension Steps

The steps above are the main list of tasks to perform to create a working Research Hub. Some of those steps require additional detail that was omitted for clarity. This section has additional information about certain steps.

## Create TDE Templates

Ideally, the related IRIs for an Entity are stored in that Entity’s headers section of the Entity Service Envelope pattern, and are projected into RDF using Templates per the Template Driven Extraction (TDE) feature of MarkLogic Server. Here are some salient points about that:

* All RDF triples refer to Entities and Concepts using a unique IRI for each.
  + The spec for an IRI is a URI extension used in RDF. All URIs are also valid IRIs.
  + In a Research Hub, we have chosen to use the Document URI as the IRI. This is somewhat unusual from and RDF standpoint, because those document URIs have a .json or .xml suffix. Despite being unusual, they are legitimate IRIs and simplify queries. More specifically, it makes it easier to retrieve IRIs using SPARQL, and use those directly to look up the corresponding Entity Documents
* Our best practice and examples are to put the related IRIs in the Entities. However, it is possible to put important keys and data in the headers, and build the IRIs using formulae in the TDE Templates. This is not recommended, in order to keep the complex transforms in one place, in the headers.sjs code.
* We speak of “headers.sjs” code in this guide, and use headers.sjs to encapsulate the creation of the headers portion of an ES envelope, but there is no restriction on the naming of files in a Data Hub Step. So the headers construction code could be put in a file with a different name.

# Data Modeling Best Practices

## Core MarkLogic data modeling practices still apply

See the MarkLogic guides and tutorials on data modeling at:

* <https://developer.marklogic.com/video/data-modeling/> (video)
* <https://mlu.marklogic.com/ondemand/f0f32816> (short video)
* <https://www.marklogic.com/learn/courses/marklogic-data-integration/> (2 day course)

For the Research Hub we have a more prescriptive view and some conventions to follow

## Major Business Entities are Entities

Major items, such as people, invoices, reports, customers, insurance claims, drugs, chemicals, arrest records, places, organizations etc. tend to be Entities. Entities are persisted as JSON or XML Documents.

A rough goal is to have the average Entity size be about 10K to 100K, as serialized text. Larger or smaller records is also acceptable. For example, you might not want a Customer’s entire purchase history in the Customer Entity, as it may span thousands of purchases and get very large.

## Contextual and minor Business Entities are Concepts

For smaller items, create sub-objects within the Entities and materialize them as Concepts stored as RDF in the triple store. The triple store is not another product – simply create TDE mappings and the triples will be projected out of the entities.

If the data desired for the triples is not available in the right format ini the Entities, create a pre-processed property in the headers. E.g. the headers might contain the full name (assembled from component names) the postal code (extracted from the zip+4), a set of skills or competencies (extracted from a resume and normalized to match a controlled vocabulary).

Concepts must have a triple associated called PRH:entityType and PRH:preferredName for the graph libraries and service in Research Hub to work. This is a required convention.

In the Research hub, we use the URI of the Entity Document to represent an Entity in the graph, as the IRI for all triples. For Concepts you are free to make up your own IRI scheme, but should try to have Concept IRIs match the URI scheme for Entities.

## Concepts can be converted to Business Entities later

Over time, Concepts can be “promoted” to become Entities by defining new Entity classes and converting the Concept classes into Entity classes. The projection of data into the RDF triple store may remain substantially or exactly the same.

## Relationships among Entities and Concepts are RDF

When an Entity has a relation to another Entity or a Concept this fact should be represented as a triple. This triple’s general format will be

<EntityIRI> PRH:<relationshipName> <EntityIRI>

These relations should be uni-directional. The SPARQL code is configured by either snippets of SPARQL or lists of predictates to query for bewteen entities. (this is being reworked as of this writing). The SPARQL code will query both directions of the relations using the “^” inverse property path operator

allEntities.getPredicates()

.map(p => `<${p}> | **^<${p}>**`) // bold part is the inverse for bi-directional query

.join(' | ')

# Extensions and Augmentation

## Ontology-based query expansion

All MarkLogic systems support Ontology-Based Query Expansion (OBQE) by allowing keywords and terms in a query string to be expanded into a larger disjunct of related or sub-terms.

Also, as a person types a query, terms and concepts from an ontology may be presented in real time, much like common queries are suggested by Google as a person starts to type in a query string.

This is common in MarkLogic systems, and the Research Hub supports this approach by supporting the loading and harmonizing of Ontologies, but there is not built-in type-ahead or query expansion code. This must be added if desired by the developers of a Research Hub.

## Business Intelligence and Reporting

Entity Services are the basis and required for all Reporting Hub data models, and automatically cause the properties in each entity (as modeled) to be projected into a row. So you can connect via the SQL/ODBC interface to the document data. Header data will not be in the views, because the Entity Services models drive TDE Templates that project only those properties modeled via Entity Services.

It may be necessary to add primary and foreign keys to the Entity Services models to ensure there are join keys which allow SQL to join related entities. SQL cannot (confirm?) access the RDF data. Without join keys, you’d use optic to access data, which can combine both document and RDF triple data in a query.

## Data Exports

Data exports can be made using standard techniques from a Research Hub just like any data hub. Use optic queries, corb, SQL, NiFi, Glue or other technology to export data in batches for use in downstream systems.

This can be very useful because the Research Hub data should be curated, mastered and have higher quality, as well as traceability and good governance. E.g. you can trace from final data properties back to the incoming records, declarative mapping rules and batches that gave rise to them. Smart Mastering will deduplicate and clean up duplicate data. Various schemas and rules can be used (as in any Data Hub) during the mapping and curation process to improve data quality.

Therefore, the hub data is ideal for export to downstream systems. In fact, exploring and selecting data with the Research Hub can set up the queries and conditions to export data for various purposes such as data science.

## Data Science

Running AI and Machine Learning models inside MarkLogic

Exporting data for downstream AI and ML

## Operational and Transactional Applications atop a RH Foundation

# MarkLogic Training and Knowledge Transfer

To build a Research Hub, the team should be familiar with all of:

* MarkLogic Data Hub Framework
* Data Modeling in MarkLogic
* Querying in MarkLogic – particularly cts.query constructs, but perhaps also Optic
* Basic SPARQL behavior
* React, Redux for web page development
* MarkLogic Grove

## Training

Data Hub Fast Start (video):

<https://developer.marklogic.com/video/data-hub/>

Data Hub (instructor led, 8 hours):

<https://www.marklogic.com/learn/courses/getting-started-with-the-marklogic-data-hub/>

Data Modeling and Integration with DHF (instructor led, 2 days)

<https://www.marklogic.com/learn/courses/marklogic-data-integration/>

Grove background (web pages)

<https://developer.marklogic.com/code/grove/>

## Knowledge Transfer Recommendation

It takes time to learn the ins and outs of all the concepts, topics and technologies involved. We recommend you work with a MarkLogic expert to guide your team in building out the first two Entities in your new Research Hub. Start small, with a moderate number of data properties per Entity, a few Concepts, some knowledge graph links, and learn the overall system. Later, the number of Entities as well as the extent of the data and number of input data sources can be expanded.

### Customer support in getting started

For customers, engage with a MarkLogic Consultant to get started

### Partner support in getting started

For partners, engage with a Partner Enablement Consultant to get started