Information about HAPI FHIR-JPA SERVER

HAPI FHIR JPA Architecture

The HAPI JPA Server has the following components:

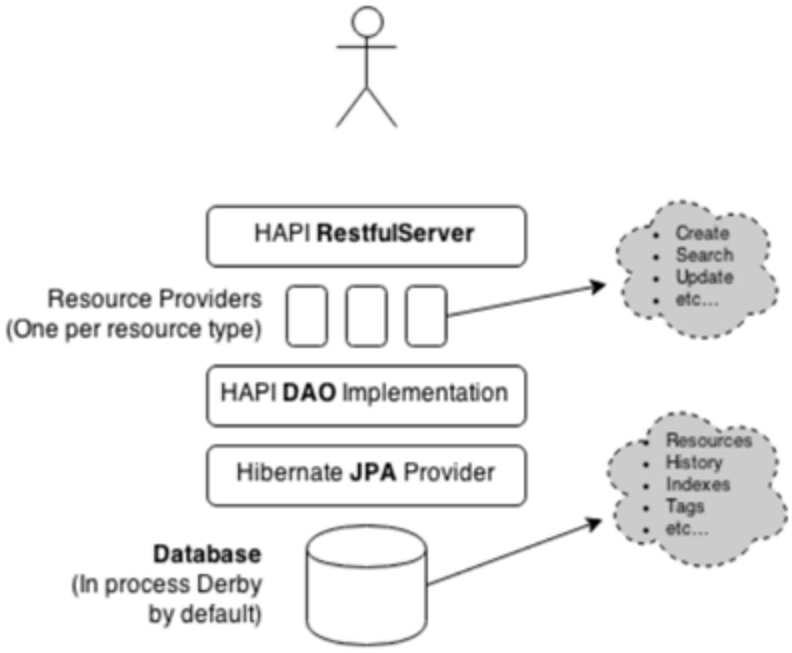
Resource Providers: Each release of FHIR provides a RESTful Server Resource Provider for each resource type. Each resource provider in the HAPI JPA Server includes a @Search method that supports every search parameter defined in the FHIR specification for its respective resource type.**The resource providers also extend a superclass which implements all of the other FHIR methods, such as Read, Create, Delete, etc.**

***The resource providers do not actually implement any of the logic in searching, updating, etc. They simply receive the incoming HTTP calls (via the RestfulServer) and pass along the incoming requests to the DAOs.**

HAPI DAOs (Data Access Objects): Handle all database logic for storing, indexing, and retrieving FHIR resources using JPA.

Hibernate: The HAPI JPA Server uses the JPA library via Hibernate. While it avoids Hibernate-specific features to allow compatibility with other providers (like EclipseLink), these alternatives are not regularly tested.

Database: The RESTful server uses an embedded Derby database, but can be configured to talk to any database supported by Hibernate.



Schema

The HAPI JPA Server uses a simple and efficient database design where all the main FHIR resource data is stored in one big table as text (which can be compressed to save space), while other smaller tables handle things like search indexes, tags, and history tracking. This setup is flexible, but it’s just one way to build a FHIR server.

HAPI FHIR JPA Schema

It contains a description of the tables within the HAPI FHIR JPA database.

Persistent IDs (PIDs)

The HAPI FHIR JPA schema uses internal persistent IDs—usually called PID and stored as 8-byte integers generated by database sequences—as primary keys to keep foreign key relationships efficient and allow business identifiers to change.

More information about the columns :
https://hapifhir.io/hapi-fhir/docs/server_jpa/schema.html

JPA Server Configuration Options

External/Absolute Resource References

If a client tries to post absolute resource references (like the following example), by default, the server will reject it, as only local references are permitted by the server. This can be changed however.

```
<Patient xmlns="http://hl7.org/fhir">
   <id value="patient-infant-01"/>
   <name>
      <use value="official"/>
      <family value="Miller"/>
      <given value="Samuel"/>
   </name>
   <managingOrganization>
      <reference value="http://example.com/fhir/Organization/123"/>
   </managingOrganization>
</Patient>
```

How to configure: https://hapifhir.io/hapi-fhir/docs/server_jpa/configuration.html

Logical References

In HAPI FHIR, logical references are identifiers (like canonical URLs) that don't point to actual server locations. We can tell HAPI to treat specific URL patterns as logical, so it won't try to resolve them.

Eg.

```
myStorageSettings.getTreatReferencesAsLogical().add("http://mysystem.com/ValueSet/cats-and-dogs");
```

Referential integrity

Referential integrity means that if one resource refers to another (like an Observation pointing to a Patient), the server will check that the referenced resource actually exists. If it doesn't, we'll get an error.

How to configure: https://hapifhir.io/hapi-fhir/docs/server_jpa/configuration.html

Search Result Caching

By default, HAPI caches search results for one minute, so repeating the same search (like Patient?name=smith) within that time will return the same results, even if new matching data was added.

-we can disable caching (globally or for a single request)

- We can also disable paging and limit results

***This also means that any new Patient resources named "Smith" within the last minute will not be reflected in the results.

Cascading Deletes

HAPI supports cascading deletes using a special interceptor called CascadingDeleteInterceptor. When we enable this on the server, we can automatically delete related resources by including either the _cascade=delete parameter in the request URL or the X-Cascade: delete header. This tells the server to also delete any resources linked to the one we're deleting.

Version Conflicts

When multiple clients update the same resource at once, the server may return a version conflict error (HTTP 409) to prevent lost updates. To reduce these errors, we can enable the UserRequestRetryVersionConflictsInterceptor, which lets the server automatically retry conflicted requests.

Controlling Delete with Expunge size

When deleting with expunge, resources are removed in batches sized according to the configured Expunge Batch Size setting.

Disabling Non Resource DB History

We can disable tracking history for non-resource data like MdmLink to improve performance, since enabling it creates extra database tables.

Prevent Conditional Updates to Invalidate Match Criteria

The JPA Server can be configured (disabled by default) to prevent conditional updates(updates that use search criteria rather than direct IDs) from invalidating match criteria, not only for the first resource version, but also for later versions.

Performance

History Counting

The `History Count Mode` setting in HAPI FHIR controls how the total count in history operations is calculated—using a cache (`COUNT_CACHED`), always accurate but expensive queries (`COUNT_ACCURATE`), or skipping the count entirely for performance (`COUNT_DISABLED`).

Bulk Loading

For servers that need to handle large data loads:

- **Tune Thread Pools**: Adjust the database and HTTP client thread counts based on your environment, to allow optimal concurrent writes and database connections.

- **Disable Deletes**: Turning off the FHIR delete operation can speed up data loading by skipping some reference checks during resource creation.

Disabling :text Indexing

The `:text` modifier in FHIR token searches (e.g., `code:text`) enables searching by human-readable display text, but indexing these values can consume significant storage and slow down write operations in large datasets.

To optimize performance, we are allowed to disable this feature either globally or selectively.

Disable Upsert Existence Check

When performing an upsert (client-assigned ID update/insert), the FHIR server normally checks if the ID exists via a `SELECT` query before inserting, which adds overhead.

By including the

X-Upsert-Extistence-Check: disabled

we can skip this check, improving write performance when you're certain the ID doesn't already exist.

However, if the ID does exist, the operation will fail with a database constraint error, preventing duplicates. This is useful for bulk data loads where IDs are known to be new.

Disabling Non Resource DB History

This setting controls whether the server tracks historical changes for non-FHIR database records like MDM links. Disabling it (by setting `Non Resource DB History = false`) improves performance by avoiding unnecessary history tables, though currently it only affects MDM operations. This is ideal when audit trails for non-resource data aren't required. FHIR resource history remains unchanged.

Enabling Index Storage Optimization

This setting can help us to save database space in large FHIR servers. When turned on, it stops storing some duplicate information in search tables for new records. This is useful when we have millions of records and we want to reduce storage.

Limitations:

1. Only works for new/updated records.
2. If we also track missing fields, we might need to add a special index.

3. Doesn't work with the "partition in search hashes" feature

---

Upgrade

HAPI FHIR JPA Upgrade Guide

When we upgrade HAPI FHIR to a newer version, we'll usually need to update our database structure too. The system includes the Migrate Database command which can be used to perform a migration from one version to the next.

To use it:

1. We need to always back up our database first.

2. Run the migrate-database command (available since version 3.5.0)

3. Specify our database type and connection details.

For large databases, we can add `--enable-heavyweight-migrations` for extra migration steps, though this usually isn't needed.

This tool can't migrate from very old versions (before 3.4.0).

Database Partition Mode

When using Database Partition Mode, we need to include the `--flags database-partition-mode` flag in our migration command to ensure proper schema setup for partitioned databases. This flag initializes new partitioned databases correctly and safely migrates existing partitioned schemas. However,we should not use it for non-partitioned databases—it cannot convert them to partitioned mode and may cause issues.

E.g.

./hapi-fhir-cli migrate-database --flags database-partition-mode -d POSTGRES_9_4 -u "[url]" -n "[username]" -p "[password]"

Database Migration

- **Before 4.2.0:** Used custom migration tool with from and to parameters.
- **Versions 4.2.0 to 6.1.x:** Used the Flyway library to manage and execute schema migrations. It tracked history in the FLY_HFJ_MIGRATION table.
- **Version 6.2.0 and later:** Replaced Flyway with its own internal migration mechanism.

    It still uses the same FLY_HFJ_MIGRATION table for tracking. When you run the migration command, HAPI FHIR:
    - Checks the table for previously executed migrations.
    - Compares that list to its own set of required schema changes.
    - Executes any new migrations that are missing, ensuring the database schema is up-to-date.

Diff Operation

The $diff operation can be used to generate a differential between two versions of a resource, or even two different resources of the same type.

Diff Instance

When the $diff operation is invoked at the instance level (meaning it is invoked on a specific resource ID), it will compare two versions of the given resource.

Parameters

- `fromVersion=[versionId]`: (*optional*) If specified, compare using this version as the source. If not specified, the immediately previous version will be compared.
- `includeMeta=true`: (*optional*) If specified, changes to Resource.meta will be included in the diff. This element is omitted by default.

To invoke:

```
GET http://fhir.example.com/baseR4/Patient/123/$diff
```

Diff Server

When the $diff operation is invoked at the server level (meaning it is invoked against the server base URL), it will compare two arbitrary resources of any type.

- `from=[reference]`: Specifies the source of the comparison. The value must include a resource type and a resource ID, and can optionally include a version, e.g. `Patient/123` or `Patient/123/_history/2`.
- `to=[reference]`: Specifies the target of the comparison. The value must include a resource type and a resource ID, and can optionally include a version, e.g. `Patient/123` or `Patient/123/_history/2`.
- `includeMeta=true`: (*optional*) If specified, changes to Resource.meta will be included in the diff. This element is omitted by default.

To invoke:

```
GET http://fhir.example.com/baseR4/$diff?from=Patient/1&to=Patient/2
```

LastN Operation

> This implementation of the `$lastn` operation requires an external Elasticsearch server implementation which is used to implement the indexes required by this operation. The following sections describe the current functionality supported by this operation and the configuration needed to enable this operation.

Functional Overview and Parameters

- **`subject/patient`**: The patient(s) to get data for. (Optional; if omitted, returns data for all patients).
- **`category`**: Filters observations by category (e.g., `vital-signs`). (Optional).
- **`code`**: Filters and, crucially, groups the results by the observation type (e.g., `blood-pressure`). The `max` parameter applies to each group. (Optional; if omitted, it groups by all codes).
- **`date`**: Filters observations by their date/time. (Optional).
- **`max`**: The number of most recent observations to return *per group* (per `code`). (Optional; defaults to 1 if not specified).

Critical Limitations

1. **Elasticsearch Version Lock:** Only officially supports Elasticsearch version 7.10.0.
2. **Limited Search Parameters:** Only the parameters listed above are supported. Others will be ignored or cause errors.
3. **Strict Grouping Rule:** The operation will not work correctly if an observation's `code` contains more than one coding. It requires a single code to group by.

Lucene/Elasticsearch Indexing

HAPI FHIR JPA requires a relational database but can add full-text search using either Elasticsearch or Lucene.

This enables two key search parameters:

- `_content`: Full-text search across all text in the resource.
- `_text`: Full-text search only within the resource's narrative section.

This does not replace the main database—it extends its search capabilities.

Performing Fulltext Search in Lucene/Elasticsearch

HAPI FHIR uses Elasticsearch or Lucene to power full-text search via the `_content` and `_text`parameters.

How It Works:

- Text is normalized during indexing: split into tokens, lowercased, and stripped of punctuation/plurality.
- Example: "Glucose [Moles/volume] in Blood" becomes tokens: `["glucose", "mole", "volume", "blood", ...]`
- Searches are normalized the same way. A search for `cancer OR tumor` matches tokenized content.

Exact Matches:

- The default cannot match exact strings with special characters (e.g., [Moles/volume]).

- Use the :contains modifier for literal substring matching:
  `GET [base]/Observation?_content:contains=[Moles/volume]`
  This bypasses tokenization and searches for the raw substring.

Customizing Full-Text Indexing

HAPI FHIR indexes full-text data automatically:

- **`_content`**: Aggregates all text from every string field in the resource into one searchable document.
- **`_text`**: Indexes only the human-readable narrative (`.text` div), ignoring tags.

Terminology

HAPI FHIR can validate codes using terminology resources (`CodeSystem`, `ValueSet`, `ConceptMap`) stored in its database. Load them via the REST API or the `hapi-fhir-cli upload-terminology`command.

Key Points:

- Versioning:
  - Each version of a `CodeSystem`, `ValueSet`, or `ConceptMap` is a separate resource, distinguished by its `.version` property.
  - Queries without a version use the most recent version.
  - Queries with a version use that specific version.
- Database Schema:
  The system uses special tables to manage terminology efficiently:
  - `TRM_CODESYSTEM` & `TRM_CODESYSTEM_VER`: Manage `CodeSystem` resources and link concepts to their specific versions. `TRM_CODESYSTEM` tracks the current version for each canonical URL.

- `TRM_VALUESET`: Stores `ValueSet` resources, uniquely identified by their URL and version.
- `TRM_CONCEPTMAP`: Stores `ConceptMap` resources for defining mappings between codes, identified by URL and version.
