SANTEDB / SANTESUITE

Software Requirements & Design Specification

Abstract

SanteSuite is a comprehensive, interoperable health systems software platform built on top of the powerful SanteDB iCDR. It provides all functions required to fully realize a patient-centered digital-health ecosystem.

Justin Fyfe et al.

justin@fyfesoftware.ca

1. Document Information

1.1.1. Revision History

Name	Date	Reason	Version
Justin Fyfe & Duane Bender (Mohawk College)	Dec 21, 2015	Initial Version	0.1
Justin Fyfe (Mohawk College)	February 1, 2016	Includes IMSI, changes to data model, and additional detail on the software interfaces from architectural spikes.	0.2
Justin Fyfe (Mohawk College)	April 28, 2016	Includes numerous implementation changes for the AMI, IMSI, and RISI.	0.3
Justin Fyfe (Mohawk College)	July 30, 2016	Include enhancements to the IMSI and FHIR sections better document the way that the OpenIZ IMS functions.	0.4
Justin Fyfe (Mohawk College)	October 24, 2016	Updated documentation to match the implementation with input from TIMR	0.5
Justin Fyfe (Mohawk College)	November 1, 2016	Updated documentation to include templates.	0.5.5
Justin Fyfe (Mohawk College)	November 20, 2016	Updated to include new instructions on AMI and IMS interfaces	0.6
Justin Fyfe (Mohawk College)	January 26, 2017	Updated design section to include PostgreSQL design specification	0.7
Justin Fyfe (Mohawk College)	March 9, 2017	Updated the design of the business rules engine.	0.7.5
Justin Fyfe (Fyfe Software Inc.)	2018-08-01	Initial fork from OpenIZ documentation.	1.0
Justin Fyfe (Fyfe Software Inc)	2018-11-25	Added and updated the security section and refactored more OpenIZ references to be more	1.1

		accurate after the code refactor.	
Justin Fyfe (Fyfe Software Inc.)	2019-04-30	Added new sections for the ADO.NET providers.	1.2

1.2. Related Documents

Related Document	Relevance
OpenIZ Design Specification (http://openiz.org/artifacts/1.0/OpenIZ%20-%20Design.pdf)	Basis of this document

1.3. License

2. Contents

3. Introduction

3.1. Purpose

The SanteSuite platform (http://santesuite.com) represents a comprehensive and cohesive software platform for realizing patient-centered digital health ecosystems. Built atop the powerful SanteDB platform, SanteSuite components are designed from the ground-up to interoperate with each other and with other software systems.

The platform provides a generic launchpad from which individual states or jurisdictions can design and customize their own software solutions that suit their use cases. However, SanteSuite also provides useful out of the box functionality that makes deployment faster and easier.

SanteDB uses an extensible, open architecture which allows for the addition of features such as materials management, analytics, authentication, outbreak management, internet of things, reporting & national data submissions, and much more.

Through this design and the implementation of plugins, it is envisioned that countries can select a package of features which work to achieve an appropriate solution for their environment. For example, a country may select a custom immunization forecasting logic module with a stock management module to support the query of immunizations and stock management capability.

3.2. Project Scope

The scope of SanteSuite is quite large, it is comprised of several platform components which further specialize the SanteDB core to perform a necessary function within a digital health ecosystem. At a high level, the components of SanteSuite are:

1. **SanteDB –** An extensible intelligent Clinical Data Repository (iCDR). SanteDB's core SDK and server infrastructure provides the necessary functions to match records, make clinical decisions, execute business rules, store and retrieve clinical data, perform security audits and privacy controls, operate on mobile and laptop devices offline, synchronize and resolve conflicts.
2. **SanteMPI –** A powerful master patient index software which leverages SanteDB's matching and MDM plugins to implement the features necessary to operate an MPI. SanteMPI provides the backbone of a master patient index and includes a series of plugins, configurations, and applets required to operate SanteDB "as a" MPI.
3. **SanteEMR –** The EMR project represents an enhancement to the SanteDB experience. It extends the SanteDB core user experience and makes it more generic and easier to implement custom primary care forms and templates.
4. **SanteInsight –** The insights tool provides a de-identified data stream to a standardized data warehouse copy of SanteDB. This allows donors and ministries of health to get real-time, de-identified, patient level reports for analysis.
5. **SanteGuard –** The guardian tool provides a complete audit repository solution for any connected SanteSuite product or any other product that uses IETF RFC3881, DICOM or FHIR audits.
6. **SanteGrid –** The grid solution allow for easy federation and peer-to-peer communication between SanteDB instances. SanteGrid provides a centralized table of contents for health data amongst connected peers and allows any connected SanteDB instance to query that index for patient information.

3.3. About The Clients

SanteSuite is a community initiative that will, potentially, produce a multitude of clients. The basis for the requirements used for the design of SanteSuite are a collection of those specified as part of projects that have been conducted around the world. This section will describe the general characteristics of such an environment.

The expected clients of this software range from low-to-medium income countries (LMIC) to individual states and provinces in developed countries. This broad base poses a variety of unique challenges. One of the primary challenges of public health professionals working in the field is that of reliable network connectivity and power. The solution shall take into account that feature-rich electronic devices may not be a viable solution and paper based or SMS based interfaces may be used as input into the system. This is true of LMIC but is also true of rural and indigenous regions in developed countries as well.

Furthermore, it is expected that the solution should be able to run on low powered, low cost hardware and server infrastructure. Great care shall be taken during the development and design stages of this project to ensure that optimal performance can be achieved on relatively low powered, low cost machines to achieve the widest possible adoption.

There can also be a severe shortage of “receptor capacity” or a capability on the part of implementing countries to deploy, manage and support these systems. Great care shall be taken in the documentation of the infrastructure, plugin architecture and installation procedures for the service. Where possible, it should be assumed that working knowledge of the underlying technical details is scarce. Multiple deployment options are also supported, including cloud options for greatest reach and custom local deployments where required to meet local legislative or other requirements.

3.4. Team Members

Being a community project, we seek to engage general partners where possible online. Upon commencement the team shall consist of the following members:

Table 1 - Team Members

Name	Role	Organization
Justin Fyfe	Architect / Lead Developer	Fyfe Software Inc.
Joseph Dal-Molin	Product Design & Evangelism	e-Cology Corp.
Paul Brown	Product Owner & Community Lead	Mohawk College
Nityan Khanna	Development	Mohawk College

4. Overall Description

This section provides a high level description of the system.

4.1. Alternative Products

Because the breadth of the SanteSuite product offerings, there are a multitude of alternative products. We believe that SanteSuite products should, at minimum, match the capabilities of these products and exceed them.

4.1.1. SanteDB Alternatives

SanteDB is an iCDR, this means that many regular CDR software products are alternative products. Two major alternatives have been identified:

- **SmileCDR** – An initiative which provides a fully functional FHIR CDR. SmileCDR offers features similar to SanteDB.
- **HEARTH** – An initiative from JEMBI health systems, which provides an open source FHIR CDR.

We feel that SanteDB offers several key advantages over these alternatives, namely:

- SanteDB does not use FHIR or any other messaging format as its storage format, it uses a more flexible clinical information model. Therefore SanteDB is insulated from changes to the FHIR standard or any other standard.
- SanteDB "speaks" standards other than FHIR. We have to be realistic, while FHIR has great potential in new developments, many environments are brownfield, meaning there are a multitude of systems which already leverage HL7v2, HL7v3, CDA, XDS, and other messaging formats. Rather than forcing these trading partners to upgrade to FHIR, we can interoperate with them on the CDR side.
- SanteDB provides insights and clinical decision support right out of the box. While other CDRs can be "hooked up" to a CDSS solution, there are several issues with this including "complete picture". A CDR may not wish to disclose that a patient has HIV, however a good CDSS system needs this information to make a determination of which vaccinations to suggest to a clinician. By integrating CDSS into the SanteDB platform, we allow the CDSS to make decisions based on the complete picture of the patient.
- SanteDB provides offline synchronization right out of the box. That is right, SanteDB's client SDK provides offline synchronization without the need for designing one from scratch. Developers simply implement their desired user interfaces in HTML5 and JavaScript and SanteDB takes care of the rest.
- SanteDB is multi-platform. We don't mean just "runs on Linux and Windows", we mean runs on Linux, Windows, MacOS, Android, Raspberry Pi, etc. Not only that but the platform can be customized to run on PostgreSQL, Oracle, FirebirdSQL, or SQLite. This allows maximum flexibility in adapting SanteDB to your licensing model and ecosystem.

4.1.2. SanteMPI Alternatives

Being a Master Patient Index, SanteMPI has a multitude of alternate products. Many are commercial offerings (such as Tiani, and Initiate), however this analysis will focus on just the open source or free alternatives.

- **OpenEMPI –** Which provides enhanced MPI functionality including graph based matching, multiple standards support and MDM functionality.
- **MEDIC CR –** The previous version of SanteMPI. This MPI provides only basic matching and storage of patient attributes, however has excellent standard support.

With SanteMPI, the major goals of the solution are to enhance the matching algorithms and incorporate machine learning algorithms such that SanteMPI provides a "better MEDIC CR". Included features to be targeted:

- SanteMPI must have an easy to use patient duplicate resolution user interface. That is to say, finding, resolving and marking duplicates should be made easy.
- SanteMPI must have a flexible matching algorithm / engine (based on SanteMatch) which applies the best practice record linkage strategies outlined here: https://www.ncbi.nlm.nih.gov/books/NBK253312/#
- SanteMPI must incorporate machine learning to "get better" at matching.

4.1.3. SanteEMR

SanteEMR has several open source and commercial alternatives. Some alternatives to SanteEMR are:

- **OSCAR –** Developed by McMaster University. While OSCAR pioneered many concepts of an EMR, its development has seemed to slow and the screens and user experience has become dated. Also, the solution is highly tailored to the Canadian market.
- **OpenMRS –** Developed by Regenstreif, OpenMRS is a world leader in terms of deployments of disease based forms.
- **OpenEHR –**

SanteEMR has several key advantages over the listed alternatives:

- It does not require any internet connection to function out of the box. Whereas OpenMRS requires specialized modules to perform individual functions offline, all of SanteEMR's functions can be performed offline without any changes to the underlying platform..
- SanteEMR synchronizes all business rules, clinical protocols, data, and schedules across server and client interfaces. This means that any new business rules, protocols, or data is automatically sent and executed by the connected clients.
- SanteEMR, being based on SanteDB's iCDR technology is capable of master data management functions (MDM) in which multiple sources of data can be maintained and aggregated to "one source of truth"
- The alternative products do not lay down a comprehensive, detailed security audit trail. SanteEMR supports detailed auditing of all user activities and these audits are shipped to the central audit repository for further analysis.
- SanteEMR support record level privacy controls and break-the-glass functionality. This functionality is supported via SanteDB's integrated IdP.

4.2. Project Principles

At a high level, this project seeks to:

- Provide developer extensibility and configuration points in all aspects of the service core,

- Provide extensive documentation for developers and users,
- Provide a disciplined approach for quality assurance, including the requirement of unit testing services on all modules in the project,
- Promote of code-reuse and standards wherever possible as integration points,
- Provide heavily normalized data model where journaling is a first class citizen and not an afterthought,
- Apply Security by design and privacy by design principles.

4.3. Project Features / Deliverables

As stated prior, the SanteDB project seeks to provide a series of highly customizable components into a series of deliverables that will be used as the basis for implementation. Envisioned as deliverables are:

1. **iCDR Backbone:** An extensible software solution that contains the majority of "online" logic required to perform immunizations, track and merge events from remote sites, perform stock management functions, etc.
2. **Administrative Portal:** A website that can be used by administrators to maintain the backbone configuration including customization of reports, stock items, antigens, etc.
3. **Disconnected Client Mobile App:** A mobile application that can operate offline for long periods and be used to collect immunization data within a clinic.
4. **Disconnected Client Desktop App:** A desktop based application which can operate in disconnected mode for long periods of time and be used to collect immunization data within a clinic.
5. **Disconnected Server:** A miniature server which is capable of servicing larger clinics which have a LAN connection but lack a WAN connection.
6. **Web App:** An application which allows online access over the internet.
7. **Developer SDK:** A development toolkit which will permit developers to create their own applications.

The need to customize edge devices speaks to one founding principle of the SanteDB project that is extensibility.

4.4. User Classes and Characteristics

This section will seek to introduce the user classes. A user class is not necessarily a technical role nor is it a single person, rather, it is a mechanism used to consider how a particular role is expected to interact with the system. A user class will have a series of skills, duties and concerns that will be addressed.

Table 2 - User Classes and Characteristics

User Class	Classifier	Priority
Clinic Staff	Low technical skill, expertise in delivery of health services.	VH
Receptionist	Low technical skill	M
Clinic Manager	Low technical skill, expertise in gathering statistics and managing stock.	H
Regional Manager	Low technical skill, expertise in gathering statistics and managing stock.	M
System Administrator	High technical skill, expertise in systems management.	M
Governing Authority	Medium technical skill, expertise in funding and strategy.	M

Audit / Privacy Officer	Medium technical skill, expertise in privacy legislation.	M
Application Developer	High technical, expertise in creating new extensions.	M
Patient	Technical skill unknown, expertise in adhering to appointments	M

4.4.1. Clinical Staff

The clinical staff user class is defined as an individual whom performs health delivery tasks at local clinics and is responsible for clinical observation (such as adverse reactions, weight, etc.).

4.4.1.1. Skills

The clinical staff within a clinic is typically quite capable of performing duties related to health delivery. The staff member will typically have moderate to low technical skill and is expected to be able to use a simple user interface to enter simple fields like date/time of an action and the result of their action (for example, date/time of immunization and the result or date/time of weight and the measure).

4.4.1.2. Duties

The clinic staff is often busy and the use of the application is often a tertiary duty. It is important that the clinical staff not be distracted from the primary duties of ensuring healthy patient population.

4.4.1.3. Concerns

The clinical staff may have many concerns with the technical solution, including:

- Time, the application must not introduce a bottleneck in the process of caring for patients.
- Accuracy, the application must provide an accurate depiction of the patient’s medical status including weight, history of medical status, and correct demographics.
- Confidentiality, the application must provide a mechanism that ensures vital information such as immune compromised is shown the officer but not shown to other generic users

4.4.2. Receptionist

The receptionist user class represents an individual who is responsible for preparing the visit by performing tasks such as demographics collection/updates, scheduling of future appointments, etc. The receptionist may be the same person as the clinical staff in some low resource settings, or may be a separate person representing the clinic.

4.4.2.1. Skills

The receptionist is expected to have lower technical skill than the clinical staff, however be adept at using simple technology such as filling in form fields, reading a calendar and following recommendations presented.

4.4.2.2. Duties

The receptionist’s primary duties include:

- Onboarding new patients that arrive at a service delivery location, or updating existing patient demographics at presentation
- Ensuring identification is accurate and up to date, including immunization cards and/or health insurance information
- Notifying the clinical staff that the patient has arrived, or placing the patient into the clinic’s “waiting room”, confirming with the patient the purpose of the encounter.

4.4.2.3. Concerns
The receptionist many concerns with the technical solution, including:

- Time: the application must not introduce a bottleneck in the process of queuing patients.
- Difficulty: the application must not be difficult to use and/or present confusing dialogues or options to the receptionist.
- Accuracy: the application must provide validation that ensures that the keystroking of the receptionist does not cause a faulty record.

4.4.3. Clinic Manager
A clinic manager user class represents an individual who is responsible with the operation of a single clinic and ensures that the clinic has sufficient stock perform daily functions, running reports to ensure adherence, etc.. The clinic manager is also responsible for the transfer of stock to/from a regional distributor, managing the clinics stock balance and ensuring that any technology used within the clinic is not operating in an error state.

4.4.3.1. Skills
The clinic manager is expected to have higher technical skill than the clinical staff does, and should be adept at stock counting, basic math (such as converting vials into doses) and reporting.

4.4.3.2. Duties
The clinic manager's primary duties include:

- Performing stock counts at set intervals and reporting current stock to a regional authority.
- Ordering and receiving stock into the facility and returning expired/unusable stock to another authority.
- Ensuring that any technology used in the clinic is operating in a non-faulted state and that any technology is ready for patient care duties (charged and functional).
- Preparing/running reports which relate to the operation of the clinic such as stock forecasts (days of remaining stock), number of patients seen, number of outstanding technical issues (failures, etc.), number of patients expected in the coming days, etc.

4.4.3.3. Concerns
The clinic manager may have concerns with the technical solution, including:

- Accuracy: The solution must provide accurate information (as is possible) related to the current stock and function of the clinic such as patient population, forecasted patients and consumption of stock, etc.
- Communication: The solution must provide a mechanism for ordering stock in a manner that the manager can see the status of their order can place additional orders, and view balances.
- Security: The solution must provide a secure access layer that prevents unauthorized access to lost/stolen tablets and must ensure the clinic manager does not view information they are not permitted to view.

4.4.4. Regional Manager
A regional manager is an individual who is responsible for the management of a collection of clinics within a specific region such as district, county, city, province, etc. The regional manager is responsible

for ensuring sufficient stock is available for the clinics in their region, and ensuring that care coverage meets specified targets.

4.4.4.1. Skills

A regional manager may be of moderate technical skill and, it is expected, should be able to interpret reports and manipulate data within a basic tool like excel, print reports and scan.

4.4.4.2. Duties

The regional manager’s primary duties include:

- Organizing stock orders, packing them for subordinate facilities and delivering or facilitating pick-up of the orders.
- Running reports for their district/region and adjusting stock, vaccination campaigns, or outreach programmes.
- Reporting to higher echelons of administration the performance of their region.
- Ordering stock from national distributors of vaccine and ensuring sufficient safety stock to supply their region.
- Device provisioning including onboarding of new users and devices for use within their region and ensuring lost devices are purged/located and user accounts locked under correct conditions.

4.4.4.3. Concerns

The regional manager may have concerns with the technical solution, including:

- Accuracy: The solution must provide accurate information (as is possible) related to the current stock and function of clinics such as patients seen, forecasted patients and consumption of stock, etc.
- Communication: The solution must provide a mechanism for relaying status of a stock order in a manner that the manager can see the status of their order, can place additional orders, and view balances, pick stock from their current store, etc.
- Security: The solution must provide a secure access layer that prevents unauthorized access to lost/stolen tablets and must ensure the regional manager does not view information they are not permitted to view such as discrete data.

4.4.5. System Administrator

A system administrator is an individual who is responsible for the planning, setup and maintenance of the solution.

4.4.5.1. Skills

The system administrator is typically a highly skilled individual who is responsible for the maintenance of several software systems within a region/district/country. The administrator in some LMIC may have less technical skill but still be familiar with basic database terminology, practices, etc.

4.4.5.2. Duties

The system administrator’s duties include:

- Backup of computer databases which contain PHI
- Maintenance of security accounts, and devices permitted to log in within a particular region.

- Setup and installation of software components and their upgrades including networking configuration.
- Advanced technical support, analysis of log files, and other diagnostic tool output.
- Planning of physical architecture, deployment timelines, OID registrations, etc. including the issuance and revocation of PKI certificates.

4.4.5.3. Concerns

A system administrator will typically have several concerns with technical solutions that may include:

- Security: What impact on the network surface area will the solution have, and how will the solution adversely affect the operation of other systems within the enterprise.
- Reliability: What is the reliability of the solution and the burden on the administration that technical support calls will place on resources?
- Cost: What is the cost of maintaining the infrastructure after initial capital costs? What are the costs related to the installation of the system and what, if any, are the licensing impacts on the operating budget and IP of other systems in the network (example: GPL)
- Auditability: What is the traceability of the solution? How easy are deployment mis-configurations to find and diagnose? How difficult are logs to obtain? Do the logs contain sufficient information to quarantine data and/or users and machines in case of breach?

4.4.6. Governing Authority / National Officers

National authority / officers are individuals who are responsible for the planning and maintenance of a national health programmes. These individuals typically have moderate technical skills and are primarily interested in stock management and reporting functions.

4.4.6.1. Skills

A national officer has data analytics skills and moderate technical skill required to customize reports and manipulate data in Excel. A national officer or programme coordinator may be interested in customizing reports themselves.

4.4.6.2. Duties

The national officer's primary duty is the running of secondary use reports and the leveraging of these reports to perform business intelligence functions. The governing authority is also responsible for the administration and creation of legislation.

4.4.6.3. Concerns

The national/governing authority's primary concern will be that of provenance and governance capabilities of the system. The national officer may be responsible for ensuring that new legislation passed can be implemented within the system and that reports are accurate and up to date.

4.4.7. Audit / Privacy Officers

Privacy officers are individuals who are responsible for the implementation and adherence of the system and its users to policies configured for the jurisdiction. The privacy officer is also concerned about security breaches, performing spot audits to ensure that users are using the system correctly.

4.4.7.1. Skills
The privacy officer is of moderate technical skill, and high domain expertise skill. The privacy officer has the ability to use Microsoft office products, as well as basic BI tools and web-interfaces for detecting security breaches.

4.4.7.2. Duties
The primary duties of the privacy officer include the setup and validation of configured policies within the core application as well as performing routine privacy audits on the system. The privacy officer is also responsible for participating in threat risk assessments and privacy impact assessments.

4.4.7.3. Concerns
The primary concerns of the privacy officer are that the system will enforce consent policies imposed by the deployment jurisdiction, and that any overrides are easily identifiable in any audit logs. The privacy officer will also be concerned with the detail of PIA and TRA assessments performed against the system and will require lots of documentation related to the security services provided by the system.

4.4.8. Application Developer / Implementer / Technical Expert
The application developer, implementer and technical expert class is used to describe those professionals who will be developing integration points for the SanteDB system for the purpose of implementing the solution in a jurisdiction.

4.4.8.1. Skills
The technical expert is of high technical skill and is able to understand application programming interfaces (API) documentation and how these APIs can be used to control the system for the purpose of their implementation.

4.4.8.2. Duties
The technical expert's primary duties include the development and customization of the SanteDB solution, as well as the deployment and configuration for a particular deployment. Technical experts may also develop plugins and/or consumer applications of the platform.

4.4.8.3. Concerns
The technical expert's primary concern is that of ease of implementation and integration of the solution. The technical expert expects the system to provide sufficient application programming interface hooks in order for them to sufficiently expand the system to complete whatever implementation work they are performing. The technical expert will also be concerned with the stability and robustness of documentation of interfaces as well as the performance of the system and availability of development tools.

4.4.9. Patient
The patient class is used to describe the consumer of the healthcare services or one of their delegates. This may include parents, relatives, guardians, etc. While patients are directly users of the system per-se, they may play a role in the use of personal portals into the system.

4.4.9.1. Skills
Patients may be of varying skill from complete computer illiteracy, to high technical shrewdness. Patient interfaces should use simple language and only display the necessary information for the patient to understand the data that they are viewing.

4.4.9.2. Duties
The primary duties of the patient class are the attending of appointments registered in the system, and the obtaining of proper patient identification to be identified within the system.

4.4.9.3. Concerns
Primary patient concerns involve the proper and accurate identification of data, the cleanliness (clarity) of interfaces and confirmation prior to submitting any changes to the system.

4.5. HDS Platform
The design of the backbone is not platform specific and could be implemented in a number of different ways. The initial version discussed in this design document will be implemented using the Microsoft server stack, making use of the following technologies:

Table 3 - HDS Implementation Platforms

Technology	**Reasoning**	**Relates To**
Microsoft .NET Framework	Execution Environment	Backbone, Web Interface, Administrative Interface
Microsoft SQL Server 2014	Database Environment	Backbone
PostgreSQL Server 9.4.x	Database Environment	Backbone
MEDIC Service Core Framework	Robust set of existing plugins available.	Backbone

There is future plans to upgrade the WCF based service core services into the MSIL implementation of HTTP handlers. This will permit operation of HTTP interfaces on Linux and MacOS X.

4.6. Web Portal Operating Environment
The web portal operating environment for SanteDB will leverage NancyFX. NancyFX can operate in a standalone manner, removing the need to setup IIS or other supporting infrastructure.

4.7. Mobile App Operating Environment
Reference mobile apps will be created for both the providers and the patients. To achieve the maximum possible device support Xamarin will be used as the platform of choice. A wrapper in Xamarin will load and execute HTML5 and JavaScript applet files.

4.8. Assumptions and Dependencies
The primary risk to implementation is the use of proprietary components upon which the stack will be based. To achieve the lowest cost deployment for LMICs, components that cannot be licensed as free and open source shall be avoided where possible.

5. Requirements

This document outlines several basic use cases upon which the SanteDB platform was created. The requirements listed here are a target of requirements that a system using the SanteDB platform could perform.

5.1. User Stories / Use Cases

5.1.1. User Logs In With Valid Credentials

A clinical staff member uses the mobile app to log into their provided user account. The device has an active internet connection. Upon login the device sends a unique device identifier to the central authentication system proving its identity. If the user credential is valid, and the device credential is valid, then the system audits the successful login.

5.1.2. User Logs In With Invalid Credentials

A clinical staff member uses the system to log into their provided user account. During the login process one of the credentials provided to the system (device or user) is invalid. The system alerts the user to the invalid credential condition and audits the invalid access attempt.

Alternate: The user continues to provide invalid credentials. After the third invalid login attempt the device does not permit another attempt for a 60 second period effectively locking the device.

5.1.3. User Resets their Password

A clinical staff member wishes to log into the device, however forgets their password. The staff member uses the system to reset their password from a registered device. The user enters their username and selects a method of reset (e-mail or SMS) providing the necessary security check. The user receives an out of band code that they enter into the forgotten password system. The forgotten password subsystem validates the out of band code provided to the user with the token generated and, if valid, permits the user to enter a new password.

5.1.4. User Reviews Appointments

If login is successful, the clinical staff member is presented with a dashboard portal. The user uses the dashboard to review the appointments in the system. The system provides a list of appointments for review and filtering by the user of the point of service device. The system does not disclose appointments for patients outside the clinical staff members's responsibility (to be applied by policy, facility, etc.)

5.1.5. User checks-in existing Patient

A patient presents to the clinic for a routine service. The receptionist scans their identification which performs an identification search within the system. The receptionist verifies the information and proceeds to "check-in" the patient.

Alternate: The patient presents without an identification card, however has attended the clinic before. The receptionist uses the system to search the local clinic's user database.

5.1.6. User copies remote demographics into the system

A patient presents to the clinic for their routine service. The patient has never presented to the clinic before, however has a national identification card. The patient presents this which is used by the receptionist to download the patient's demographic data. The system queries the national identification system and presents a series of results to the user. The receptionist selects the appropriate record and indicates that the patient should be imported.

The receptionist continues to check-in the patient.

Variation: The receptionist updates the patient's demographic data and submits the changes. The system conveys this change to the national patient registry.

5.1.7. User registers a new Patient

A previously unregistered patient presents to the clinic. The patient has never visited the clinic before and does not have a national or regional identification from another clinic. The receptionist gathers the user's demographic details and enters them into the system. The receptionist saves the new demographic record which results in a new patient record within the system. The receptionist saves any existing clinical data the patient has in the system. The system calculates a care schedule for the patient and schedules appointments if necessary. The receptionist reviews the created schedule, and if necessary, continues to check-in the patient.

Variation: This new identification is posted to the national records system which results in a new jurisdictional identifier for the patient.

Variation: The new patient's demographics exactly match the demographics of another patient already registered in the SanteDB system. The receptionist is shown a warning confirming that this is in fact a new patient or if the patient is a duplicate.

Variation: The new patient's demographics exactly match the demographics of another patient already registered in the SanteDB system. The patient is registered. At a later time, a district officer retrieves a list of conflicts and resolves the duplicate record. The duplicate record's clinical history data is copied into the new patient master file.

5.1.8. Patient presents and is past due / has no appointment

A patient presents to the clinic without an appointment, or is late for an existing scheduled appointment. The receptionist uses the system to look up the patient's records, and looks at the missing (past-due) events within the system. The receptionist requests the system to generate an on-demand care plan/recommendation, the system creates an appointment with the past-due events scheduled on the current date. The receptionist checks-in the patient for the created appointment.

Variation: The system displays the past due vaccination and automatically generates an updated schedule and displays any warnings if appropriate (ex: some vaccines may be unsafe or may require a different dosing).

5.1.9. Patient presents and provides new demographics

A patient presents to the clinic for their routine immunization. The patient informs the receptionist that their demographics information (phone number, address, etc.) has changed. The receptionist keys the changed data into the system and saves the patient's demographic information.

5.1.10. Clinical Staff performs encounter

After being checked in, the patient waits in the "waiting room" for some amount of time. The clinical staff member calls the patient into a private area to discuss their medical history, and reviews the actions to be taken for the specific encounter. The physician starts the encounter recording measurements (such as height, weight, etc.) and adjusts the list of actions to be performed based on what is considered safe.

5.1.11. Patient has an adverse reaction after encounter
After receiving an immunization, the patient is instructed to wait a certain time period before leaving the clinic (discharge). During this time the patient develops a rash/fever/other reaction. The clinical staff member uses the system to record the adverse event (i.e. updates the immunization encounter).

Alternative: After going home, the patient starts to develop an adverse reaction to the vaccine given during a previous encounter. The patient returns to the clinic. The Clinical Staff Member amends the previous encounter entering an adverse reaction.

5.1.12. National officer enables a new application
After reviewing an application on the mobile application store, the national officer decides that an application meets criteria for a need within their jurisdiction. The national officer enables the application on their service by allowing the application key and selecting the user roles / application functions that the application is allowed to operate. This information is communicated to the SanteDB backend where it is then distributed to all connected mobile devices.

5.1.13. District / Regional / National Officer runs summary reporting
A regional officer wishes to determine the performance of their immunization programme within their jurisdiction. The officer uses the reporting engine of the solution to run a series of reports which illustrate the performance of their region.

Alternate: The national officer uploads a new report to the reporting engine and selects which users may view the report and specified parameters they are permitted to view.

5.2. Other Non-Functional Requirements

5.2.1. Performance Requirements
List any performance requirements if available. State any performance requirements and their rationale. This will help implementers understand the intent and make suitable design choices.

1. The system SHALL be capable of performing simple queries and returning resources from the local data storage device in a reasonable amount of time.
2. The system SHALL provide a mechanism for compressing inbound and outbound data.
3. The system SHALL provide a mechanism for fragmenting and bundling data. The system SHALL allow consumers to dictate how this bundling occurs in order to minimize traffic. This requirement is waived when standardized interfaces are implemented.

5.2.2. Safety Requirements
Specify requirements that are concerned with the possible loss, damage or harm that could result from the use of the deliverables of this project. Refer to any policies that are being enforced.

4. The system SHALL persist all outbound messages and SHALL track the response to outbound messages. Unsuccessful messages SHALL be flagged and the system SHALL provide a mechanism for re-sending data.

5. The system SHALL persist all inbound messages and their responses for any operation which modifies data. The system MAY persist the entire inbound request and/or response message but SHALL at least persist the unique message identifier. This functionality is related to exec-once requirements.
6. The system SHALL NOT communicate PHI over unsecured channels and SHALL reject any messages which are not sent over encrypted channels.
7. The system SHALL use node authentication when communicating with other infrastructure components. Node authentication SHOULD be used for end-user devices.
8. The system SHOULD use a local root authority for node authentication purposes but SHALL at minimum allow the trusting of a list of certificates if a root authority is not supported.

5.2.3. Security Requirements

Identify any requirements related to security or privacy issues. Define any user identity authentication and authorization requirements. Refer to any policies or regulations that are being enforced

5.2.4. Quality Assurance Requirements

Specify any quality characteristics of the software that are important to either the implementer, or customer. Some examples are: adaptability, availability, correctness, flexibility, interoperability, maintainability, portability, reliability, reusability, robustness, testability, and usability. Write these to be specific, quantitative and verifiable requirements when possible. At the least, clarify the relative preference for various attributes such as ease of use over ease of learning.

6. Interface Considerations

This section outlines the requirements of any external interfaces required to implement the project.

6.1. User Interfaces

Describe the logical requirements of a user interface that are required. This may include prototype screen captures, diagrams, product style guidelines, layout constraints, standard buttons that will appear on screens. Keyboard shortcuts and error message standards may also be listed here.

6.2. Software Interfaces

Identify any software interface that this project will provide. Include database services, libraries, tools, and integrated components.

Table 4 - Software Interfaces

Software Package	Type	Provider	License
Microsoft AjaxMin	Class Library	Microsoft Inc.	MS-PL
Newtonsoft JSON.NET	Class Library		
NHAPI	Class Library		
Everest Framework	Class Library	Mohawk College	Apache 2.0
AtnaAPI	Class Library	Mohawk College	Apache 2.0
XDSApi	Class Library	Mohawk College	Apache 2.0
Antlr3	Class Library		
ExpressionEvaluator	Class Library		
RestSharp	Class Library		
StackExchange.Redis	Class Library		
Twilio.Api	Class Library		
SQLite.NET	Class Library		
SQLCipher	Class Library		
SwaggerWCF	Class Library		

6.3. Communications Interfaces

Describe any requirements associated with communications functions required by this product. This could include e-Mail, web-browsers, network server communications, protocols, etc…

Table 5 - Communications Interfaces

Interface	Service	Method / Standard	Provider
Health Data Management	SanteDB iCDR	FHIR STU3 / HDSI	SanteDB
Clinical Protocol Management	SanteDB CDSS	FHIR STU3 / HDSI	SanteDB
User Accounts	SanteDB IdP	OAUTH	SanteDB
In-Application Reporting	SanteDB ReportR	SQL	SanteDB
FHIR Service Core	Core FHIR Services	FHIR DSTU	MEDIC SVC Core
Auditing	ATNA Auditing	ATNA + DICOM	SanteGuard
HMIS Reporting	TBD	TBD	TBD
Patient Identity Source	Patient Identity	PIX	SanteMPI

Patient Identity Consumer	Patient Identity	PIX	SanteMPI
Patient Demographics Search	Patient Identity	PDQ	SanteMPI

7. Solutions Architecture

7.1. Solution Architecture

SanteSuite / SanteDB provides a loosely coupled open system architecture. Figure 1 illustrates the major components of the platform where each bidirectional arrow represents a communications channel over an open standard.

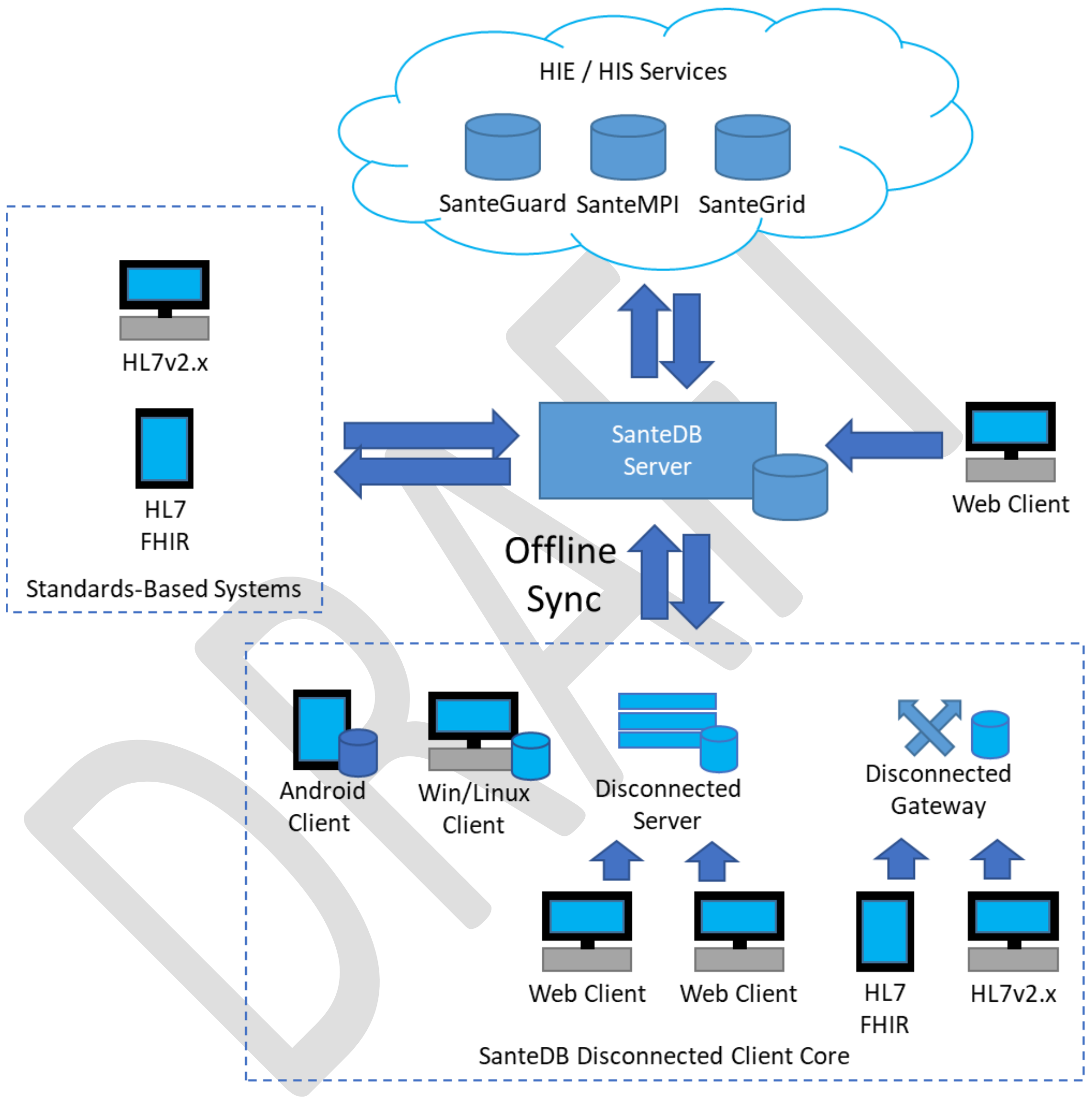


Figure 1 – SanteSuite / SanteDB System Architecture

The major components of the architecture are:

- **SanteDB Server (iCDR)**: The iCDR is the primary platform component of the SanteSuite platform. The SanteDB iCDR is responsible (at a high level) for:
    - Maintenance of individuals' medical records
    - Scheduling and maintenance of medical appointments
    - Forecasting schedules and demand

    - Integration with infrastructural systems such as Logistics Management Information Systems (LMIS), Health Management Information Systems (HMIS), educational systems, etc.
- **SanteDB Disconnected Client Core (dCDR):** These software pieces represent the offline capacities of the SanteSuite / SanteDB platform. These include:
    - **SanteDB Disconnected Client:** A thick client application that operates offline and hosts SanteDB applets and applications. This client operates on Android, Linux, Windows and provides complete miniature version of the iCDR for use offline, synchronizing data when appropriate.
    - **SanteDB Disconnected Gateway:** A version of the Disconnected Client which exposes standards based interfaces (no user interface) for applications that would otherwise require an internet connection, to function appropriately using FHIR or HL7v2.
    - **SanteDB Disconnected Server:** A version of the disconnected client that can be used at clinics which require local connectivity (between systems) while not being connected to the internet (LAN but no WAN)
- **Standards Based Systems:** Represents third party, existing clinic assets such as admitting systems, EMRs, etc. which integrate directly with the iCDR using one of its many standards based interfaces.
- **SanteDB Applications:** Represent applications such as EMRs, HISs, Mobile Applications, and custom websites which use the HDS to convey data to end users. This also include s the reference implementations of the patient and provider mobile applications.
- **SanteSuite HIE Offerings:** These are specialized, purpose focused instances of SanteDB's iCDR to perform a particular function.

7.1.1. SanteDB / SanteSuite Pre-Packaged Solutions

SanteDB operates at the core of the SanteSuite product offerings. SanteSuite community assets are then customized for particular operational contexts.

- **SanteEMR:** Is a fully functioning offline-first EMR leveraging SanteDB's matching, storage, privacy and security controls to offer a complete clinic management solution.
- **SanteMPI:** Is a fully functioning Master Patient Index (MPI) which leverages SanteDB's powerful standards based interfaces and matching plugins to operate as an MPI.
- **SanteGuard:** Is a fully functioning security audit repository which leverages SanteDB's data storage layer and communications capabilities to operate as a fully functioning RFC-3881, DICOM or FHIR R4 security audit repository.
- **SanteInsights:** Is a reporting plugin solution that allows SanteSuite products (via SanteDB) to automatically de-identify inbound data and submit to a centralized data warehouse service.
- **SanteGrid:** Is a federation solution which allows multiple SanteDB services and products to be federated geographically, by program area, etc.

7.2. Network / Physical Architecture

The SanteDB iCDR is designed to support a wide range of deployment options. This will increase the scalability of the solution across environments. There are envisioned to be three types of deployments supported by the SanteDB infrastructure:

1. Single Server – In this deployment all necessary functions run on one physical or virtual server. This will be the default installation supported for developer installations and staging environments.
2. Simple Multi-Server – In this deployment functional components are split across servers to balance load. This deployment will see use of multiple application servers, multiple database servers, and shared memory caching.
3. Federated – In this deployment a series of HDS environments are linked together in a federation of servers. This type of deployment is an envisioned future state.

7.2.1. Single Server Deployment

The single server deployment option simply places the CDR, caching, databases, and supporting tools onto one server. A sample single server deployment is illustrated in .

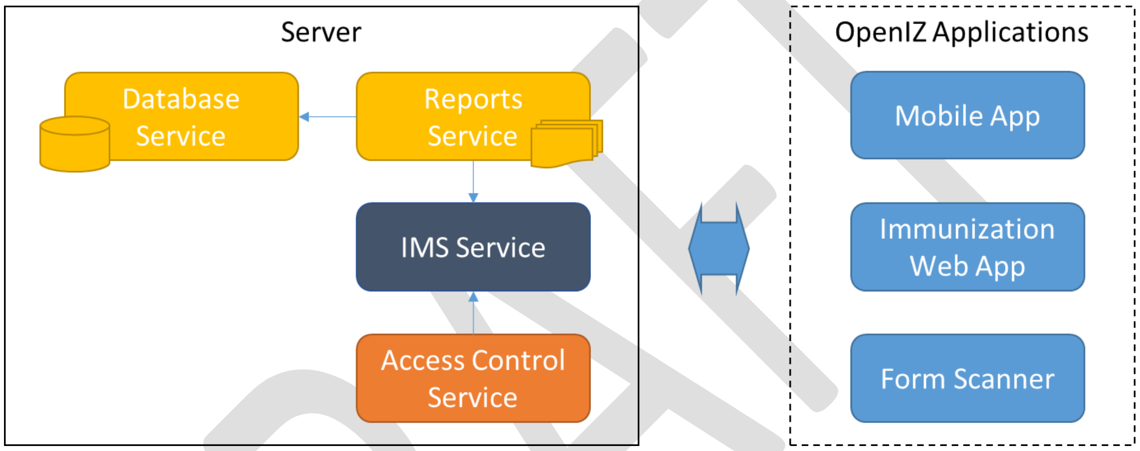


Figure 2 - Single Server Deployment

7.2.2. Multi-Server Deployment

A multi-server deployment of SanteDB is also supported. Each of the application server pieces has been designed with the goal that they can split apart based on role. A multi-server deployment will require some planning and will depend solely on the environment into which the service is being deployed.

Illustrates a simple multi-server deployment whereby application services are split across physical servers and the database is not scaled.

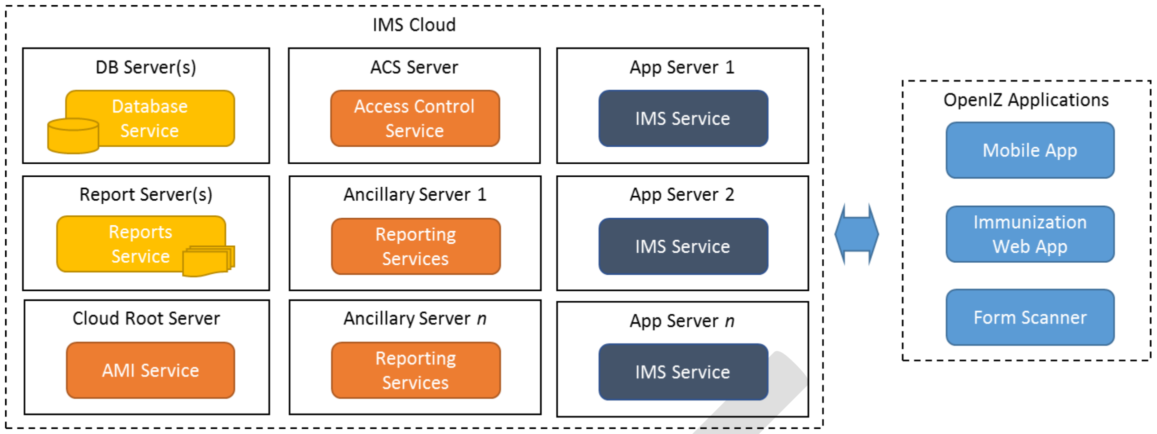


Figure 3 - Multi Server Deployment Example

This form of deployment can also be scaled out to meet a much larger environment as well. Illustrates the extreme stress test environment (ESTE) used for SanteDB testing with approximately 4 million fake patients. The ESTE environment illustrates database, application, and ancillary server scale out opportunities.

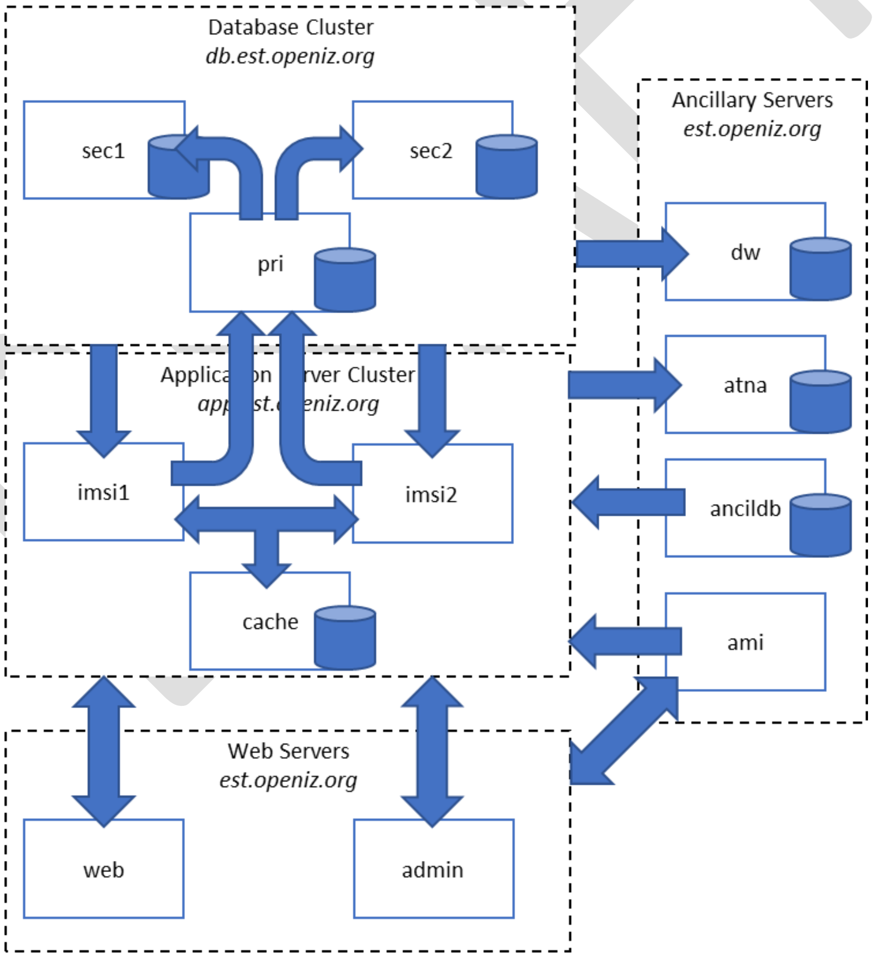


Figure 4 - High capacity scale-out

7.3. Software Architecture

7.3.1. SanteDB's Clinical Data Repository Architecture

The CDR portion of SanteDB is based heavily upon the micro-services architecture. In this architecture, a series of pluggable services implement a series of contracts. Whenever a function unit wishes to perform a unit of work it will ask the host context (IServiceProvider) to get the currently configured service provider.

The service types provided by SanteDB's HDS are illustrated in Figure 5.

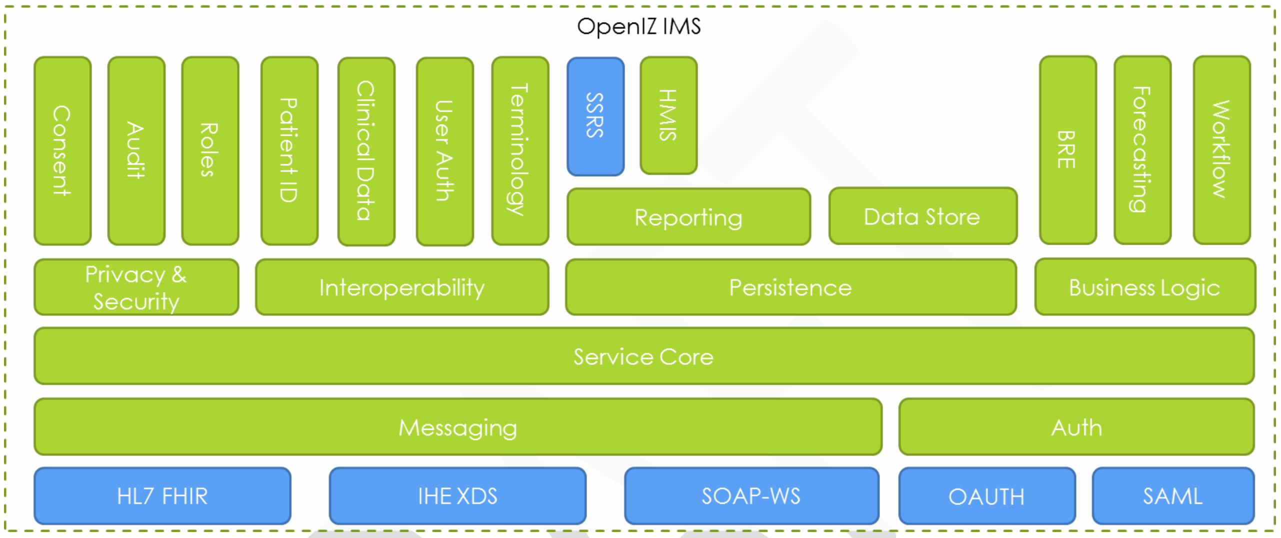


Figure 5 - SanteDB HDS Component Architecture

Each service is described in more detail in Table 6 with those services provided by the MARC-HI Service Core framework marked in red.

Table 6 - SanteDB HDS Services (To be updated)

Services	Contract	Description
Messaging	IMessageHandlerService	The message handler service is started upon application start/stop and is used to receive messages, parse them into a canonical form.
	IMessagePersistenceService	The message queue service allows messaging services to queue inbound messages that need re-processing.
Authorization	IIdentityProviderService	Responsible for authorizing and interacting with the identity management system configured in SanteDB. For example, if LDAP authentication was preferred then there would be an LdapIdentityProviderService
	IDeviceIdentityProviderService	Responsible for authorizing and managing the principals related to security device and node authentication.

	IApplicationIdentityProviderService	Responsible for authorizing and managing application principals (OAUTH Client keys) to be used for validating third party applications can communicate with SanteDB.
	IRoleProviderService	Responsible for maintaining and managing roles on the identity provider.
ServiceCore	IStockManagementService, IMailMessagePersistenceService	These clinical data services are responsible for the orchestration of underlying functions to perform the specified operations they define. For example: The appointment scheduling service would be responsible for finding recommended dates for a particular clinic.
	IConceptService	The concept management service is responsible for managing the internal concept dictionary found within the SanteDB database.
Consent	IPolicyDecisionService	The policy decision service is responsible for ultimately deciding the outcome of the policies registered for a particular object (called a securable) and telling the enforcement service how it thinks the data should be handled (Grant, Elevate, Deny)
	IPolicyInformationService	The policy information service is responsible for maintaining the linkage between policies and securables.
	IPolicyEnforcementService	The policy enforcement service is responsible for the actual enforcement of the policy. The PEP is responsible for masking data, raising audits, blocking access.
Audit	IAuditRepositoryService (storage)	The audit repository service is responsible for storing local copies of audits in the SanteDB server that is running. The audit repository service allows for querying and insertion of audits
	IAuditorService	The auditor service is responsible for shipping audits to a central audit authority.
Repositories	IRepositoryService<>	The repository services are responsible for the storage and business logic steps of storing a retrieving data to/from the data layer. All presentation layers use the repository layer to interact with objects.

Data Store	IDataPersistenceService<>	The data persistence service is responsible for taking the internal canonical model of the SanteDB HDS and translating that data into the physical data storage unit.
Forecasting	IClinicalProtocolRepository IClinicalProtocolService	The forecasting service is used for creating care plans and for maintaining and managing the central list of clinical protocols that can be used in the SanteDB platform.
BusinessRules	IBusinessRulesService<>	The IBusinessRules services provide an opportunity to alter the behavior of SanteDB within the context of an applet. IBusinessRules support being executed on certain data triggers (Before/After) (Insert/Update/Obsolete/Query/Retrieve).
Notification	IClientRegistryNotificationService	The notification service is used to alert other systems of real-time data storage events. This is the hook that will most likely be used when implementing pure ODD in SanteDB HDS or when merges or patient registrations occur.

Figure 6 illustrates each of the services contained within SanteDB server and their relationship to one another.

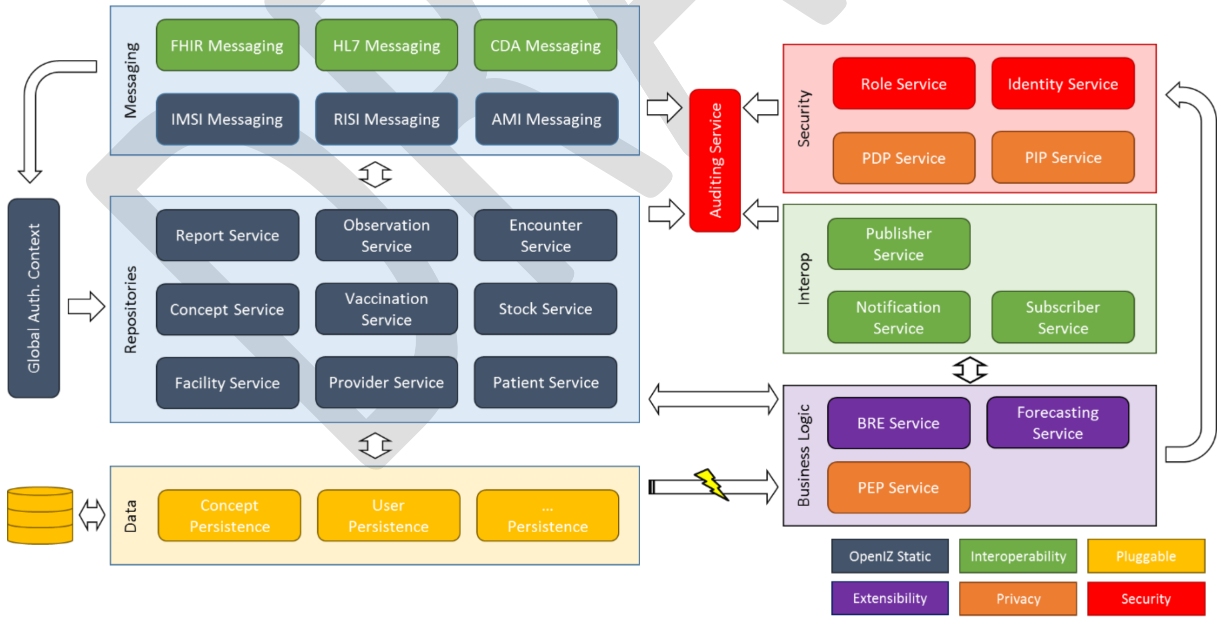


Figure 6 - HDS Component Execution Flow

7.3.1.1. Daemon Services

The IDaemonService is not a service contract per-se, rather it is a scaffold interface (contract) which can be used by services which need to operate as a daemon within the SanteDB application context.

Daemon services are started as application context start and shut-down at application context stop. Daemon services are started in the order in which they appear in the application host's configuration file. If daemon services require another daemon to be started they can subscribe to the dependent daemon's "Started" event.

7.3.1.2. Job Services

A job service represents a piece of C# code that can be executed at-will by an administrator. Examples of job services include:

- Exporting data from the main database to a data warehouse
- Synchronizing data from SanteDB to another system that may or may not support messaging
- Creating a global, country-wide forecast of a particular care protocol

Job services implement the IJob interface. Jobs can declare the parameters (types and name of parameters) that they support. Parameters will be exposed/collected from the user prior to executing the administrative job.

7.3.1.3. Timer Services

Timer service jobs are implemented via the ITimerJob interface. Their schedules are dictated by the timer service's configuration mechanism. The default timer job defined in the MARC-HI ServiceCore framework uses the application configuration file to manage the execution of timer jobs. SanteDB may implement additional functionality in future releases to allow database based configuration to occur.

7.3.1.4. Business Rules Services

The business rules services are responsible for the execution of business rules based on system events. There are two types of services which can be classified as business rules:

1. **Event Based:** These services implement IDaemonService and subscribe to system events for which they are interested. For example, a BRE which validates a user's password is of correct length would subscribe to the PasswordChanging event of the IIdentityProvider.
2. **Explicit Call:** These services implement the IBusinessRulesService<T> and is executed on-demand. These type of business rules services are only called from repository services that require explicit business functions to be performed. An example of this would be an IBusinessRulesService<Patient> which may provide functions to detect merges, or validation functions

By default, SanteDB has a basic business rules engine service which provides access to both services as JInt engine.

7.3.1.4.1 JavaScript Business Rules Engine

The JavaScript business rules engine allows implementers to describe their business rules as a series of JavaScript functions. These have access to the SanteDB JavaScript model objects and can subscribe to pre/post events on any model object.

The BRE service executes the JavaScript file which can register its rules via the SanteDBBre service. By default the business rules engine exposes the following interfaces. Technical documentation is provided in the JavaScript documentation.

The SanteDBBre module allows rule scripts to register two types of handlers:

- **Rules:** These are functions which have access to data being persisted, or queried prior to or after the event occurs. A pre event allows the rule to modify the object before the event occurs, whereas a post event allows the rule to handle events after the object has been persisted.
- **Validators:** These are functions which are those which add validation errors to a return array for a particular type.

The events to which rules can subscribe is outlined in .

Trigger	Event	Description
Insert	Pre / Post	Fired either before or after a new record is being created. This is fired with a call to insert, regardless if the insert ultimately resulted in an update.
Update	Pre / Post	Fired either before or after an existing record is explicitly updated. This is not fired when the update occurred due to an insert.
Obsolete	Pre / Post	Fired before or after an object is obsoleted from the database.
Query	Post	Fired after a result set has been retrieved from the database, but before the data is returned to an external party. This allows privacy controls such as masking, redaction, or pseudonymization.
Retrieve	Post	Fired after a record was specifically retrieved.

Illustrates the use of the SanteDBBre service to register a pre-event trigger which ensures that all patients do not have a name.

```
/**
 * Sample Business Rule for Patient
 */
SanteDBBre.AddBusinessRule("Patient", "AfterInsert", { statusConcept: StatusKeys.ACTIVE
}, function (patient) {
    // No patients may have a name
    patient.Name = null;
    return simplePatient;
});
```

Here, the business rule is being added for "Patient" to be trigger "AfterInsert", and only applied to patients who have a status code of ACTIVE.

Illustrates the use of the SanteDBBre service to register a validation handler which will return a warning about the use of the registration of males in a female only programme.

```
SanteDBBre.AddValidator("SubstanceAdministration", function (act) {

    var retVal = new Array();

    if (act.participation.RecordTarget.playerModel.genderConceptModel.mnemonic != "F")
        retVal.push(new SanteDBBre.DetectedIssue("Only females may receive vaccines!",
SanteDBBre.IssuePriority.Warning));

    return retVal;
});
```

7.3.1.5. Clinical Protocol Services

SanteDB's forecasting services (IClinicalProtocolService) is implemented by service classes which perform on-demand forecasting / scheduling. Passive forecasting (based on events) should be done by a daemon service which subscribes to events on the persistence layer and/or an IJobService instance that performs the operation.

Forecasting services generate acts or alerts with min/max times representing the minimum safe date, maximum safe date and suggested date for an action to occur. The forecasting service's proposal method accepts a parameter of type Patient and optionally any relevant data (existing vaccination SubstanceAdministrations, Observations representing AEFIs).

There are two major concepts for the forecasting services in SanteDB:

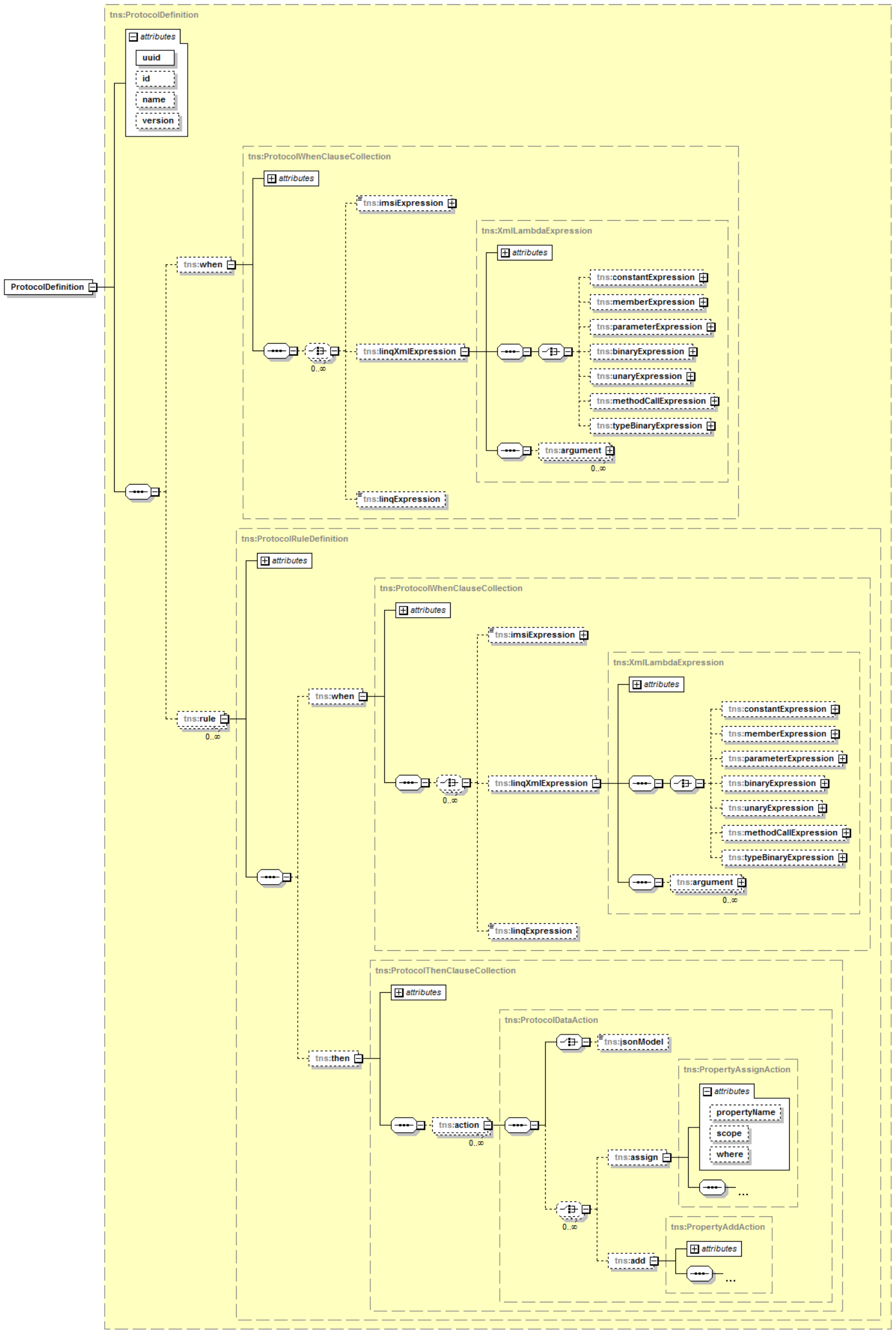
- **Care Protocol:** A protocol represents a series of instructions conveyed as when/then conditions which are concerned with a particular aspect of a patient's care. For example, in SanteDB, a protocol may be a particular antigen (OPV protocol) or other actions such as weight.
- **Care Plan:** The care plan represents an instantiation of a series of protocols which have been determined for a particular patient. The care plan is an execution of the care protocols placed into a coherent series of proposals.

In terms of execution, forecasting is handled by two separate interfaces:

- **Protocol Definition:** Which is responsible for defining the protocol. These are the clinical or logical description of the definition without necessarily describing the protocol.
- **Clinical Protocol Implementation:** The implementation is the protocol handler which actually has executable instructions in the form of when/then conditions which are executed to construct the care plan.

7.3.1.5.1 Xml Protocol Provider

The default implementation of the protocol handler for SanteDB is the XML based protocol handler. This handler defines a series of protocols in the http://santedb.org/cdss namespace. This namespace is illustrated in .



Element / Path	Cardinality	Description
@uuid	1..1	Uniquely identifies the clinical protocol in the global scope of all SanteDB instances.

@id	0..1	Identifies the clinical protocol within the local SanteDB instance.
@name	0..1	A human readable name for the clinical protocol. Example: "OPV Standard Schedule"
@version	0..1	Identifies the version of the protocol. Example: May 2008 WHO
When	1..1	The "when" condition which guards entry into the protocol. This when condition is executed once, if the result is false all rules in the protocol are skipped.
when/@evaluation	0..1	Identifies how the "when" condition should be evaluated. Examples of values are: and (all conditions must equate to true), or (any condition must equate to true) or xor (only one of the conditions should evaluate to true.
when/hdsiExpression	0..*	Represents a single guard condition expressed as an HDSI query expression (see documentation of the HDSI documentation grammar)
when/linqXmlExpression	0..*	Represents a single guard condition expressed as a LINQ expression serialized as XML. This representation is preferable to string linqExpression when the full expressivity of LINQ is required.
when/linqExpression	0..*	Represents a simplified string representation of a linq expression.
Rule	1..*	One or more rules which should be evaluated which represent the individual steps in a clinical protocol.
rule/@uuid	1..1	Uniquely identifies the clinical step within the global scope of SanteDB.
rule/@id	0..1	Identifies the rule in the local context of the protocol itself.
rule/@repeat	0..1	Identifies the number of times that the rule should be applied. For each iteration a value named "$index" is incremented.
rule/when/@evaluation	0..1	Identifies how the when condition of the rule should be evaluated.
rule/when/HDSiExpression	0..*	Represents the HDSI query grammar for the rule guard condition.
rule/when/linqXmlExpression	0..*	Represents the XML serialized LINQ expression of the rule guard condition.
rule/when/linqExpression	0..*	Represents a LINQ expression of the rule guard condition.
rule/then/@repeat	0..*	Identifies the number of times that the "then" condition should be applied. This

		differs from the @repeat attribution on "when" as this @repeat results in the when condition being evaluated once before performing the "then" action.
rule/then/jsonMondel	0..1	The JSON representation of the model which should be proposed when the "when" condition evaluates to true.
rule/then/assign	0..*	Instructs the engine to assign one or more properties to the specified values. These values (the text of the element) are LINQ expressions which are evaluated to set the specified properties.
rule/then/assign/@where	0..1	Identifies the guard condition which should be applied in order for the assignment to occur.
rule/then/assign/@propertyName	1..1	Identifies the property in the result model (the then clause) which should be set to the result of the LINQ expression.
rule/then/assign/@scope	0..1	Identifies the value of the then model which carries the scope. This is used for the special scope keyword in the LINQ expression and can be used to copy values from other proposals.
rule/then/add	0..*	Instructs the engine to add an instance of the result of the LINQ expression to the current propertyName expression.

7.3.1.6. Configuration of the CDR

The first version of the SanteDB HDS backbone will leverage the MARC-HI ServiceCore framework components heavily. These components are configured via application configuration files. This will introduce some overhead on large-deployments as configuration files will need to be shared among the application hosts performing a particular role within the SanteDB infrastructure.

Some components of SanteDB such as forecasting and protocols are configured via a central database (or rather, their behavior is controlled by the central data store for SanteDB).

7.3.1.7. Plugin Management

Plugin management is performed via a series of assembly attributes which are embedded in the assembly manifest of the iCDR plugins. The following attributes are to be used for identifying plugin metadata:

Table 7 - Plugin Management Attributes

Attribute	**Use**	**Description**
AssemblyVersion	R	Identifies the version (major.minor.revision.build) of the plugin. This information is used for dependency information.

AssemblyInformationalVersion	O	An informational version which is displayed on the administration and management service interface.
AssemblyDescription	R	A human readable description of the plugin to appear on the administrative interface.
AssemblyCopyright	O	Copyright information and/or use restriction messages.
Plugin	R	Identifies the assembly as a plugin. The plugin attribute identifies the minimum version of the SanteDB core which is required to run the plugin.
PluginDependency	O	Identifies the name and version of a dependency upon which the plugin must have installed.

Additionally plugins may embed database modification scripts into their assembly manifest. These database scripts are stored in an XML information format similar to Liquibase whereby “features” are identified and relevant “install” and “uninstall” SQL commands are included. These SQL statements may have guard conditions that are maintained by the database configuration technology selected.

Each feature file is also assigned an RDBMS invariant name that indicates the database management system for which the installation script is intended (in the case that a plugin works with more than one RDBMS, for example: the ADO.NET message persistence schemas).

SanteDB plugins may also embed a Plugin.xml resource into their assembly manifest. This plugin manifest describes the service providers the plugin provides, as well as defines the configuration parameters for that plugin.

For more information about the how plugins can expose configuration options to the SanteDB server configuration system see .

7.3.1.8. Security Architecture

All the components of SanteDB are designed to consider how data is access securely from each layer and between each component. This architecture requires that all access to method calls to secured services pass an instance of IPrincipal which represents the authenticated user context within the current execution pipeline.

There are four major concepts to the SanteDB security architecture:

- **Identities:** Represent an identification of a security asset such as user, device or application. For example, the user jsmith would represent a user identity.
- **Principals:** Represent an authenticated identity (or collection of identities) representing a single session. Principals have an identity (the user/device/application accessing the HDS) as well as a series of claims about the identity (such as role/device/application/authentication method/etc.)
- **Policies:** Represent a definition of some action or group of actions applied against the SanteDB HDS system (such as login, create role, etc.) or some securable within the SanteDB HDS data store (such as privacy policies applied to data). Policy definitions are maintained by policy information providers.
- **Permissions:** Represent a granting of access or rights to a policy for a principal. The decision on whether a principal is granted a permission to perform an operation is made by a policy decision provider.

The creation of an IPrincipal instance can be from a local authority (such as simple SQL database authentication) or from a remote authority (such as SWT, JWT, etc.).

7.3.1.8.1 Provenance

All objects at all layers of the SanteDB iCDR persistence layer use the concept of provenance to attributing all data actions performed on the system. The SanteDB provenance object structure is always written by the iCDR server, though clients may "suggest" values (which are captured in the server's provenance object.

The properties of the provenance object and their purpose (how they are set) is outlined in.

Property	Description	Source
User	Captures the SID of the user identity that was attached to the principal when the action occurred.	Server Authentication Context – User Identity
Application	Captures the SID of the software application / vendor attached to the principal when the action occurred. With this data it is possible to determine the software application responsible for the change.	Server Authentication Context – Application Identity
Device	Captures the SID of the physical node/device attached to the principal when the action occurred. With this data it is possible to determine the software device responsible for the action.	Server Authentication Context – Device Identity
Session	Captures the actual session attached to the request. This is used for tracking or correlating actions across requests and actions.	Server Authentication Context – Session Identity
Establishment	Captures the date / time that the database transaction that made the change started.	Server Timestamp
External Ref	Captures the provenance or user SID that the client/submitter "claims" created the data. Note that this value is for reference only.	Client Claim
External Ref Type	Identifies the type of object that the external security reference points to. Can be a SecurityUser or SecurityProvenance object.	Calculated

7.3.1.8.2 Basic Security

The default SanteDB messaging services (FHIR, HDSI, etc.) can be configured to use HTTP basic authentication. This authentication mechanism is tied into the WCF pipeline and uses the current implementation of IIdentityProvider to authenticate username and password in the HTTP header. The device identity is established via the TLS client certificate sent in the HTTP request.

Applications connecting to a HTTP Basic security service are furthermore required to send their application public identifier and application secret in the X-SanteDBClient-Authorization HTTP header. This header has the same format as the BASIC Auth header and includes the client id and secret as a base64 encoded string.

Claims can also be sent using this scheme via the X-SanteDBClient-Claim HTTP header. Claim values are base64 encoded in format: claimURI=claimValue. The X-SanteDBClient-Claim HTTP header values repeat and the authentication pipeline ensures that the user is permitted to make the claim provided.

For example, the HTTP headers for user “Jsmith” on client e612f88c-3ba3-40fe-8cd6-792836b2088c making claim that purpose of use is TREATMENT would be:

```
POST /fhir
Authorization: BASIC anNtaXRoOnBhc3N3b3Jk
X-SanteDBClient-Authorization: BASIC
ZTYxMmY4OGMtM2JhMy00MGZlLThjZDYtNzkyODM2YjIwODhjOjc0MzQyYTE1MTY4YTQxODNhOWU2ZTllZTFmMGUxZWQ0
X-SanteDBClient-Claim: dXJuOm9hc2lzOm5hbWVzOnRjOnhhY21sOjIuMDphY3Rpb246cHVycG9zZT1UUkVBVE1FTlQ=
Content-Type: application/json+fhir
Content-Length: 2394

{
```

7.3.1.8.3 Federated Security

Figure 7 illustrates how a remote client can obtain a token from a federated security token service (STS) representing an IPrincipal and pass it to the SanteDB HDS. The creation of a local IPrincipal is controlled by a local IIdentityProviderService implementation. It is imperative that the ACS generate a token format which is suitable for the HDS messaging interface to consume (i.e. the configurations match), otherwise the HDS will have no mechanism for verifying tokens.

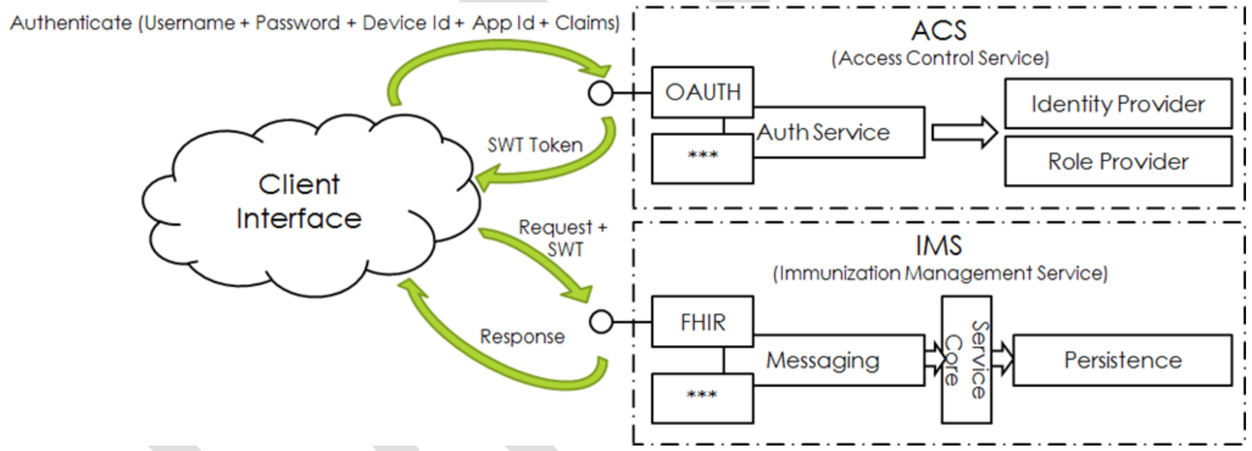


Figure 7 - Security Architecture

Any ACS service can be used with SanteDB, however it is recommended that the ACS being used support the OAuth token service’s password grant and provide client/device authentication via TLS and/or HTTP basic auth.

7.3.1.8.4 Default OAuth ACS Implementation

SanteDB provides an implementation of an OAuth STS which generates JSON Web Tokens (JWT) compatible with SanteDB. The default implementation of the OAuth STS only supports password and token refresh grant types.

The returned value is a JWT token which may subsequently be used by the client to access HDS service interfaces. The JWT token validator is inserted into the WCF’s WIF pipeline and ensures that the token is signed by a trusted ACS and that the token has not expired.

The default ACS implementation performs node authentication (authentication of the device) using the TLS certificate passed in the SSL transport layer. The device certificate used to connect to the ACS forms the basis of authenticating the node and may be explicit (using the DeviceEvidence field in the SecurityDevice table) or chained (to a root CA that the ACS trusts).

Applications are authenticated using the HTTP BASIC auth scheme described in the OAuth 2.0 specification. The application is expected to pass its client_id and client_secret as a username/password in HTTP Authorize header.

The client can make claims about the request by using the X-SanteDBClient-Claim HTTP header. This header is in the format claimType=claimValue and is base64 encoded. Multiple claims are separated by a comma.

The following example represents a request for token for user jsmith from client e612f88c-3ba3-40fe-8cd6-792836b2088c making claim that purpose of use is TREATMENT.

```
POST /oauth2_token
Content-Type: application/x-www-urlform-encoded
Authorization: BASIC
ZTYxMmY4OGMtM2JhMy00MGZlLThjZDYtNzkyODM2YjIwODhjOjc0MzQyYTE1MTY4YTQxODNhOWU2ZTllZTFmMGUxZWQ0
X-SanteDBClient-Claim: dXJuOm9hc2lzOm5hbWVzOnRjOnhhY21sOjIuMDphY3Rpb246cHVycG9zZT1UUkVBVE1FTlQ=
Content-Length: 204

grant_type=password&username=jsmith&password=password123&scope=http://demo.openiz.org/fhir
```

7.3.1.8.5 Claim Types

The implementations of IPrincipal should be claims based. In a claims based principal, the authenticated user information contains a series of claims about that user such as their name, organization, the reason for access, etc. The claims used in SanteDB are listed in Table 8.

Table 8 - Claim Types

Claim	**Value**	**Use**
urn:oasis:names:tc:xacml:2.0:resource:resource-id	String	The identifier of the resource to which the claim is about.
urn:oasis:names:tc:xacml:2.0:action:purpose	PurposeOfUse	Indicates the reason why data is being queried. Used for policy enforcement decisions. Valid values are drawn from the PurposeOfUse concept set.
urn:oasis:names:tc:xacml:2.0:subject:role	String	The clinical roles that the user has.
urn:oasis:names:tc:xspa:1.0: subject:facility	Url	The facility identifier to which the principle belongs.
urn:oasis:names:tc:xspa:1.0: subject:organization-id	String	The organization identifier to which the principal belongs.

urn:oasis:names:tc:xacml:1.0: subject:subject-id	String	The distinguished name of the principal.
http://openiz.org/claims/grant	String	The policies to which the user has been granted access by the ACS.
http://openiz.org/claims/device-id	String	The identifier for the security device from which the principal is operating.
http://openiz.org/claims/application-id	String	The identifier for the security application from which the principal is operating.
http://schemas.microsoft.com/ws/2008/06/identity/claims/role	String	Security roles to which the user belongs.
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name	String	The user name of the principal.
http://schemas.xmlsoap.org/ws/2005/05/identity/claims/authentication	String	The authentication result of the principal.
http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationinstant	DateTime	The instant in time when the principal was authenticated.
http://schemas.microsoft.com/ws/2008/06/identity/claims/authenticationmethod	String	The method of authentication used.
http://schemas.microsoft.com/ws/2008/06/identity/claims/expiration	DateTime	The date/time that the principal's authentication no longer is valid.
`http://schemas.xmlsoap.org/ws/2005/05/identity/claims/sid`	UUID	The security identifier of the principal. This is the UUID of the user.

7.3.1.9. Policy / Privacy Enforcement Architecture

The enforcement of privacy and policies is handled through a series of services within the SanteDB solution. From a high level, three different types of services are involved:

- **Policy Information Provider (PIP)** – Is responsible for storing information related to the policies. The information point is responsible for maintaining a list of IPolicy objects which contain the name, oid, handler (C# class which is executed upon policy decision), and elevation control.
- **Policy Decision Point (PDP)** – Is responsible for making a decision related to a policy (or series of policies) for a given securable. The decision outcome is one of the following options:
    - **Deny** – The principal has no authorization to access the requested securable or policy.
    - **Elevate** – The principal can access the securable or policy however they require additional authentication (such as 2nd level password, TFA, etc.)
    - **Grant** – The principal is granted access to the specified securable or policy.
- **Policy Enforcement Point (PEP)** – Is responsible for listening to events from the SanteDB system and leveraging the decision and information points to enforce the policy decision. This implementation can vary between jurisdictions however by default involves either the masking

(i.e. there is something here you can't see), redaction (i.e. removal of information), or partial disclosure of records.

The process for enforcement is illustrated in Figure 8.

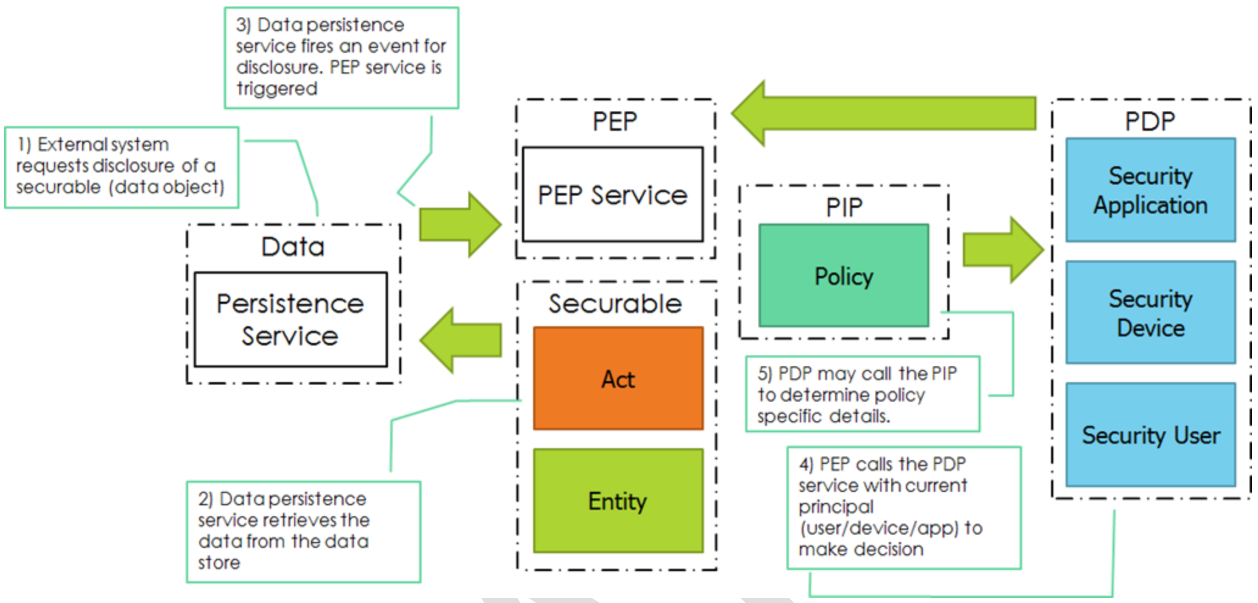

3) Data persistence service fires an event for disclosure. PEP service is triggered
1) External system requests disclosure of a securable (data object)
PEP
PEP Service
PDP
Security Application
Security Device
Security User
PIP
Policy
Data
Persistence Service
Securable
Act
Entity
5) PDP may call the PIP to determine policy specific details.
4) PEP calls the PDP service with current principal (user/device/app) to make decision
2) Data persistence service retrieves the data from the data store

Figure 8 - Policy Enforcement Architecture

Policy enforcement may happen declaratively via enforcement of security attributes on code (most notably the PolicyPermission and PolicyPermissionAttribute classes). The default policies included in SanteDB are listed in . The HDS is expected to be aware of all policy identifiers, clients and services accessing the HDS are merely to be aware of local policies which may have an impact on their function.

Table 9 - SanteDB Policies

Name	OID	Description
Superuser	1.3.6.1.4.1.33349.3.1.5.9.2	Identities which possess this policy permission are granted access to all functions in SanteDB.
Access Administrative Function	1.3.6.1.4.1.33349.3.1.5.9.2.0	Identities which possess this policy permission are granted access to all administrative functions of SanteDB.
Change Password	1.3.6.1.4.1.33349.3.1.5.9.2.0.1	Allows an identity to change any other user's password.
Create Role	1.3.6.1.4.1.33349.3.1.5.9.2.0.2	Allows an identity to arbitrarily create a user role.
Alter Role	1.3.6.1.4.1.33349.3.1.5.9.2.0.3	Allows an identity to modify roles, including role membership.
Create Identity	1.3.6.1.4.1.33349.3.1.5.9.2.0.4	Allows an identity to create arbitrary identities (users).

Create Device	1.3.6.1.4.1.33349.3.1.5.9.2.0.5	Allows an identity to create arbitrary devices.
Create Application	1.3.6.1.4.1.33349.3.1.5.9.2.0.6	Allows an identity to create arbitrary applications.
Administer Concept Dictionary	1.3.6.1.4.1.33349.3.1.5.9.2.0.7	Allows an identity to create and modify concept definitions
Alter Identity	1.3.6.1.4.1.33349.3.1.5.9.2.0.8	Allows an identity to alter already created identities.
Alter Policy	1.3.6.1.4.1.33349.3.1.5.9.2.0.9	Allows an identity to create or alter a policy.
Administer Data Warehouse	1.3.6.1.4.1.33349.3.1.5.9.2.0.10	Allows an identity to administer the data warehouse
Login	1.3.6.1.4.1.33349.3.1.5.9.2.1	Grants an identity the login permission.
Login Service	1.3.6.1.4.1.33349.3.1.5.9.2.1.0	Grants an identity permission to login without user interaction. User screens should not demand this permission.
Unrestricted Clinical Data	1.3.6.1.4.1.33349.3.1.5.9.2.2	Identities which possess this policy permission are granted access to all clinical functions of the SanteDB HDS.
Query Clinical Data	1.3.6.1.4.1.33349.3.1.5.9.2.2.0	Allows an identity to execute any query against clinical data.
Write Clinical Data	1.3.6.1.4.1.33349.3.1.5.9.2.2.1	Allows an identity to create and/or update clinical data.
Delete Clinical Data	1.3.6.1.4.1.33349.3.1.5.9.2.2.2	Allows an identity to obsolete clinical data.
Read Clinical Data	1.3.6.1.4.1.33349.3.1.5.9.2.2.3	Allows an identity to fetch arbitrary records.
Export Clinical Data	1.3.6.1.4.1.33349.3.1.5.9.2.2.4	Allows an identity to export sensitive clinical data to an external device (may be un-encrypted)
Override Disclosure	1.3.6.1.4.1.33349.3.1.5.9.2.3	Allows a user to override a disclosure deny.
Unrestricted Metadata	1.3.6.1.4.1.33349.3.1.5.9.2.4	Allows an identity unrestricted access to metadata (extension types, places, etc.)
Read Metadata	1.3.6.1.4.1.33349.3.1.5.9.2.4.0	Allows an identity to read metadata.
Client Administrator	1.3.6.1.4.1.33349.3.1.5.9.2.10	Allows a user to access a client administration function.
Unrestricted Warehouse	1.3.6.1.4.1.33349.3.1.5.9.2.5	Allows a user unrestricted access to the datawarehouse
Write Warehouse Data	1.3.6.1.4.1.33349.3.1.5.9.2.5.0	Allows a user to read data from the datawarehouse

Delete Warehouse Data	1.3.6.1.4.1.33349.3.1.5.9.2.5.1	Allows a user to delete data from the data warehouse
Read Warehouse Data	1.3.6.1.4.1.33349.3.1.5.9.2.5.2	Allows a user to read discrete warehouse data
Query Warehouse Data	1.3.6.1.4.1.33349.3.1.5.9.2.5.3	Allows a user to execute warehouse stored queries (which may return less data than discrete reads)
Unrestricted MDM	1.3.6.1.4.1.33349.3.1.5.9.2.6	Allows a user, application or device to do anything with an MDM record
Write MDM MAster Record	1.3.6.1.4.1.33349.3.1.5.9.2.6.1	Allows a user, device or application to create/update/obsolete an MDM master record directly (normally user/device/applications can only update their LOCAL record)
Read MDM Local Records	1.3.6.1.4.1.33349.3.1.5.9.2.6.2	Allows a user, device, application to read all MDM local records. Typically these can only read their own local record, or the master record.
Merge MDM Master Record	1.3.6.1.4.1.33349.3.1.5.9.2.6.3	Allows a user, device, application to merge a MDM master record explicitly. Typically applications can only merge records that they themselves have created (their provenance)

7.3.1.9.1 Most-Restrictive Enforcement

SanteDB's default policy decision service provider operates on a basis of most-restrictive with default DENY. In this evaluation scheme policy decisions are created as follows:

- If the principal has no data associated with the policy, the result of the decision is DENY,
- If the principal has one rule associated with the policy via role, device, or application then the result of the decision is the rule's configuration,
- If the principal has multiple rule instances configured via role, device or application then the result of the decision is the most restrictive option.

For example, John Smith (user jsmith) is a member of USERS, CLINICAL and is accessing SanteDB from application ReaderApp.

Policy	**From USERS**	**From CLINICAL**	**From ReaderApp**	**Effective Set**
Access Administrative Function				***DENY***
Change Password				DENY

Create Role				DENY
Alter Role				DENY
Create Identity				DENY
Login	GRANT		GRANT	GRANT
Unrestricted Clinical Data		***GRANT***		***GRANT***
Query Clinical Data		GRANT (implied)		GRANT (implied)
Write Clinical Data		GRANT (implied)	DENY	DENY
Delete Clinical Data		GRANT (implied)	DENY	DENY
Read Clinical Data		GRANT (implied)		GRANT (implied)
Override Disclosure		GRANT	DENY	DENY

7.3.1.10. Report Services

It is expected that SanteDB implementations will leverage several different types of reporting engines based on features of the report engine. For example, Jurisdiction A may choose SSRS whilst another may choose JasperReports.

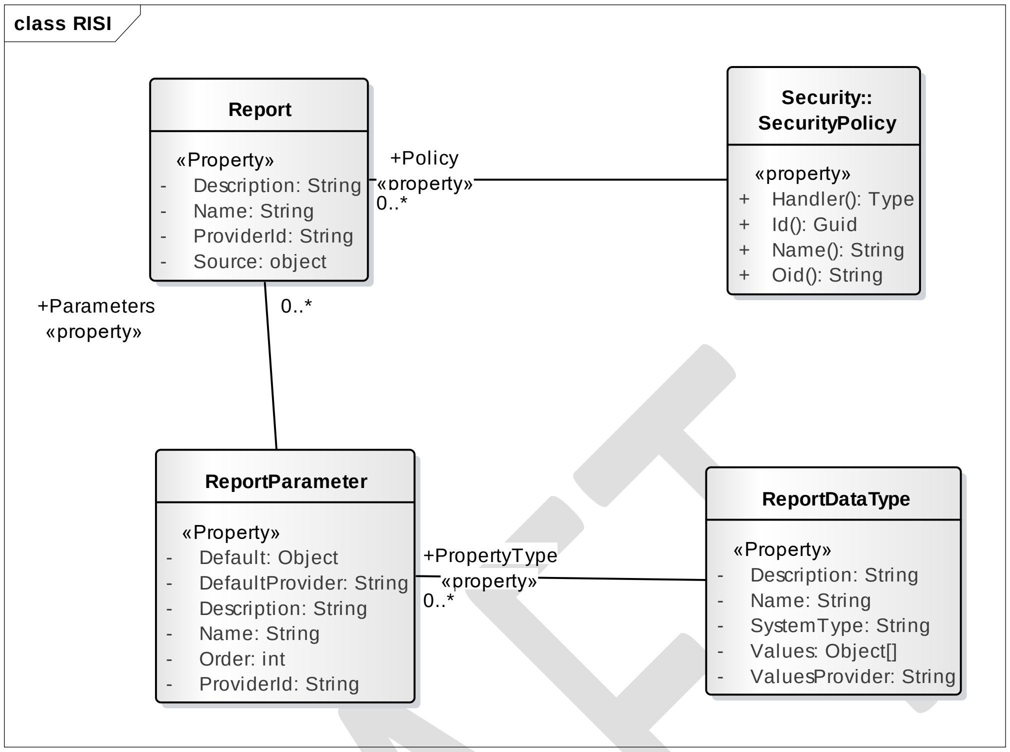
In order to ensure that client applications are given a consistent interface with which to generate reports, the SanteDB HDS provides the IReportProvider service. This service is responsible for:

1. Exposing an enumeration of report objects which can be used by callers to determine the installed reports on the report manager.
2. Maintaining the security attributes of the report, providing a the ability to restrict report access based on policies.
3. Exposing the parameters that a particular report can accept for generation.
4. Executing the report and exposing the resultant files to callers.
5. Installing / reflecting reports to the backing report engine. For example, the IReportProvider may provide a mechanism for reflecting JRXML or RDL report files and deciphering parameters and report titles.

The IReportProviderService service is merely an extension of the IDataPersistenceService<Report> interface.

7.3.1.10.1 Model

The IReportProvider service exposes a series of canonical objects which describe the reports contained in the execution engine. Illustrates the objects exposed and their relations.



Describes the report canonical model in more detail.

Class	**Property**	**Type**	**Description**
Report	(N/A)	VersionedEntityData	The report class identifies a single report within the report execution engine which can be executed by a principal.
	Source	XElement	The complete source of the report. This is the RDL or JRXML, etc. which comprises the report.
	Name	String	The name of the report.
	Description	String	A long form description of the report, its intended purpose, etc.
	ProviderId	String	The identifier by which the execution engine knows the report as.
	Policy	SecurityPolicy[]	One or more policies which a user must posses in order to execute the report. These policies are AND.
	Parameters	ReportParameter[]	One or more report parameters which can be used to render the report.
ReportParameter	(N/A)	VersionedAssociation	The report parameter class identifies a single report parameter which can be applied to report.

	Name	String	The human readable name of the report parameter.
	Description	String	A long form description of the report parameter such as described in a help document.
	ProviderId	String	Identifies the id of the parameter as the report execution engine understands the report parameter.
	Order	Int32	The order in which the report parameter should be displayed.
	Policy	SecurityPolicy[]	One or more policies which the principal must posses in order to populate the parameter.
	Default	Object	The default value for the report parameter.
	DefaultProvider	IValueProvider	The IValueProvider which can be used to ascertain a default value at runtime.
	Type	ReportDataType	The type which represents the type of data in the parameter.
ReportDataType	(N/A)	IdentifiedData	The report data type class is used to describe a type of data which can be populated into a report. This can be simple types like: Date, String, etc. or complex types like SecurityUserSelector, VillageSelector, etc.
	Name	String	A human readable name for the report data type.
	SystemType	Type	The underlying system type which this datatype represents.
	Description	String	The description of the report data type.
	Values	Object[]	One or more values which are acceptable for the report parameter type.
	ValuesProvider	IValueProvider	Represents a value provider which can be used in lieu of a static list of allowed values. This is often used for validating report parameter values as well as a source for auto-complete information.

7.3.1.10.2 Value Providers

A value provider represents a simple implementation of the IValueProvider interface which is used by report parameters and parameter types to either provide values for an auto-complete list, default value, etc.; or validate a value given by a caller. IValueProvider extends the IEnumerable<Object> interface.

The methods provided by a value provider are listed in .

Method	Return	Parameters	Description
GetEnumerator()	IEnumerator<Object>	None	Gets a list of allowed or acceptable values based on the current authentication context.
Validate	Bool	Object	Validates that the provided object is valid according to the current security context.

7.3.2. SanteDB's Administration & Configuration Architecture

TODO:

- Discuss deployment
- Discuss cloud controller
- Discuss concept of REALM

7.3.2.1. Administration Management Interface (AMI)

The administration management interface (AMI) is used to control and harmonize the configuration of SanteDB instances. The AMI function is three fold:

1. **Cloud Control** – The AMI is responsible for distributing the configuration of an SanteDB cluster (example: in a private cloud) across application servers.
2. **Configuration –** The AMI is used by an administration tool to control the service interface and configuring an instance of SanteDB.
3. **Device Management –** The AMI is used by administrators to on-board new devices, manage their security certificates.

The service represents a hybrid of configuration, management of the Security* tables (applications, devices, etc.) as well as control over security certificate services. The AMI service uses CAS to ensure that the access to the service is only performed by administrators with appropriate functions. The exception to this rule is are the security user functions for update whereby a user may update their own information.

7.3.2.1.1 Operations

Illustrates the operations that are supported by the administration management interface (AMI).

Resource	Operation	Description
/	OPTIONS	Gets options for the AMI service. Returns options for the AMI service.
	PING	Ping the service to determine up/down
/?xsd={schemaId}	GET	Gets the schema for the administrative interface. The id of the schema to be retrieved.Returns the administrative interface schema.
/alert	GET	Gets a list of alert for a specific query. Returns a list of alert which match the specific query.

	POST	Creates an alert. The alert message to be created.Returns the created alert.
/alert/{alertId}	GET	Gets a specific alert. The id of the alert to retrieve.Returns the alert.
	PUT	Updates an alert. The id of the alert to be updated.The alert containing the updated information.Returns the updated alert.
/applet	GET	Gets a list of applets for a specific query. Returns a list of applet which match the specific query.
	POST	Creates an applet. The pak data.Returns the created applet manifest info.
/applet/{appletId}	GET	Gets a specific applet. The id of the applet to retrieve.Returns the applet.
	DELETE	Deletes an applet. The id of the applet to be deleted.Returns the deleted applet.
	HEAD	Return just the headers of the applet id
	PUT	Updates an applet. The id of the applet to be updated.The applet containing the updated information.Returns the updated applet.
/applet/{appletId}/pak	GET	Downloads the applet. The applet identifier.Stream.
/application	GET	Gets a list applications for a specific query. Returns a list of application which match the specific query.
	POST	Creates a security application. The security application to be created.Returns the created security application.
/application/{applicationId}	GET	Gets a specific application. The id of the application to retrieve.Returns the application.
	DELETE	Deletes an application. The id of the application to be deleted.Returns the deleted application.
	PUT	Updates an application. The id of the application to be updated.The application containing the updated information.Returns the updated application.
/assigningAuthority	GET	Gets a list of assigning authorities for a specific query. Returns a list of assigning authorities which match the specific query.
	POST	Creates an assigning authority. The assigning authority to be created.Returns the created assigning authority.

/assigningAuthority/{assigningAuthorityId}	GET	Gets a specific assigning authority. The id of the assigning authority to retrieve.Returns the assigning authority.
	DELETE	Deletes an assigning authority. The id of the assigning authority to be deleted.Returns the deleted assigning authority.
	PUT	Updates an assigning authority. The id of the assigning authority to be updated.The assigning authority containing the updated information.Returns the updated assigning authority.
/audit	POST	Create audit in the HDS' audit repository
/certificate	GET	Gets a list of certificates. Returns a list of certificates.
/certificate/{id}	GET	Gets a specific certificate. The id of the certificate to retrieve.Returns the certificate.
/certificate/{id}/revokeReason/{reason}	DELETE	Deletes a specified certificate. The id of the certificate to be deleted.The reason the certificate is to be deleted.Returns the deletion result.
/changepassword/{id}	PUT	Changes the password of a user. The id of the user whose password is to be changed.The new password of the user.Returns the updated user.
/codeSystem	GET	Gets the code systems. Returns a list of code systems.
	POST	Creates the code system. The code system.Returns the created code system.
/codeSystem/{codeSystemId}	GET	Gets the code system. The code system identifier.Returns a code system.
	DELETE	Deletes the code system. The code system identifier.Returns the deleted code system.
	PUT	Updates the code system. The code system identifier.The code system.Return the updated code system.
/crl	GET	Gets the certificate revocation list. Returns the certificate revocation list.
/csr	GET	Gets a list of submitted certificate signing requests. Returns a list of certificate signing requests.
	POST	Submits a specific certificate signing request. The certificate signing request.Returns the submission result.
/csr/{certId}	DELETE	Rejects a specified certificate signing request. The id of the certificate signing request to be rejected.The reason the

		certificate signing request is to be rejected.Returns the rejection result.
/csr/{id}	GET	Gets a specific certificate signing request. The id of the certificate signing request to be retrieved.Returns the certificate signing request.
	PUT	Accepts a certificate signing request. The id of the certificate signing request to be accepted.Returns the acceptance result.
/device	GET	Gets a list of devices. Returns a list of devices.
	POST	Creates a device in the HDS. The device to be created.Returns the newly created device.
/device/{deviceId}	GET	Gets a specific device. The id of the security device to be retrieved.Returns the security device.
	DELETE	Deletes a device. The id of the device to be deleted.Returns the deleted device.
	PUT	Updates a device. The id of the device to be updated.The device containing the updated information.Returns the updated device.
/extensionType	GET	Gets the extension types. Returns a list of extension types.
	POST	Creates the type of the extension. Type of the extension.Returns the created extension type.
/extensionType/{extensionTypeId}	GET	Gets the type of the extension. The extension type identifier.Returns the extension type, or null if no extension type is found.
	DELETE	Deletes the type of the extension. The extension type identifier.Returns the deleted extension type.
	PUT	Updates the type of the extension. The extension type identifier.Type of the extension.Returns the updated extension type.
/policy	GET	Gets a list of policies. Returns a list of policies.
	POST	Creates a security policy. The security policy to be created.Returns the newly created security policy.
/policy/{policyId}	GET	Gets a specific security policy. The id of the security policy to be retrieved.Returns the security policy.

	DELETE	Deletes a security policy. The id of the policy to be deleted.Returns the deleted policy.
	PUT	Updates a policy. The id of the policy to be updated.The policy containing the updated information.Returns the updated policy.
/role	GET	Gets a list of security roles. Returns a list of security roles.
	POST	Creates a security role. The security role to be created.Returns the newly created security role.
/role/{roleId}	GET	Gets a specific security role. The id of the security role to be retrieved.Returns the security role.
	DELETE	Deletes a security role. The id of the role to be deleted.Returns the deleted role.
	PUT	Updates a role. The id of the role to be updated.The role containing the updated information.Returns the updated role.
/sherlock	GET	Gets a server diagnostic report. The diagnostic report to be created.Returns the created diagnostic report.
	POST	Creates a diagnostic report. The diagnostic report to be created.Returns the created diagnostic report.
/tfa	GET	Gets the list of TFA mechanisms. Returns a list of TFA mechanisms.
	POST	Creates a request that the server issue a reset code
/user	GET	Gets a list of security users. Returns a list of security users.
	POST	Creates a security user. The security user to be created.Returns the newly created security user.
/user/{userId}	GET	Gets a specific security user. The id of the security user to be retrieved.Returns the security user.
	DELETE	Deletes a security user. The id of the user to be deleted.Returns the deleted user.
	PUT	Updates a security user. The id of the security user to be retrieved.The user containing the updated information.Returns the security user.

7.3.2.2. Sherlock Integration with JIRA

The Sherlock service on the administration management interface is used to collect logs from publishers which submit bug reports to the configured HDS system. By default the HDS which is configured will save the application diagnostic report to the hard drive, however there are configurations which are able to send these diagnostic reports to a JIRA system.

The configuration for the Sherlock JIRA integration requires a JIRA server running version 6 of JIRA or greater and will create issues via the REST integration interface. All issues are created under the configured project with the label SanteDBAuto label. This allows the automatically created JIRA issues to be filtered from regular issues created within the JIRA user interface.

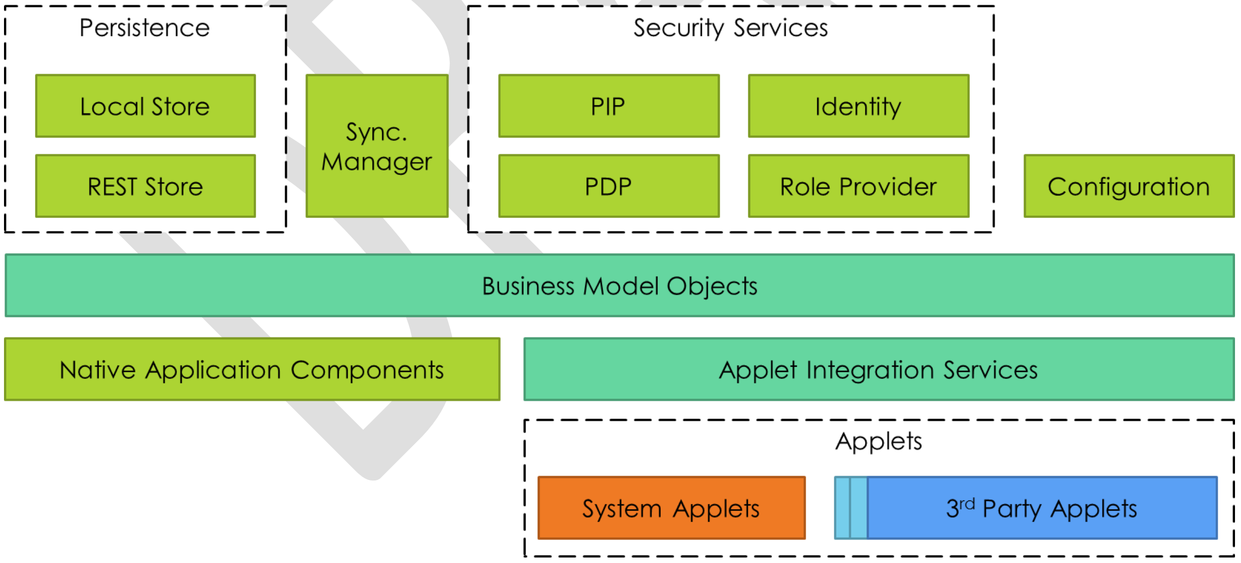
7.3.3. Disconnected Client Architecture

SanteDB is designed as an open architecture, therefore consumer applications may be implemented by third parties. However, to demonstrate the capabilities of the SanteDB data architecture and features, a series of reference clients have been developed.

The primary client of the SanteDB immunization management system is the SanteDB Disconnected Client. The disconnected client is designed to consume SanteDB HDSI services in a variety of capacities to showcase how the system operates.

The disconnected client is designed as a cross Xamarin/HTML5 hybrid application. All heavy lifting functions are performed in native C# / Xamarin functions whereas the user interfaces are rendered in HTML5.

The architecture of the disconnected client is illustrated in .



The major components of the application architecture are:

- **Business Model Objects :** The HDS core object model is shared between the server and disconnected client using .NET PCL libraries.
- **Persistence Services :** These services are responsible for persisting data in memory to either the database (in offline mode) or the HDS backend (in online mode).

- **Synchronization Services :** The synchronization services are responsible for managing the synchronization mailbox for the disconnected client.
- **Security Services :** Are responsible for enforcing security on objects in the persistence store and actions on the application.
- **Configuration Services :** Provide a consistent mechanism for configuration of application and the applet features in the mobile application.
- **Native Application Components :** Represent the native components such as splash screen, actions, authentication handlers, etc.
- **Applet Integration Services :** Provide a series of common interfaces with which the SanteDB mobile applets can interact with the native mobile applications.
- **Applets :** These are the user screens for the SanteDB DC.

The applet infrastructure allows a jurisdiction to easily customize the SanteDB user experience without the need for having C#/Xamarin programmers on staff. It also allows a jurisdiction to deploy applets to mobile devices easily.

7.3.3.1. Synchronization

The synchronization of the SanteDB disconnected database and the SanteDB back-end are performed using the subscription interface. The client is responsible for storing which local resources it has subscribed to and exhausting the most important data from the server when it becomes available.

The subscription interface is unique in that it can only be connected to via devices in the synchronization role. The subscription interface allows a device to subscribe and un-subscribe to data for which it is interested. The device then exhausts its message box

This process is triggered several times throughout the lifecycle of the application:

- On application start-up : The mobile device is responsible for sending HDSI STATE commands (HTTP HEAD) to the ControlAct resource. The HEAD operation is described in more detail in section

7.3.3.2. Applet Architecture

SanteDB applets are shared resources between the server and disconnected client. An applet contains one or more of the following types of “assets”:

- **User Interfaces –** Which are written using HTML5 and JavaScript. These files use the specialized tags which are rendered at runtime to maximize code reuse. The default infrastructure for applets is AngularJS, however this can be overridden by using a different core module.
- **Business Rules –** Business rules are JavaScript files within the applet which dictate specialized validation / behaviors on both the client and server. Business rules are described in more detail in 7.3.1.4.1 (on page 30).
- **Protocols –** Protocols are clinical protocols which will be used by SanteDB to calculate a care plan. These protocols are designed using protocol XML script and is described in more detail in 7.3.1.5.1 (on page 32).
- **Templates –** Templates are pre-defined snippets of data which define a common structure for data. Templates will have an input form and a view. These are used by common patient history components to render the data in question properly.

- **Widgets –** A widget is a small piece of user interface code which is injected into a master page. Widgets are defined by their type (tab, panel, etc.) and their scope (patient, facility etc.).
- **Reports –** Mobile reports are defined using the mobile reporting engine in the disconnected client. This engine is used to render simple tabular reports and can do things like pivot and group report data.

7.3.3.2.1 Applet Manifest

The applet manifest is used to provide meta-data about the applet being deployed or compiled. Applet manifests contain vital information such as the templates exposed by the applet, the version, menus, and localization strings.

The applet manifest

7.4. Communications / Interoperability Architecture

The primary mechanism of communication with the SanteDB HDS is via the interoperability and communications layer. These services allow multiple user interfaces, or point of service devices, to connect with the HDS. The overall communications architecture for the HDS is outlined in

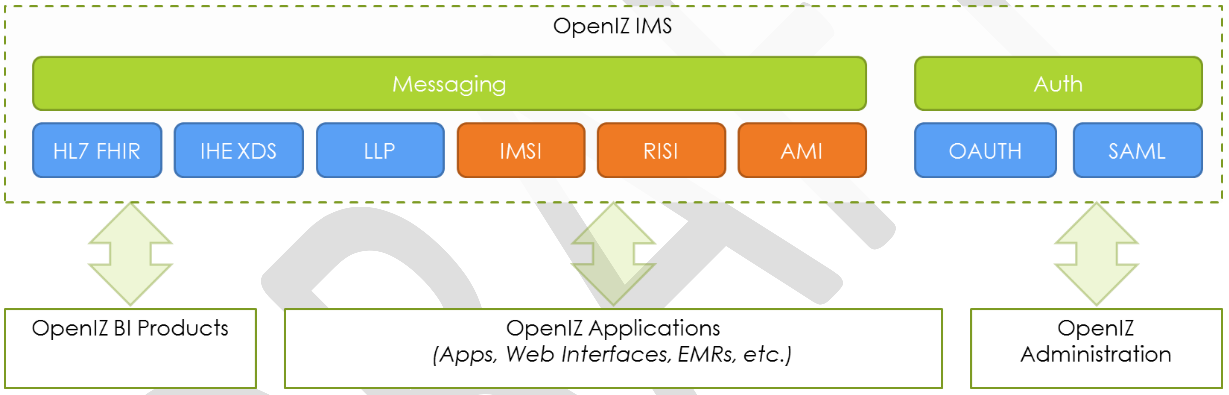


Figure 9 - HDS Messaging Interfaces

The interfaces are detailed in Table 10.

Table 10 - HDS Messaging Interfaces

Interface	Standards	Version	Description
HDSI	Proprietary	1.0	The Immunization Management Service Interface provides a raw 1-1 mapping between REST and the backing HDS data store.
RISI	Proprietary	1.0	The Report Integration Services Interface provides a direct aggregate view of the backing database as well as queries supporting reporting services such as MS-SSRS or JasperReports. The RISI interface is readonly and intended to be used for BI purposes only.
AMI	Proprietary	1.0	The Administration Management Interface allows access to a variety of administrative

			functions including restarting services, deploying plugins, configuration and meta-data editing. It has no access to the clinical store and is available to administrators only.
LLP	HL7 V2 over MLLP	2.5	The HL7v2 interface provides ADT, QBP, VXQ, VXU and SRM messaging support and is intended to provide a bridge to HDS functionality.
XDS	IHE XDS + IHE IC	XDS.b	The XDS.b interface supports the export and import Immunization Content (IC) CDA documents.
FHIR	HL7 FHIR	1.4	The HL7 FHIR interface supports the manipulation of HDS resources using the HL7 FHIR standard. The version of FHIR implemented on the HDS is FHIR STU3.

7.4.1. Communicating with the HDS

All interfaces to the HDS are secured with one of the allowed authorization schemes. These include SAML, OAUTH or Basic AUTH. Communications with the HDS protocols are made using the Windows Communication Foundation (WCF) framework and/or the NHAPI HL7v2 framework for LLP.

There are six interfaces that the HDS will eventually support:

- Immunization Management Service Interface (HDSI) – A proprietary, REST based interface which provides a near 1-to-1 mapping of clinical functionality in the HDS to clients.
- Report Integration Services Interface (RISI) – A proprietary, REST based interface which provides a series of interfaces for acquiring the location and parameterization of system reports for use in the HDS.
- Administration Management Interface (AMI) – A proprietary, SOAP based interface which is used to perform administrative operations on the nodes in an SanteDB cluster.
- HL7® FHIR™ - Fast Health Interoperability Resources (FHIR) is a standardized REST interface for interacting with the HDS.
- HL7® Version 2 – HL7v2 interfaces provide VXU capability within the HDS and are implemented per CDC specification for immunization reporting.
- HL7® CDA – Interface which provides and imports CDA documents containing immunization summaries.

7.4.2. Immunization Management Service Interface (HDSI)

The HDSI interface provides a 1-to-1 mapping of business model artifacts to a wire level representation. The HDSI is primarily designed for clients which will require sending and receiving large amounts of RAW data from the HDS for purposes like offline synchronization.

The HDSI does no “messaging” of data and data values in the underlying business data model (i.e. values are not directly deep-serialized). The HDSI interface’s principles are:

- Provide referentially valid data to clients as it exists in the HDS data store.
- Accept referentially valid data from clients as it was presented to the HDS.

- Faithfully store and reproduce the objects received on the wire with minimization of duplication.

7.4.2.1. Transport

The HDSI service is a RESTful service which exposes the resources listed in . The semantics REST API are described in .

Operation	**URL**	**Method**	**Description**
Create	/{resource}	POST	Instructs the HDS to create the specified resource.
Create / Update	/{resource}/{id}	POST	Instructs the HDS to update the specified resource if it exists or create it if it doesn't.
Update	/{resource}/{id}	PUT	Instructs the HDS to update the specified resource.
Patch	/{resource}/{id}	PATCH	Instructs the HDS to apply a patch to the specified object.
Touch	/{resource}/{id}	TOUCH	Instructs the HDS to set the current modification time of the specified resource to the current system time. This does not result in a new version, as no clinical data changes.
Query	/{resource}?{query}	GET	Instructs the HDS to perform the specified query.
Read	/{resource}/{id}	GET	Instructs the HDS to fetch the most recent version of the specified resource.
History Query	/{resource}/{id}/history	GET	Instructs the HDS to fetch all history of the specified item.
History Read	/{resource}/{id}/history/{vid}	GET	Instructs the HDS to fetch a specific version of the record.
History	/{resource}/history	GET	Instructs the HDS to fetch all changes performed on the specified resource.
Obsolete	/{resource}/{id}	DELETE	Instructs the HDS to obsolete the specified resource.
State	/{resource}?{query}	HEAD	Returns headers which represent the latest resource id, version id and date of modification for the particular resource.
History	/history	GET	Retrieves the complete history of changes on the server matching. The result is a collection of control acts which represent the change.

Patch	/{resource}/{id}	PATCH	Indicates that the client wishes to issue a patch to an existing resource.
Options	/{resource}	OPTIONS	Retrieves a list of options which the server supports.

7.4.2.2. Response Codes

The HDSI will respond with a series of HTTP response codes which detail the outcome of the response. In addition to the HTTP response code, an object will be returned which allows the client to compute the reason why the HTTP error was returned.

Lists the response codes and their meaning in the context of the ISMI.

Code	**Error**	**Description**
200	OK / No Error	This error condition indicates that the entire operation succeeded and the response represents the desired operation.
201	Created	This status code indicates that the resource was created on the server and additional processing may occur.
302	Moved	This status code indicates a redirect. This is used when a request is made to an HDSI interface on a server where a remote IDataPersistence is configured (such as in a load balancing or migration scenario).
400	Bad Request	This response code indicates that there was no possible way for the HDSI to comprehend the request sent to it. This is typically done when a low level processing instruction fails (such as bad compression data)
401	Unauthorized	This response code indicates that the requested resource requires permissions which are above the current permission set of the user. This may indicate that the current user is ANONYMOUS and is trying to access a protected resource, OR may indicate a desire by the HDSI interface for the client to elevate themselves.
403	Forbidden	This response code indicates that the authorized user is not permitted to access the requested resource. This represents a full DENY policy decision.
404	Not Found	This response code indicates that the resource requested cannot be found.
405	Method not allowed	This response code indicates that the client attempted to perform an operation on the HDSI interface which is not permitted.
409	Conflict	This response code indicates that an update failed because of a formal validation constraint. Or a patch application failed due to test failure.
410	Gone	This response code indicates that the resource being requested DID exist, however has since been obsoleted.
415	Unsupported Media Type	This response code indicates that it is not possible for the HDSI to process the request content. This error typically

		occurs when the client is submitting data which is not JSON nor XML.
422	Entity could not be processed	The server understood the request (content/type and content) however was unable to process the request due to some state issue or business rule violation.
500	Server Error	This error indicates that the server encountered an error while trying to perform the operation.
503	Service Unavailable	This error indicates that the HDSI service is temporarily unavailable due to current startup or partial startup (i.e. an error occurred starting the HDS).

7.4.2.3. Resource Types

The HDSI exposes resources contained in the underlying business model API layer. In addition to these resources, the HDSI exposes several "meta" resources which wrap complexities around appointment/scheduling. The HDSI specific resources are found in the http://openiz.org/HDSi rather than the http://openiz.org/model namespace.

Table 11 - Resource Types

Resource	**Namespace**	**Description**
Concept	http://openiz.org/model	Represents a concept such as "arm" or "OPV".
ReferenceTerm	http://openiz.org/model	Represents the wire representation of a concept such as a CVX or ICD code.
Act	http://openiz.org/model	Represents a general action that is or was performed such as a concern.
Observation	http://openiz.org/HDSi	A classification of Act which represents the observing of some value. Sub-classed resources are TextObservation, CodedObservation, and QuantityObservation.
PatientEncounter	http://openiz.org/model	A classification of Act which represents an encounter that the patient has or will have with a health provider.
SubstanceAdministration	http://openiz.org/model	Represents the administration of a substance to a patient.
Entity	http://openiz.org/model	A generic class representing an unclassified entity.
Patient	http://openiz.org/model	A classification of an entity representing a Patient.
Provider	http://openiz.org/model	A classification of an entity representing a healthcare provider.

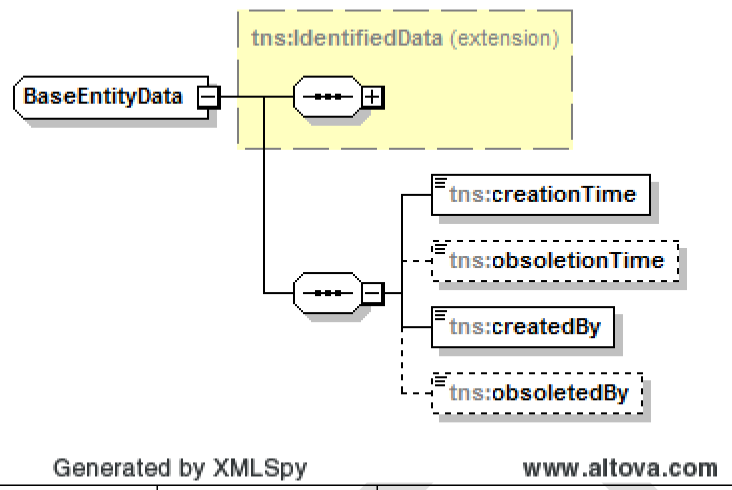
Organization	http://openiz.org/model	A classification of an entity representing an organization.
Place	http://openiz.org/model	A classification of an entity which represents a place where health services are delivered.
Material	http://openiz.org/model	A classification of entity which represents some sort of material such as a kit, classification of drug or boxed item.
ManufacturedMaterial	http://openiz.org/model	A classification of entity which represents a manufactured material. Manufactured materials are those acquired from a manufacturer.
Bundle	http://openiz.org/model	A collection of SanteDB elements bundled for the purpose of transporting referenced objects.
ConceptSet	http://openiz.org/model	A series of concepts which are grouped together for some purpose such as roles, statuses, etc.
ExtensionType	http://openiz.org/model	Represents a definition of an extension which can be applied when communicating with the particular server.
AssigningAuthority	http://openiz.org/model	Represents the registered assigning authorities available on the HDSI server.
ConceptRelationshipType	http://openiz.org/model	Represents classifications of the concept relationship types.
PhoneticAlgorithm	http://openiz.org/model	Represents phonetic algorithm information.
Procedure	http://openiz.org/model	Represents procedure information (see logical data design)

7.4.2.3.1 Base Types

The HDSI resources extend a series base classes which, while not resources per se, are included here to ensure the type definitions which follow are more concise.

Table 12 provides a summary of the base entity data class. The base entity data class is use to convey data which is related to a data entity where the data is attributed.

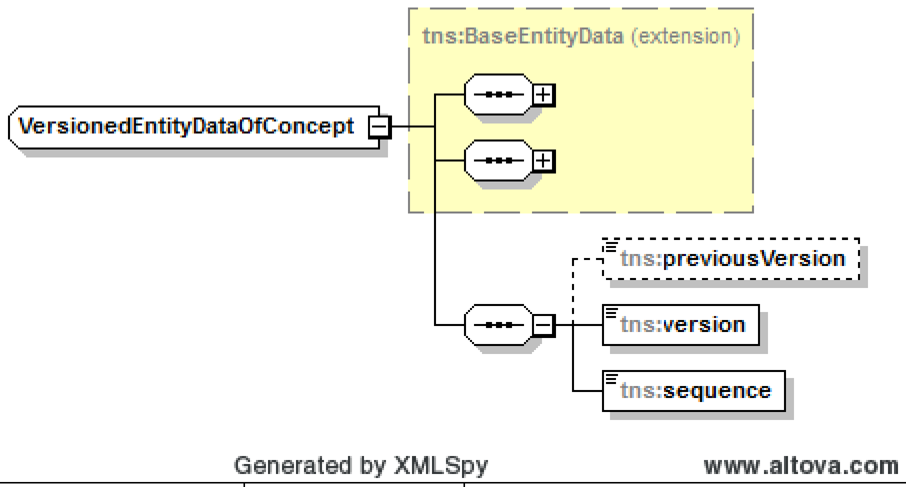
Table 12 - Base Entity Data



Element	Type	Description
id [1..1]	UUID	Uniquely identifies the business model entity.
creationTime [1..1]	DateTime	Identifies the full timestamp (from the service) when the specified data was created.
obsoletionTime [0..1]	DateTime	When present, indicates the date/time that the entity became or will become obsolete.
createdBy [1..1]	UUID	Identifies the user who was responsible for the creation of the data model entity.
obsoletedBy [0..1]	UUID	Identifies the user who was responsible for the obsoletion of the data model entity.

Table 13 illustrates the versioned entity data class is a generic class and indicates common data elements which are required for entities which are versioned in the SanteDB system.

Table 13 - Versioned Entity Data

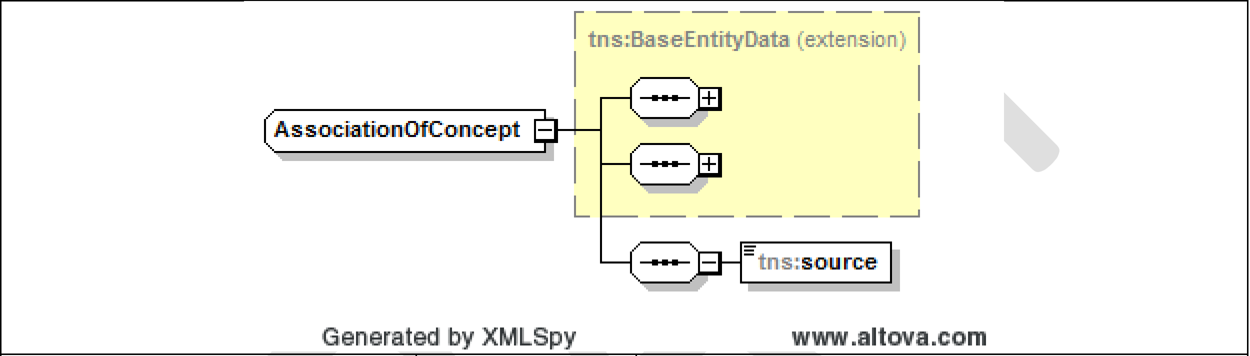


Element	Type	Description

previousVersion [0..1]	UUID	The version identifier of the previous version of this entity.
version [1..1]	UUID	The unique identifier for the represented version.
sequence [1..1]	int	An incrementing serial number which is used to identify the order in which the version was created. This number is not consistent between systems in value, however must be consistent in ordering.

Table 14 conveys how the association class is used to track a simple association between a source and the current item. The association classes point to the source of their association.

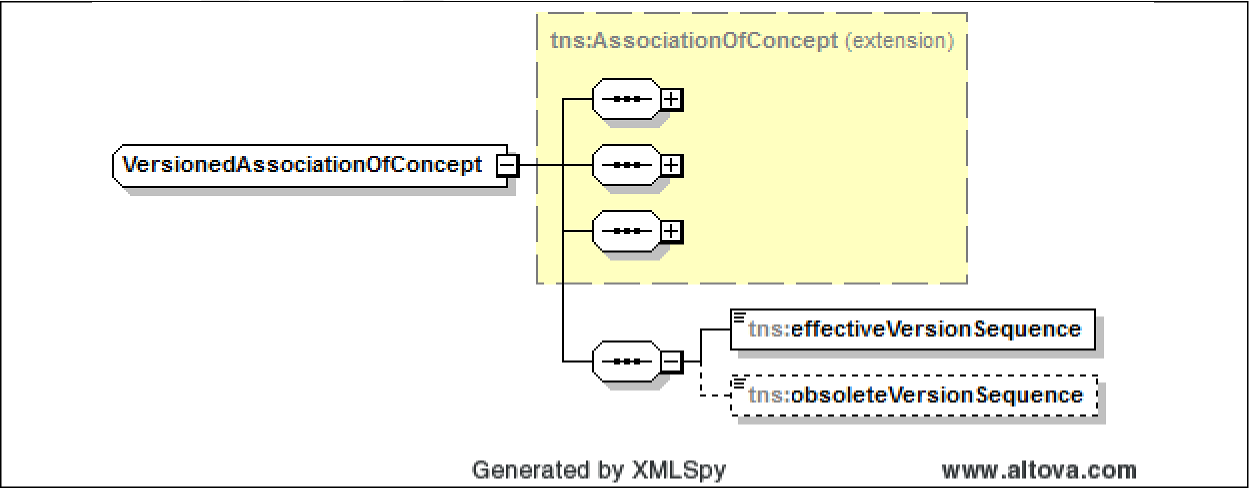
Table 14 - Association



Element	Type	Description
source [1..1]	UUID	The source data entity to which the associative class is associated.

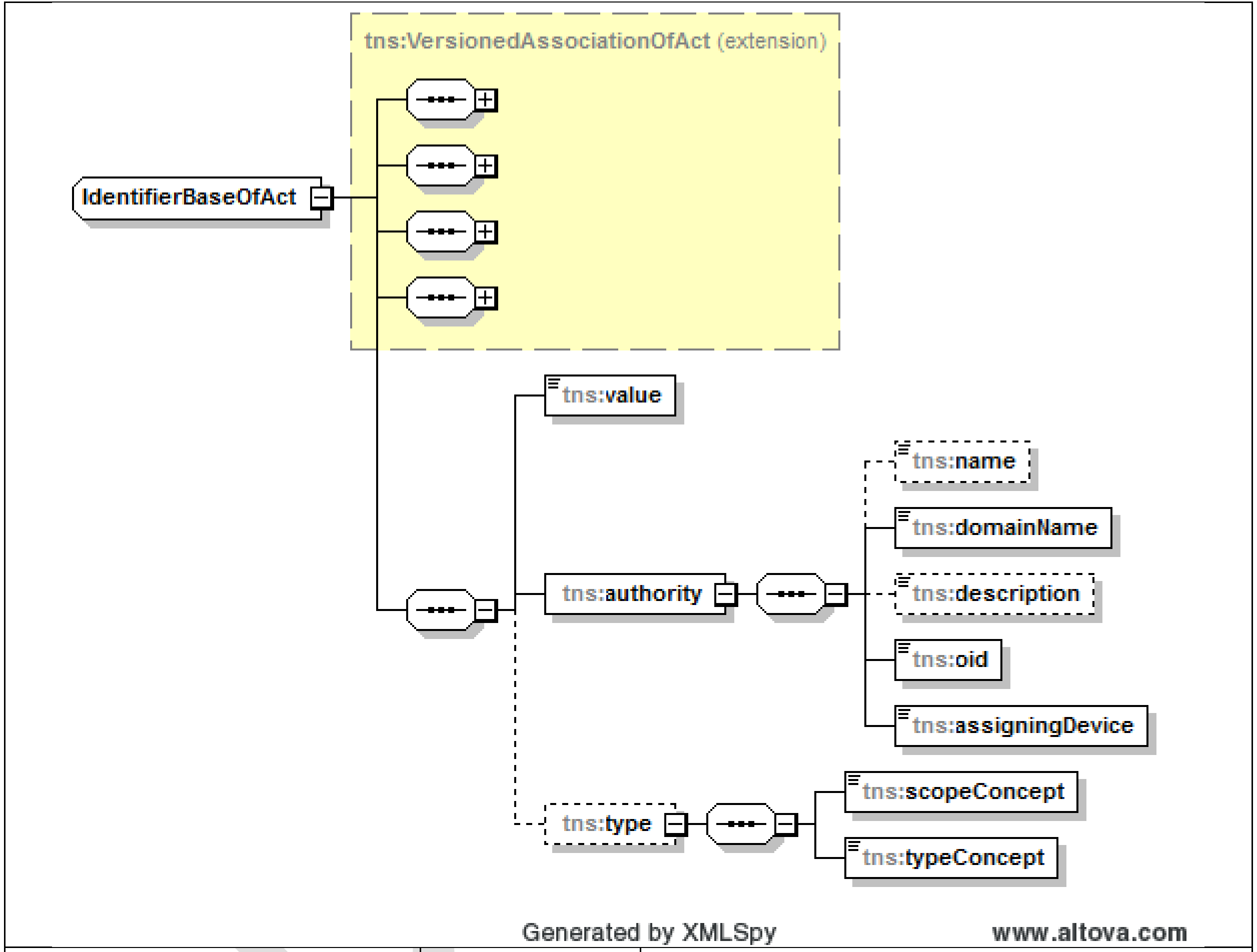
The versioned association class represents a special association whereby the associative entity is applied to a specified series of versions (Table 15).

Table 15 - Versioned Association


tns:AssociationOfConcept (extension)
VersionedAssociationOfConcept
tns:effectiveVersionSequence
tns:obsoleteVersionSequence
Generated by XMLSpy
www.altova.com

Element	Type	Description
effectiveVersionSequence [1..1]	int	The effective version of the ***source*** entity to which the associative entity is active.
obsoleteVersionSequence [0..1]	int	The version of the ***source*** entity where the associative entity is no longer active.

Alternate identifiers (identified as element "identifier") are represented in entities as illustrated in .



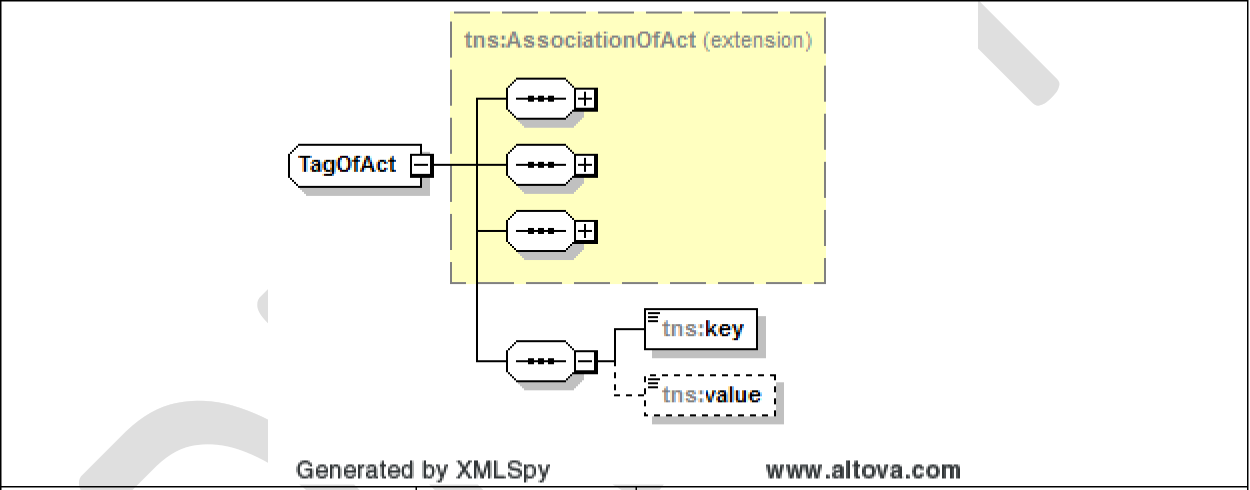
Element	Type	Description
value [1..1]	String	The value of the identifier as assigned by the specified authority.
authority [1..1]	Authority	Represents information about the source authority from which the identifier is assigned.
authority.name [0..1]	String	The optional name of the authority. For example: Good Health Hospital Systems
authority.domainName [1..1]	String	The domain name of the authority. In HL7v2 speak the CX.4 value. Example: GHHS
authority.description [0..1]	String	A human readable description of the assigning authority.
authority.oid [1..1]	String	The OID of the authority. Example: 1.2.3.4.5

authority.assigningDevice [0..1]	UUID	The identifier of the SecurityDevice which is allowed to assign identifiers.
type [0..1]	Type	Identifies the type of the identifier. Example: Business, Stock, etc.
type.scopeConcept [1..1]	UUID	Identifies the allowed scope of identifier use. This maps to the classCode on Entity and Act classes where the identifier can be used.
type.typeConcept [1..1]	UUID	Identifies the type of identifier. Example: Stock, Business, etc.

7.4.2.3.2 Extensions & Tags

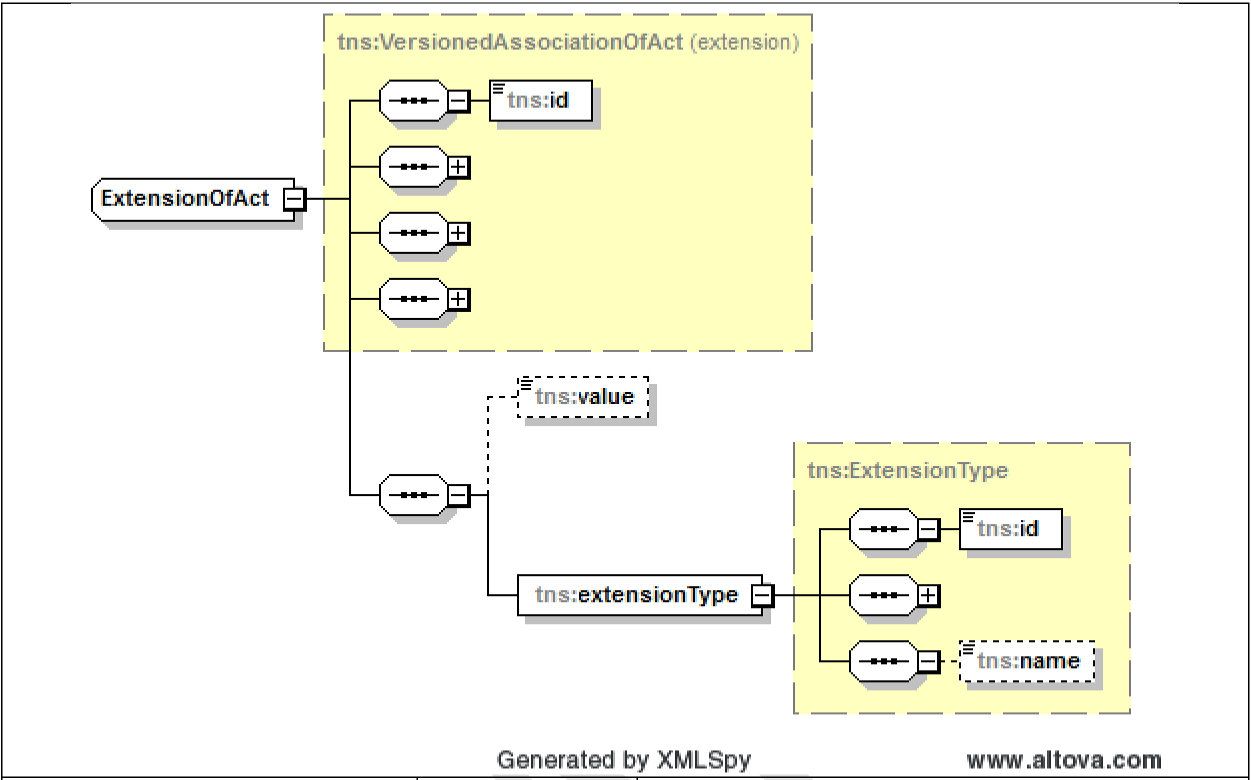
The HDSI interface represents the extension and tag values stored in the underlying data model. Extensions and tags are used to extend HDS data model classes to support use cases not envisioned in the original design.

Tags are version independent data elements which are attached to acts and entities and used for things such as workflow control, and security.



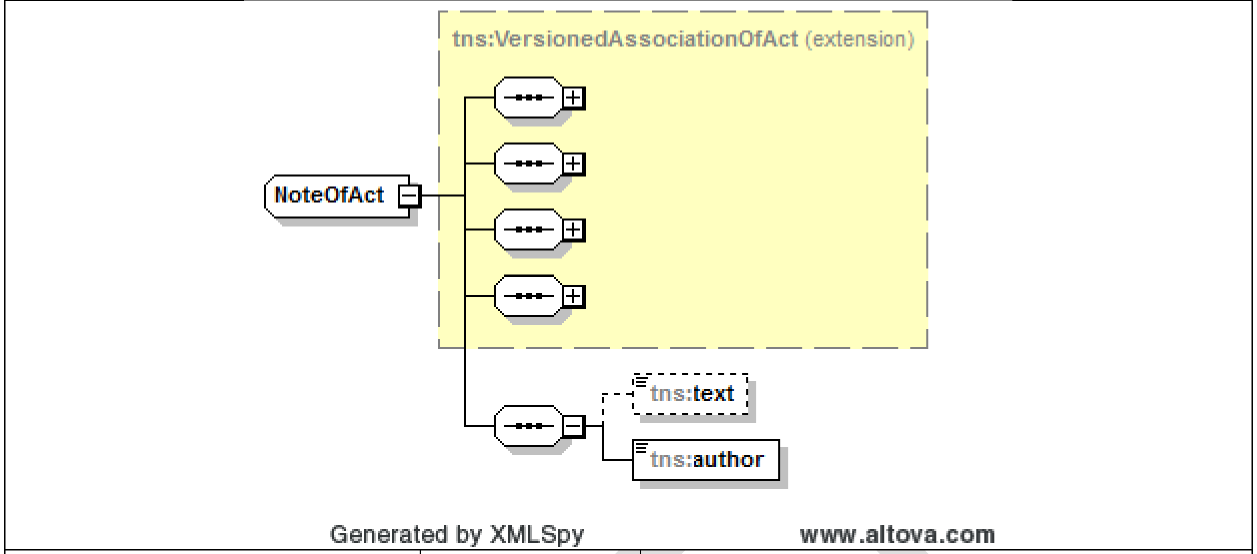
Element	Type	Description
key [1..1]	String	Identifies the type of tag applied to the act/entity.
value [0..1]	String	Carries the value of the tag. Some tags may be indicator flags (i.e. the presence of the tag indicates something).

Extensions are used to represent data elements which are associated with the clinical meaning of an act or entity. Unlike tags, extensions can carry more robust data and typically result in new versions of the associated entity and acts.



Element	Type	Description
value [0..1]	Base64binary	The serialized value of the extension represented in the instance.
extensionType [1..1]	Extension Type	The type of extension. Indicates how the value of the extension should be serialized and/or parsed and interpreted.
extensionType.id [1..1]	UUID	Uniquely identifies the extension type.
extensionType.name [0..1]	String	The human readable name of the extension type.

Notes are textual information which are intended to be displayed verbatim to human participants. Notes, like tags, are version independent.

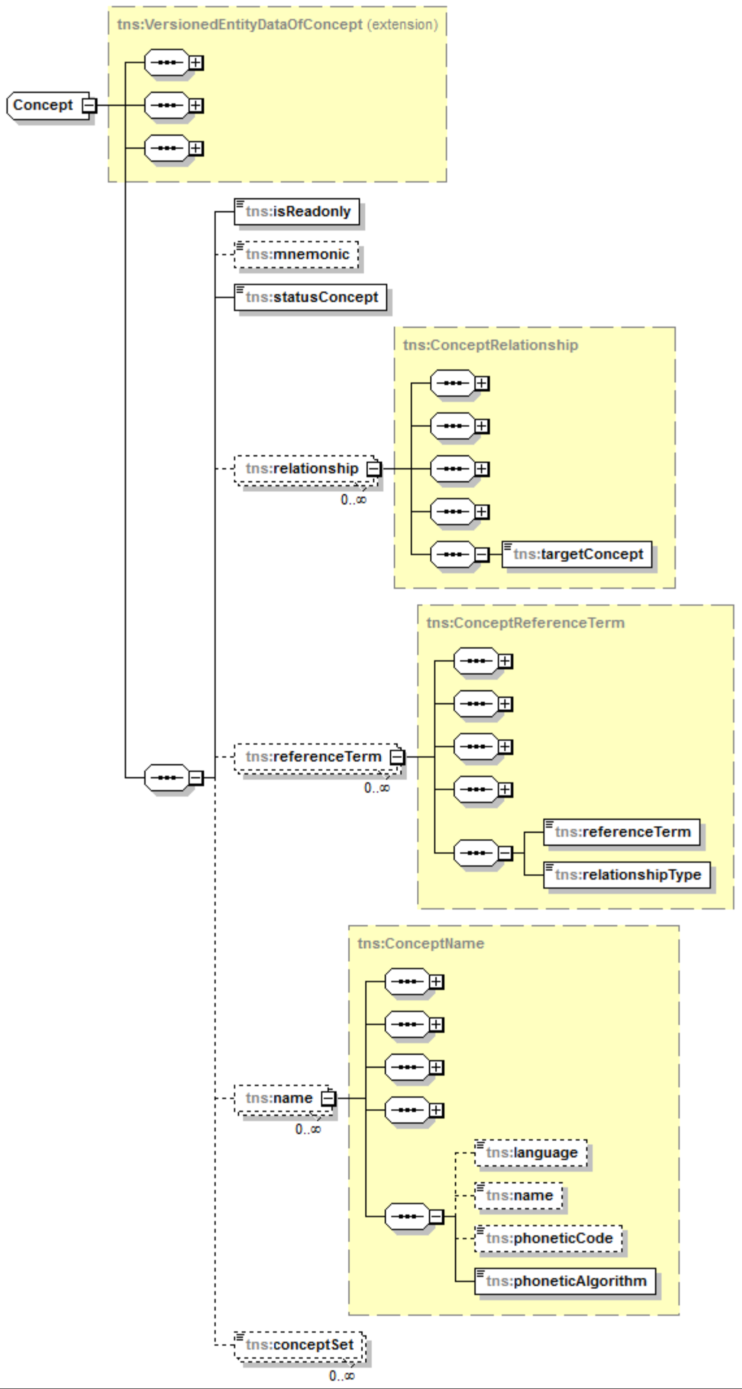


Element	Type	Description
text [0..1]	String	The textual content of the note attached to the act or entity.
author [1..1]	UUID	Identifies the author of the note.

7.4.2.3.3 Concept

The concept resource can be used to get, search, fetch history, create, update and obsolete clinical concepts used in the SanteDB HDS system. Concepts are used to represent abstract facets of care delivery. For example: arm, Polio Vaccine, Allergy, etc.

Table 16 - Concept



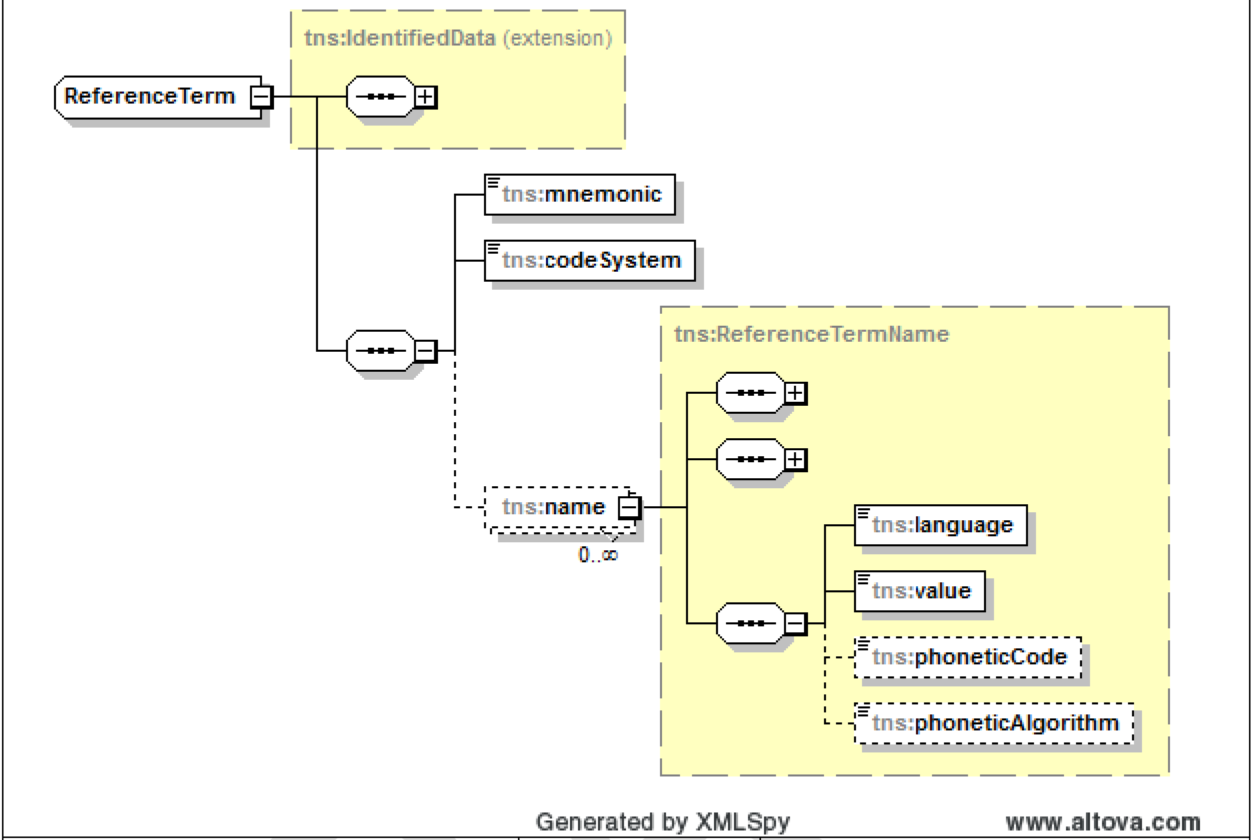
Element	**Type**	**Description**
isReadonly [1..1]	bool	Indicates whether the concept is readonly (i.e. no changes can be made)
mnemonic [0..1]	String	A invariant string which can be used to reference the concept in queries.
statusConcept [1..1]	UUID	The identifier of the status concept which represents the current status of

		the concept as of the represented version.
relationship [0..*]	Relationship	One or more relationships that this concept has with other concepts.
relationship.targetConcept [1..1]	UUID	Identifies the target concept which this concept is associated with.
relationship.relationshipType [1..1]	UUID	Identifies the concept relationship type (same-as, inverse-of, etc.)
referenceTerm [0..*]	ReferenceTerm	One or more reference terms (wire terms) which can be used to represent the concept.
referenceTerm.term [1..1]	UUID	Identifies the term identifier which contains
referenceTerm.relationshipType [1..1]	UUID	Identifies the type of relationship the concept reference term has to the concept. Example: narrower-than, same-as.
name [0..*]	Name	One or more names which can be displayed to represent the concept.
name.language [1..1]	String	The language code in which the name value is represented.
name.value [1..1]	String	The value of the name which can be displayed.
name.phoneticCode [0..1]	String	The phonetic code (for searching) of the name.
name.phoneticAlgorithm [1..1]	UUID	The identifier of the phonetic algorithm which can be used to generate the phonetic code value.
conceptSet [0..*]	UUID	The identifier(s) of concept sets to which the current concept belongs.

7.4.2.3.4 ReferenceTerm

The reference term resource can be used to create, search, obsolete, and get information related to reference terms used in the SanteDB system. Reference terms are used to represent concepts on the wire and are primarily referenced for interoperability reasons.

Table 17 - Reference Term

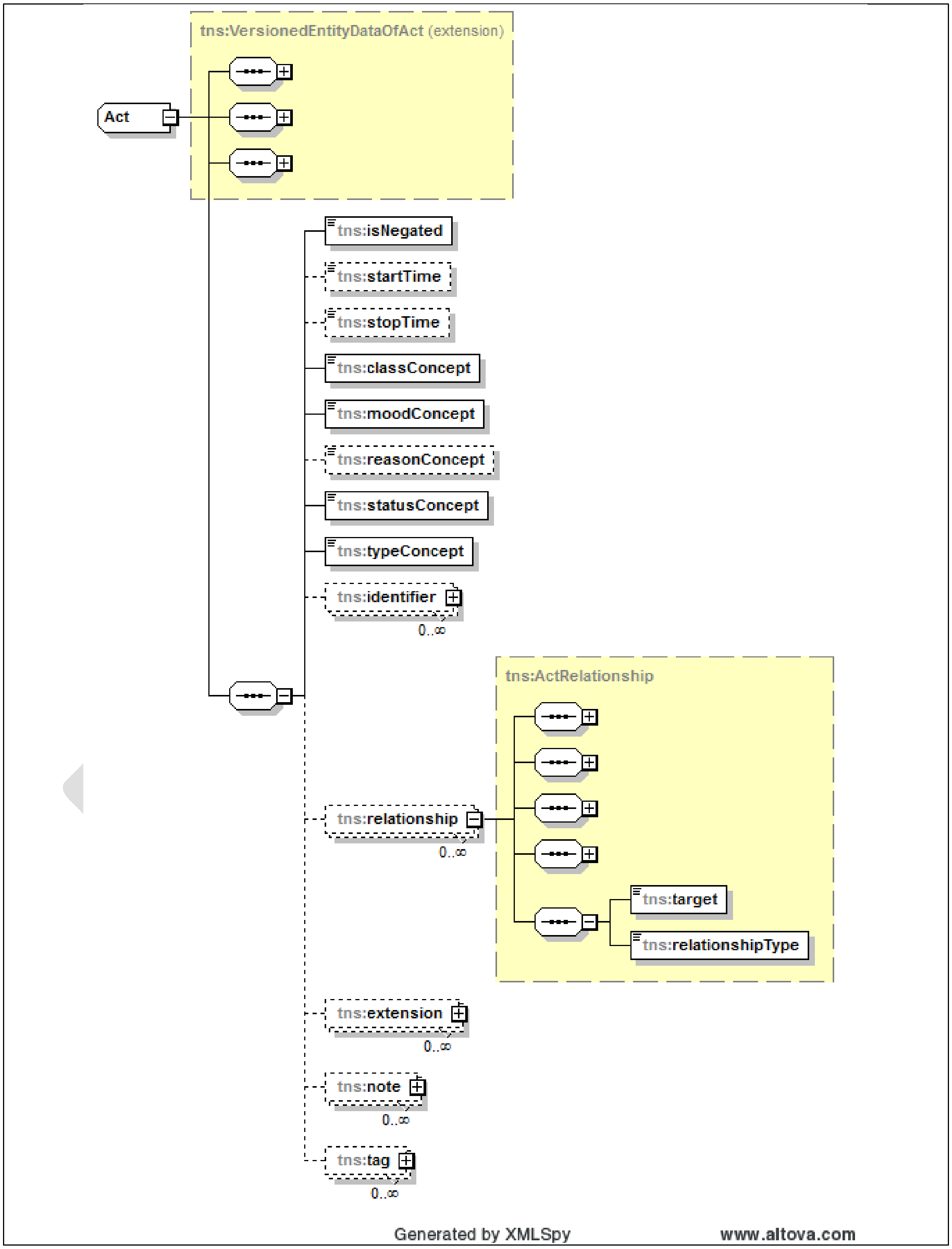


Element	Type	Description
mnemonic [1..1]	String	A mnemonic which is used to represent the reference term on the wire.
codeSystem [1..1]	UUID	Identifies the code system to which the reference term belongs (example: ICD10, LOINC, etc.)
name [0..*]	Name	One or more displayable names from the reference term's standard. These display names are used in interoperability contexts.
name.language [1..1]	String	The ISO language code in which the value of the name is represented.
name.value [1..1]	String	The value of the reference term name in the specified language.
name.phoneticCode [0..1]	String	The phonetic code of the name.
name.phoneticAlgorithm [0..1]	UUID	The phonetic algorithm identifier used to generate the phoneticCode.

7.4.2.3.5 Act

Identifies acts which cannot be classified into one of the concrete act types (observation, substance administration, etc.).

Table 18 - Act


tns:VersionedEntityDataOfAct (extension)
Act
tns:isNegated
tns:startTime
tns:stopTime
tns:classConcept
tns:moodConcept
tns:reasonConcept
tns:statusConcept
tns:typeConcept
tns:identifier
0..∞
tns:ActRelationship
tns:relationship
0..∞
tns:target
tns:relationshipType
tns:extension
0..∞
tns:note
0..∞
tns:tag
0..∞
Generated by XMLSpy
www.altova.com

Element	Type	Description
isNegated [1..1]	Boolean	Indicates whether the act which is represented in the response is a negation. Example: OPV NOT given.
actTime [1..1]	DateTime	Identifies the time that the act occurred.
startTime [0..1]	DateTime	Identifies the time when the act started or is scheduled to start.
stopTime [0..1]	DateTime	Identifies the time when the act ended or is scheduled to end.
classConcept [1..1]	UUID	Identifies concept which classifies the act. For example: classifies observations, administrations, etc.
moodConcept [1..1]	UUID	Identifies the mood of the act. For example: classifies events from intent to perform acts.
reasonConcept [0..1]	UUID	Identifies a codified reason as to why the act took place (or didn't)
statusConcept [1..1]	UUID	Identifies the codified status of the act at the current status.
typeConcept [1..1]	UUID	Identifies the type of act. Further classifies the act (i.e. makes observation a reaction observation)
identifier [0..*]	Identifier	Provides one or more alternate identification schemes for the act.
relationship [0..*]	Relationship	Provides one or more relationships between the current act and other acts. This is used to link acts together by encounter.
relationship.target [1..1]	UUID	The target act of the relationship.
relationship.relationshipType [1..1]	UUID	Identifies the type of relationship such as "component of", "subject of", etc.
participation [0..*]	Participation	Provides one ore more links to entities which participate in the carrying out of the act.
participation.player [1..1]	UUID	The entity which plays the role in the act.
participation.participationRole [1..1]	UUID	Identifies the type of participation such as "performer" or "location", etc.
participation.quantity [0..1]	INT	Identifies the number of entities which participated in the act.
extension [0..*]	Extension	One or more extensions which are used to extend the clinical data contained in the act.

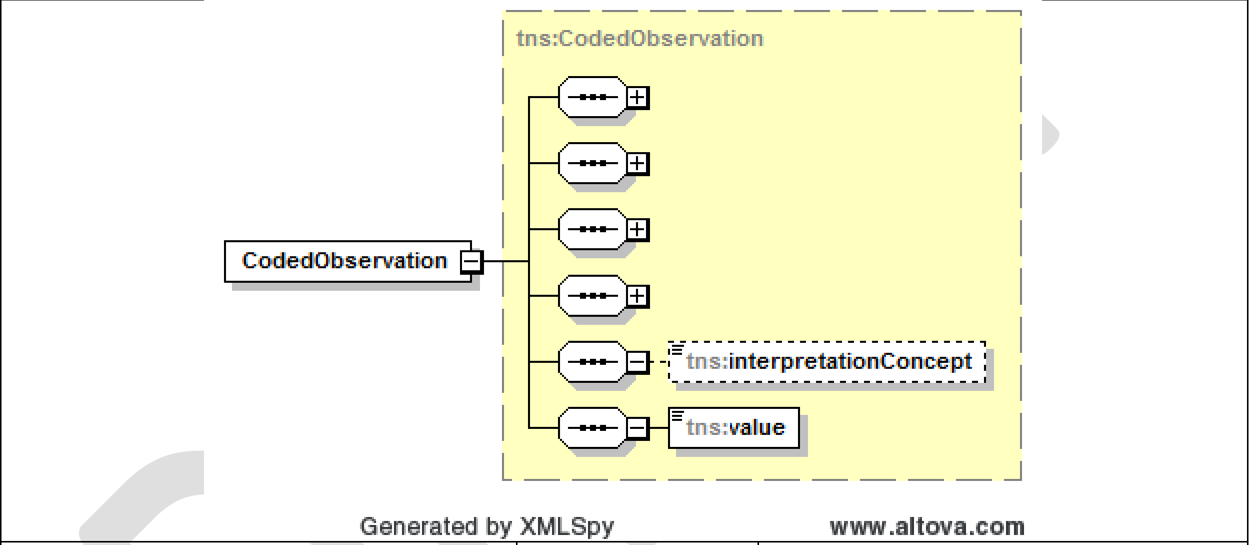
note [0..*]	Note	One or more notes which are used to provide human readable data related to an act.
tag [0..*]	Tag	One or more tags which are used to further classify an act.

7.4.2.3.6 Observation

The observation resource is an HDSI specific observation class which encapsulates the functionality of the three types of observations for convience. Observation allows a consumer to quickly query any three of the observation types.

A coded observation represents an observation whose value is a concept. This type of observation is used whenever an observation is made which can be codified. For example: Observed reaction, severity, etc.

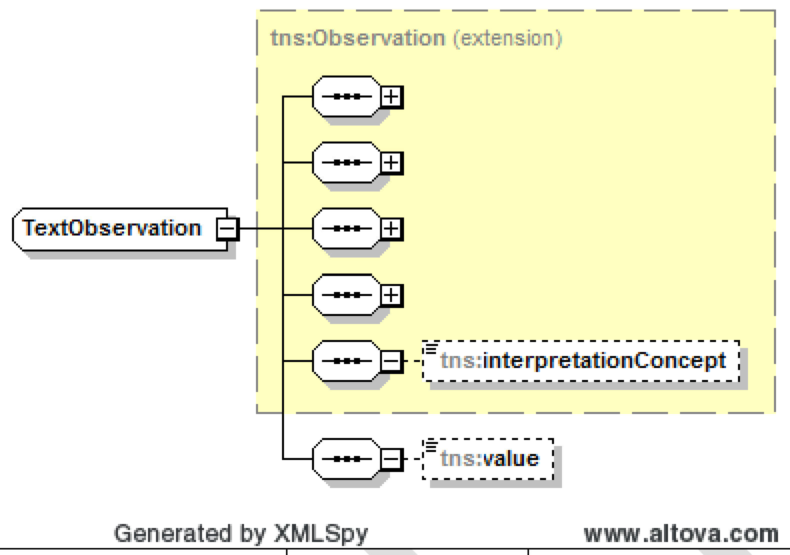
Table 19 - Coded Observation



Element	**Type**	**Description**
interpretationConcept [0..1]	UUID	A concept which provides an interpretation of the observation's value (High, Low, Nominal, etc.)
value [1..1]	UUID	The identifier of the code which represents the value of the observation.

A text observation is used to store an observation whose primary value is textual. These observations are not intended to be used by processing routines, rather are intended for human consumption.

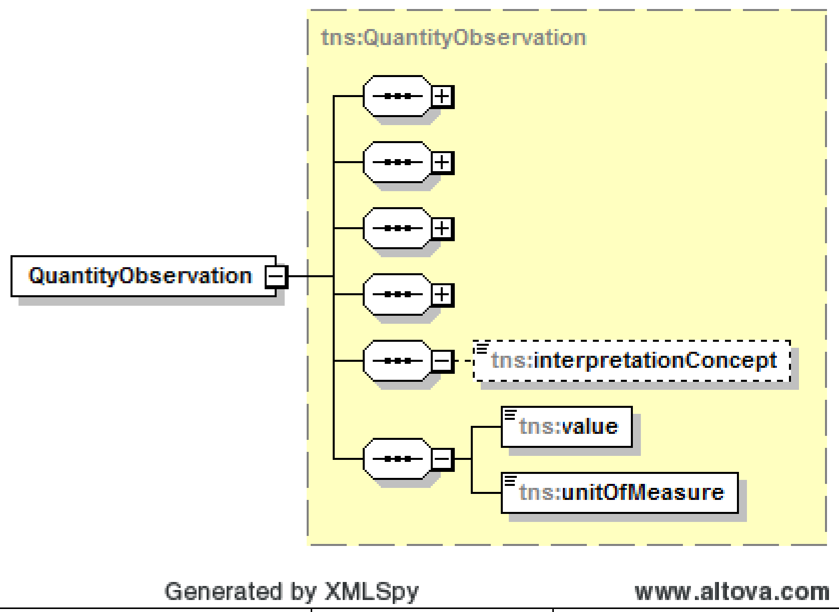
Table 20 - Text Observation



Element	Type	Description
value [0..1]	String	The textual value of the observation.

Finally, quantity observations are used to convey an observation where the observation value was obtained by quantitative measurement. For example: weight, height, bmi, etc.

Table 21 - Quantity Observation

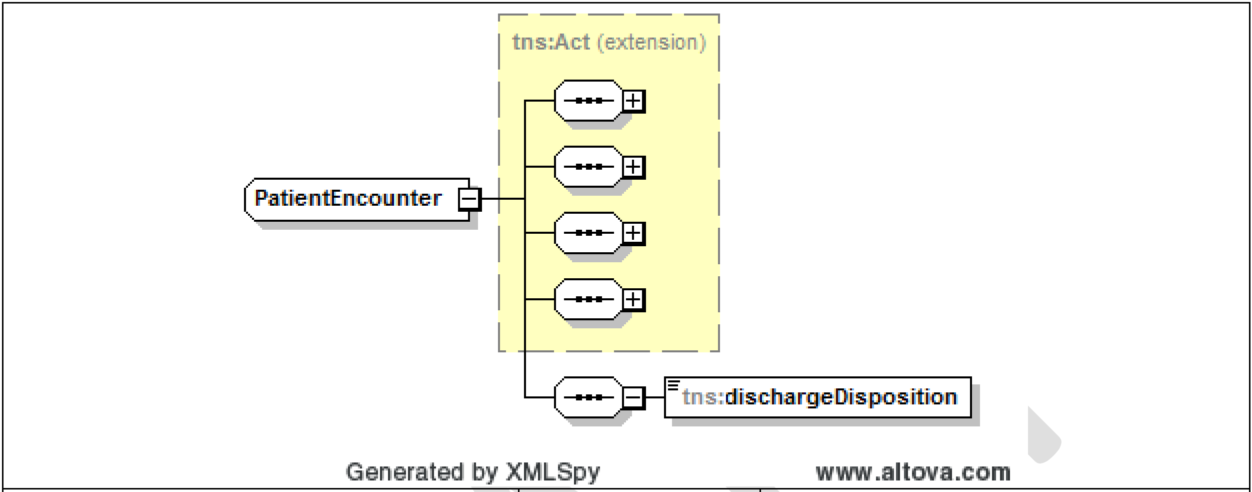


Element	Type	Description
Value [1..1]	Decimal	The numerical value of the observation.
unitOfMeasure [1..1]	UUID	A concept identifier which indicates the units of the value measure.

7.4.2.3.7 PatientEncounter

Patient encounters are used as a means to convey information related to an encounter which the patient has with the health system or is intended to occur (in the case of an appointment).

Table 22 - Patient Encounter

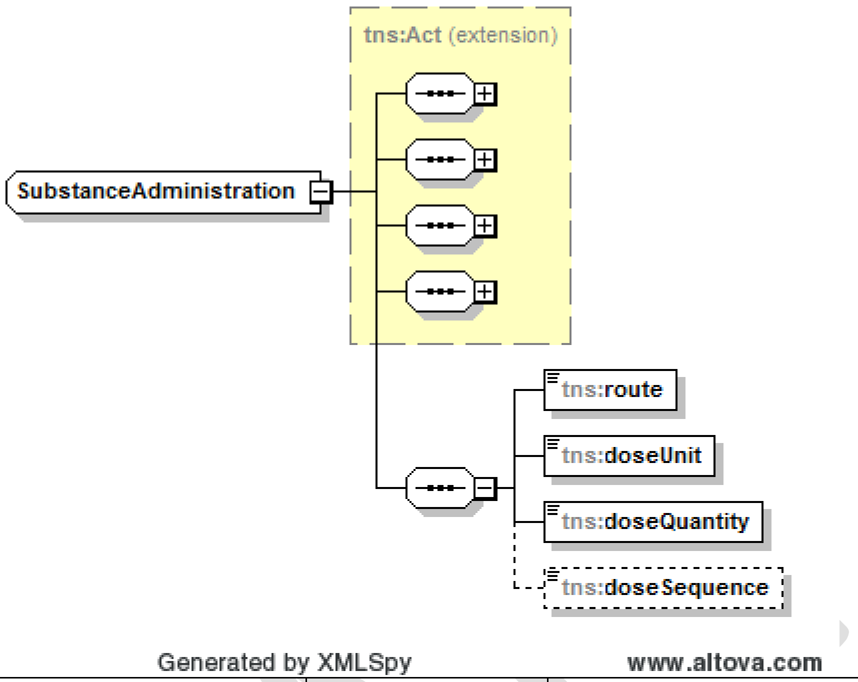


Element	Type	Description
dischargeDisposition [1..1]	UUID	A concept which indicates the method in which the patient left the encounter.

7.4.2.3.8 SubstanceAdministration

Substance administrations represent the administration of any substance to a patient in the context of an encounter. These can include vaccinations, booster shots, EpiPen (anti-histamine) administrations, etc.

Table 23 - Substance Administration

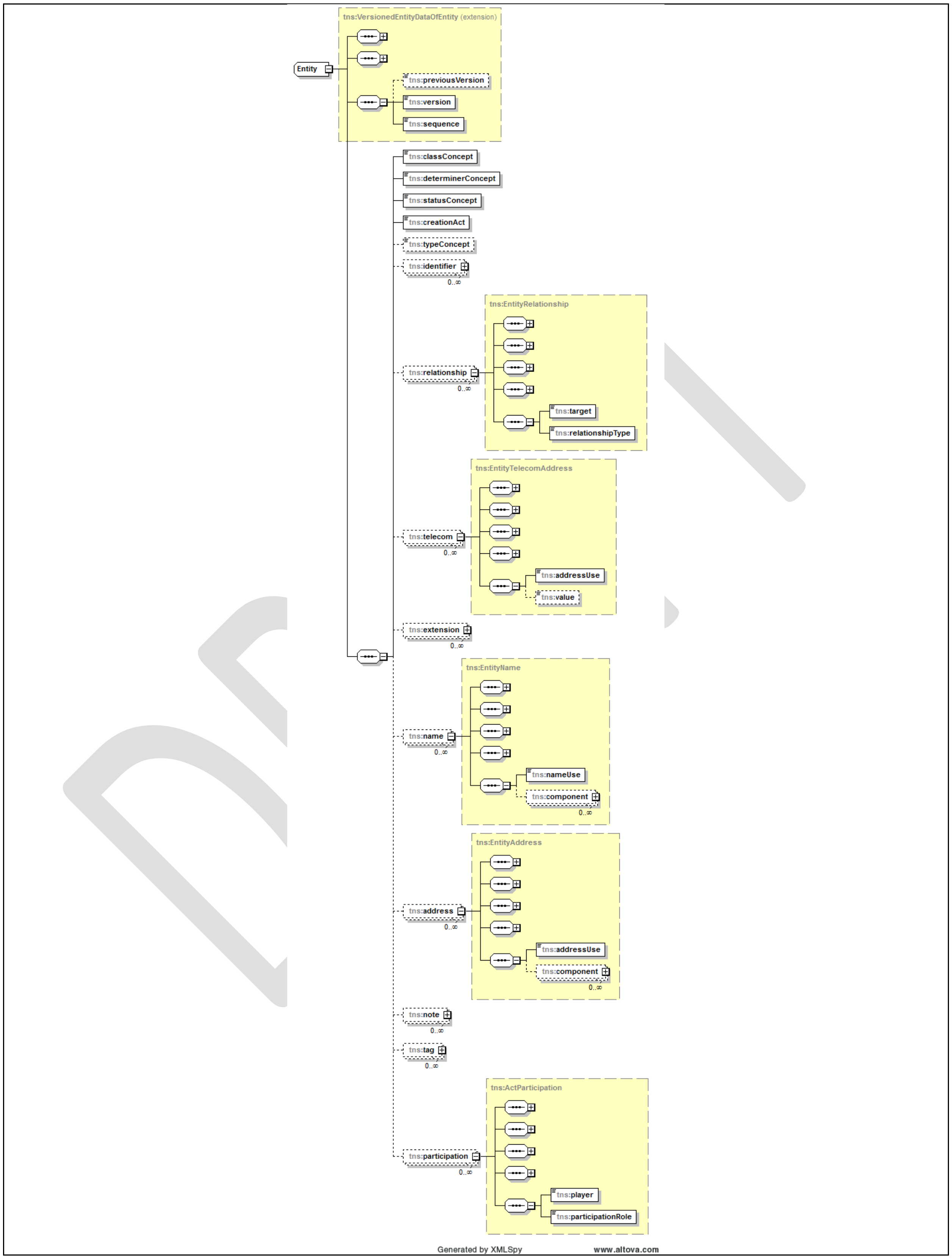


Element	**Type**	**Description**
route [1..1]	UUID	The concept which identifies the route in which the substance was administered.
doseUnit [1..1]	UUID	The unit of measure in which the doseQuantity was administered.
doseQuantity [1..1]	Decimal	The quantity of substance administered.
doseSequence [0..1]	Int	The sequence of the dose.

7.4.2.3.9 Entity

The entity resource is used to represent entities which cannot be classified in one of the other entity subclasses (patient, material, etc.).

Table 24 - Entity



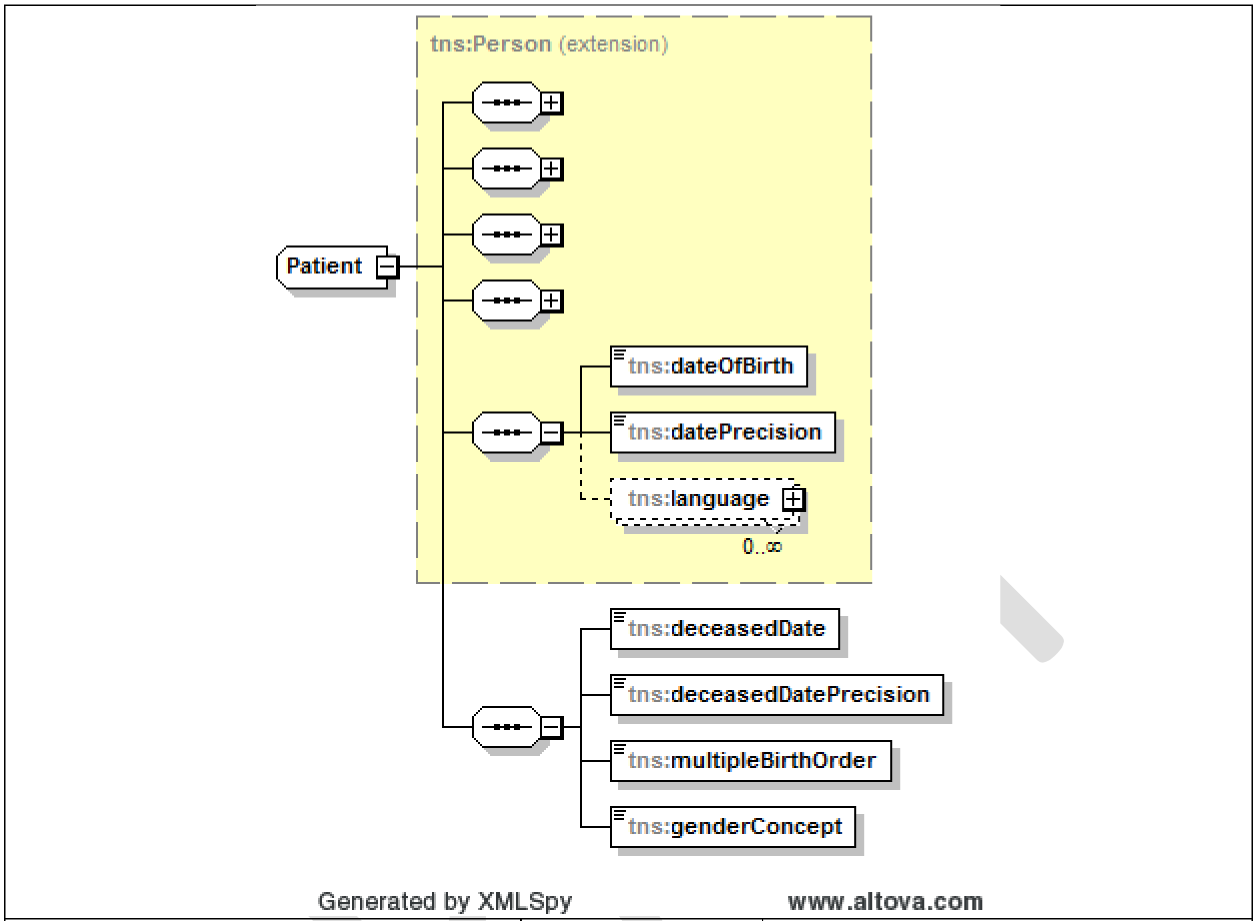
Element	Type	Description
classConcept [1..1]	UUID	Identifies the classification of the entity being represented.
determinerConcept [1..1]	UUID	Identifies whether the entity represents a specific instance of a thing or a type of thing.
statusConcept [1..1]	UUID	Represents the current status of the entity (Active, New, Obsolete, etc.)
creationAct [0..1]	UUID	If applicable, links the entity to the act which created the entity.
typeConcept [0..1]	UUID	Provides a further classification of the type of entity. For example if the class of the entity is Place, the type concept might be “Hospital”
identifier [0..*]	Identifier	When populated, provides a reference to the external identifiers which are associated with the entity.
identifier.authority [1..1]	AssigningAuthority	Provides a link to the authority which has assigned the particular identifier
identifier.authority.domain [1..1]	String	The unique domain name of the authority which assigned the identifier.
identifier.value [1..1]	String	The actual value of the external identifier.
relationship [0..*]	Entity Relationship	Provides one or more links between this entity and other entities which are related.
relationship.target [1..1]	UUID	The identifier of the entity to which this entity is related.
relationship.relationshipType [1..1]	UUID	The concept which classifies the type of relationship between the two entities.
relationship.quantity [0..1]	INT	Identifies the number of target entities involved in the relationship.
telecom [0..*]	Entity Telecom	Used to represent one or more telecommunications addresses which the entity can be contacted using.
telecom.use [1..1]	UUID	Classifies the use of the particular telecom address. (home, work, etc.)
telecom.type [0..1]	UUID	When present, classifies the type of telecommunications address (fax, phone, email, etc.)
telecom.value [1..1]	String	Identifies the actual telecommunications address.
extension [0..*]	Extension	Provides additional data not included in the core patient resource.

name [0..*]	Entity Name	Provides one or more names by which the entity is known.
name.use [1..1]	UUID	Classifies the use of the name, for example: official, legal, artist, etc.
name.component [0..*]	Name Component	Represents each part of the name.
name.component.type [1..1]	UUID	Classifies the type of name component being represented such as Given or Family.
name.component.value [1..1]	String	The actual value of the name component.
address [0..*]	Entity Address	One or more addresses for the entity. An address can be a physical place (street address), a mailing address (PO Box), or directions to a place.
address.use [1..1]	UUID	Classifies the use of the address (example: Postal Address, Physical Address, etc.)
address.component [0..*]	Address Component	One or more components which make up the total address.
address.component.type [1..1]	UUID	Classifies the type of address component (street address, city, postal code, etc.)
address.component.value [1..1]	String	The textual value of the address component.
note [0..*]	Entity Note	One or more notes about the entity such as additional observations or details that a human could meaningfully consume.
note.author [1..1]	UUID	Identifies the author of the textual content.
note.text [1..1]	String	The value of the note
note.creationTime [1..1]	DateTime	The time that the note was authored.

7.4.2.3.10 Patient

The patient resource represents a specialization of the person resource which is used to represent data related to a patient role.

Table 25 - Patient

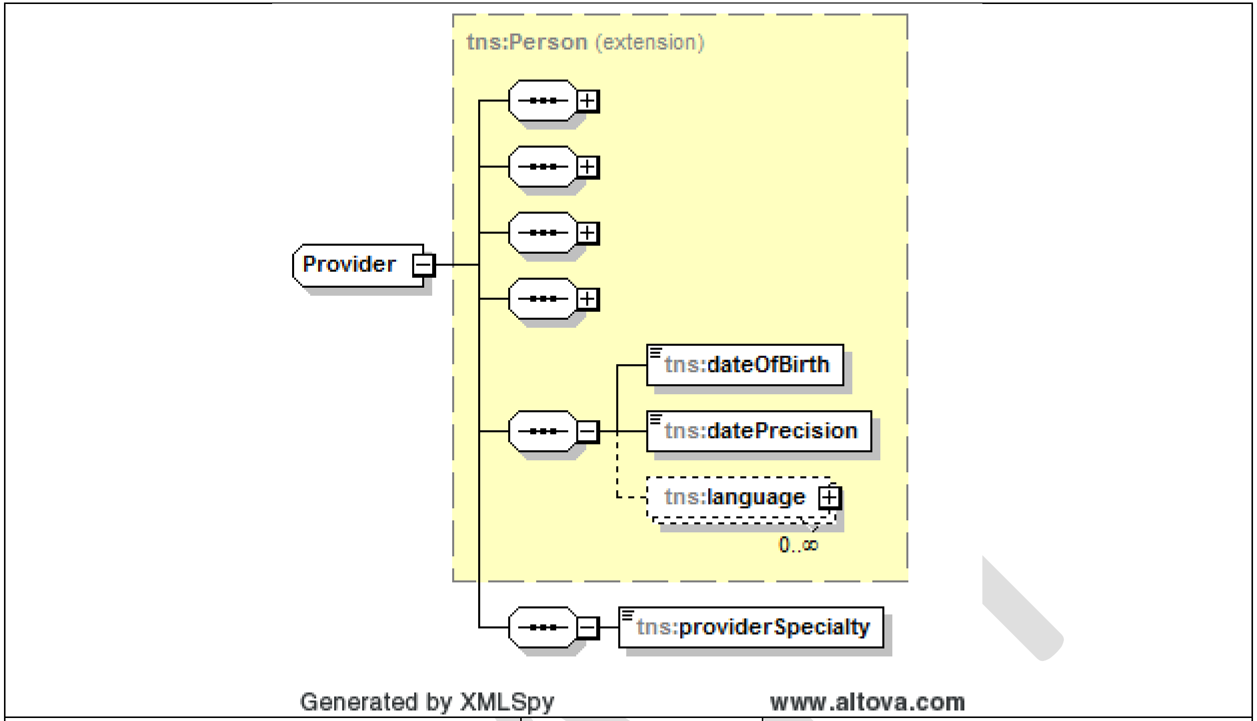


Element	**Type**	**Description**
deceasedDate [0..1]	DateTime	When populated, identifies the date that the patient died.
deceasedDatePrecision [0..1]	DatePrecision	Identifies the precision of the deceased date.
multipleBirthOrder [0..1]	Int	When populated, indicates that the patient was a part of a multiple birth.
genderConcept [1..1]	UUID	Identifies the gender of the patient.

7.4.2.3.11 Provider

A provider represents a specialization of a person which is charged with the delivery of health care services to patients.

Table 26 - Provider

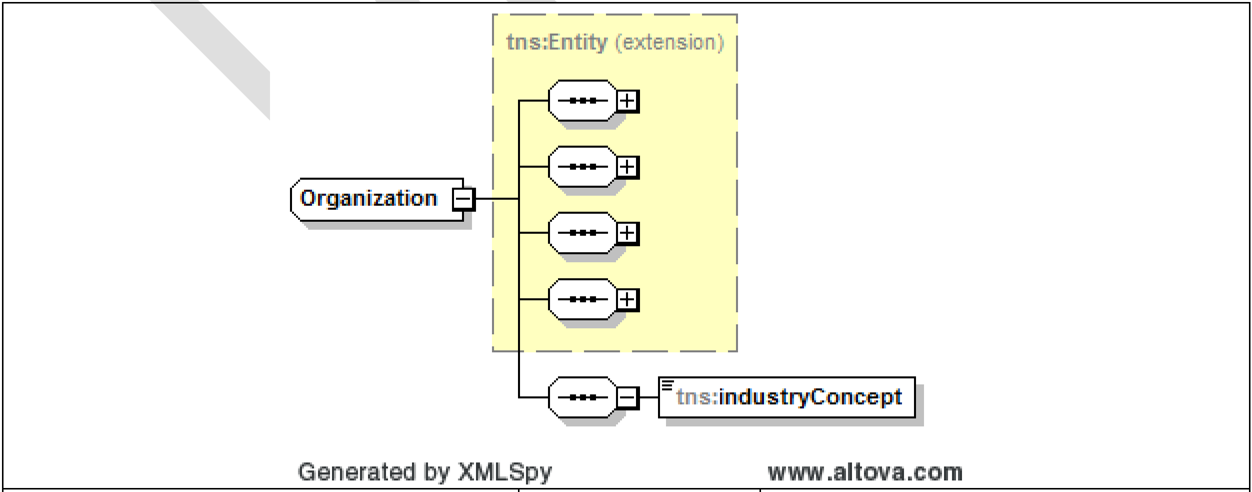


Element	Type	Description
providerSpecialty [1..1]	UUID	A code indicating the type of provider represented by the provider entity. (example: Nurse, CHW, etc.).

7.4.2.3.12 Organization

An organization represents an administrative construct which is used to deliver or organize care, delivery and manufacture of other entities.

Table 27 - Organization



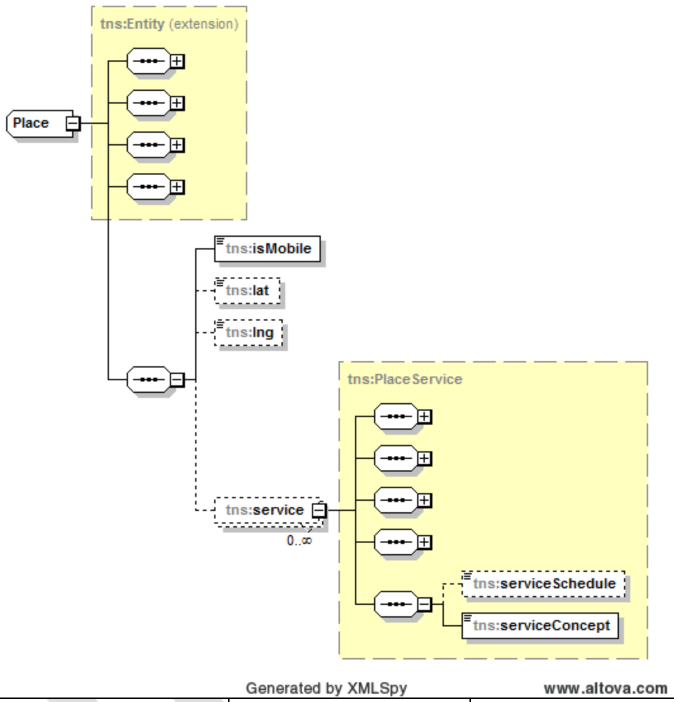
Element	Type	Description

industryConcept [1..1]	UUID	A code indicating the industry in which the organization operates. Examples: supply, hospital, etc.

7.4.2.3.13 Place

A place represents an entity which is a physical location where health delivery services are delivered such as clinics, hospital, mobile immunization clinics, etc.

Table 28 - Place

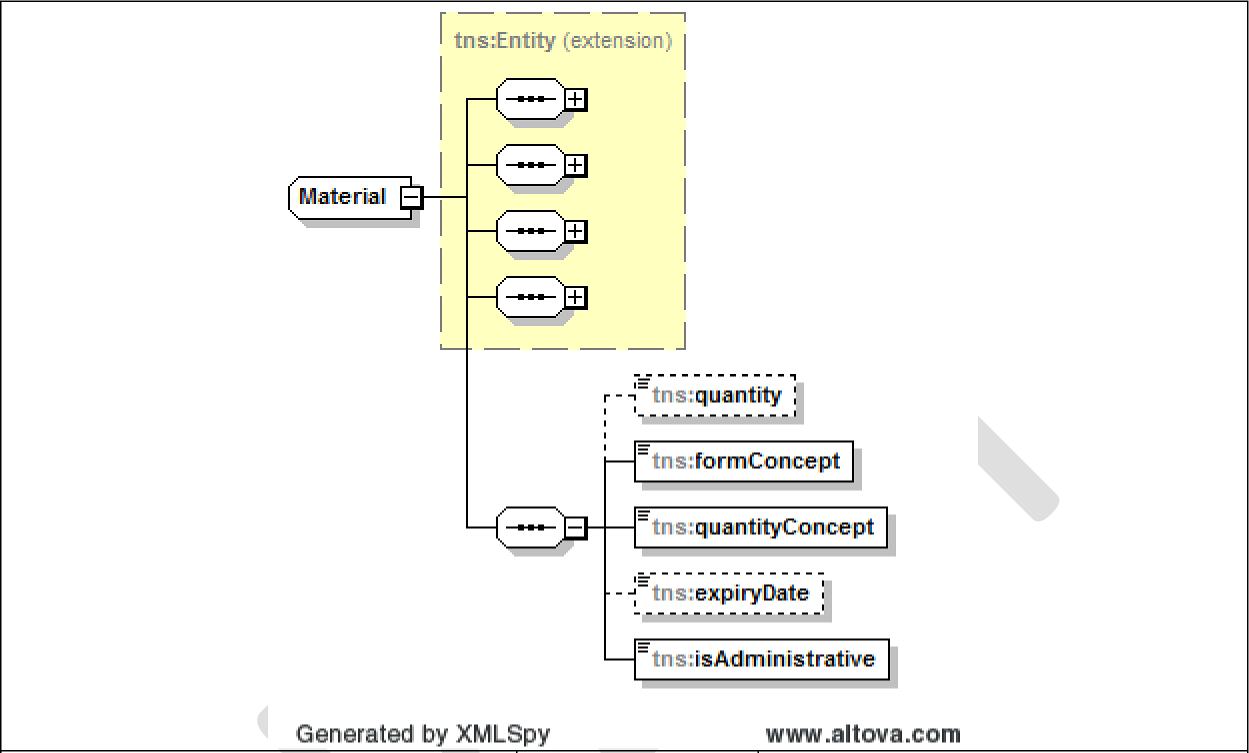


Element	Type	Description
isMobile [1..1]	Boolean	An indicator which identifies whether the delivery location is a mobile location.
lat [0..1]	Float	Indicates the latitude of the place.
lng [0..1]	Float	Indicates the longitude the place.
service [0..*]	Service	One or more services which the place offers.
service.serviceSchedule [0..1]	Xml	An XML representation of the scheduled of the service availability.
service.serviceConcept [1..1]	UUID	The type of service provided.

7.4.2.3.14 Material

A material represents an entity which can be distributed, consumed, packaged, or used. A material can be a physical material such as a distribution kit, device, etc.

Table 29 - Material

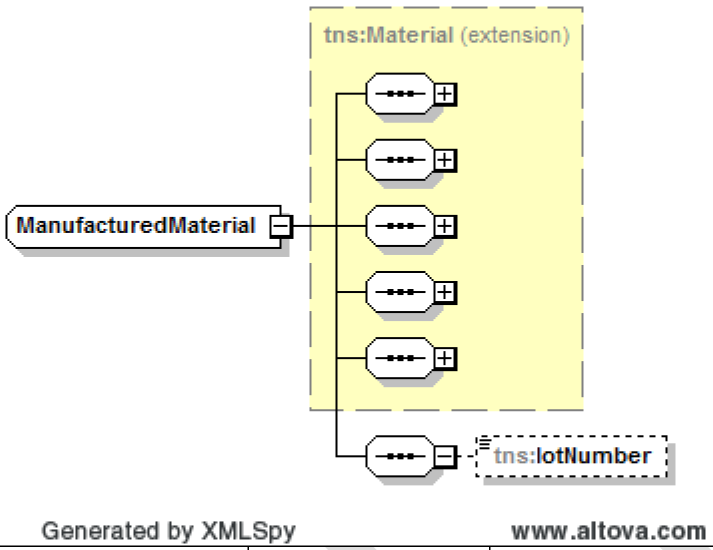


Element	Type	Description
quantity [0..1]	Decimal	Identifies the quantity of material in a container or group if populated.
formConcept [1..1]	UUID	Identifies the form of the material. For example: liquid, gas, boxed, etc.
quantityConcept [1..1]	UUID	Identifies the units of measure for the material when administered or consumed.
expiryDate [0..1]	DateTime	The time when the material will expire or did expire.
isAdministrative [1..1]	Boolean	When true, indicates that the material is not a consumable and is merely an administrative construct.

7.4.2.3.15 ManufacturedMaterial

A manufactured material represents a material which is acquired from a manufacturer.

Table 30 - Manufactured Material

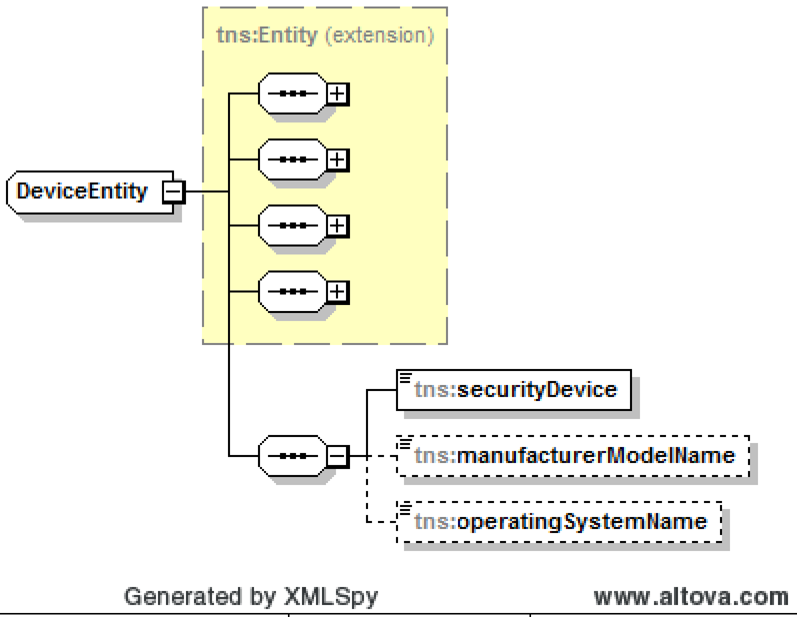


Element	Type	Description
lotNumber [0..1]	UUID	The lot number of the manufactured material.

7.4.2.3.16 Device

A device represents an entity which is physically used. A device entity is differentiated from a SecurityDevice in that the DeviceEntity is used primarily as information augmenting clinical data whereas a SecurityDevice contains security identity information.

Table 31 - Device



Element	Type	Description

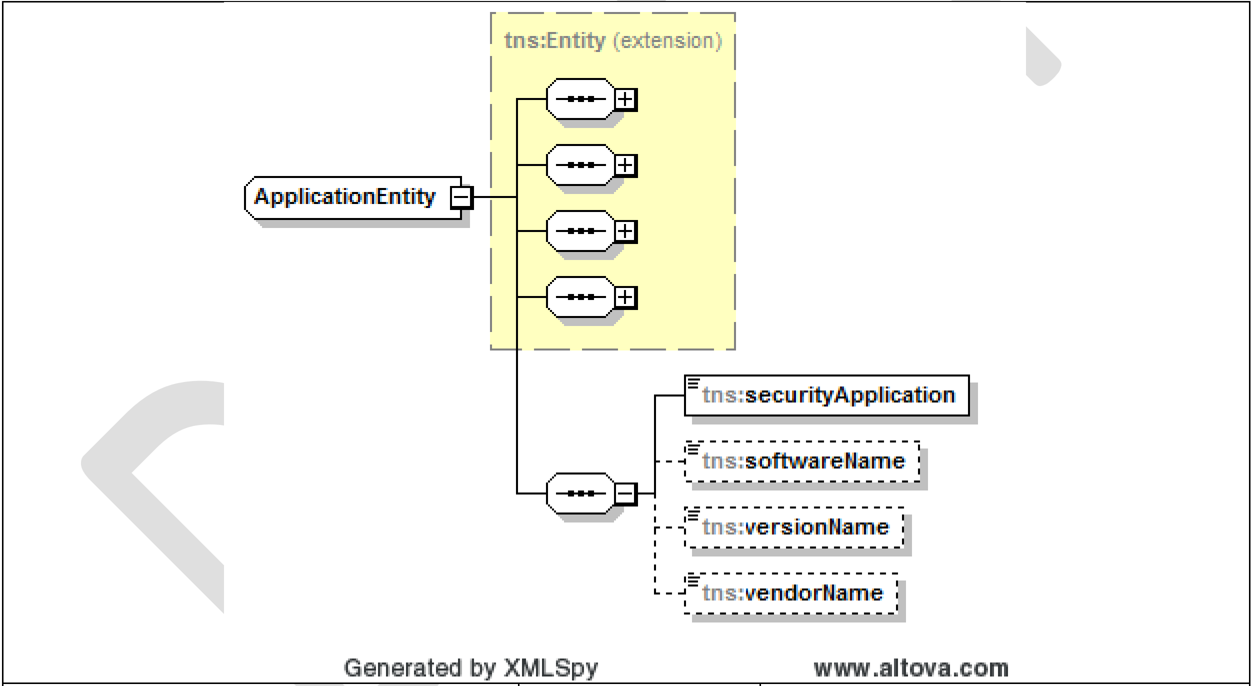
securityDevice [0..1]	UUID	Identifies the security device which the device entity represents. If applicable.
manufacturerModelName [0..1]	String	Identifies the manufacturer name of the device entity.
operatingSystemName [0..1]	String	Identifies the operating system of the device entity.

7.4.2.3.17 User

The user entity represents information in the clinical context of the SanteDB data store. It is differentiated from a SecurityUser in that the user entity is focused more on the clinical data related to a user (used for provenance) whereas a SecurityUser is primarily concerned with the authentication and security enforcement.

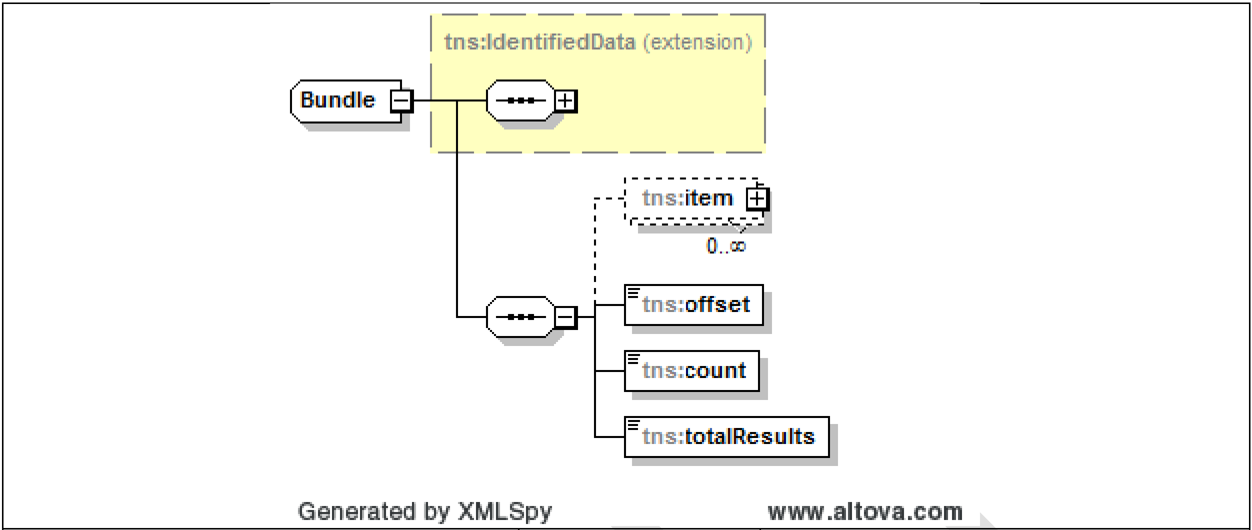
7.4.2.3.18 Application

The application entity represents information in the clinical context of the SanteDB clinical data store.



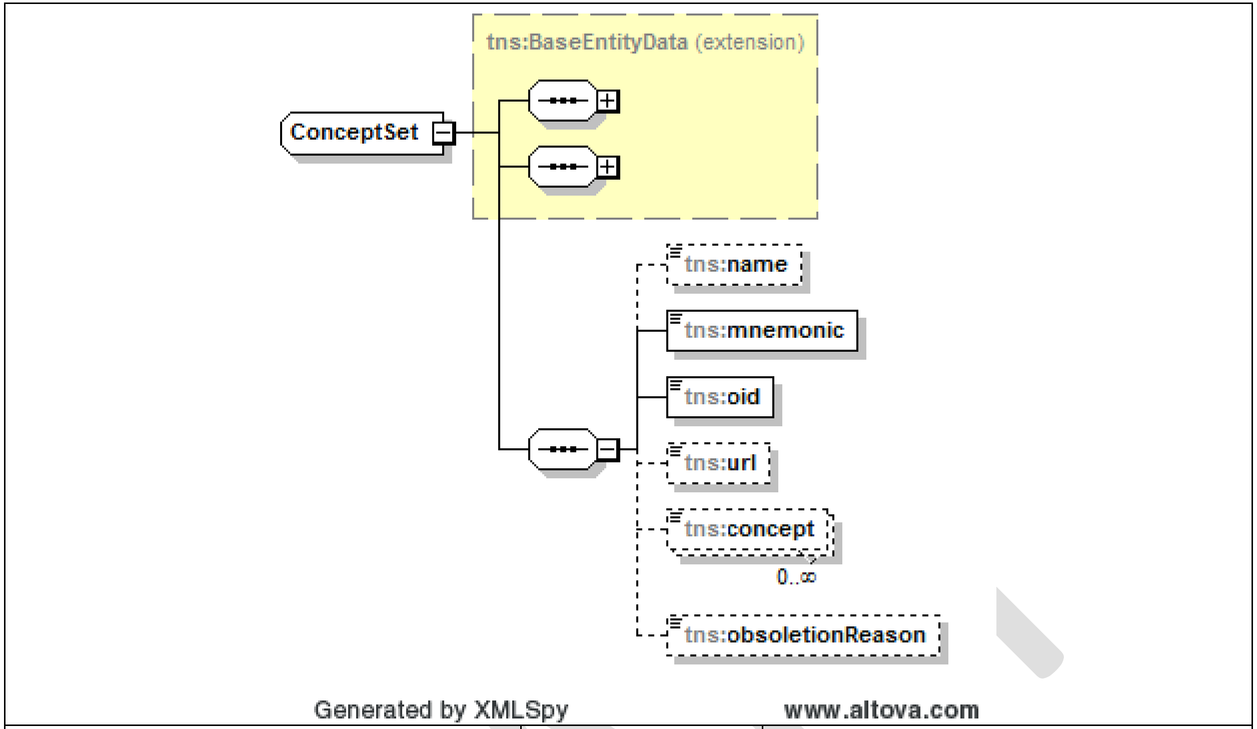
Element	Type	Description
securityApplication [0..1]	UUID	Identifies the SecurityApplication which the application entity represents if applicable.
softwareName [0..1]	String	The name of the software package.
versionName [0..1]	String	The name of the version of the software package.
vendorName [0..1]	String	The name of the vendor which manufactured the software application entity.

7.4.2.3.19 Bundle



Element	Type	Description
item [0..*]	Resource	Represents the contents of the bundle which are the resources being transported to another system.
offset [1..1]	Int	The offset of the bundle in the case of a large result set.
count [1..1]	Int	The count of the primary items in the bundle.
totalResults [1..1]	Int	The total number of primary items in the bundle.
entryItem [0..1]	UUID	Identifies the entry point in the bundle when the bundle is used to package a single result with referenced items.

7.4.2.3.20 ConceptSet



Element	Type	Description
mnemonic [1..1]	String	A mnemonic which is used to represent the reference term on the wire.

7.4.2.4. HTTP Conditional Headers

The HDSI supports the HTTP conditional headers If-Modified-Since and If-None-Match. These headers control the amount of data sent back when there are no modifications to any data. The If-None-Match header is either the version key or the key of the last modified object in the case of a bundled query, or the primary result in the case of the GET operation.

If the “If” condition fails, then the HDS server will respond with an HTTP 304 (not modified).

The HTTP HEAD operation is used to fetch metadata about a particular resource without fetching the resource itself. When a client sends HTTP head, the HDSI interface will respond with data such as version-identifier and sequence of the last edited resource or the metadata of the resource in question. For example, an HTTP HEAD operation on a resource instance would yield the following:

```
HEAD /Patient/fb00e97a-bdfc-403d-8f62-52f8e6846a16
Authorization: BASIC ZmlkZGxlcjpmaWRkbGVy
Host: demo.openiz.org:8080
If-Modified-Since: Fri, 15 Jul 2016 10:09:39 GMT

HTTP/1.1 200 OK
Content-Length: 0
Server: Microsoft-HTTPAPI/2.0
X-GeneratedOn: 2016-06-22T19:21:30.6871719-04:00
Last-Modified: 2016-06-22T19:21:30.6871719-04:00
ETag: 094941e9-a3db-48b5-862c-bc289bd7f86c
```

7.4.2.5. Bundling Resources

All HDS data contract members are independent and do not include deeply nested data. Rather, each HDS object contains a reference to other objects via properties identified by UUID. When fetching and/or querying a record, callers can instruct the HDS to bundle these references into a resource bundle which contains the specified object.

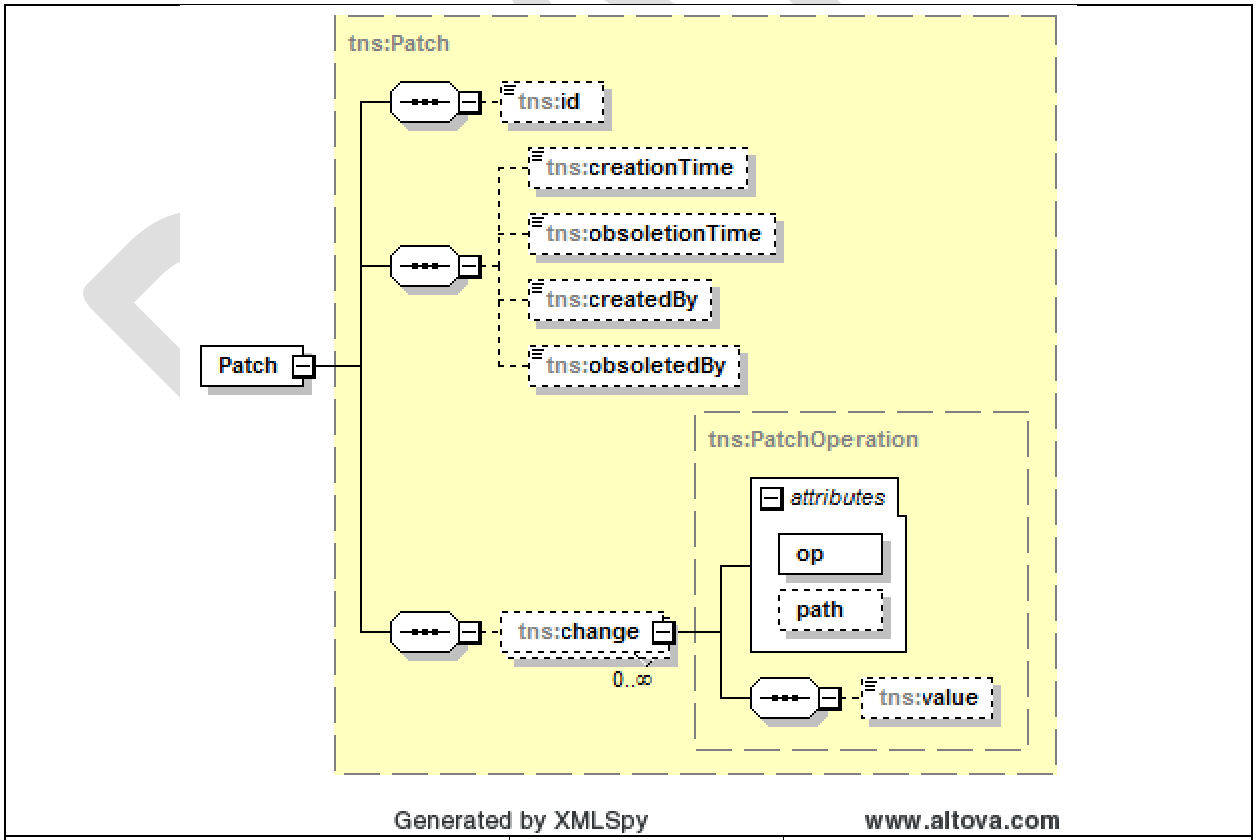
This pattern is used because it prevents deep nesting and maintains references to data elements even when JSON is used.

7.4.2.6. Patching Resource

The default behavior of the HDS is to replace an object, in its entirety with the value sent to it. For example, if a patient exists in the HDS, and a client sends an PUT on the patient resource, the HDS will perform the necessary actions required to ensure that the HDS patient in its data store matches the patient resource sent.

This means that the PUT operation is really a replacement (similar to uploading a new copy of a file). This behavior requires clients to send a complete copy of the resource with modifications to the HDS which can be troublesome when the client only wishes to update part of a resource.

In order to overcome this, SanteDB's HDS implements the HTTP PATCH method specified in IETF RFC5789. This method allows a client to send a delta of changes to be made to a particular object. The PATCH format is illustrated in , and closely follows the IETF RFC6902 specification.



Element	Type	Description

Id [1..1]	UUID	Uniquely identifies the patch operation and the patch.
creationTime [1..1]	DateTime	The time at which the patch was generated.
obsoletionTime [0..1]	DateTime	The time when the patch is no longer valid.
createdBy [1..1]	UUID	The user which was responsible for the generation of the patch.
obsoletedBy [0..1]	UUID	When populated, indicates the user which was responsible for the obsoletion of the patch.
change/ [1..*]		Indicates one or more changes which should be applied to the target object.
change/@op [1..1]	PatchOperationType	The type of patch operation to be performed. Add | Remove | Replace | Test
change/@path [1..1]	String	The target property which is to be patched.
change/value [1..1]	Object	The value to patch or value to test.

The operations which can be performed are identified in .

Operation Type	**Description**
Test	Indicates that the patch operation on the server should test that the value in @path equals value before continuing the patch operation. Failure of this assertion will result in a 409. This operation is included in order for the client to perform a sanity check before performing subsequent actions.
Add	Indicates that the patch operation on the server should add the value provided in <value> should be added to a collection.
Remove	Indicates that the patch operation on the server should remove an item from the collection whose identifier or other information matches the <value> element.
Replace	Indicates that the patch operation should replace the value currently held in @path with the value in <value>

The patch operation requires the passing of the If-Match HTTP header. The If-Match header indicates that the patch should be applied against the specified version of the resource. If If-Match does not match the current version of the object then a 409 Conflict response is passed back to the client. A client may override a 409 by issuing a PATCH with the X-Patch-Force header set to true, which instructs the HDS to ignore the conflict and attempt the patch operation anyway.

7.4.2.7. MIME Types / Message Encodings

The HDSI supports both XML and JSON encodings. The control over which encoding is to be used is performed via the Accept header in the HTTP message. The allowed content types are listed in .

Content-Type	Description
application/xml	An XML formatted message will be sent or is included in the body.
application/json	A JSON formatted message will be sent or is included in the body.
application/xml+openiz-patch	An XML patch
application/json+sdb-viewmodel	A JSON formatted message in the simplified view model format will be returned or is included in the body.

7.4.2.8. View Model Content

The HDSI interface, by default, responds back to callers in a method which is optimized to reduce bandwidth usage. This means that the HDSI will not duplicate data in messages, instead relying on identifier pointers for data. While this is useful for optimizing transmission to synchronization clients, it can prove to be quite inconvenient for clients which are seeking to display the contents of the message directly.

This is the purpose for the application/json+sdb-viewmodel mime type. This mime type indicates that the view model representation (according to the JavaScript API) is being used. The view model that is used is controlled by the ?_viewModel= parameter or the X-SanteDB-ViewModel header.

7.4.2.9. Compression

The HDSI interface supports two-way compression for all communications. This means that a client can not only request compressed data from the HDS but can submit compressed data to the HDS. This functionality is expected to be useful when mobile clients are disconnected for long periods of time and are required to submit large amounts of data.

Requesting the HDSI to perform compression on a response requires the use of the Accept-Encoding header. Submitting compressed data to the HDSI requires the Content-Encoding header.

The HDSI will accept and produce compressed data formats listed in . If accept-encoding is not understood the X-CompressResponseStream header will be set to "no-known-accept" and uncompressed data will be sent to the requestor. Additionally if a Content-Encoding is sent to the HDSI which it cannot process a 400 (Bad Request) response will be sent to the requestor.

Algorithm	Content-Encoding	Support (Version)
GZIP Compression	gzip	0.7 +
Deflate	deflate	0.7 +
BZip-2	bzip2	0.9.7 +
LZip / LZMA	lzma	0.9.7 +

7.4.2.10. *Query Parameters*

7.4.2.10.1 Element Filters

All query parameters in the HDSI interface are mapped directly to data contract objects. For example, if a resource contains an element named "createdBy" the filter of the same name will perform a filter of data by the createdBy.

It is also possible to chain query parameters using a dotted notation. For example, to query by the name of the user which created a resource one can filter "createdBy.userName". In this scenario any chained parameters match the data element name on type. Since createdBy is of type SecurityUser, elements in SecurityUser can be chained.

7.4.2.10.2 Control Parameters

HDSI queries also provide a series of response control parameters. All control parameters begin with an underscore.

Parameter	**Operations**	**Description**
_bundle	Get, Version Get	Instructs the HDSI to bundle resource references.
_expand	Search, History, Get, Version Get	Instructs the HDSI service to expand properties and include them in the result. Using this option will force the delay-load of properties in the HDS model.
_all	Get, Version Get	Instructs the HDSI to expand all properties and include them in the result. This option forces the delay load of all data in the HDS model and can have performance implications.
_since	History	Instructs the HDSI to only return versions in the version history which have been created since the specified version identifier.
_offset	Search	Instructs the HDSI interface to offset search results by the specified number of result.
_count	Search	Instructs the HDSI interface to include only the indicated number of results in the return bundle.
_orderBy	Search	Instructs the HDSI interface to order the result set by a series of order parameters. The :asc and :desc qualifiers are added to the query.

7.4.2.10.3 Modifiers

Modifiers are applied to operators of particular type and can be used by consumers to perform simple filter tasks. The list of modifiers are found in .

Modifier	**Operator**	**Example**
Equality	?a=b	?userName=SYSTEM
Less Than	?a=<b	?creationTime=<2016-01-01
Less Than or Equal To	?a=<=b	?value=<=6.3
Greater Than	?a=>b	?value=>2

Greater Than or Equal To	?a=>=b	?creationTime=>=2016-01-01
Approximately	?a=~b	?mnemonic=~FOO
Starts With	?a=^b	?mnemonic=^FE
Not	?a=!b	?userName=!smithj
Guard	?a[guard]=b	?name.component[Family].value=Smith
OR	?a[guard|guard]=b	?name.component[Prefix|Suffix].value=MR
Cast	?a@type=b	?participation[Location].player@Place.service.mnemonic=F
Function	?:(function|parameters)value	?identifier.value=:(substr|1,2)203-203-203

7.4.2.10.4 And / Or Semantics

Boolean logic semantics on the HDSI are limited to a simple scheme. If more than one parameter of the same name appears in the query string, the values are ORed to one another. Unique filter parameters are ANDed. For example:

```
?createdBy.userName=SYSTEM&mnemonic=~EVN
```

Equates to any record created by a user with username SYSTEM and having a mnemonic approximately matching EVN, whereas:

```
?createdBy.userName=SYSTEM&mnemonic=~EVN&mnemonic=~Event
```

Matches any record created by a user with username SYSTEM having mnemonic approximately matching EVN or mnemonic approximately matching Event.

7.4.2.10.5 Guard Conditions

There are several times when queries will require filtering on a chained parameter where the query executor wishes to guard or only filter a specific type of traversal. These guard conditions are used to select only traversals whose classifiers match the guard condition. For example, to filter on only Patients whose legal given name is John, the query would be as follows:

```
/Patient?name[Legal].component[Given].value=John
```

Here the query guards that only names with NameUse.Mnemonic (the classifier for EntityName) matching L are filtered and the component having ComponentType.Mnemonic (the classifier for EntityNameComponent) has a value of GIV. In the query filter builder for the HDSI the actual expression is as follows:

```
o => o.Names.Where(guard => (guard.UseConcept.Mnemonic == "Legal")).Any(name =>
name.Component.Where(guard => (guard.TypeConcept.Mnemonic ==
"Given")).Any(component => (component.Value == "John")))
```

Guard conditions are useful when filtering collections where classifier carries some semantic meaning. For example: name, addresses, participations, entity relationships, etc.

7.4.2.10.6 Casting

There are several instances where an element type in the HDS data model is bound to an abstract class or class which is not being requested. For example, a relationship may point to an Entity, however properties from SubstanceAdministration are desired. To issue a cast, a client uses the @ symbol after the element name.

For example, to find all substance administrations given to a patient born after 2017-01-01, the following query is used:

```
/Act?classConcept.mnemonic=SubstanceAdministration&participation[RecordTarget].player@Patient.dateOfBirth=>2017-01-01
```

The type name is the XML schema name of the type being cast to. The query processor will convert this to a TypeAs expression and it is equivalent to:

```
o => o.ClassConcept.Mnemonic == "SubstanceAdministration" && o.Participations.Where(guard => guard.ParticipationRole.Mnemonic == "RecordTarget").Any(a => (a.PlayerEntity as Patient)?.DateOfBirth > new DateTime(2017,01,01))
```

7.4.3. Report Integration Services Interface(RISI)

The RISI interface is primarily designed to abstract applications from the reporting technology used in an SanteDB deployment. The RISI exposes data related to reports, and facilitates the execution of reports in a secure manner.

7.4.3.1. Transport

The RISI interface is very simple and designed solely for the purpose of exposing data for reporting purposes.

The operations for the RISI are described in .

Resource	Operation	Description
/datamart	GET	Gets a list of all datamarts from the warehouse
	POST	Create a datamart
/datamart/{datamartId}/data	GET	Execute adhoc query
	POST	Create warehouse object
/datamart/{datamartId}/data/{objectId}	GET	Get warehouse object
/datamart/{datamartId}/query	GET	Get stored queries
	POST	Create a stored query
/datamart/{datamartId}/query/{queryId}	GET	Executes a stored query
/datamart/{id}	GET	Gets a specified datamart
	DELETE	Delete data mart
/format	GET	Gets the report formats. Returns a list of report formats.
	POST	Creates a report format. The report format to create.Returns the created report format.
/format/{id}	GET	Gets a report format by id. The id of the report format to retrieve.Returns a report format.
	DELETE	Deletes a report format. The id of the report format.Returns the report deleted report format.
	PUT	Updates a report format. The id of the report format to update.The updated report format.Returns the update report format.
/report	GET	Gets a list of report definitions based on a specific query. Returns a list of report definitions.

	POST	Creates a new report definition. The report definition to create.Returns the created report definition.
/report/{id}	GET	Gets a report definition by id. The id of the report definition to retrieve.Returns a report definition.
	DELETE	Deletes a report definition. The id of the report definition to delete.Returns the deleted report definition.
	PUT	Updates a report definition. The id of the report definition to update.The updated report definition.Returns the updated report definition.
/report/{id}/format/{format}	POST	Executes a report. The id of the report.The output format of the report.The report parameters.Returns the report in raw format.
/report/{id}/parm	GET	Gets a list of report parameters. The id of the report for which to retrieve parameters.Returns a list of parameters.
/report/{id}/parm/{parameterId}/values	GET	Gets a list of auto-complete parameters which are applicable for the specified parameter. The id of the report.The id of the parameter for which to retrieve detailed information.Returns an auto complete source definition of valid parameters values for a given parameter.
/report/{id}/source	GET	Gets the report source. The id of the report for which to retrieve the source.Returns the report source.
/type	GET	Gets a list of all report parameter types. Returns a list of report parameter types.
	POST	Creates a new parameter type. The parameter type to create.Returns the created parameter type.
/type/{id}	GET	Gets a report parameter by id. The id of the report parameter to retrieve.Returns a report parameter.
	DELETE	Deletes a report parameter type. The id of the report parameter type to delete.Returns the deleted report parameter type.
	PUT	Updates a parameter type definition. The id of the parameter type.The parameter type to update.Returns the updated parameter type definition.

7.4.3.2. Response Codes

See 7.4.2.2 for response codes.

7.4.3.3. Compression

The RISI interface supports compression of requests and responses. See 7.4.2.8 for more information.

7.4.4. HL7 FHIR

The SanteDB HDS supports the HL7 FHIR standard for the communication of HDS objects to/from the backend. The implementation of FHIR and restrictions on the SanteDB HDS are descried here in more detail.

7.4.4.1. Supported Resources

The HL7 FHIR interface exposes the standard conformance resource either via the /Conformance resource or via an OPTIONS HTTP request on the base url of the FHIR interface. The resources implemented by the HDS are listed here for completeness. The FHIR concept of "un-delete" is not currently planned to be supported for the SanteDB HDS.

Resource	Operations	Description / Restrictions
/Immunization	GET	The immunization resource exposes the underlying substance administrations which have a mood code of event occurrence. That is to say, the immunization resource represents immunizations have "actually" occurred (either actually administered or actually not administered)
/ImmunizationRecommendation	GET	Gets the immunization proposals from the forecaster along with the status of the forecast. This represents immunizations which were proposed and for which there are no fulfillments.
/Patient	GET, POST, PUT, DELETE	Gets or registers patient resources in the HDS system. This operation will result in complete business rules (including forecasting) occurring on post and put.
/Practictioner	GET	Gets practitioners in the SanteDB HDS. The get operation is supported to facilitate export from the SanteDB HDS into other systems. This maps to the Provider resource in the HDS data store.
/Organization	GET	Gets the organizations which are registered in the SanteDB HDS and maps to the Organization resource in the HDS data store.
/Location	GET	Gets the locations (villages, cities, regions, etc.) that are registered in the SanteDB HDS.
/AdverseEvent	GET	Gets the adverse events (AEFIs) recorded in the SanteDB HDS. By

		default this filters to all Acts having classConcept of Condition.
/AllergyIntolerance	GET	Gets the allergies and intolerances recorded in the SanteDB HDS including food, drug, and environmental allergies and their severity.
/Medication	GET	Gets a list of all medications from SanteDB HDS. This maps to all ManufacturedMaterials in the concept set “medication types”.
/Substance	GET	Gets a list of all substances from the SanteDB HDS. This maps to all Materials in the concept set “substance types”
/Observation	GET	Gets a list of all observations (coded, textual, or quantified) having a type concept in the concept set VitalSigns.
/MedicationAdministration	GET	Gets a list of all medication administrations that are not immunizations (for example, supplements)
/Condition	GET	Gets a list of all problems (coded observations having type concept in problem types concept set) recorded for a patient.

7.4.4.2. Meta Data

All FHIR resources exposed by the SanteDB HDS expose metadata in the HL7 <meta> field. This field contains the following data:

- The version UUID of the resource returned in the message
- The last updated time of the resource
- The profile http://openiz.org/fhir
- All security policies applied against the object
- All custom Tags attached to the object.

An example of this meta attribute is provided in .

```
<meta>
   <versionId value="9f7a3850-e3fc-4d3d-ba42-d6f0a909473b"/>
   <lastUpdated value="2017-07-09T12:52:07.8010+03:00"/>
   <profile value="http://openiz.org/fhir"/>
   <security>
      <system value="http://openiz.org/security/policy"/>
      <code value="1.3.6.1.4.1.33349.3.1.5.9.2.2.3"/>
   </security>
   <tag>
      <system value="http://openiz.org/tags/fhir/hasDuplicateFix"/>
      <code value="True"/>
   </tag>
</meta>
```

7.4.4.3. Extensions

All extensions attached to the SanteDB HDS object will be exposed via FHIR with the exception of the extension http://openiz.org/extensions/core/jpegPhoto which is rendered in the photo or image tag of the resource if available. Naturally, HDS extensions are mapped to their FHIR counterparts listed in .

SanteDB Extension Handler	FHIR Extension Value	Notes
BinaryExtensionHandler	base64Binary	The value of the binary extension is simply serialized verbatim to its FHIR counterpart.
BooleanExtensionHandler	boolean	The value is translated to a FHIR Boolean.
DateExtensionHandler	dateTime	The value is translated to a FHIR Date/Time object.
DecimalExtensionHandler	decimal	The value is translated to a FHIR decimal.
DictionaryExtensionHandler	base64Binary	The dictionary (JSON) value is serialized as base64binary.
StringExtensionHandler	string	The string is represented verbatim.

8. Data Architecture

8.1. Conceptual Data Model

This section seeks to describe and discuss the concepts found within the SanteDB data design. It does not describe the data entities, nor their fields and seeks to provide context to the logical data model which follows.

The SanteDB data model can be described as a series of conceptual domains. These domains are used to conceptualize how the entirety of the IZTW data model functions. The logical domains within the SanteDB data model can be described as:

- **Security:** Security tables are primarily used for the purpose of securing the SanteDB data and enforcing of policies. SanteDB's policy system is described in the solution architecture section of this document. Security tables also deal with users in roles.
- **Clinical:** These tables represent the primary way in which clinical data is stored. The SanteDB clinical data model is a derivation of the HL7® Reference Information Model (RIM). The HL7 RIM can be described as a series of base classes describing all events as "entities playing roles participating in acts".
- **Protocol / Workflow:** These tables represent metadata that is used by forecasting and DSS tools to enforce vaccination protocols. Patients are enrolled into a series of protocols or workflows that can be used to track adherence to best practices.
- **Concepts:** These tables are used to control the vocabulary used within the SanteDB data model. SanteDB uses a series of core concepts in the clinical tables and the concept tables permit the mapping of these internal SanteDB concepts to wire-level codes in HL7® FHIR™, or HL7® CDA™.

8.1.1. Clinical Domain

The clinical domain within SanteDB is loosely based on the HL7 reference information model (RIM). The SanteDB clinical domain does not represent a holistic implementation of the RIM, however borrows some of its concepts to represent and organize clinical data.

Figure 10 graphically illustrates how the elements within the clinical domain relate to one another.

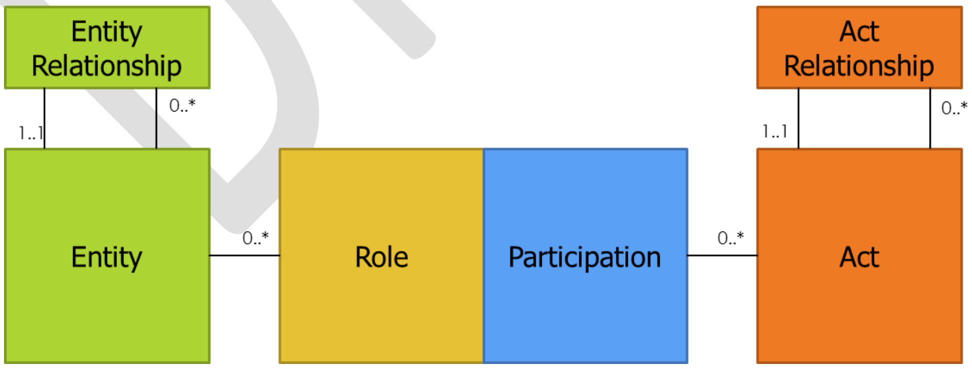


Figure 10 - Conceptual Clinical Model

- **Entities:** An entity represents a person, place, organization, or thing (syringe, antigen, vaccine, etc.). Entities can be related to one another via an entity relationship such as place belonging to an organization or series of materials belonging to a vaccination kit.

- **Roles:** Roles represent a type of part an entity plays. For example, a Person entity may play the role of a provider or a patient.
- **Participations:** Participations are the link between an act and an entity via a role. A participation is used to describe how an entity participates in the carrying out of an act.
- **Acts:** An act represents an action performed. An example of an Act may be an encounter the patient has with a provider to receive a weight or immunization. Acts may be related to one another, for example an encounter may fulfill an appointment, or an observation may be a part of an encounter.

In this context we can represent many different clinical scenarios. In order to make conceptualizing data elements in the SanteDB persistence store easier, much data documentation leverages an information model cards as illustrated in Figure 11.

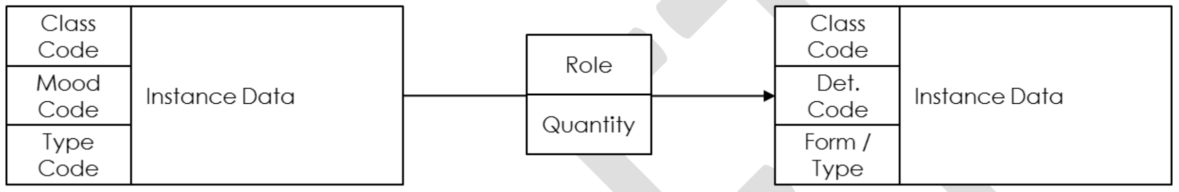


Figure 11 - Sample information model representing Participation

8.1.1.1. Entities

An entity within the SanteDB data model is used to represent a person, place, or thing. Entities represent the who, which and where aspects of an action. Entities are further classified into several subclasses illustrated in Figure 12.

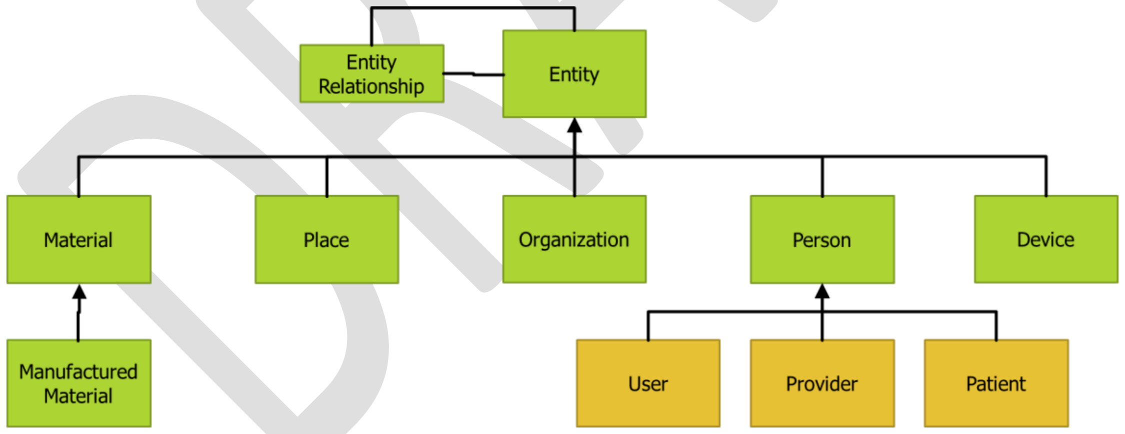


Figure 12 - Entity Classes

Entities are further classified by their class code and determiner code. The determiner code of an entity is responsible from differentiating a type of an entity (i.e. antigen, dose number, material type, etc.) and an instance or series of instances of an entity (an actual vial of vaccine, a box of syringes, etc.). Most entities within the SanteDB system are expected to be stored as instances of entities, classes of entities will primarily be restricted to materials whereby a class of antigens (OPV for example) will have both instances (vials of OPV) and sub-classes of representing dose numbers (OPV0 – OPV3). Provides a summary of the entity classes and how they are classified in the data model.

Entity Class	Class Code	Description
Entity	ENT	An entity is the base class used to represent a person/place/thing in the SanteDB data model.
Material	MAT	A material represents a physical thing to which participates in the delivery of care. For example: a syringe, a box of vaccine, etc.
Manufactured Material	MMAT	A manufactured material represents a material which is manufactured such as diluent, vaccine, syringes, etc.
Place	PLC	A place represents a physical location where health services are provided.
Organization	ORG	An organization represents an administrative structure which employs providers, operates clinics, etc.
Person	PSN	A person represents a human being.
Patient	PAT	Represents a person who receives health services.
Provider	PVD	Represents a person who provides health services.
User	USR	Represents a person who actively uses the system.
Device	DEV	Represents a physical object on which health services data is entered, stored, etc.
Application	APP	Represents a piece of software which is used to access, record or update medical data.

Determiner codes are used to classify whether a tuple in the entity table represents an actual thing or a classification of things. There are three types of determiner classifications for objects listed in .

Determiner	Description	Example
Instance	The entry in the database represents a one or more occurrences of an entity.	A patient, a facility, an organization, etc.
Kind	The entry represents a classification of entities.	A type of material, a type of organization, a type of device.
Quantified Kind	Represents a kind of object which is quantified such as X number of Y.	A dose of vaccine (5ml, etc.)

Figure 13 represents a sample information model of a kit of BCG vaccine. The model illustrates how a single tuples in the SanteDB model are related to one another to represent a kit. The material “BCG Vaccine” is comprised of one dose of “BCG Antigen”, one “Syringe”, and one dose of “BCG Diluent”. A box of “BCG Vaccine” represents 25 single doses of “BCG Vaccine” (meaning 25 syringes, 25 antigen and 25 diluent). Finally, from this we can order (via instantiation or inheriting a kind) a kit of GSK 5ml BCG vaccine. This derivation will have the same rules applied (in that this vaccine must have a syringe, antigen, and diluent.

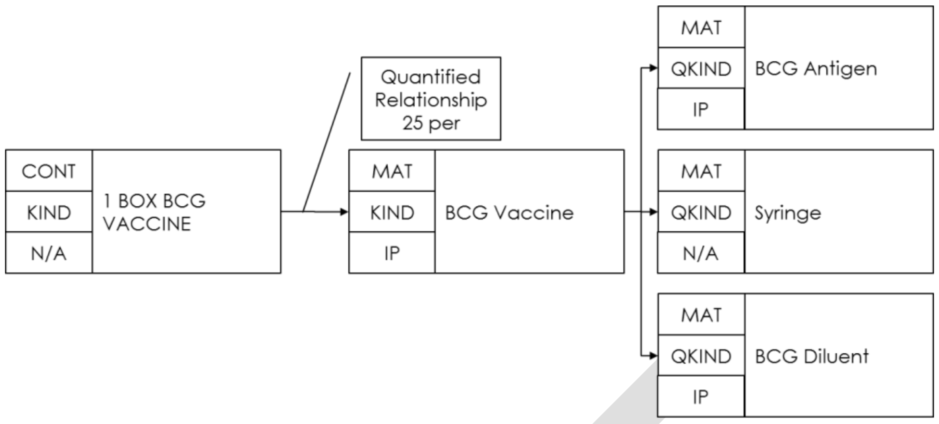
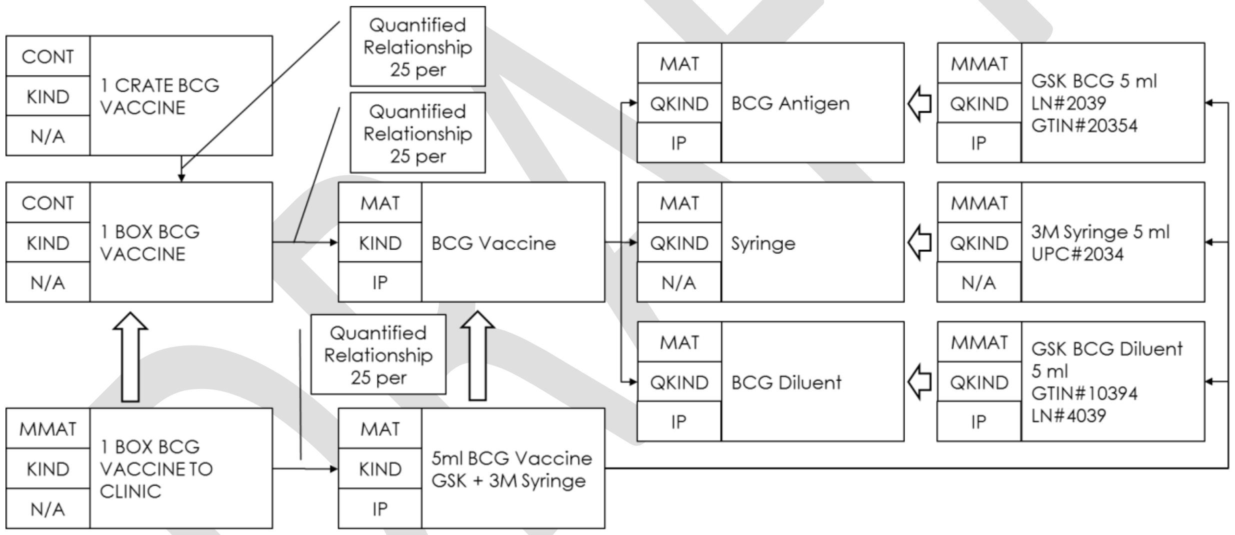


Figure 13 - Representing BCG as a combination of materials

Figure 14 represents a complex scenario of kitting a vaccine and then deriving a specific type of kit. The relationship between the instantiation and the generic concepts of the material would have a type of "Manufactured" meaning the "GSK 5 ml BCG" is a manufactured representation of the generic BCGvaccine.



CONT
KIND
N/A
1 CRATE BCG VACCINE
Quantified Relationship 25 per
Quantified Relationship 25 per
CONT
KIND
N/A
1 BOX BCG VACCINE
MAT
KIND
IP
BCG Vaccine
MAT
QKIND
IP
BCG Antigen
MAT
QKIND
N/A
Syringe
MAT
QKIND
IP
BCG Diluent
MMAT
QKIND
IP
GSK BCG 5 ml LN#2039 GTIN#20354
MMAT
QKIND
N/A
3M Syringe 5 ml UPC#2034
MMAT
QKIND
IP
GSK BCG Diluent 5 ml GTIN#10394 LN#4039
Quantified Relationship 25 per
MMAT
KIND
N/A
1 BOX BCG VACCINE TO CLINIC
MAT
KIND
IP
5ml BCG Vaccine GSK + 3M Syringe

Figure 14 - Instantiating BCG vaccine kit to Manufactured materials

The SanteDB model also supports more simplistic representation of a "BCG Vaccine" if kits are not required by the jurisdiction that is implementing SanteDB. Figure 15 illustrates a valid representation of BCG in a jurisdiction where only the antigen is tracked.

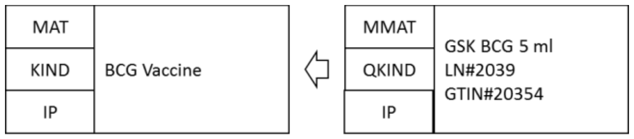


Figure 15 - Simple BCG to manufactured material

8.1.1.1.1 States

Entities in SanteDB also supports the attachment of an entity status.

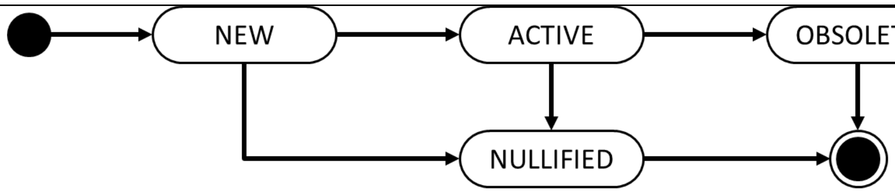


Figure 16 - Entity States

State	Description
NEW	The entity is a new registration or submission and no business processes or review has occurred on the entity.
ACTIVE	The entity is registered and actively available for use, search, query, etc.
OBSOLETE	The entity information is no longer valid and has since been amended, replaced, or is no longer available.
NULLIFIED	The entity was created in error and never existed.

8.1.1.2. Acts

Acts in the SanteDB data model represent actions taken by entities to other entities. In this lens, we can say that an act represents everything that "happens" to entities. This mechanism of storage allows the SanteDB data model to adapt to different jurisdictions with relative ease.

There are five different types of acts that are supported by the default SanteDB schema (Figure 17).

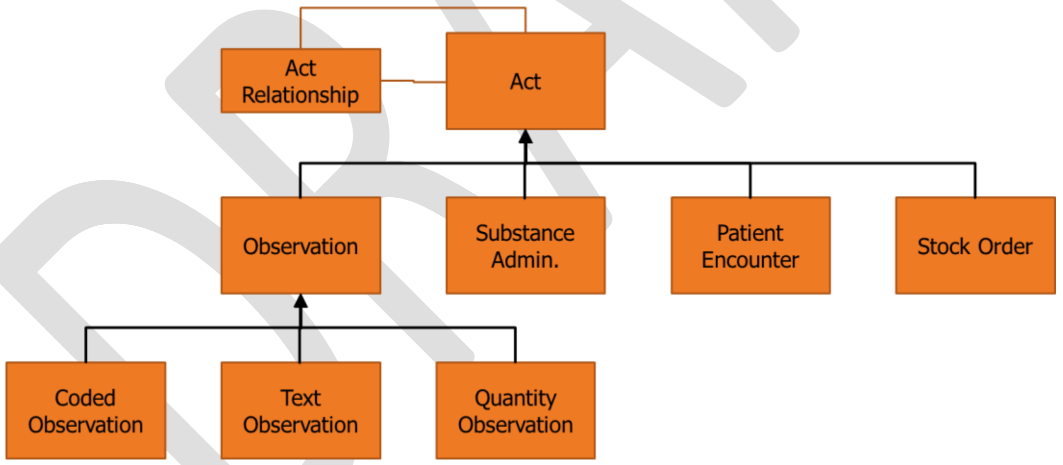


Figure 17 - Act classes

- **Patient Encounters:** Represent an act whereby a patient presents, or is intended to present for care.
- **Observations:** Represent an act whereby an entity observes something. Observations can include codified observations such as diagnoses, textual values such as a free-text description of an event, or a quantity such as weight or heartrate.
- **Substance Administrations:** A substance administration is representing an act whereby a substance, such as a vaccine is administered to a patient.
- **Transfer Order:** An order is a special type of stock act and represents a request by an entity to transfer stock from one place to another.

- **Procedures:** A procedure is a type of act in which the physical state of the record target or target entity is changed.

Acts are classified into one of these four types based upon their ClassConcept code. This dictates what type of Act the particular tuple in the database represents. Additionally, Acts are classified by Mood via the MoodConceptId. The mood of an act identifies the mood or method of operation of the act, moods include:

- **Proposal:** A proposal to perform the act, typically a forecaster or some automated DSS process will create acts with this mood code.
- **Intent:** Represents an intent to perform the act. These represent a human entering an intent to attend an encounter, or perform an observation.
- **Event:** Represents an act that *actually* occurred. This represents an action taken by a human.
- **Request:** Represents a request by one human to another human perform the action. Examples may be requesting a transfer order or requesting an encounter be done (i.e. referral).

Often time acts are related to one another via the ActRelationship entity. Act relationships are important, as typically acts cannot be standalone. For example, one cannot simply "Observe" something without having an encounter. An example of several types of interactions within SanteDB are described below:

- **Patient Presents to receive an Immunization:**
  In this use case the primary act is a PatientEncounter whereby participants include the clinician performing the vaccination (performer), the patient (record target), the clinic (service delivery location). The patient may be weighed prior to receiving the immunization that represents a component act of type Observation, and the administration of the vaccine represents a substance administration with relations to one or materials administered (Figure 18).

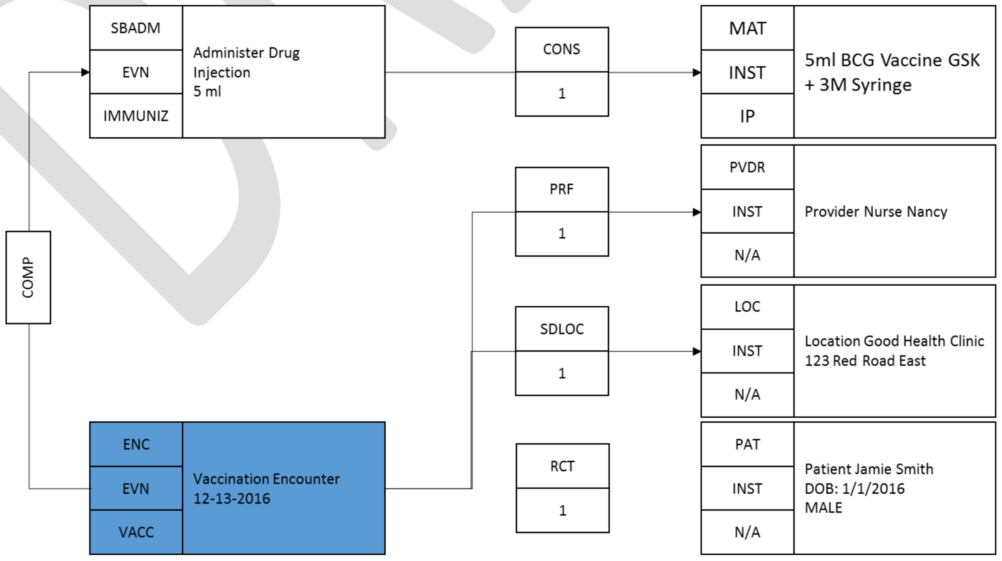

SBADM
EVN
IMMUNIZ
Administer Drug Injection 5 ml
CONS
1
MAT
INST
IP
5ml BCG Vaccine GSK + 3M Syringe
PRF
1
PVDR
INST
N/A
Provider Nurse Nancy
COMP
SDLOC
1
LOC
INST
N/A
Location Good Health Clinic 123 Red Road East
ENC
EVN
VACC
Vaccination Encounter 12-13-2016
RCT
1
PAT
INST
N/A
Patient Jamie Smith DOB: 1/1/2016 MALE

Figure 18 - Sample Vaccination Encounter model

- **Scheduling a second dose of an antigen:**
  In this use case the primary act is a PatientEncounter with mood of *Intent* (i.e. I intend to have an encounter) and date in the future. The PatientEncounter may have one or more substance administrations with mood of *Intent* that represent the vaccinations that are intended to be given at the appointment (Figure 19).

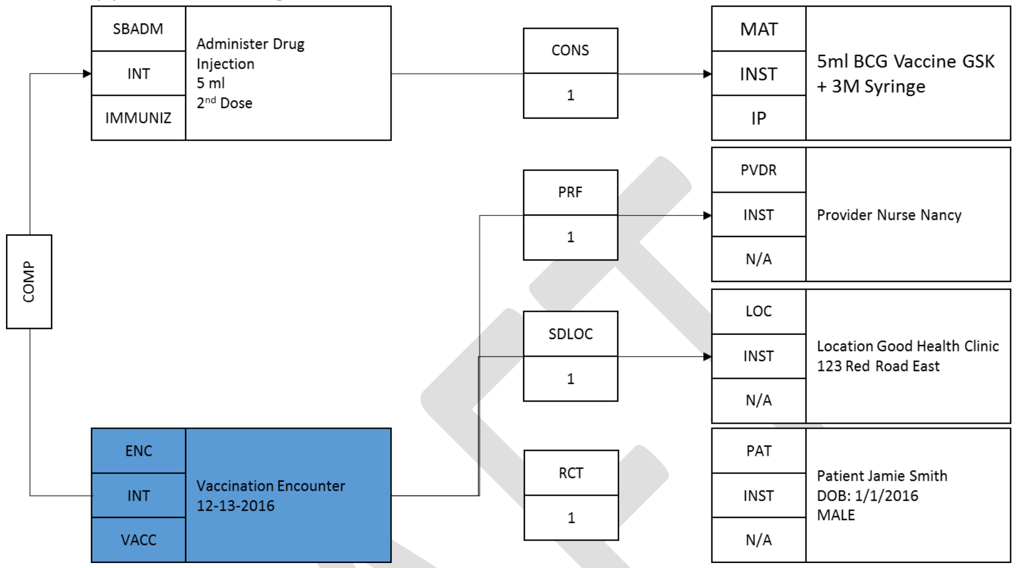

SBADM
INT
IMMUNIZ
Administer Drug Injection 5 ml 2nd Dose
CONS
1
MAT
INST
IP
5ml BCG Vaccine GSK + 3M Syringe
PRF
1
PVDR
INST
N/A
Provider Nurse Nancy
COMP
SDLOC
1
LOC
INST
N/A
Location Good Health Clinic 123 Red Road East
ENC
INT
VACC
Vaccination Encounter 12-13-2016
RCT
1
PAT
INST
N/A
Patient Jamie Smith DOB: 1/1/2016 MALE

Figure 19 - Vaccination Appointment model

- **Patient Presents for Appointment:**
  In this use case, the patient presents for the scheduled appointment. The encounter fulfills (FLFLS) the appointment request (Figure 20).

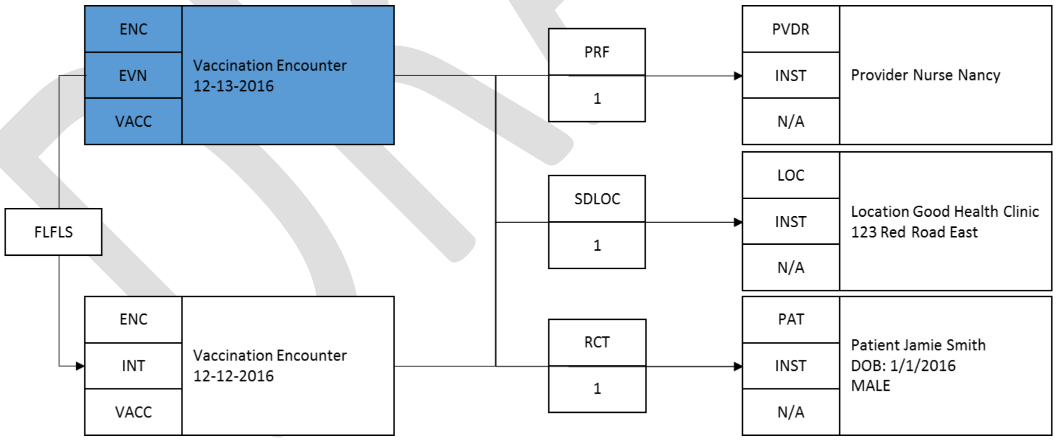

ENC
EVN
VACC
Vaccination Encounter 12-13-2016
PRF
1
PVDR
INST
N/A
Provider Nurse Nancy
FLFLS
SDLOC
1
LOC
INST
N/A
Location Good Health Clinic 123 Red Road East
ENC
INT
VACC
Vaccination Encounter 12-12-2016
RCT
1
PAT
INST
N/A
Patient Jamie Smith DOB: 1/1/2016 MALE

Figure 20 - Vaccination appointment fulfilling an appointment

- **Forecasted vaccination:**
  In this use case a decision support system identifies a patient which needs to receive a vaccine. The forecasting engine may create a PatientEncounter with mood of *Proposed* that indicates that a computer system is proposing an action to occur. When the patient presents for their vaccination the clinician creates a new PatientEncounter with mood of *Event* that fulfills the

PatientEncounter with mood of *Proposed* (i.e. the clinician is saying "I am acting on your proposal").

8.1.1.2.1 States

SanteDB Acts also carry a current state which dictate the status of the act. Act states are idenfieid in Figure 21.

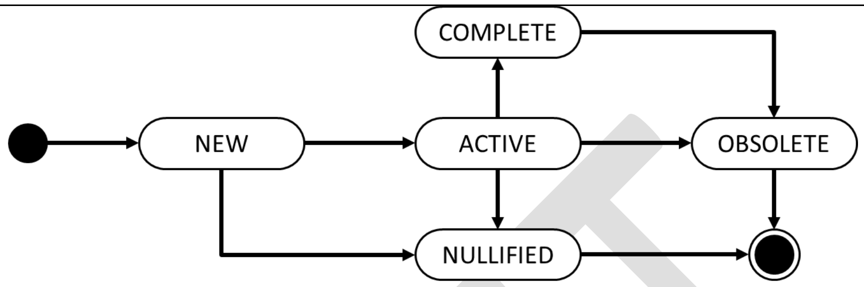


Figure 21 - Act States

State	Description
NEW	The act is a new submission and no formal review or rules have been applied to it.
ACTIVE	The act is currently occurring or is ongoing.
COMPLETE	The act has completed and no further action is required.
OBSOLETE	The act did occur, however the current information has since been amended, replaced, or otherwise is no longer accurate.
NULLIFIED	The act was created in error and never occurred.

8.1.1.3. Templates

SanteDB Acts and Entities can be templated to ease the creation and reuse of common structures. A template is a particular set of rules and pre-set data elements which are used to perform some use case with the underlying base type.

For example, weight and height are two separate types of a QuantityObservation. By creating a "weight" and "height" template, we allow any business processes or user interface elements to understand the rules for a particular entity. It allows software to understand "what kind" of element it is looking at without having to guess based on type/class/mood/determiner/etc.

8.2. Logical Data Design

This section seeks to describe and discuss the methodology behind the SanteDB data design and to describe the logical data design and schema. Note that the logical data design is often different from the physical schema as the physical schemata are adapted for the restrictions of the RDBMS in which they operate. The logical data design is also separate from the business data model design though the concepts are translatable.

8.2.1. Design Notes

This section outlines several design principles which were used in the creation of the SanteDB data model. Please note that any diagrams are informational, the source of truth are the detailed data dictionary tables.

8.2.1.1. Versioning

All data tables in SanteDB are versioned. This includes everything from display names of concepts, entities, acts, etc. This versioning allows an administrator to view a record as it was when it was created by a clinician.

This is important not only for a medical/legal reason, but for understanding why an action was taken and/or submitted. The versioning of major elements is performed using a few core concepts:

- **Object Table:** The core table contains the object's identifier and any immutable attributes of the object. For example, the Entity table contains the immutable "class code" attribute in the object table.
- **Object Version Table:** The object version table contains the object's versioned attributes. That is, attributes which may change over the lifetime of the object. Each version is assigned a unique identifier and a version sequence number. The version table also keeps track of version history via a replaces column so that versions are linked to one another.
- **Associative Tables:** All associative tables to a versioned object carry an EffectiveVersionSequenceId and ObsoleteVersionSequenceId which allow the SanteDB queries to establish whether a relationship was active at a particular time an action was taken, and also allows SanteDB to establish the user which was responsible for adding/removing the association.

Some tables are versioned in a different manner; these tables are typically tables where versioning is not as important for tracing purposes. These tables typically have a CreationTime/CreatedBy and ObsoletionTime/ObsoletedBy column and may have a pointer to a previous record that represents an edit.

The key message is that no UPDATE statements (with very few exceptions for Tags) are, and should never be, performed on the SanteDB database.

8.2.1.2. Tags and Extensions

Often times there are country specific requirements that cannot be stored in the default SanteDB data model. The data model has two methods of storing and reproducing such data.

Tags represent version independent, non-structural data associated with an Act or Entity. Tags are often used for tagging workflow values associated with an Act (such as editing approval process), security tags, and/or attachments. Tags are primarily used for storing metadata.

Extensions are structural attributes that modify the meaning of an Act or Entity. Extensions differ from tags in that they add meaning to an entity or act, and thus, are versioned. Extensions can be thought of as representing data specifically associated with the entity. Examples of extensions include 10-cell address of a patient, a stock consumption policy of a place. Extensions should be used as a last resort, when no other mechanism exists to store the data in the natural data model.

8.2.1.3. Constraint Notation

The columns in the logical data model follow a cardinality notation similar to that used by HL7 profiles. This notation can be described in the format:

[0|1..1|*] (= default value) (~ vocabulary binding) (? (check constraint))

For example, gender may be represented as :

Gender
[0..1] = NoInformation ~ AdministrativeGenderCode

A check constraint for obsoletion time may be represented as

ObsoletionTime
[0..1] ?(> CreationTime)

8.2.2. Storage Patterns

Starting with SanteDB version 1.2.x, the software persistence layer supports three primary modes of storing data to/from the primary data store.

- **Resource Storage:** This is the classic method of storage. In resource based storage, the data persistence layer will faithfully store the last copy of the resource presented to it. This is best suited for situations where SanteDB is being used as a single coherent system (like an EMR or EIR)
- **Merge Storage:** This method of storage is introduced in 1.2.0 of SanteDB's storage engine. When enabled, the merge storage service will merge changes provided via resource updates into the master copy of the resource in the data store. This means that multiple sources of updates are folded into a master record. This method of storage is useful for simple MPI and SHR solutions.
- **Master Data Management (MDM) Storage:** This method of storage is allows local copies of a resource to be maintained by individual suppliers of information. These suppliers have complete control over their copy of the record and the MDM module synthetically generates a master record from the local records on demand. This allows for powerful data management, merging, and unmerging of records.

8.2.2.1. Resource Storage

TODO

8.2.2.2. Merge Storage

By default, SanteDB supports the resource storage mechanism, however there are cases where SanteDB will act as a central data store rather than a simple resource data repository. To illustrate this, imaging a SanteDB instance running which receives requests using HL7v2 from two sources.

If this SanteDB instance received a PID segment for patient John Doe, ID #1234^123 and NID #2 from source 1, under resource storage, a Patient resource is inserted with this data. However, if the same SanteDB instance received a PID segment for John Doe, ID #4321^321 and NID #2, the SanteDB instance would ***replace*** the existing Patient resource with the new information.

This is expected behavior for a resource data store, however it is not expected behavior for a central shared repository. The merge storage interface rectifies this. Whenever an update is performed to an existing patient, the merge storage engine will determine the changes that need to be applied to the existing record and will append them (append only).

To enable merge storage, the MergeStorageDaemon needs to be enabled in the service host's configuration. This daemon is controlled using the configuration file as illustrated below:

```
<santedb.mss>
  <resources>
    <add resource="Patient" matchConfiguration="default" />
  </resources>
</santedb.mss>
```

Here matchConfiguration is used to determine which existing records in the datastore are to be updated based on incoming messages. Potential duplicate records are marked as duplicates using the Duplicate relationship type.

8.2.2.3. Master Data Management (MDM) Storage

One of the biggest improvements of the new SanteDB storage enhancements was the addition of the master data management (MDM) functionality. In this storage scheme individual records are inserted to the primary data store. A post-execution trigger is then fired to determine whether a record is a duplicate of an existing master record, or a new record entirely.

If a record is deemed a new record, a new Entity or Act is created with classification MDM-Master, the local copy of the record is then linked to the master entity or act with a relationship type of MASTER. Any other master records which are potential duplicates are related with relationship DUPLICATE.

Figure 22 illustrates an example of these relationships for a definitive match. Here there are two local copies of the patient John Doe, both point at the synthetic master record, whose content is determined based on the security principal querying the record at runtime.

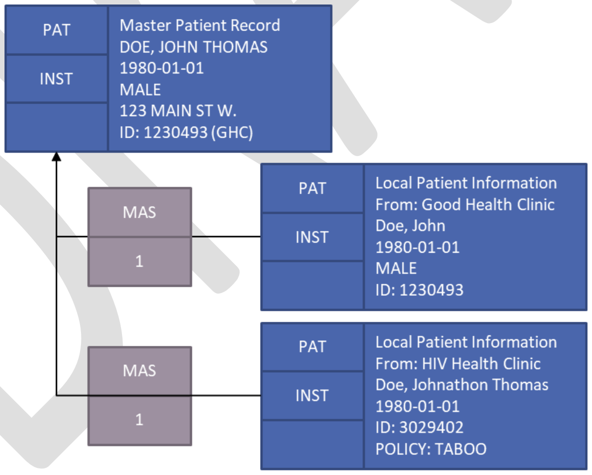


Figure 22 - MDM Relationships

MDM storage can be configured on the HDSI server instance only, and simply requires the registration of the MdmDaemonService in the host's service list. If you are running multiple copies of the HDSI server it is strongly recommended that this service daemon be enabled on all application servers.

The daemon service will process the configuration for resources which are to be placed under MDM storage mechanisms. Below, an example of an MDM configuration for Patients, note that the matchConfiguration attribute is used to determine the configuration name for the

IRecordMatchingService instance, this determines how the MDM service classifies master records opposed to duplicates.

```
<santedb.mdm>
  <resources>
    <add resource="Patient" autoMerge="true" matchConfiguration="default" />
  </resources>
</santedb.mdm>
```

8.2.3. Entity Relationships

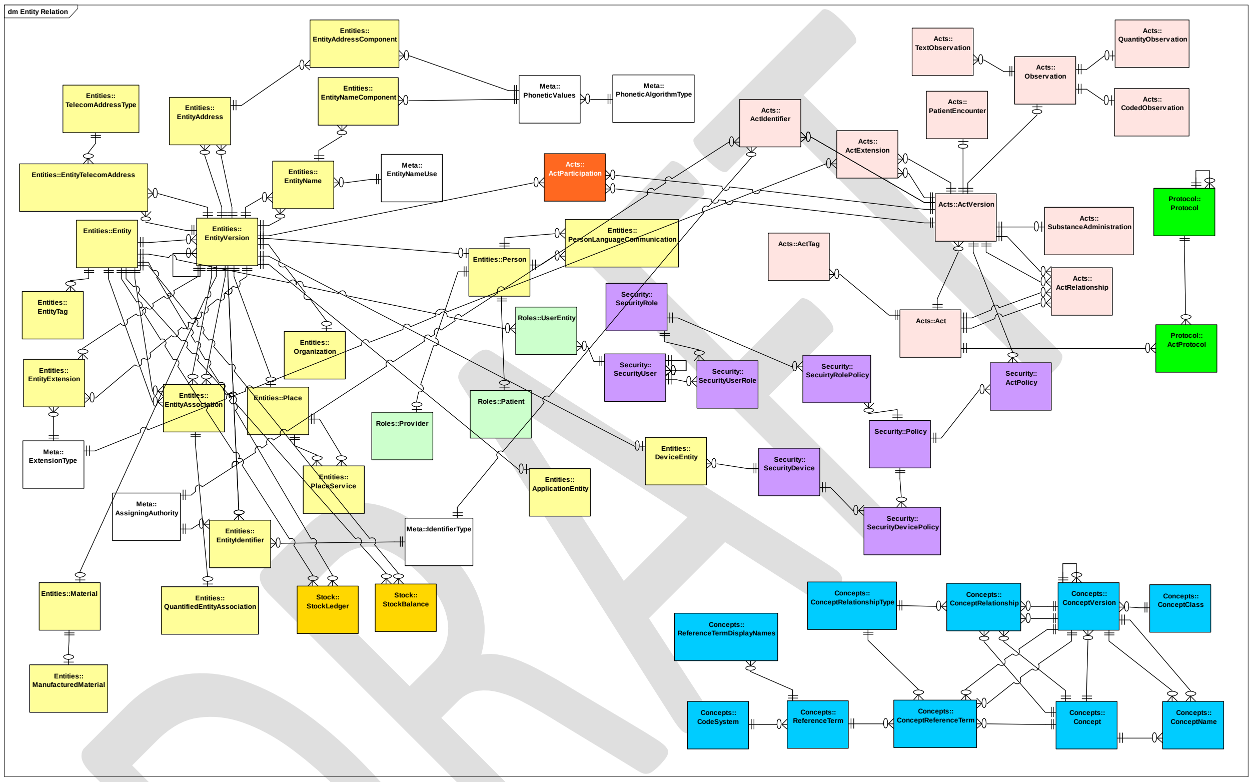


Figure 23 - SanteDB Entity Relations

Table 32 - SanteDB Entity Dictionary

Entity	Classification	Purpose
Entity	Clinical:Entity	The entity table provides a master list of all entity data within the data model. This table is used to store attributes related to the abstract superclass "Entity".
EntityVersion	Clinical:Entity	The entity version table provides data related to the versioning of entity data within the SanteDB system. All key entity attributes (minus classification) are stored in the entity version table.
EntityTelecommunicationsAddress	Clinical:Entity	The entity telecommunications address table provides a 1..n relationship through which telecommunications addresses

		(email addresses, phone numbers, fax numbers, etc.) can be stored and related to a particular entity version.
TelecommunicationsAddressType	Metadata	The telecommunications address type table is used to store data related to the type of telecommunications addresses used. This can be configured so that handlers may “contact” these addresses.
EntityAddress	Clinical:Entity	The entity address table is used to store addresses that are related to a particular entity. These addresses may include physical addresses, mailing addresses such a PO Boxes, etc.
EntityAddressComponent	Clinical:Entity	The entity address component table is used to store the component pieces of an entity address. These components may include the street address, city, village, zip code, geo-locator, etc. All address components are phonetically searchable.
EntityName	Clinical:Entity	The entity name table is used to store 1..n names related to an entity. An entity name may be a place name such as “Good Health Hospital”, a person name such as “Legal”, “Maiden”, etc.
EntityNameComponent	Clinical:Entity	The entity name component table is used to store the components of an entity name. These may include the given, family, suffix or prefix portions of a name.
EntityNameUse	Metadata	The entity name use table is used to store a master list of allowed uses of an entity name. This table is also responsible for ensuring that an appropriate name use is used in conjunction with the associated entity. For example, a “Person” entity name may not use an “Organizational” name use.
EntityIdentifier	Clinical:Entity	The entity identifier table is used to store a list of 1..n identifiers associated with an entity. An entity identifier can be thought of in many ways, for example: - An MRN for a Patient - A GTIN or UPC for a Material - An OU name for an Organization - A license number for a Provider
EntityIdentifierType	Metadata	The entity identifier type table is used to identify how an entity identifier is to be used and selected. An example, a GTIN

		entity identifier type may not be used with a Person entity as a Person does not carry a GTIN.
AssigningAuthority	Metadata	An assigning authority an a globally unique domain identifier which qualifies which system, organization or authority has the ability to assign new identifiers within a domain. An example of an assigning authority may be a system which generates MRNs, a jurisdictional EMPI, a global trade organization, etc.
EntityExtension	Metadata	An entity extension represents a non-classified, extended field for the entity. Extension fields are used primarily to store data related to an entity that does not belong in the standard SanteDB datastore. For example, there may be a jurisdictional requirement to store the Race or EthnicGroup of a Person.
EntityExtensionType	Metadata	An entity extension type qualifies the type of extension represented in the EntityExtension table. The type identifies an extension handler responsible for de-persisting the data.
EntityTag	Metadata	An entity tag represents a simple Key/Value pair data that is version-independent for a particular entity, that is to say that the updating of the tag does not produce a new version of the entity. Tags can be used for things like workflow tagging, quarantine control, or any other application specific use.
Person	Clinical:Entity	The person table is used to store data related to the Person subclass of the Entity superclass.
PersonLanguageCommunication	Clinical:Role	Identifies the languages of communication in which the person can be contacted.
Patient	Clinical:Role	The Patient table is used to store data related to the Patient subclass of the Person superclass. The patient table adds fields for date of birth, multiple birth order, deceased time, etc.
Provider	Clinical:Role	The provider role table is used to store data related to the provider subclass of the Person superclass.

Organization	Clincial:Entity	An Organization is a table used to store data related to the Organization subclass of the Entity superclass. An organization represents a legal entity, which is not a person, who has a mandate to deliver healthcare.
Place	Clinical:Entity	A Place is a table used to store data related to a physical place where care is delivered.
PlaceService	Metadata	The PlaceService table is used to store data related to the services available at a particular place. The services in the scope of SanteDB may represent immunization services, weight services, emergency services, etc.
Material	Clinical:Entity	The Material table is used to store data related to the Material subclass of the Entity superclass. A material represents a physical thing or supply that is used in the delivery of care. This can include syringes, diluents, vaccines, kits, gloves, etc.
ManufacturedMaterial	Clinical:Entity	A manufactured material represents a material that can be ordered from a manufacturer. This includes syringes, diluents, gloves, etc. This is a specialization for a Material in that a material can technically be a self-assembled group of materials which may not necessarily be orderable by a manufacturer (for example: a vaccine kit which is an administrative material)
EntityRelationship	Clinical:Entity	The entity association table is used to associate entities to one another. This can be used in a variety of ways including - Linking places to an organization - Linking materials together in a kit - Linking persons together, for example a mother/father of a patient - Linking places together in an administrative hierarchy. - Linking providers with patients.
ActParticipation	Clinical:Participation	The ActParticipation associative entity class is used to associate a particular version of an act with a version of an entity participating in the act. Example of

		participations include : RecordTarget, Author, Performer, Subject, etc.
Act	Clinical:Acts	The Act table is used to store the abstract data related to a particular act.
ActVersion	Clinical:Acts	The ActVersion table is used to store abstract data related to a particular point-in-time version of an Act or subclass of an act.
ActRelationship	Clinical:Acts	The ActRelationship table is used to link two acts together in some way. Examples of an ActRelationship include fulfillment (i.e. an encounter fulfills an appointment), componentization (i.e. encounter has an observation), etc.
SubstanceAdministration	Clinical:Acts	The SubstanceAdministration table is used to store the administration of substances (like vaccines) to the patient.
Observation	Clinical:Acts	The Observation table is used to store additional data related to an observation that occurs during the course of an encounter.
QuantityObservation	Clinical:Acts	The QuantityObservation table is used to store additional data related to an observation whose value is a quantity. For example, an observation of weight, blood pressure, height, etc.
CodedObservation	Clinical:Acts	The CodedObservation table is used to store additional data related to an observation whose value is a code. For example, observing the patient has blue eyes.
TextObservation	Clinical:Acts	The TextObservation table is used to store additional data related to observations which have textual content such as observation
PatientEncounter	Clinical:Acts	The PatientEncounter table is used to store additional data related to an encounter that the patient has with the health system. For example, an encounter may represent a single fulfillment of 6 week vaccinations.
ActTag	Metadata	The ActTag table represents a series of version independent metadata tags applied to the act. These can be workflow, security or categorization tags.
ActExtension	Metadata	The ActExtension table represents a series of extended data elements that can

		assigned to an act's version. This can be to store additional data related to the act itself not representable in the core data framework.
StockLedger	StockManagement	The stock ledger is used to store stock transactions (like a bank account) performed against a Place's stock level. A stock ledger has a 1..1 mapping to a deduction, deposit or transfer of materials between places.
StockBalance	StockManagement	The stock balance table is used to store a running total or balance of a material at a particular place.
StockOrder	StockManagement	A stock order represents a special type of Act (a non-clinical act) which is a request to order stock to/from a place. Mood codes for this include: - Propose – When a planning function proposes an order - Intend – When an order should be placed - Event – When an order has been placed
QuantifiedActParticipation	Clinical:Act	A quantified act participation is a specialization of an act participation that has the ability to convey the number of entities participating in the act. In particular, quantified act participations represent entities which are classes instead of instances of entities.
QuantifiedEntityRelationship	Clinical:Entity	Represents a specialization of an entity association whereby the container entity has a specified number of Source in Target entity.
TemplateDefinition	Clinical:Metadata	Represents a definition of a template which is used to track which template a particular entity or act implements.

8.2.4. SanteDB Concept Model

One of the central design principles in Open Immunize is that of customizable code and valuesets to be used within the SanteDB system. The concept subset of tables within the datamodel are used to store the master dictionary of concepts used within the SanteDB system.

There are four major facets of the concept system in SanteDB:

- **Concept Classes:** A concept class is a classification of what a concept represents. Concept classes are used for validation and organizing the concept dictionary. Examples of a concept class could be UnitOfMeasure for concept which are units of measure.
- **Concepts:** A concept is a central idea of an object or attribute of an object within the SanteDB database. A concept is, technically speaking, an abstract object which describes a real world thing. For example, the concept of an Arm, or the concept of OPV.
- **Reference Terms:** A reference term is a wire level representation of a concept. A reference term represents a manner in which a concept can be represented to another system. For example, the concept of OPV in the HL7 CVX reference term set is ‘01’.
- **Concept Sets:** A concept set represents a collection of concept that may be used for a particular purpose. Concept sets are used to restrict the complete set of concepts within the SanteDB database to those applicable in a particular scope. For example, an entity classification concept must be selected from the EntityClass concept set.

Figure 24 illustrates the relationships between the concepts tables found within the SanteDB database.

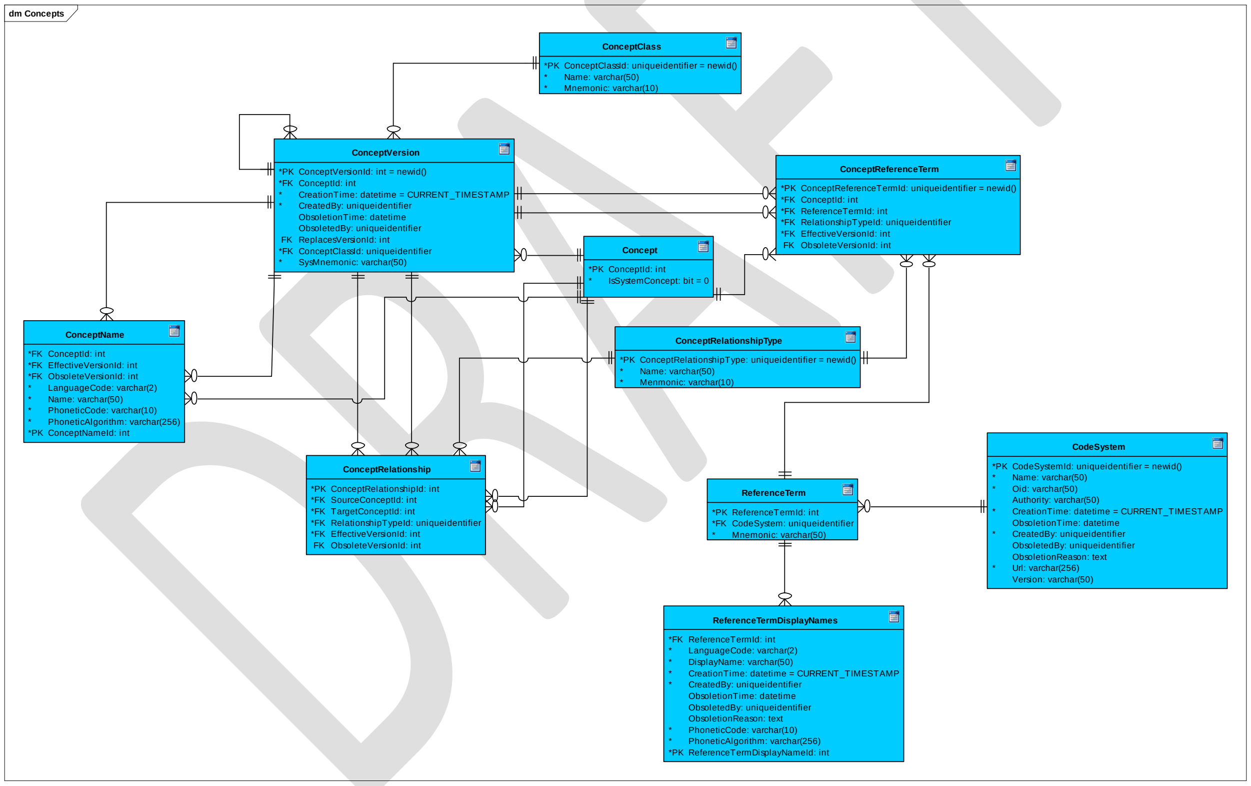

dm Concepts
ConceptClass
*PK ConceptClassId: uniqueidentifier = newid()
* Name: varchar(50)
* Mnemonic: varchar(10)
ConceptVersion
*PK ConceptVersionId: int = newid()
*FK ConceptId: int
* CreationTime: datetime = CURRENT_TIMESTAMP
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier
FK ReplacesVersionId: int
*FK ConceptClassId: uniqueidentifier
* SysMnemonic: varchar(50)
ConceptReferenceTerm
*PK ConceptReferenceTermId: uniqueidentifier = newid()
*FK ConceptId: int
*FK ReferenceTermId: int
*FK RelationshipTypeId: uniqueidentifier
*FK EffectiveVersionId: int
FK ObsoleteVersionId: int
Concept
*PK ConceptId: int
* IsSystemConcept: bit = 0
ConceptName
*FK ConceptId: int
*FK EffectiveVersionId: int
*FK ObsoleteVersionId: int
* LanguageCode: varchar(2)
* Name: varchar(50)
* PhoneticCode: varchar(10)
* PhoneticAlgorithm: varchar(256)
*PK ConceptNameId: int
ConceptRelationshipType
*PK ConceptRelationshipType: uniqueidentifier = newid()
* Name: varchar(50)
* Menmonic: varchar(10)
ConceptRelationship
*PK ConceptRelationshipId: int
*FK SourceConceptId: int
*FK TargetConceptId: int
*FK RelationshipTypeId: uniqueidentifier
*FK EffectiveVersionId: int
FK ObsoleteVersionId: int
ReferenceTerm
*PK ReferenceTermId: int
*FK CodeSystem: uniqueidentifier
* Mnemonic: varchar(50)
CodeSystem
*PK CodeSystemId: uniqueidentifier = newid()
* Name: varchar(50)
* Oid: varchar(50)
Authority: varchar(50)
* CreationTime: datetime = CURRENT_TIMESTAMP
ObsoletionTime: datetime
* CreatedBy: uniqueidentifier
ObsoletedBy: uniqueidentifier
ObsoletionReason: text
* Url: varchar(256)
Version: varchar(50)
ReferenceTermDisplayNames
*FK ReferenceTermId: int
* LanguageCode: varchar(2)
* DisplayName: varchar(50)
* CreationTime: datetime = CURRENT_TIMESTAMP
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier
ObsoletionReason: text
* PhoneticCode: varchar(10)
* PhoneticAlgorithm: varchar(256)
*PK ReferenceTermDisplayNameId: int

Figure 24 - SanteDB Concept Table Relations

By default, several concepts are included in the default installation of SanteDB including language codes, vaccine codes, entity and act classifications, status codes, etc. Additional codes may be added by users and/or may be customized to their environment.

Table 33 - SanteDB Concept Data Dictionary

Table	Column	Type	Description

ConceptClass	(None)	N/A	The concept class table stores a complete list of concept classifications.
	ConceptClassId [1..1]	UUID	Represents a unique identifier for the concept classification.
	Name [1..1]	VARCHAR	Represents a human readable name for the concept classification. Example: Class Codes
	Mnemonic [1..1]	VARCHAR	Represents a system mnemonic for the concept class. The mnemonic does not change even if the human readable Name column does.
Concept	(None)	N/A	The Concept table stores the key data related to a concept. The Concept table represents immutable concept properties that cannot be changed once a concept is created.
	ConceptId [1..1]	UUID	A unique identifier for the concept.
	IsSystemConcept [1..1] = false	BIT	An indicator which identifies whether the concept is a system concept (i.e. no further versions can be created, cannot be obsoleted, etc.).
ConceptVersion	(None)	N/A	The concept version table is used to store mutable properties of a concept. All edits to a concept’s attributes result in a new version being created in the ConceptVersion table.
	ConceptVersionId [1..1]	UUID	A unique identifier for the concept version.
	VersionSequenceId [1..1]	INT	A sequence identifier for the version which

			allows for establishing a time-independent record of version order.
	ConceptId [1..1]	UUID	The concept to which the version applies.
	CreationTime [1..1]	DATETIME	The instant in time when the concept version became active (was created). Should default to the current database timestamp.
	CreatedBy [1..1]	UUID	The user who was responsible for the creation of the version, or the system user if the installation process created the concept version.
	ObsoletionTime [0..1] ? (> CreationTime)	DATETIME	When present, identifies the time when the concept version did become obsolete. This is used whenever a new version is created, the old version is obsoleted.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Indicates the user who was responsible for the obsoletion of the record.
	ReplacesVersionId [0..1]	UUID	Identifies the concept version that the current version of the concept replaces.
	ConceptClassId [1..1] = Other	UUID	Identifies the classification of the concept as of the version tuple.
	Mnemonic [0..1]	VARCHAR	A unique mnemonic used by the system to lookup the concept. The system mnemonic is primarily used for validation purposes where a concept's identifier does not

			represent a consistent identifier across deployments.
ConceptName	(None)	N/A	The concept name table represents a series of human readable names for the concept at a particular version. This facilitates searches as well as translation.
	ConceptNameId [1..1]	UUID	A unique identifier for the concept name
	ConceptId [1..1]	UUID	The concept to which the concept name applies.
	EffectiveVersionSequenceId [1..1]	UUID	The version of the concept when the name did become active.
	ObsoleteVersionSequenceId [0..1]	UUID	The version of the concept when the concept name was obsoleted.
	Name [1..1]	VARCHAR	The human readable display name for the concept.
	LanguageCode [1..1] ~ ISO639-2 = en	CHAR	The ISO639-2 language code for the concept display name.
	PhoneticCode [0..1]	VARCHAR	The phonetic code for the display name. This is used for phonetic “sounds-like” searches of concepts.
	PhoneticAlgorithmId [0..1] ?(PhoneticCode)	UUID	The phonetic algorithm used to generate the phonetic code. This allows deployments to use METAPHONE, SOUNDEX or custom phonetic algorithms appropriate for the language used.
ConceptRelationship	(None)	N/A	The concept relationship table is used to link concepts to

			one another. Concept relationships can represent equivalency between concepts, parent/child relationships, etc.
	ConceptRelationshipId [1..1]	UUID	The unique identifier for the relationship.
	SourceConceptId [1..1]	UUID	The concept that represents the source of the relationship.
	TargetConceptId [1..1]	UUID	The concept which represents the target of the relationship
	EffectiveVerisonId [1..1]	UUID	Identifies the version of the source concept where the relationship did become active.
	ObsoleteVersionSequenceId [0..1]	UUID	Identifies the version of the source concept where the relationship is no longer active.
	RelationshipTypeId [1..1]	UUID	Identifies the type of relationship the concepts have.
ConceptRelationshipType	(None)	N/A	The concept relationship type represents allowed types of relationships that a concept can have.
	ConceptRelationshipTypeId [1..1]	UUID	The unique identifier for the concept relationship
	Name [1..1]	VARCHAR	The human readable name of the concept relationship type.
	Mnemonic [1..1]	VARCHAR	An invariant value that represents the type of relationship typically used by software components.
ReferenceTerm	(None)	N/A	A reference term represents a wire level code that can be used

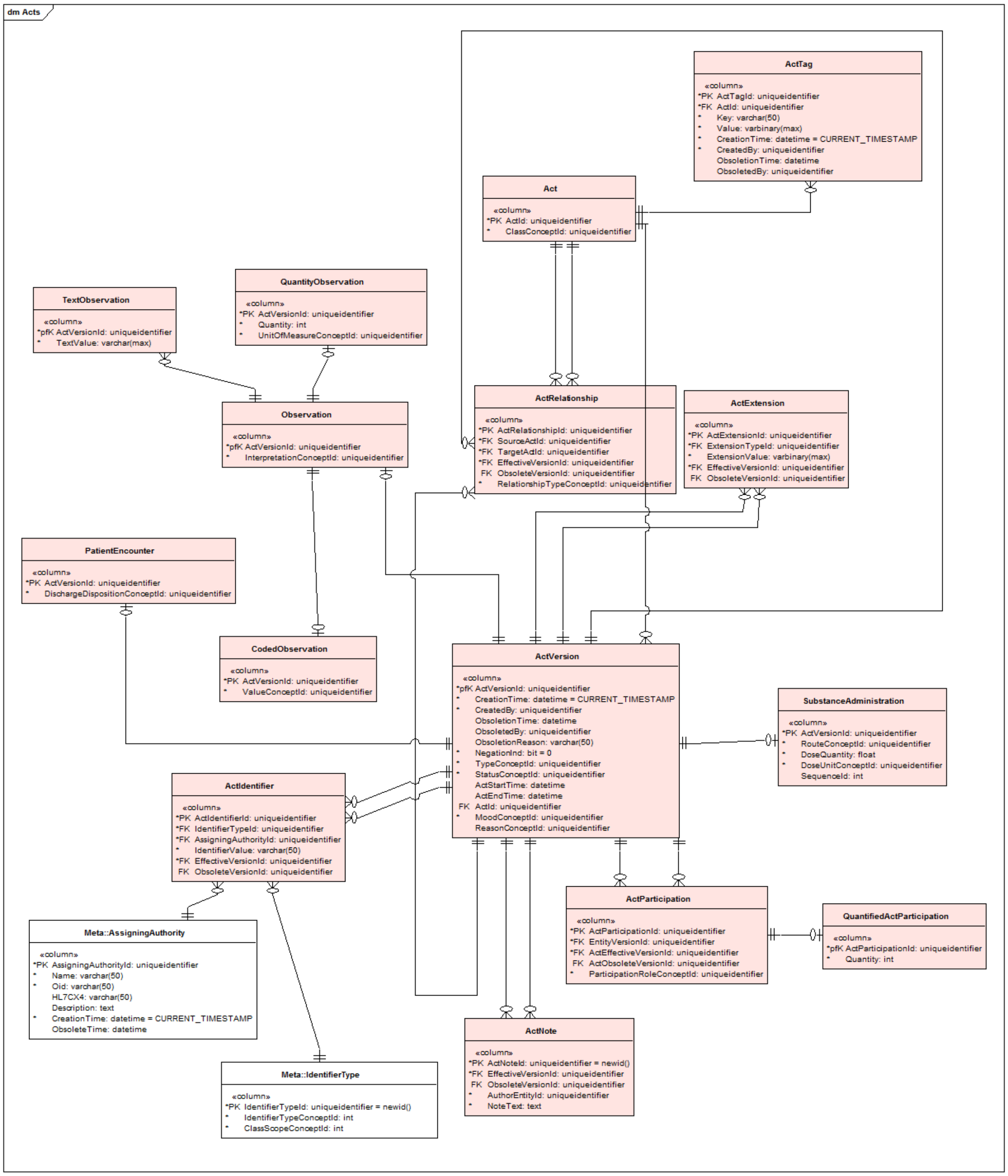
			to represent the concept.
	ReferenceTermId [1..1]	UUID	A unique identifier for the reference term.
	CodeSystemId [1..1]	UUID	The code system in which the reference term belongs.
	Mnemonic [1..1]	VARCHAR	The wire level code mnemonic for the reference term.
ConceptReferenceTerm	(None)	N/A	An associative entity that links a concept to one or more reference terms and indicates the strength of the map.
	ConceptReferenceTermId [1..1]	UUID	A unique identifier for the concept reference term map
	ConceptId [1..1]	UUID	The concept to which the reference term is linked.
	EffectiveVersionSequenceId [1..1]	UUID	The version of the concept where the reference term map became effective.
	ObsoleteVersionSequenceId [0..1]	UUID	The version of the concept where the reference term map became obsolete.
	ReferenceTermId [1..1]	UUID	The reference term which is associated with the concept.
	RelationshipTypeId [1..1]	UUID	Identifies the relationship (or strength) that the reference term has with the concept. For example: SAME_AS, NARROWER_THAN, etc.
CodeSystem	(None)	N/A	The code system table represents a master list of all code systems from which a reference term can be drawn.
	CodeSystemId [1..1]	UUID	A unique identifier for the code system entry.

	Name [1..1]	VARCHAR	A human readable name for the code system. For example: ICD10
	Oid [1..1]	VARCHAR	The object identifier that identifies the code system in an interoperable manner.
	Authority [1..1]	VARCHAR	The unique assigning authority of the particular code system. Example CVX or SNOMEDCT
	CreationTime [1..1]	DATETIME	The time when the code system entry was created. Default to the current timestamp in the RDBMS.
	CreatedBy [1..1]	UUID	The user that was responsible for the creation of the code system.
	ObsoletionTime [0..1] ? (>CreationTime)	DATETIME	When populated, indicates the time when the code system record is obsolete.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user who was responsible for obsoleting the record.
	ObsoletionReason [0..1] ?(ObsoletionReason)	VARCHAR	The textual description as to why the record was obsoleted.
	Url [1..1]	VARCHAR	A URI that uniquely identifies the code system. This is primarily used when exposing the code system over REST interfaces.
	Version [0..1]	VARCHAR	A textual description of the version of the code system that this record represents.
ReferenceTermDisplayNames	(None)	N/A	Like the ConceptName table, the reference term display name table is used to identify human readable display

			names associated with the reference term.
	ReferenceTermDisplayNameId [1..1]	UUID	The unique identifier for the reference term.
	ReferenceTermId [1..1]	UUID	The reference term to which the display name applies.
	LanguageCode [1..1] ~ ISO639-2 = en	CHAR	The ISO639-2 language code that identifies the language in which the display name is represented.
	DisplayName [0..1]	VARCHAR	The human readable name for the reference term.
	CreationTime [1..1]	DATETIME	The time when the display name became active.
	CreatedBy [1..1]	UUID	Identifies the user that was responsible for the creation of the reference term display name.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When present, identifies the time when the record should no longer be used.
	ObsoletedBy [0..1] ?(CreationTime)	UUID	Identifies the user that was responsible for obsoleting the display name.
	ObsoletionReason [0..1] ?(CreationTime)	VARCHAR	A textual description as to why the display name was obsoleted.
	PhoneticCode [0..1]	VARCHAR	Represents a phonetic code that can be used in "sounds-like" queries.
	PhoneticAlgorithmId [0..1] ?(PhoneticCode)	UUID	Identifies the phonetic algorithm that was used to generate the phonetic code. This allows METAPHONE or SOUNDEX or some other custom language appropriate phonetic algorithm to be used.

8.2.5. SanteDB Act Model

The act data model describes the tables and fields required for the tracking of acts in the SanteDB logical model.



dm Acts
ActTag
«column»
*PK ActTagId: uniqueidentifier
*FK ActId: uniqueidentifier
* Key: varchar(50)
* Value: varbinary(max)
* CreationTime: datetime = CURRENT_TIMESTAMP
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier
Act
«column»
*PK ActId: uniqueidentifier
* ClassConceptId: uniqueidentifier
TextObservation
«column»
*pfK ActVersionId: uniqueidentifier
* TextValue: varchar(max)
QuantityObservation
«column»
*PK ActVersionId: uniqueidentifier
* Quantity: int
* UnitOfMeasureConceptId: uniqueidentifier
Observation
«column»
*pfK ActVersionId: uniqueidentifier
* InterpretationConceptId: uniqueidentifier
ActRelationship
«column»
*PK ActRelationshipId: uniqueidentifier
*FK SourceActId: uniqueidentifier
*FK TargetActId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
* RelationshipTypeConceptId: uniqueidentifier
ActExtension
«column»
*PK ActExtensionId: uniqueidentifier
*FK ExtensionTypeId: uniqueidentifier
* ExtensionValue: varbinary(max)
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
PatientEncounter
«column»
*PK ActVersionId: uniqueidentifier
* DischargeDispositionConceptId: uniqueidentifier
CodedObservation
«column»
*PK ActVersionId: uniqueidentifier
* ValueConceptId: uniqueidentifier
ActVersion
«column»
*pfK ActVersionId: uniqueidentifier
* CreationTime: datetime = CURRENT_TIMESTAMP
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier
ObsoletionReason: varchar(50)
* NegationInd: bit = 0
* TypeConceptId: uniqueidentifier
* StatusConceptId: uniqueidentifier
ActStartTime: datetime
ActEndTime: datetime
FK ActId: uniqueidentifier
* MoodConceptId: uniqueidentifier
ReasonConceptId: uniqueidentifier
SubstanceAdministration
«column»
*PK ActVersionId: uniqueidentifier
* RouteConceptId: uniqueidentifier
* DoseQuantity: float
* DoseUnitConceptId: uniqueidentifier
SequenceId: int
ActIdentifier
«column»
*PK ActIdentifierId: uniqueidentifier
*FK IdentifierTypeId: uniqueidentifier
*FK AssigningAuthorityId: uniqueidentifier
* IdentifierValue: varchar(50)
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
ActParticipation
«column»
*PK ActParticipationId: uniqueidentifier
*FK EntityVersionId: uniqueidentifier
*FK ActEffectiveVersionId: uniqueidentifier
FK ActObsoleteVersionId: uniqueidentifier
* ParticipationRoleConceptId: uniqueidentifier
QuantifiedActParticipation
«column»
*pfK ActParticipationId: uniqueidentifier
* Quantity: int
Meta::AssigningAuthority
«column»
*PK AssigningAuthorityId: uniqueidentifier
* Name: varchar(50)
* Oid: varchar(50)
HL7CX4: varchar(50)
Description: text
* CreationTime: datetime = CURRENT_TIMESTAMP
ObsoleteTime: datetime
ActNote
«column»
*PK ActNoteId: uniqueidentifier = newid()
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
* AuthorEntityId: uniqueidentifier
* NoteText: text
Meta::IdentifierType
«column»
*PK IdentifierTypeId: uniqueidentifier = newid()
* IdentifierTypeConceptId: int
* ClassScopeConceptId: int

Figure 25 - SanteDB Act Model

The data dictionary for the SanteDB act data model is provided in Table 34.

Table 34 - SanteDB Act Data Model

Table	**Column**	**Type**	**Description**
Act	(None)	N/A	The act table is represents the immutable attributes of an act.
	ActId [1..1]	UUID	A unique identifier for the act.
	TemplateDefinitionId [1..1]	UUID	Identifies the template definition that the particular act implements.
	ClassConceptId [1..1] ~ ActClass	UUID	Identifies a concept that classifies the act. This determines the type of act, for example an Observation, PatientEncounter, etc.
	MoodConceptId [1..1] ~ ActMood	UUID	Identifies the mood, or method of the act's performance.
ActTag	(None)	N/A	A table for storing tags related to acts. A tag represents a version independent piece of data attached to a tag.
	ActTagId [1..1]	UUID	A unique identifier for the tag.
	ActId [1..1]	UUID	Identifies the act to which the tag is applied.
	Key [1..1]	VARCHAR	A unique key identifier for the type of tag. A tag's key is used to convey the type of data.
	Value [1..1]	VARBINARY	Contains the binary data of the tag.
	CreationTime [1..1]	DATETIME	Identifies the time when the tag became active, or was created.
	CreatedBy [1..1]	UUID	Identifies the user that was responsible for the creation of the tag.

	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When present, identifies the time that the tag data is no longer valid.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user who obsoleted the act tag.
ActVersion	(None)	N/A	Act events are versioned. The act version table is used to track mutable data.
	ActVersionId [1..1]	UUID	A unique identifier for the version.
	VersionSequenceId [1..1]	INT	A sequence identifier for the version which allows for a time independent mechanism for establishing version order.
	ActId [1..1]	UUID	Identifies the act to which the version data applies.
	CreationTime [1..1]	DATETIME	Identifies the time when the act version became active (was created)
	CreatedBy [1..1]	UUID	Identifies the user that was responsible for creating the version.
	ObsoletionTime [0..1] ? (>CreationTime)	DATETIME	When present, identifies the time when the version of the act is no longer active.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user who was responsible for the obsoletion of the version.
	NegationInd [1..1] = false	BIT	When present, indicates that the act's value is not true. For example, when attempting to convey that a vaccine was not given, the negationInd would be set to true.
	TypeConceptId [0..1]	UUID	Identifies the type of act. This is a type that is

			a subclass within the major classification. For example, if the class is a substance administration, the type concept may represent an Immunization.
	StatusConceptId [1..1] ~ActStatus	UUID	Identifies the status of the act as of the current version.
	ActTime [0..1] ?(ActTime | ActStartTime | ActStopTime)	DATETIME	Identifies the time that the act did occurred, should occur.
	ActStartTime [0..1] ?(ActTime | ActStartTime | ActStopTime)	DATETIME	Identifies the start time of the act.
	ActStopTime [0..1] ?(ActTime | ActStartTime | ActStopTime)	DATETIME	Identifies the stop time of the act.
ActRelationship	(None)	N/A	The act relationship table is used to track the relationship of acts to one another.
	ActRelationshipId [1..1]	UUID	Uniquely identifies the act relationship.
	SourceActId [1..1]	UUID	Identifies the source act of the relationship.
	TargetActId [1..1]	UUID	Identifies the target act of the relationship.
	EffectiveVersionSequenceId [1..1]	UUID	Identifies the version of the source act where this relationship did become active.
	ObsoleteVersionSequenceId [0..1]	UUID	Identifies the version of the source act where this relationship is no longer active.
	RelationshipTypeConceptId [1..1] ~ActRelationshipType	UUID	Identifies the type of relationship that the two acts have to one another.
ActParticipation	(None)	N/A	The ActParticipation table is used to track how entities participate in a particular act.

	ActParticipationId [1..1]	UUID	Uniquely identifies the act participation.
	ActId [1..1]	UUID	Identifies the act that the participation is for.
	EffectiveVersionSequenceId [1..1]	UUID	Identifies the version of the act when the participation is active.
	ObsoleteVersionSequenceId [0..1]	UUID	When present, identifies the version of the act when the participation is no longer active.
	ParticipationRoleConceptId [1..1] ~ActParticipationType	UUID	Qualifies what role the entity played in the carrying out of the act.
	Quantity [1..1] = 1	INT	Identifies the number of entities that are included in the playing of the role.
ActIdentifier	(None)	N/A	The act identifier table is used to store alternate identifiers for the act. This may include vaccine event identifiers, external order identifiers, etc.
	ActIdentifierId [1..1]	UUID	Uniquely identifies the alternate act identifier.
	IdentifierTypeId [0..1]	UUID	Identifies the type of identifier this particular identifier instance represents (order #, etc.)
	AssigningAuthorityId [1..1]	UUID	Identifies the authority that assigned the identifier.
	IdentifierValue [1..1]	VARCHAR	The actual external identifier value.
	EffectiveVersionSequenceId [1..1]	UUID	Identifies the version of the act where the alternate identifier became active.
	ObsoleteVersionSequenceId [0..1]	UUID	When present, identifies the version of the act whereby the

			alternate identifier is no longer active.
ActExtension	(None)	N/A	The act extension table is used to store extensions attached to acts.
	ActExtensionId [1..1]	UUID	A unique identifier for the act extension.
	ExtensionTypeId [1..1]	UUID	Identifies the type of extension represented. This includes information on how the extension should be serialized to/from the ExtensionValue column.
	ExtensionValue [1..1]	VARBINARY	Carries the value of the extension.
	ExtensionDisplay [1..1]	VARCHAR	A human comprehendible display value for the extension.
	EffectiveVersionSequenceId [1..1]	UUID	Indicates the version of the act where this extension became active.
	ObsoleteVersionSequenceId [0..1]	UUID	When present, indicates the version of the act where the extension is no longer active.
Observation	(None)	N/A	The observation table is used to store extended values related to observation act types.
	ActVersionId [1..1]	UUID	Identifies the act version to which this extended data applies.
	InterpretationConceptId [0..1] ~ActInterpretation	UUID	Identifies the concept that represents the interpretation of the observation.
QuantityObservation	(None)	N/A	The quantity observation table is used to store extended information related to the observations that carry quantified values.

	ActVersionId [1..1]	UUID	Identifies the observation act version to which the quantified observation data applies.
	Quantity [1..1]	DECIMAL	A decimal value that contains the value of the observation quantity.
	QuantityPrecision [1..1]	INT	Identifies the precision of the Quantity field.
	UnitOfMeasureConceptId [1..1] ~UnitOfMeasure	UUID	Identifies the concept that identifies the units of measurement.
TextObservation	(None)	N/A	The text observation table is used to store additional information related to a text valued observation.
	ActVersionId [1..1]	UUID	Identifies the observation version id that this text observation data is attached to.
	TextValue [1..1]	TEXT	A textual field that contains the observation data.
CodedObservation	(None)	N/A	The coded observation table is used to store additional data related to observations that are coded. Problems and Allergies would qualify as coded observations.
	ActVersionId [1..1]	UUID	Identifies the version of the observation that this coded observation value data applies.
	ConceptValueId [1..1]	UUID	Identifies the concept that represents the value of observation.
SubstanceAdministration	(None)	N/A	The substance administration table is used to store data related to substance administrations to a

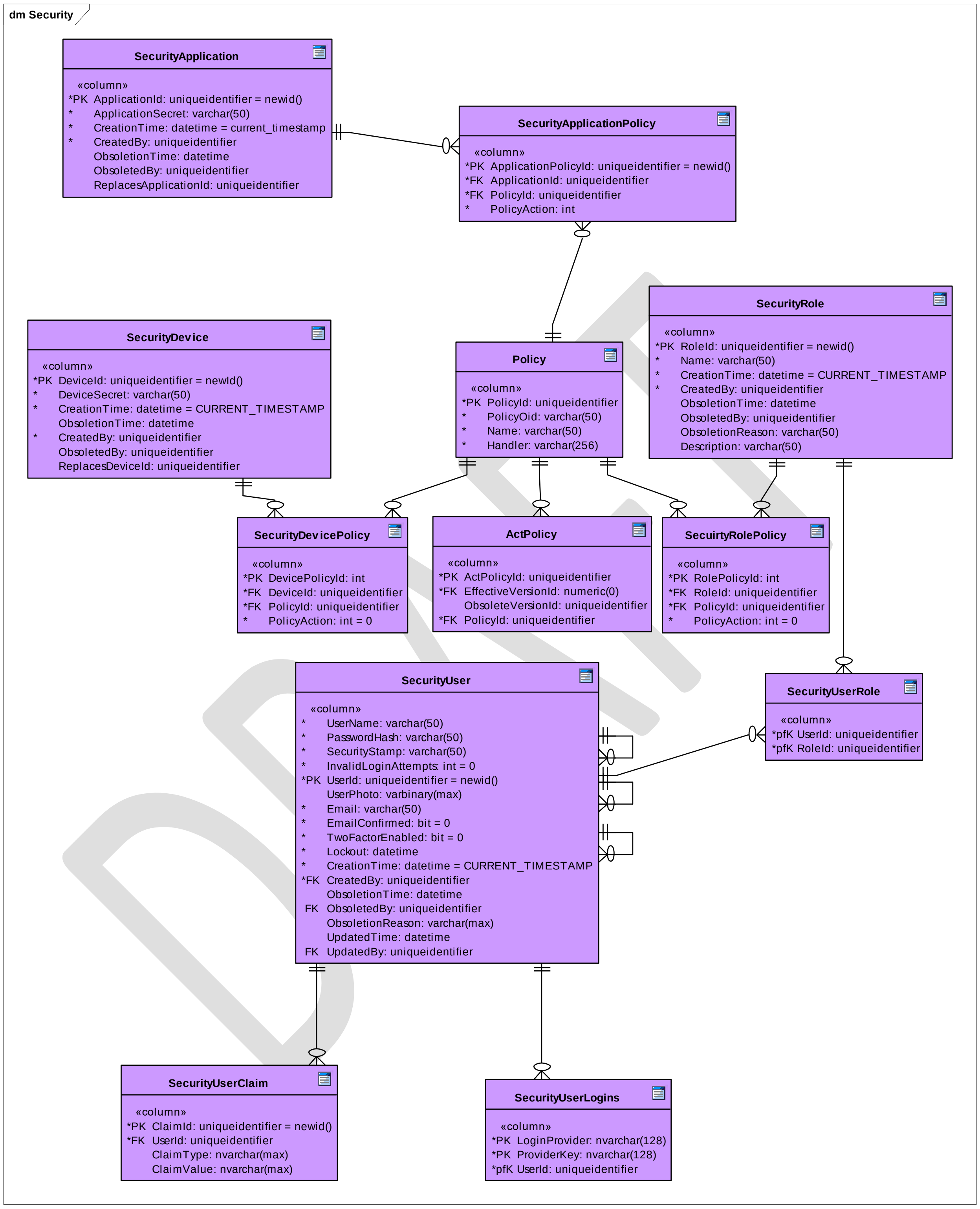
			patient. This include vaccines or any type of Epupin injections for
	ActVersionId [1..1]	UUID	Identifies the act version to which the substance administration data applies.
	RouteConceptId [0..1] ~RouteConcept	UUID	Identifies the concept that describes the route that was taken to administer the substance. This may be drawn from the default route if not supplied.
	DoseQuantity [1..1]	DECIMAL	Identifies the dosage that was given to the patient.
	DoseQuantityPrecision [1..1]	INT	Identifies the precision of the dose quantity.
	DoseUnitConceptId [1..1] ~UnitOfMeasure	INT	Identifies the dose unit of measure that was given to the patient.
	SequenceId [0..1]	INT	Identifies the sequence of this dose if it is a part of a sequence of doses.
PatientEncounter	(None)	N/A	The patient encounter table is used to store additional data related to an act that represents a patient encounter.
	ActVersionId [1..1]	UUID	Identifies the act to which the extended patient encounter data applies.
	DischargeDispositionConceptId [0..1] ~DischargeDisposition	UUID	Identifies the disposition in which the patient left the encounter.
ActNote	(None)	N/A	The act note table is used to store notes associated with an act.
	ActNoteId [1..1]	UUID	A unique identifier for the note.

	EffectiveVersionSequenceId [1..1]	UUID	The version whereby the note became effective
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, indicates the version of the act where the note is no longer active.
	AuthorEntityId [1..1]	UUID	The identifier of the entity that wrote the note.
	NoteText [1..1]	TEXT	The textual content of the note.
Procedure	(None)	N/A	Used to track acts which alter the physical state of an entity (i.e. surgeries, etc.)
	ActVersionId [1..1]	UUID	Points to the version of the Act which this procedure is describing.
	MethodCodeId [0..1] ~ProcedureTechniqueCode	UUID	Identifies the formal method for performing the procedure.
	ApproachSiteCodeId [0..1] ~BodySiteOrSystemCode	UUID	Identifies the manner in which the target site was approached for the procedure.
	TargetSiteCodeId [0..1] ~BodySiteOrSystemCode	UUID	Identifies the body system/part which was the target of the procedure.

8.2.6. SanteDB Security Model

One of the key tenants of the SanteDB immunization management system is privacy and security by design. To that end, SanteDB's HDS supports not only external policy enforcement decisions and role providers, but also provides access to internal policy engines (when external policy decision points are not available).

Figure 26 illustrates the relationships between the various security sub systems tables found in the SanteDB data model.



dm Security
SecurityApplication
«column»
*PK ApplicationId: uniqueidentifier = newid()
* ApplicationSecret: varchar(50)
* CreationTime: datetime = current_timestamp
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier
ReplacesApplicationId: uniqueidentifier
SecurityApplicationPolicy
«column»
*PK ApplicationPolicyId: uniqueidentifier = newid()
*FK ApplicationId: uniqueidentifier
*FK PolicyId: uniqueidentifier
* PolicyAction: int
SecurityRole
«column»
*PK RoleId: uniqueidentifier = newid()
* Name: varchar(50)
* CreationTime: datetime = CURRENT_TIMESTAMP
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier
ObsoletionReason: varchar(50)
Description: varchar(50)
SecurityDevice
«column»
*PK DeviceId: uniqueidentifier = newId()
* DeviceSecret: varchar(50)
* CreationTime: datetime = CURRENT_TIMESTAMP
ObsoletionTime: datetime
* CreatedBy: uniqueidentifier
ObsoletedBy: uniqueidentifier
ReplacesDeviceId: uniqueidentifier
Policy
«column»
*PK PolicyId: uniqueidentifier
* PolicyOid: varchar(50)
* Name: varchar(50)
* Handler: varchar(256)
SecurityDevicePolicy
«column»
*PK DevicePolicyId: int
*FK DeviceId: uniqueidentifier
*FK PolicyId: uniqueidentifier
* PolicyAction: int = 0
ActPolicy
«column»
*PK ActPolicyId: uniqueidentifier
*FK EffectiveVersionId: numeric(0)
ObsoleteVersionId: uniqueidentifier
*FK PolicyId: uniqueidentifier
SecuirtyRolePolicy
«column»
*PK RolePolicyId: int
*FK RoleId: uniqueidentifier
*FK PolicyId: uniqueidentifier
* PolicyAction: int = 0
SecurityUser
«column»
* UserName: varchar(50)
* PasswordHash: varchar(50)
* SecurityStamp: varchar(50)
* InvalidLoginAttempts: int = 0
*PK UserId: uniqueidentifier = newid()
UserPhoto: varbinary(max)
* Email: varchar(50)
* EmailConfirmed: bit = 0
* TwoFactorEnabled: bit = 0
* Lockout: datetime
* CreationTime: datetime = CURRENT_TIMESTAMP
*FK CreatedBy: uniqueidentifier
ObsoletionTime: datetime
FK ObsoletedBy: uniqueidentifier
ObsoletionReason: varchar(max)
UpdatedTime: datetime
FK UpdatedBy: uniqueidentifier
SecurityUserRole
«column»
*pfK UserId: uniqueidentifier
*pfK RoleId: uniqueidentifier
SecurityUserClaim
«column»
*PK ClaimId: uniqueidentifier = newid()
*FK UserId: uniqueidentifier
ClaimType: nvarchar(max)
ClaimValue: nvarchar(max)
SecurityUserLogins
«column»
*PK LoginProvider: nvarchar(128)
*PK ProviderKey: nvarchar(128)
*pfK UserId: uniqueidentifier

Figure 26 - SanteDB Security Model

Describes the data dictionary for the SanteDB security model.

Table	Column	Type	Description

Policy	(None)	N/A	The policy table is a complete dictionary of policies that can be applied to acts within the SanteDB HDS.
	PolicyId [1..1]	UUID	Uniquely identifies the policy within the SanteDB system.
	PolicyOid [1..1]	VARCHAR	A globally unique identifier in the form of an OID for the policy.
	Name [1..1]	VARCHAR	A human readable name for the policy.
	Handler [1..1]	VARCHAR	An assembly qualified name (AQN) of an IPolicyHandler implementation which is triggered when the policy rule fires.
SecurityUser	(None)	N/A	The security user table is used to store a master list of users that have secured access to the SanteDB HDS functions.
	UserId [1..1]	UUID	A unique identifier for the user.
	UserName [1..1]	VARCHAR	A unique identifier for the security user that a human may use to access the SanteDB HDS system.
	PasswordHash [1..1]	VARCHAR	A SHA256 hash of the user's password.
	SecurityStamp [1..1]	VARCHAR	A unique security stamp for the user account. This can include a salt for the user password, or some other security tag for the user.
	InvalidLoginAttempts [1..1] = 0	INT	Identifies the number of times that a person has attempted to access the SanteDB HDS with invalid credentials.
	UserPhoto [0..1]	VARBINARY	An optional photograph for the user.
	Email [0..1]	VARCHAR	Identifies an electronic mail telecommunications address that can be used to contact the user.
	EmailConfirmed [1..1] = False	BIT	Indicates whether the email address of the user has been confirmed.

	TwoFactorEnabled [1..1] = False	BIT	Indicates whether the user account requires two-factor authentication. The TFA mechanism is enabled by the ITwoFactorAuthenticationService implementation.
	LockoutTime [0..1]	DATETIME	Indicates the time when the account is locked out until. A DateTime.MaxValue indicates that the account is perminently locked out.
	CreationTime [1..1]	DATETIME	Identifies the time when the user account was created.
	CreatedBy [1..1]	UUID	Identifies the user who was responsible for the creation of the security user.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When populated, indicates the time when the user account did or will become obsolete.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	The identifier of the user who was responsible for obsoleting the record.
	ObsoletionReason [0..1] ?(ObsoletionTime)	VARCHAR	Identifies the reason why the security user was obsoleted.
	UpdatedTime [0..1] ?(>CreationTime)	DATETIME	Identifies the last timestamp that the user record was updated.
	UpdatedBy [0..1] ?(UpdatedTime)	UUID	Identifies the user who was responsible for the last edit of the security user.
SecurityUserClaims	(None)	N/A	The security user claims table is used to store claims associated with a user account. These claims can be things like TFA secrets, refresh tokens, etc.
	ClaimId [1..1]	UUID	A unique identifier of the claim
	UserId [1..1]	UUID	Identifies the user to which the claim applies.
	ClaimType [1..1]	VARCHAR	Identifies the type or classification of claim that has been made.
	ClaimValue [1..1]	VARCHAR	Identifies the value of the claim token
SecurityUserLogins	(None)	N/A	The security user logins table is used to track external

			authorization providers associated with a user account.
	LoginProvider [1..1]	VARCHAR	The provider (google, Microsoft, etc.) which holds the external credential.
	ProviderKey [1..1]	VARCHAR	The key of the user identifier in the provider system.
	UserId [1..1]	UUID	Identifies the user to which the external login applies.
SecurityRole	(None)	N/A	The security role table is used to store security (user) roles that can be used in policy based decisions.
	RoleId [1..1]	UUID	Uniquely identifies the security role.
	Name [1..1]	VARCHAR	A human readable name for the role.
	CreationTime [1..1]	DATETIME	Identifies the moment in time when the security role was created.
	CreatedBy [1..1]	UUID	Identifies the user who was responsible for the creation of the role.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When present, identifies the date/time when the role became obsolete.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user who was responsible for the obsoletion.
	ObsoletionReason [0..1] ?(ObsoletionTime)	VARCHAR	Indicates the reason for the obsoletion of the record.
SecurityUserRole	(None)	N/A	An associative entity table between a security user and role.
	UserId [1..1]	UUID	Identifies the user of the association.
	RoleId [1..1]	UUID	Identifies the role to which the association applies.
SecurityRolePolicy	(None)	N/A	The security role policy is an associative entity table that links security roles to policies which can be used in a policy decision.
	RolePolicyId [1..1]	UUID	Uniquely identifies the tuple
	RoleId [1..1]	UUID	Identifies the role to which the security role policy association applies.

	PolicyId [1..1]	UUID	Identifies the policy that is being applied to the role.
	Grant [1..1] = False	INT	Indicates the grant level for the particular policy either deny, grant or elevate.
	CanOverride [0..1] = True	BIT	When true, indicates that when a policy decision is made, a user within the role can override the decision.
SecuritySession	(None)	N/A	Table that stores security sessions established with the server.
	SessionId [1..1]	UUID	Uniquely identifies the session.
	UserId [0..1]	UUID	The user SID which holds the session.
	DeviceId [0..1]	UUID	The device SID that holds the session.
	ApplicationId [1..1]	UUID	The application SID that holds the session.
	NotBefore [1..1]	DATETIME	Identifies the time when the session becomes valid.
	NotAfter [1..1]	DATETIME	Identifies the time that the session is invalid.
	RefreshToken [1..1]	VARBINAR Y	Identifies the refresh token that can be used to refresh the session.
ActPolicy	(None)	N/A	The ActPolicy table is used to associate a policy with an act.
	ActPolicyId [1..1]	UUID	A unique identifier for the policy identifier.
	ActId [1..1]	UUID	Identifies the act to which the association applies
	EffectiveVersionSequenceI d [1..1]	INT	Identifies the version of the act whereby the policy is active.
	ObsoleteVersionSequenceI d [1..1]	INT	Indicates the version of the act where the policy no longer applies.
	PolicyId [1..1]	UUID	Identifies the policy that is associated with the act.
EntityPolicy	(None)	N/A	The EntityPolicy table is used to associate security policies with entities.
	EntityPolicyId [1..1]	UUID	The unique identifier of the entity policy association.

	EntityId	UUID	Identifies the entity to which the policy is applied.
	EffectiveVersionSequenceId [1..1]	INT	Identifies the version of the entity when the policy became effective.
	ObsoleteVersionSequenceId [1..1]	INT	Identifies the version of the entity where the policy is no longer effective.
	PolicyId [1..1]	UUID	Identifies the policy that applies to the entity.
SecurityDevice	(None)	N/A	The security device table is used to store data related to an authorized device that can access the SanteDB HDS.
	DeviceId [1..1]	UUID	Uniquely identifies the device.
	DeviceSecret [1..1]	VARBINARY	A secret that is used to verify whether the device can connect.
	CreationTime [1..1]	DATETIME	Indicates the time when the record was created.
	CreatedBy [1..1]	UUID	Identifies the user responsible for the creation of the record.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When present, indicates the time when the device record became or will become obsolete.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user that is responsible for the obsoletion of the device.
	ReplacesDeviceId [0..1]	UUID	Indicates the old device that the current device would replace.
SecurityDevicePolicy	(None)	N/A	An associated entity that links a security device to a policy.
	DevicePolicyId [1..1]	UUID	A unique identifier for the device policy association.
	DeviceId [1..1]	UUID	Identifies the device to which the association applies.
	PolicyId [1..1]	UUID	Indicates the policy to which the association applies.
	IsDeny [1..1] = False	BIT	When true, instructs the decision engine to deny access to an act or policy.
SecurityApplication	(None)	N/A	The security application table is used to store records associated with an application.

	ApplicationId [1..1]	UUID	Uniquely identifies the application.
	ApplicationSecret [1..1]	VARBINARY	A secret that is used by the application to authenticate itself.
	CreationTime [1..1]	DATETIME	The time when the application was created.
	CreatedBy [1..1]	UUID	The user responsible for registering the application.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	The time that the application record did become or will become obsolete.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Indicates the user that the obsoleted the record.
	ReplacesApplicationId [0..1]	UUID	Identifies the application that this current version of the application record replaces.
SecurityApplicationPolicy	(None)	N/A	An associated entity that links a security application to a policy.
	ApplicationPolicyId [1..1]	UUID	A unique identifier for the application policy association.
	ApplicationId [1..1]	UUID	Identifies the application to which the association applies.
	PolicyId [1..1]	UUID	Indicates the policy to which the association applies.
	IsDeny [1..1] = False	BIT	When true, instructs the decision engine to deny access to an act or policy.
SecurityProvenance	(None)	N/A	The security provenance table is responsible for storing data related to the provenance of a particular version of an object.
	ProvenanceId [1..1]	UUID	The unique identifier of the provenance object.
	SessionId [0..1]	UUID	The identity of the session which was active when the provenance object was created.
	UserId [0..1]	UUID	The identity of the user that was authenticated when the provenance object was established.
	ApplicationId [1..1]	UUID	The identity of the application that was authenticated when the provenance object was established.
	DeviceId [0..1]	UUID	The identity of the device that was authentication when the

			provenance object was established.
	Established [1..1]	DATETIME	The date/time that the database transaction started which resulted in the object being created.
	SecurityObjectRef [0..1]	UUID	The identifier of the user/provenance object on the client system.
	SecurityObjectType [0..1] ?(U,P)	CHAR	The type of object that the security object reference points to.

8.2.7. SanteDB Stock Model

Stock management within SanteDB is performed via a series of linkages between entities (Places own Materials) in a quantity holding the current balance. Any adjustments, orders or transfers are performed using account management acts.

8.2.8. SanteDB Entity Model

The SanteDB entity model represents a series of tables which are responsible for the tracking of entities within the SanteDB data model. Entities represent people, places, organizations, things, etc. and are responsible for participating within acts in some capacity.

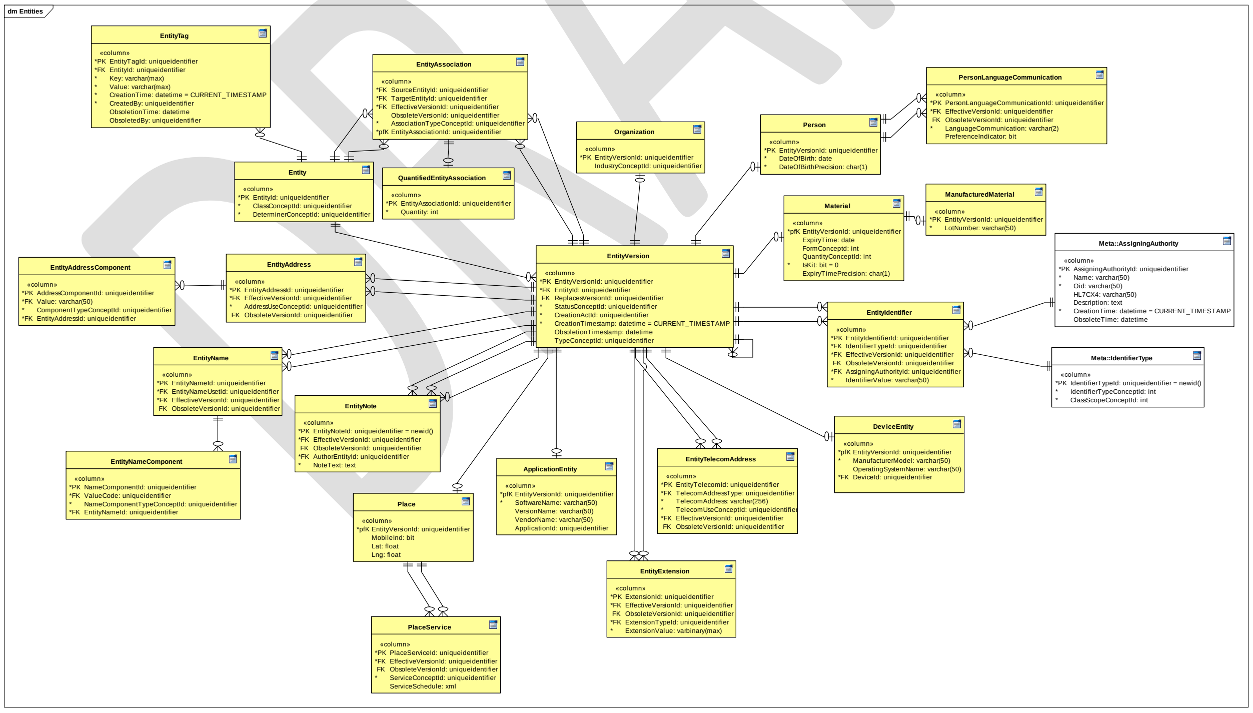

dm Entities
EntityTag
«column»
*PK EntityTagId: uniqueidentifier
*FK EntityId: uniqueidentifier
* Key: varchar(max)
* Value: varchar(max)
* CreationTime: datetime = CURRENT_TIMESTAMP
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier
EntityAssociation
«column»
*FK SourceEntityId: uniqueidentifier
*FK TargetEntityId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
ObsoleteVersionId: uniqueidentifier
* AssociationTypeConceptId: uniqueidentifier
*pfK EntityAssociationId: uniqueidentifier
Entity
«column»
*PK EntityId: uniqueidentifier
* ClassConceptId: uniqueidentifier
* DeterminerConceptId: uniqueidentifier
QuantifiedEntityAssociation
«column»
*PK EntityAssociationId: uniqueidentifier
* Quantity: int
Organization
«column»
*PK EntityVersionId: uniqueidentifier
IndustryConceptId: uniqueidentifier
Person
«column»
*PK EntityVersionId: uniqueidentifier
* DateOfBirth: date
* DateOfBirthPrecision: char(1)
PersonLanguageCommunication
«column»
*PK PersonLanguageCommunicationId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
* LanguageCommunication: varchar(2)
PreferenceIndicator: bit
Material
«column»
*pfK EntityVersionId: uniqueidentifier
ExpiryTime: date
FormConceptd: int
QuantityConceptId: int
* IsKit: bit = 0
ExpiryTimePrecision: char(1)
ManufacturedMaterial
«column»
*PK EntityVersionId: uniqueidentifier
* LotNumber: varchar(50)
Meta::AssigningAuthority
«column»
*PK AssigningAuthorityId: uniqueidentifier
* Name: varchar(50)
* Oid: varchar(50)
HL7CX4: varchar(50)
Description: text
* CreationTime: datetime = CURRENT_TIMESTAMP
ObsoleteTime: datetime
EntityAddressComponent
«column»
*PK AddressComponentId: uniqueidentifier
*FK Value: varchar(50)
* ComponentTypeConceptId: uniqueidentifier
*FK EntityAddressId: uniqueidentifier
EntityAddress
«column»
*PK EntityAddressId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
* AddressUseConceptId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
EntityVersion
«column»
*PK EntityVersionId: uniqueidentifier
*FK EntityId: uniqueidentifier
FK ReplacesVersionId: uniqueidentifier
* StatusConceptId: uniqueidentifier
* CreationActId: uniqueidentifier
* CreationTimestamp: datetime = CURRENT_TIMESTAMP
ObsoletionTimestamp: datetime
TypeConceptId: uniqueidentifier
EntityIdentifier
«column»
*PK EntityIdentifierId: uniqueidentifier
*FK IdentifierTypeId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
*FK AssigningAuthorityId: uniqueidentifier
* IdentifierValue: varchar(50)
Meta::IdentifierType
«column»
*PK IdentifierTypeId: uniqueidentifier = newid()
* IdentifierTypeConceptId: int
* ClassScopeConceptId: int
EntityName
«column»
*PK EntityNameId: uniqueidentifier
*FK EntityNameUsetId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
EntityNote
«column»
*PK EntityNoteId: uniqueidentifier = newid()
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
*FK AuthorEntityId: uniqueidentifier
* NoteText: text
DeviceEntity
«column»
*pfK EntityVersionId: uniqueidentifier
* ManufacturerModel: varchar(50)
OperatingSystemName: varchar(50)
*FK DeviceId: uniqueidentifier
EntityNameComponent
«column»
*PK NameComponentId: uniqueidentifier
*FK ValueCode: uniqueidentifier
* NameComponentTypeConceptId: uniqueidentifier
*FK EntityNameId: uniqueidentifier
ApplicationEntity
«column»
*pfK EntityVersionId: uniqueidentifier
* SoftwareName: varchar(50)
VersionName: varchar(50)
VendorName: varchar(50)
ApplicationId: uniqueidentifier
EntityTelecomAddress
«column»
*PK EntityTelecomId: uniqueidentifier
*FK TelecomAddressType: uniqueidentifier
* TelecomAddress: varchar(256)
* TelecomUseConceptId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
Place
«column»
*pfK EntityVersionId: uniqueidentifier
MobileInd: bit
Lat: float
Lng: float
EntityExtension
«column»
*PK ExtensionId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
*FK ExtensionTypeId: uniqueidentifier
* ExtensionValue: varbinary(max)
PlaceService
«column»
*PK PlaceServiceId: uniqueidentifier
*FK EffectiveVersionId: uniqueidentifier
FK ObsoleteVersionId: uniqueidentifier
* ServiceConceptId: uniqueidentifier
ServiceSchedule: xml

Table	Column	Type	Description
Entity	(None)	N/A	The entity table is responsible for the

			storage of immutable attributes of an entity.
	EntityId [1..1]	UUID	Uniquely identifies the entity within the context of the SanteDB implementation.
	TemplateDefinitionId [0..1]	UUID	Identifies the template which the entity instance implements.
	ClassConceptId [1..1] ~EntityClassConcept	UUID	Identifies the concept that classifies the entity by a type. The classifier is used to determine “WHAT TYPE” of entity the tuple represents such as a person, material, manufactured material, organization, place, etc.
	DeterminerConceptId [1..1] ~EntityDeterminerConcept	UUID	Identifies the concept that classifies or determines the type of entity. This is either an INSTANCE or CLASS concept identifier.
EntityTag	(None)	N/A	The entity tag table is used to store version independent tags associated with an entity. A tag does not result in new versions of the entity and is used to track additional data related to security and/or workflow related metadata.

	EntityTagId [1..1]	UUID	Uniquely identifies the entity tag.
	EntityId [1..1]	UUID	Identifies the entity to which the tag is associated.
	Key [1..1]	VARCHAR	Qualifies the type of tag associated with the entity. That is to say, type of tag is represented in the tuple of the determiner.
	Value [1..1]	VARCHAR	A value that carries the data associated with the tag value.
	CreationTime [1..1]	DATETIME	Indicates the date/time at which time the tag was created.
	CreatedBy [1..1]	UUID	Identifies the user that was responsible for the creation of the tag.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When populated, indicates the time when the tag is no longer associated with the entity.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user that was responsible for obsoleting the tag.
EntityVersion	(None)	N/A	The entity version table is used to store the mutable attributes of an entity, that is to say, any fields associated with an entity that may evolve over the lifespan of the entity are tracked in this table.
	EntityVersionId [1..1]	UUID	Uniquely identifies the version of the entity represented in the tuple.

	EntityId [1..1]	UUID	Identifies the entity to which this version applies.
	ReplacesVersionId [0..1]	UUID	Identifies the version of the entity that the current tuple is responsible for replacing.
	StatusConceptId [1..1] ~EntityStatusConcept	UUID	Identifies the status of the entity as of the version represented in the tuple.
	CreationTime [1..1]	DATETIME	Indicates the time when the entity was created.
	CreatedBy [1..1]	UUID	Identifies the user that was responsible for the creation of the entity.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When populated, indicates the time when the entity version became obsolete.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user that was responsible for the obsoleting of the record.
	TypeConceptId [0..1]	UUID	Indicates the concept that classifies the subtype of entity. For example, an entity may be a provider; however, the sub-type may be a “physiotherapist”.
EntityRelationship	(none)	N/A	The entity association table is used to associate two or more entities together. An association is made between a source entity and a target entity.

	EntityRelationshipId [1..1]	UUID	Uniquely identifies the entity association.
	SourceEntityId [1..1]	UUID	Identifies the source of the entity association.
	TargetEntityId [1..1]	UUID	Identifies the target of the entity association.
	EffectiveVersionSequenceId [1..1]	UUID	Indicates the version of the source entity at which time this entity association was created or became effective.
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, indicates that the entity association is no longer active, and indicates the version of the source entity where the association ceased to be applicable.
	RelationshipTypeConceptId [1..1] ~EntityRelationshipType	UUID	Classifies the relationship between the two entities. Can indicate ownership roles such as "Place OWNS Material", or relationship "Patient CHILD OF Person".
	Quantity [1..1] = 1	INT	Indicates the quantity of target entities contained within the source entity.
EntityNote	(None)	N/A	The entity note table is used to store textual notes related to an etity.
	EntityNoteId [1..1]	UUID	Uniquely identifies the note.
	EffectiveVersionSequenceId [1..1]	UUID	Identifies the version of the entity

			to which the note applies.
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, indicates the version of the entity where the note is no longer relevant.
	AuthorEntityId [1..1]	UUID	Identifies the entity that was responsible for the authoring of the note.
	NoteText [1..1]	TEXT	Indicates the textual content of the note.
EntityAddress	(None)	N/A	The entity address table is used to store address information (physical addresses) related to an entity.
	EntityAddressId [1..1]	UUID	Uniquely identifies the entity address.
	EffectiveVersionSequenceId [1..1]	UUID	Identifies the version of the entity whereby the address information became active.
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, indicates the version of the entity whereby the address is no longer applicable.
	AddressUseConceptId [1..1] ~AddressUseType	UUID	Indicates the desired use of the address. Examples include physical visit, vacation home, contact, mailing, etc.
EntityAddressComponent	(None)	N/A	The entity address component table is used to store the address components associated with a particular entity address.
	EntityAddressComponentId [1..1]	UUID	Uniquely identifies the entity address component.

	Value [1..1]	VARCHAR	Identifies the value of the of the address component
	ComponentTypeConceptId [1..1] ~NameComponentType	UUID	Classifies the type of address component represented in the value field. For example: street name, city, country, postal code, etc.
	EntityAddressId [1..1]	UUID	Identifies the entity address to which the entity address component applies.
EntityName	(None)	N/A	The entity name table is used to store master list of names associated with an entity.
	EntityNameId [1..1]	UUID	Uniquely identifies the entity name.
	EntityNameUseId [1..1]	UUID	Classified the intended use of the entity name. Examples: maiden name, legal name, license name, artist name, etc.
	EffectiveVersionSequenceId [1..1]	UUID	Identifies the version of the entity when this name became active.
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, identifies the version of the entity where the name is no longer active.
EntityNameComponent	(None)	N/A	The entity name component table is responsible for the storage of name components that comprise an entity name.
	NameComponentId [1..1]	UUID	Uniquely identifies the name component.

	ValueId [1..1]	UUID	Indicates the phonetic value tuple that stores the name value.
	NameComponentTypeConceptId [1..1] ~EntityComponentType	UUID	Classifies the type of name component represented. Examples: first name, title, family name, etc.
	EntityNameId [1..1]	UUID	Indicates the entity name to which the name component applies.
EntityIdentifier	(None)	N/A	The entity identifier is table is responsible for the storage of alternate identifies associated with the entity.
	EntityIdentifierId [1..1]	UUID	Uniquely identifies the entity identifier.
	IdentifierTypeId [1..1]	UUID	Classifies the type of identifier that is represented by the entity identifier. Examples: business identifier, mrn, primary identifier, etc.
	EffectiveVersionSequenceId [1..1]	UUID	Indicates the version of the entity when the identifier became active.
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, indicates the version of the entity where the identifier is no longer active.
	AssigningAuthorityId [1..1]	UUID	Identifies the authority that was responsible for the assigning of the identifier.
	IdentifierValue [1..1]	VARCHAR	Indicates the value of the entity identifier.

Place	(None)	N/A	The place table represents a specialization of the Entity table which is used to represent physical places such as clinics, outreach activity sites, etc.
	EntityVersionId [1..1]	UUID	Identifies the version of the entity to which the place data applies.
	MobileInd [1..1] = False	BIT	Indicator that is used to identify that a place is mobile.
	Lat [0..1]	FLOAT	The latitudinal position of the place expressed in degrees latitude.
	Lng [0..1]	FLOAT	The longitudinal position of the place expressed in degrees longitude.
PlaceService	(None)	N/A	The place service table is used to identify the services that are provided at a particular place. Services may include stocking, transfer depots, immunization.
	PlaceServiceId [1..1]	UUID	A unique identifier for the place service.
	EffectiveVersionSequenceId [1..1]	UUID	The version of the place entity where the service entry is active.
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, indicates the version of the place entity where the service entry is no longer valid.
	ServiceConceptId [1..1] ~ServiceType	UUID	Indicates a concept that describes the service offered.

	ServiceSchedule [0..1]	XML	An XML expression of the service schedule.
	ServiceScheduleType [0..1]	VARCHAR	Identifies the type of data stored in the service schedule column (iCal, GTS, etc.)
ApplicationEntity	(None)	N/A	The application entity table is used to store entity data related to an application. An application is a software program that runs on a device. This differs from a security application, in that an application may be referenced clinically without needing access to the SanteDB system. For example: The patient uses MyPHR
	EntityVersionId [1..1]	UUID	Identifies the version of the entity to which the application data applies.
	SoftwareName [0..1]	VARCHAR	Identifies the name of the software package (“EMR Package” is an example)
	VersionName [0..1]	VARCHAR	Identifies the version of the software (example: “1.0”)
	VendorName [0..1]	VARCHAR	The name of the vendor which distributes the software application (example: “ABC Corp”)

	ApplicationId [0..1]	UUID	When populated, links the application entity to a security application.
EntityExtension	(None)	N/A	The entity extension table is used to store additional, clinically relevant, versioned data attached to an entity that cannot be stored in the native data model.
	EntityExtensionId [1..1]	UUID	Uniquely identifies the extension.
	EffectiveVersionSequenceId [1..1]	UUID	Indicates the version of the entity when the extension data did become active.
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, indicates the version of the entity where the extension value is no longer applicable.
	ExtensionTypeId [1..1]	UUID	Indicates the type, or handler, for the extension data.
	ExtensionData [1..1]	VARBINARY	Serialized data that contains the raw value of the extension (serialized and de-serialized by the handler).
	ExtensionDisplay [1..1]	VARCHAR	A textual, human readable expression of the extension value which can be displayed on reports, etc.
EntityTelecomAddress	(None)	N/A	The entity telecommunications address table is used to store data related to telecommunications addresses (email,

			fax, phone, etc.) for an entity.
	EntityTelecomId [1..1]	UUID	Uniquely identifies the telecommunications address.
	TelecomAddressType [1..1] ~TelecomAddressType	UUID	Classifies the type of address represented (example: phone, fax, email, etc.)
	TelecomAddress [1..1]	VARCHAR	The value of the telecommunications address in RFC-2396 format.
	TelecomUseConceptId [0..1] ~TelecomAddressUse	UUID	Identifies the intended use of the telecom address. (Example: home, work, etc.)
	EffectiveVersionSequenceId [1..1]	UUID	Identifies the version of the entity whereby the telecom address became effective.
	ObsoleteVersionSequenceId [0..1]	UUID	When populated, identifies the version of the entity where the telecom address is no longer valid.
DeviceEntity	(None)	N/A	The device table is used to store clinical information related to a physical device. Like an application entity, this table is used to describe the clinical attributes of a device used in the provisioning of care. Example: Bob's Insulin Pump. The insulin pump itself may have no security device as it doesn't require access to SanteDB.

	EntityVersionId [1..1]	UUID	Indicates the version of the entity to which the device data applies.
	ManufacturerModel [0..1]	VARCHAR	Indicates the name of the manufacturer of the device.
	OperatingSystemName [0..1]	VARCHAR	Indicates the name of the operating system installed on the device.
	DeviceId [0..1]	UUID	When populated, identifies the security device associated with the device entity.
Material	(None)	N/A	A material represents a physical thing (syringe, drug, etc.) which participates in an act or is assigned to a person.
	EntityVersionId [1..1]	UUID	Identifies the version of the entity to which the material data applies.
	ExpiryTime [1..1]	DATETIME	Indicates the time when the material will expire.
	ExpiryTimePrecision [1..1]	CHAR	Indicates the precision that the expiry time has.
	FormConceptId [0..1] ~MaterialForm	UUID	Identifies a concept that denotes the form that the material takes. Examples: capsule, injection, nebulizer, etc. For drugs and vaccines, the form will imply the route of administration.
	QuantityConceptId [0..1] ~UnitOfMeasure	UUID	Indicates the unit of measure for a single unit of the material.

			Examples: dose, mL, etc.
	Quantity [0..1] = 1	NUMERIC	Indicates the reference quantity in UOM. For example, BCG MMAT is 5 mL of BCG Antigen
	IsAdministrative [1..1] = false	BIT	An indicator that is used to identify whether the material is a real material or an administrative material for the purpose of management.
ManufacturedMaterial	(None)	N/A	A manufactured material is a specialization of a material that is manufactured.
	EntityVersionId [1..1]	UUID	Indicates the version of the material to which the specialized data applies.
	LotNumber [1..1]	VARCHAR	Indicates the manufacturer lot for the material.
Person	(None)	N/A	Person represents a specialization of Entity representing a person.
	EntityVersionId [1..1]	N/A	The version of the entity to which the person data applies.
	DateOfBirth [0..1]	DATE	Indicates the date on which the person entity was born.
	DateOfBirthPrecision [0..1]	CHAR	Indicates the precision of the date of birth field.
PersonLanguage Communication	(None)	N/A	The person language communication table is used to store information related

			to the person's language preferences. This can be used by the user interface to determine which language to display, however is also clinically relevant to indicate the language in which a patient wishes to receive communciations.
	PersonLanaugeCommunicationId [1..1]	UUID	Uniquely identifies the language of communication.
	EffectiveVersionSequenceId [1..1]	UUID	Indicates the version of the person entity whereby the language of communication is effective.
	ObsoleteVersionSequenceId [0..1]	UUID	When present, indicates the version of the person entity where the language of communication is no longer effective.
	LanguageCommunication [1..1] ~ISO639-2	VARCHAR	An ISO-639-2 language code indicating the language preference.
	PreferenceIndicator [0..1] = False	BIT	Indicates whether the person prefers the language for communications.
Organization	(None)	N/A	The organization table represents a specialization of an entity representing a logical organization.
	EntityVersionId [1..1]	UUID	Indicates the version of the entity to which the organization

			specialization applies.
	IndustryConceptId [0..1] ~IndustryConcept	UUID	Indicates the industry in which the organization operates. Examples: logistics, healthcare, etc.

8.2.8.1. Relationships between Entities

There are two types of relationships that can exist between entities (quantified and unquantified). An unquantified relationship represents a 1:1 relationship between things and merely identifies that two items are related in some manner. For example, one may say John [Patient] is related to Mary [Person] by way that Mary is John's mother.

Quantified relationships are used for expressing when a certain number of entities are related to a parent. For example, a Box of BCG [Container] contains 25 BCG [Material]. Quantified relationships are only used to represent per relationships and do not take the quantity UOM into consideration (for that use the quantity field on the actual Entity). For example, a box of GSK BCG vial of 5 mL may contain 25 5 ml BCG vials, where each BCG vial is 5 ml of BCG. This more complex relationship is illustrated in Figure 27.

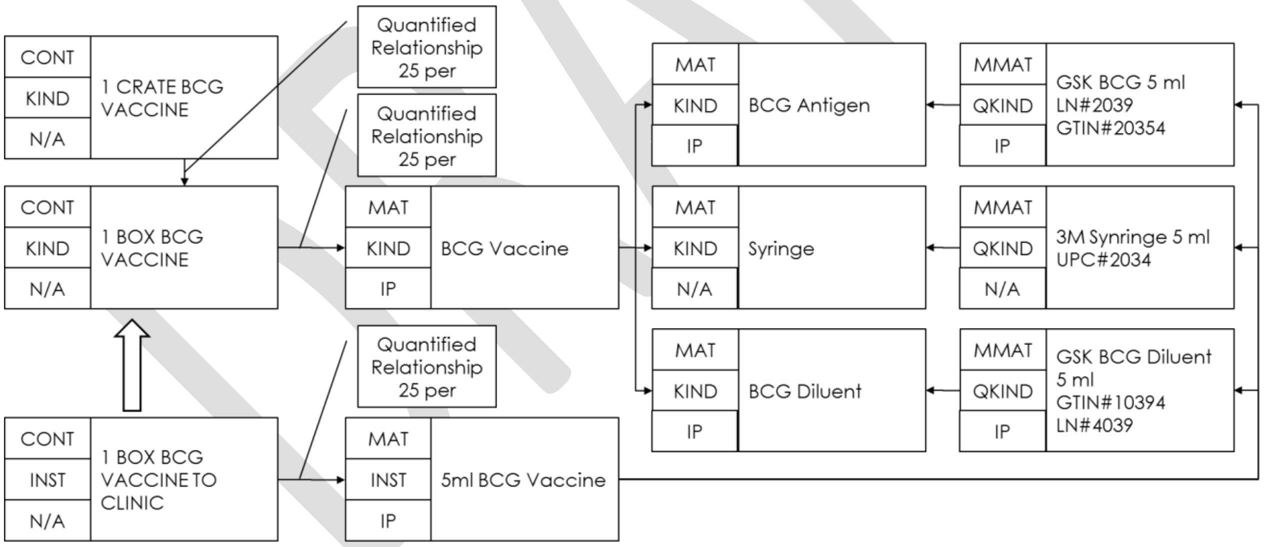

CONT
KIND
N/A
1 CRATE BCG VACCINE
Quantified Relationship 25 per
Quantified Relationship 25 per
CONT
KIND
N/A
1 BOX BCG VACCINE
MAT
KIND
IP
BCG Vaccine
Quantified Relationship 25 per
CONT
INST
N/A
1 BOX BCG VACCINE TO CLINIC
MAT
INST
IP
5ml BCG Vaccine
MAT
KIND
IP
BCG Antigen
MAT
KIND
N/A
Syringe
MAT
KIND
IP
BCG Diluent
MMAT
QKIND
IP
GSK BCG 5 ml LN#2039 GTIN#20354
MMAT
QKIND
N/A
3M Synringe 5 ml UPC#2034
MMAT
QKIND
IP
GSK BCG Diluent 5 ml GTIN#10394 LN#4039

Figure 27 - Entity Relationships

Additionally the association is qualified by the role code in which the target plays in the source. Examples of relationship types are:

Relationship	Mnemonic	Description
Family Member	NextOfKin	
Owned Stock	OwnedEntity	
Personal Relationship	PersonalRelationship	
Employee	Employee	

Dedicated Location	DedicatedServiceDeliveryLocation	
Manufactured Product	RegulatedProduct	
Used Entity	Assigned	

In the example above. 3M Syringe 5ml is an manufactured material of Syringe, while Syringe is an part of BCG Dose, BCG Dose is a part for box of BCG and so on.

8.2.9. SanteDB Protocol Model

The SanteDB protocol model is used to track the types of clinical protocols and links between an encounter and a clinical protocol. Protocols can be expressed in a variety of manners and are adhered to by their protocol handler. The protocol handler identifies which piece of code should be invoked to handle the particular workflow.

Protocol handlers are pieces of code which run in the backbone HDS application and may schedule future events, analyse prior events and execute tasks, etc. Whenever an Act is associated with an Act protocol any addition, modification, or link made to that Act will trigger the execution of the protocol handler.

Examples of protocols can include vaccination schedules, appointment scheduling, adverse event treatment, etc.

The protocol handler and associative entity tables are illustrated in Figure 28 and described in .

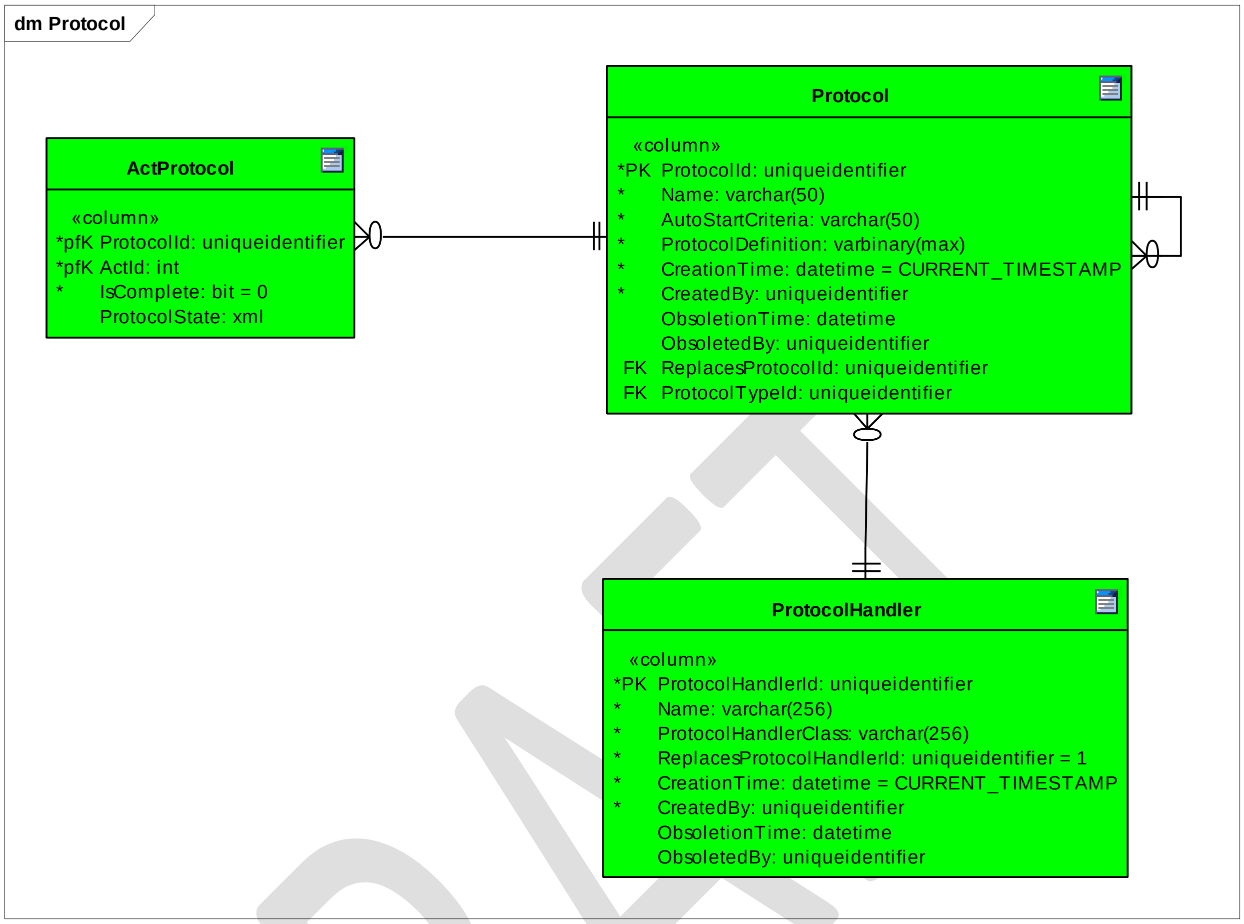

dm Protocol
ActProtocol
«column»
*pfK ProtocolId: uniqueidentifier
*pfK ActId: int
* IsComplete: bit = 0
ProtocolState: xml
Protocol
«column»
*PK ProtocolId: uniqueidentifier
* Name: varchar(50)
* AutoStartCriteria: varchar(50)
* ProtocolDefinition: varbinary(max)
* CreationTime: datetime = CURRENT_TIMESTAMP
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier
FK ReplacesProtocolId: uniqueidentifier
FK ProtocolTypeId: uniqueidentifier
ProtocolHandler
«column»
*PK ProtocolHandlerId: uniqueidentifier
* Name: varchar(256)
* ProtocolHandlerClass: varchar(256)
* ReplacesProtocolHandlerId: uniqueidentifier = 1
* CreationTime: datetime = CURRENT_TIMESTAMP
* CreatedBy: uniqueidentifier
ObsoletionTime: datetime
ObsoletedBy: uniqueidentifier

Figure 28 - SanteDB Protocol Tables

Table	Column	Type	Description
Protocol	(None)	N/A	The protocol table is used to store discrete protocols related to immunization, reporting, etc.
	ProtocolId [1..1]	UUID	Uniquely identifies the protocol within the SanteDB data model.
	Name [1..1]	VARCHAR	A human readable name for the protocol. Example: “BCG Immunization Scheduling”
	ProtocolDefinition [0..1]	VARBINARY	An implementation specific definition of the protocol. This can be BPMN, RulesML, JavaScript, etc. The

			protocol definition must be understood by the associated handler.
	CreationTime [1..1]	DATETIME	Identifies the time when the protocol definition was created.
	CreatedBy [1..1]	UUID	Identifies the user who was responsible for the creation of the protocol definition.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When present, indicates the time when the protocol definition is or was no longer active.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user who was responsible for obsoleting the protocol.
	ReplacesProtocolId [0..1]	UUID	Identifies the protocol which the current tuple replaces.
	ProtocolHandlerId [0..1]	UUID	Identifies the protocol handler which should be used to execute the protocol.
	OID [0..1]	VARCHAR	Provides the globally unique object identifier for the protocol
ProtocolHandler	(None)	N/A	The protocol handler table is used maintain a registration of all handlers which are responsible for the execution of protocols.
	ProtocolHandlerId [1..1]	UUID	Uniquely identifies the protocol handler.
	Name [1..1]	VARCHAR	A human readable name for the protocol handler type.
	ProtocolHandlerClass [1..1]	VARCHAR	The assembly qualified name of the handler which executes the

			protocol. Must implement IProtocolHandler
	CreationTime [1..1]	DATETIME	Identifies the time when the protocol handler was registered.
	CreatedBy [1..1]	UUID	Identifies the user which was responsible for the creation of the protocol handler.
	ObsoletionTime [0..1] ?(>CreationTime)	DATETIME	When present, identifies the time when the protocol handler is no longer active.
	ObsoletedBy [0..1] ?(ObsoletionTime)	UUID	Identifies the user that was responsible for the obsoleting of the protocol handler entry.
	ReplacesProtocolHandlerId [0..1]	UUID	Identifies the protocol handler registration that the current tuple replaces.
ActProtocol	(None)	N/A	The act protocol table is used to associate a protocol definition with an Act. This association identifies the "why" an act was created (i.e. it was associated with an protocol)
	ProtocolId [1..1]	UUID	Identifies the protocol definition to which the association applies.
	ActId [1..1]	UUID	Identifies the act which the protocol is associated with.
	IsComplete [0..1] = False	BIT	Identifies whether the act completed the protocol (terminated it).
	ProtocolState [0..1]	VARBINARY	A handler specific piece of data which can be used to store

			application specific state related to the instance of the protocol definition.

8.3. Physical Data Design

Because SanteDB can leverage a variety of data storage mechanisms via the IDataPersistenceService<T> implementations, all SanteDB plugins and core functions use a business model. IDataPersistenceService implementations are, therefore, responsible for the conversion of queries and data entering/exiting the implementation. This mechanism is illustrated in Figure 29.

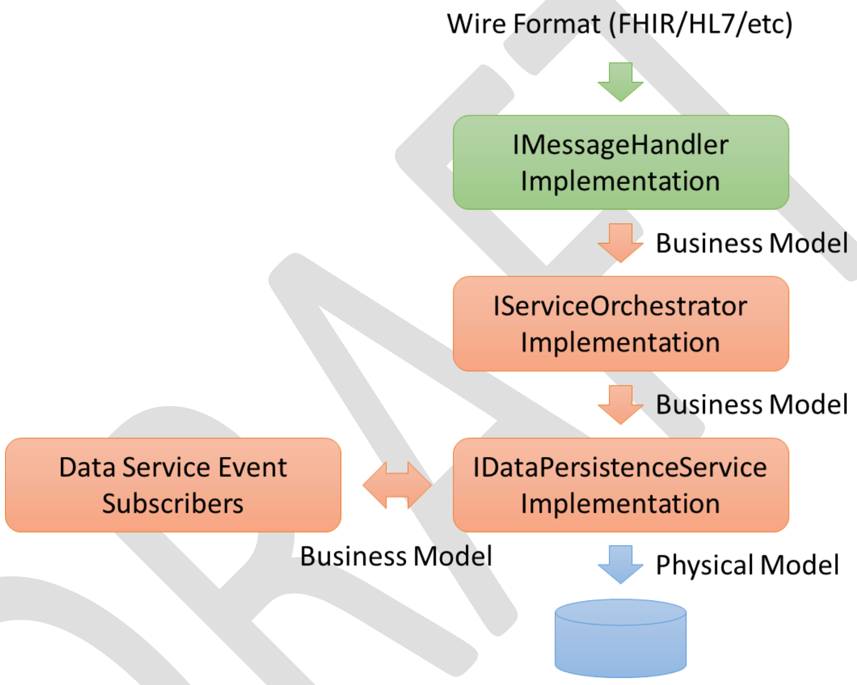

Wire Format (FHIR/HL7/etc)
IMessageHandler Implementation
Business Model
IServiceOrchestrator Implementation
Business Model
Data Service Event Subscribers
IDataPersistenceService Implementation
Business Model
Physical Model

Figure 29 - Business / Physical Model Flow

The SanteDB business model exposes a series of simpler structures than those of the data structures listed in the Data Model section of this document.

8.3.1. ADO Providers

SanteDB HDS includes an implementation of a persistence layer which includes support for generic ADO based databases. This provider uses the ADO.NET wrapper to invoke necessary functions and select data from tables. In order to leverage the ADO.NET persistence service, the ADO.NET provider must support:

- Limit and offset queries
- Invocation of stored procedures and functions
- Multi-threaded access to the database

Currently SanteDB iCDR works with the following OrmLite data providers:

Database System	**Invariant**

PostgreSQL 9.4 – 10.0	npgsql
FirebirdSQL 3.0	fbsql
SQLite 3.3	sqlite

Regardless of the RDBMS system feeding the ADO provider, it must follow the schema specified in this section.

8.3.2. Patching / Changing Data Schema

Whenever a plugin wishes to make changes to the SanteDB database, they must embed assembly manifest resources. These resources contain a specific header that will instruct the ADO data service layer to apply changes to the database.

These changes should register themselves in the database to ensure that duplicate executions of the update are not executed.

Provides a sample database update script

```
/**
 * <update id="PATCHID" applyRange="0.2.0.3-0.9.0.3"  invariantName="npgsql">
 *      <summary>Adds BAD to the type of name uses</summary>
 *      <remarks>Fixes issue with locked accounts</remarks>
 *      <isInstalled>select ck_patch('PATCHID')</isInstalled>
 * </update>
 */

 BEGIN TRANSACTION ;

 -- DO YOUR UPDATES HERE

SELECT REG_PATCH('PATCHID');


COMMIT;
```

The patch schema is defined in .

Element / Path	Cardinality	Description
@id	1..1	Uniquely identifies the update. This should be a globally unique value.
@applyRange	1..1	Identifies the version range (database version not software version) that this patch was designed to work with.
@invariantName	1..1	Identifies the name of the invariant provider this patch works with.
summary	1..1	Provides a summary of what the patch is doing.
remarks	0..1	An optional area where you can specify additional details about the update.
isInstalled	0..1	When specified, overrides the default check mechanism to ensure the patch can be applied.

8.3.3. Data Schema

The physical schema of the ADO provider is designed such that many different ADO providers can support the data structures. Because of the limitations on some RDBMS systems, the ADO schema uses a short-hand for table names. The general guidelines of the shorthand are as follows:

- All objects are suffixed with the database structure they represent.
  - TBL = Table entities
  - VW = Views
  - CDTBL = Code / Lookup Tables
- Names less four characters or less are represented in full (example: NAME or TAG)
- Names which represent a classified object will carry their classification
  - Example: Observation table => OBS_TBL
  - SubstanceAdminsitration table => SBADM_TBL
- Associative entities carry ASSOC in the name
- Other tables use acronyms, and usually have vowels removed
  - Example: Concept Definition table => CD_TBL
  - Code System Table => CS_TBL

Additionally, versions can take on three different forms: inherited, versioned and non-versioned.

- **Inherited:** These types of relationships are directly linked to their base class via a parent version key. This restricts child classes of versioned tables to be linked via version key. The primary key of these types of tables always point to the primary key of the parent table.
- **Non Versioned**: These types of associative table point to their source via a simple key. This source key is usually either to a versioned item (_vrsn_id) or the non-versioned portion of the item (_id). Non versioned associations are considered simple joins as the persistence layer will attempt to join these tables via the primary key relationship.
- **Versioned**: These types of associative tables have a direct foreign key link to the non-versioned table which represents their source. Two additional columns (efft_vrsn_seq and obslt_vrsn_seq) dictate the effective/obsolete versions of that source object when the association was active. These types of tables will always be loaded according to the following logic:
  - **id == a.x_id AND a.efft_vrsn_seq <= versionSequence AND (a.obslt_vrsn_seq IS NULL OR a.obslt_vrsn_seq > versionSequence)**

  The PostgreSQL provider defines an index on entity id, version sequence and obsolete sequence as this allows the query planner to create a better query strategy for the associative load.

  Each of the sections that follow simply translate the physical data model for ADO based RDBMS systems to the logical data model. For a complete description of what each table does, refer to section 8.2.

8.3.3.1. ADO Act Tables

All ADO.NET based implementations of the logical information model implement Act derived tables as illustrated in .

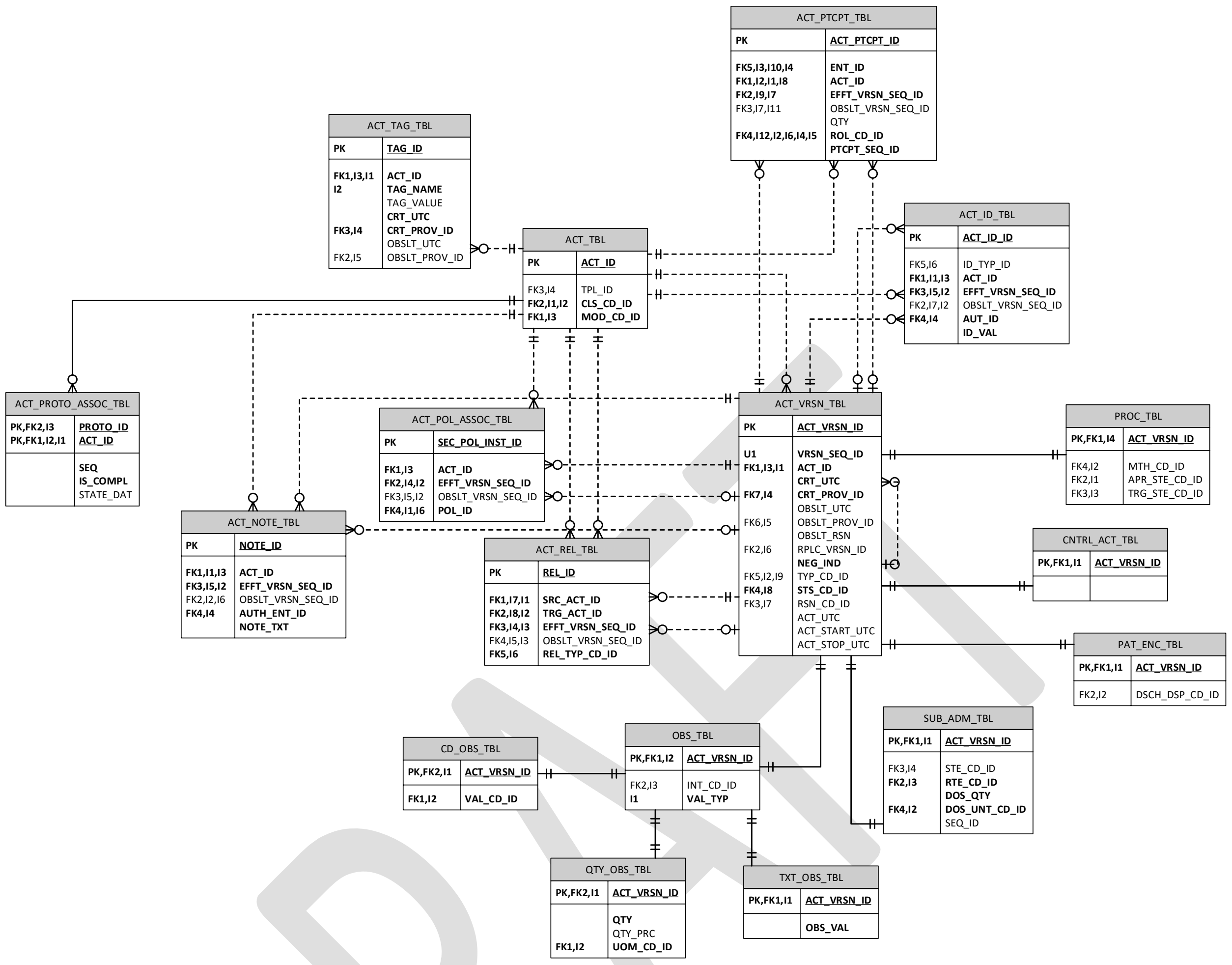


Figure 30 - ADO.NET Act Derived Tables

Table	**Implements**	**Description**
act_tbl	Act	This table is used to store the non-versioned aspects of an act. These are things that are not supposed to change over the lifespan of an act
act_vrsn_tbl	Act (Versions)	This table is used to store the versioned portions of an act. The version sequence column should be a sequence which increments for each tuple (at minimum) or each new version of an object (best option)
act_id_tbl	Act Identifiers	This is an associative table which is used to store identifiers by which an act is referenced externally.
act_ext_tbl	Act Extensions	Extensions table.

act_proto_tbl	Act Protocols	Act protocol implementation table.
act_ptcpt_tbl	Act Participation	Act participation table (links acts to entities)
act_tag	Act Tags	Act tags table
act_ext_tbl	Act Extensions	Act extensions table (value should be a binary array or BLOB)
act_note_tbl	Act Notes	Act notes
act_pol_assoc_tbl	Act Policies	Policies which apply to the act.
act_rel_tbl	Act Relationships	Relationships table. Note that effective/obsolete sequence columns refer to the source act.
sub_adm_tbl	Substance Administrations	Substance administration child table which is linked directly to the act table.
obs_tbl	Observation	Observation child table which is linked to the act table.
qty_obs_tbl	Quantity Observations	Quantity Observation table which is linked with the observation table.
txt_obs_tbl	Text Observations	Text observations table which is linked with the observation table.
cd_obs_tbl	Coded Observations	Coded observations table which is linked to the observation table.
pat_enc_tbl	Patient Encounters	Patient encounters table which is linked to the act table.
proc_tbl	Procedures	Procedures tablet which is linked to the act table.

8.3.3.2. ADO Entity Tables

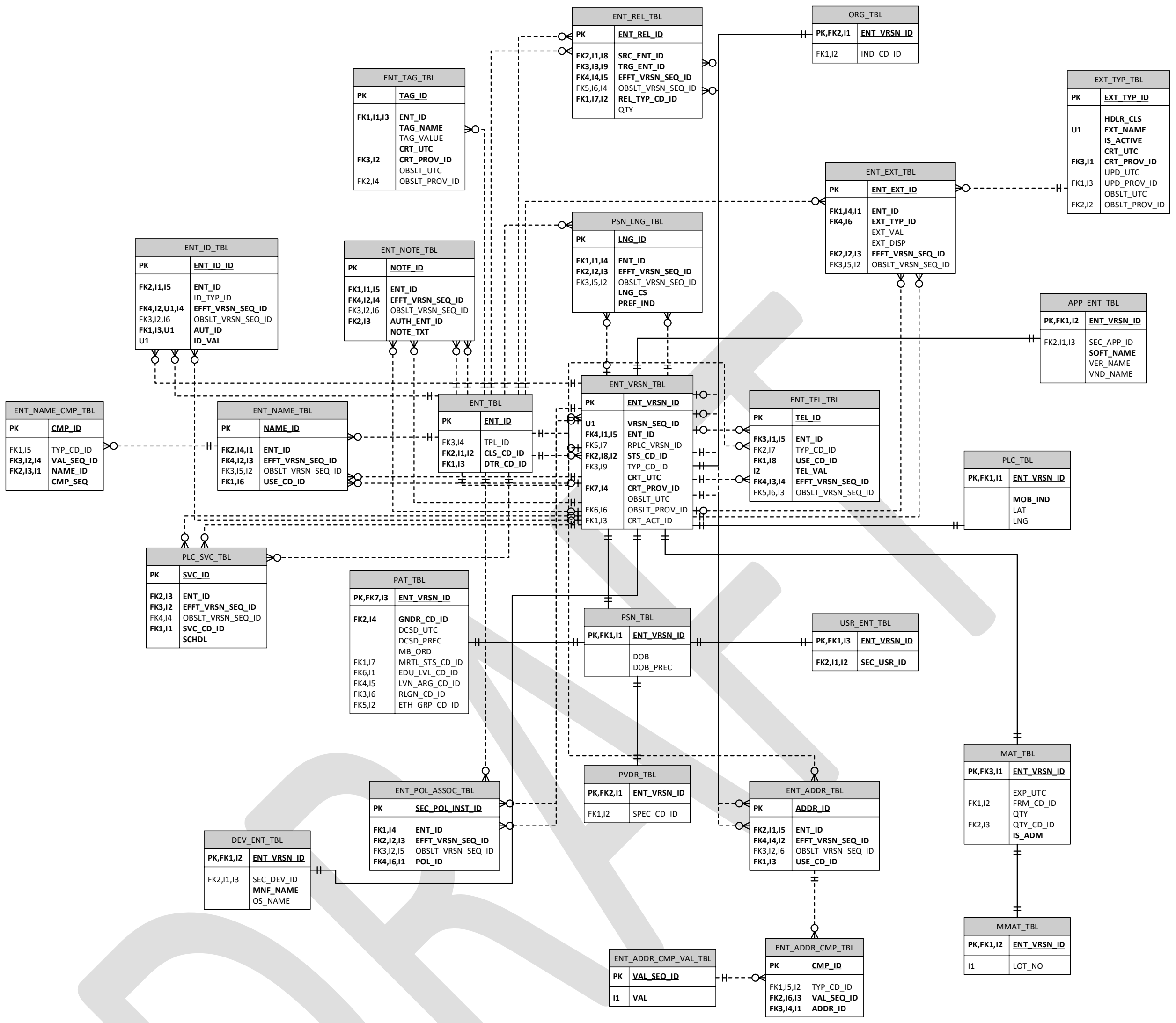


Table	Implements	Description
app_ent_tbl	Application Entity	This table is used to describe the clinical attributes of an application such as vendor, version etc. This differs from the security definition of an application. A security application may exist without a clinical application existing.
dev_ent_tbl	Device Entity	This table is used to describe the clinical attributes of a device. For example an insulin pump. Like an application, a clinical device may not be a security device or vice versa. For example when describing an

		event in which a device was involved , but that device does not need access to the system.
ent_addr_cmp_tbl	Entity Address Component	This table is used to store entity address components. In order to normalize addresses, the values of the addresses themselves are not stored in this table, rather, stored in a unique datatable.
ent_addr_cmp_val_tbl	N/A	This table stores the unique values of all address components found in the SanteDB system.
ent_addr_tbl	Entity Address	Stores information related to an entity's physical, mailing or other postal addresses
ent_ext_tbl	Entity Extension	Stores the data related to entity extensions. Entity extensions are versioned, additional values for a particular entity.
ent_id_tbl	Entity Identifier	The entity identifier table is used to store alternate, external identifiers associated with a particular entity
ent_name_cmp_tbl	Entity Name Component	The entity name component table is used to store the individual components (family, given, prefix, suffix) of an entity's name.
ent_name_tbl	Entity Name	The entity name table is used to store the names associated with a particular entity. For example, a patient may have a married and maiden name.
ent_note_tbl	Entity Notes	The entity note table is used to store freetext notes about a patient.
ent_rel_tbl	Entity Relationship	Entity relationships are relationships between one or more entities with one another. For example, a service delivery location may have one or more employees.
ent_tag_tbl	Entity Tags	Entity tags are non-versioned extensions of an entity. That is their value can change however

		changes are not tracked over time.
ent_tbl	Entity	Represents a non-versioned parts of the core entity such as class, determiner, etc.
ent_tel_tbl	Entity Telecom Addresses	Stores telecommunications addresses related to the entity and their associated type (phone, email, etc) and use (work, home, mobile, etc.)
ent_vrsn_tbl	Entity Versions	This table is used to store the data which can change over time for an entity such as status, additional classifier, etc.
mat_tbl	Material	This table is used to store information related to materials.
mmat_tbl	Manufactured Materials	This table is used to store information related to manufactured materials.
org_tbl	Organization	This table is used to store information related to organizations, which are groupings of people, places for administrative purposes.
pat_tbl	Patient	This table is used to store information related to patients, or those who are receiving care.
plc_svc_tbl	Place Services	This table is used to store information related to the services which occur at a particular place.
plc_tbl	Place	This table is used to store information about a physical place. In Open IZ places can represent delivery locations (hospitals, clinics, etc.) or administrative units (states, cities, etc.)
psn_lng_tbl	Person Languages	This table is used to store information related to the languages which a person (including patients) speak.
psn_tbl	Person	This table is used to store information related to persons (including patients and providers).

pvdr_tbl	Provider	This table is used to store information related to providers (persons who deliver care).
usr_ent_tbl	User Entity	This table is used to link the security subsystem for users to the clinical subsystem of entities (as entities are the ones who participate in Acts rather than Security Users).

8.3.3.3. ADO Security Tables

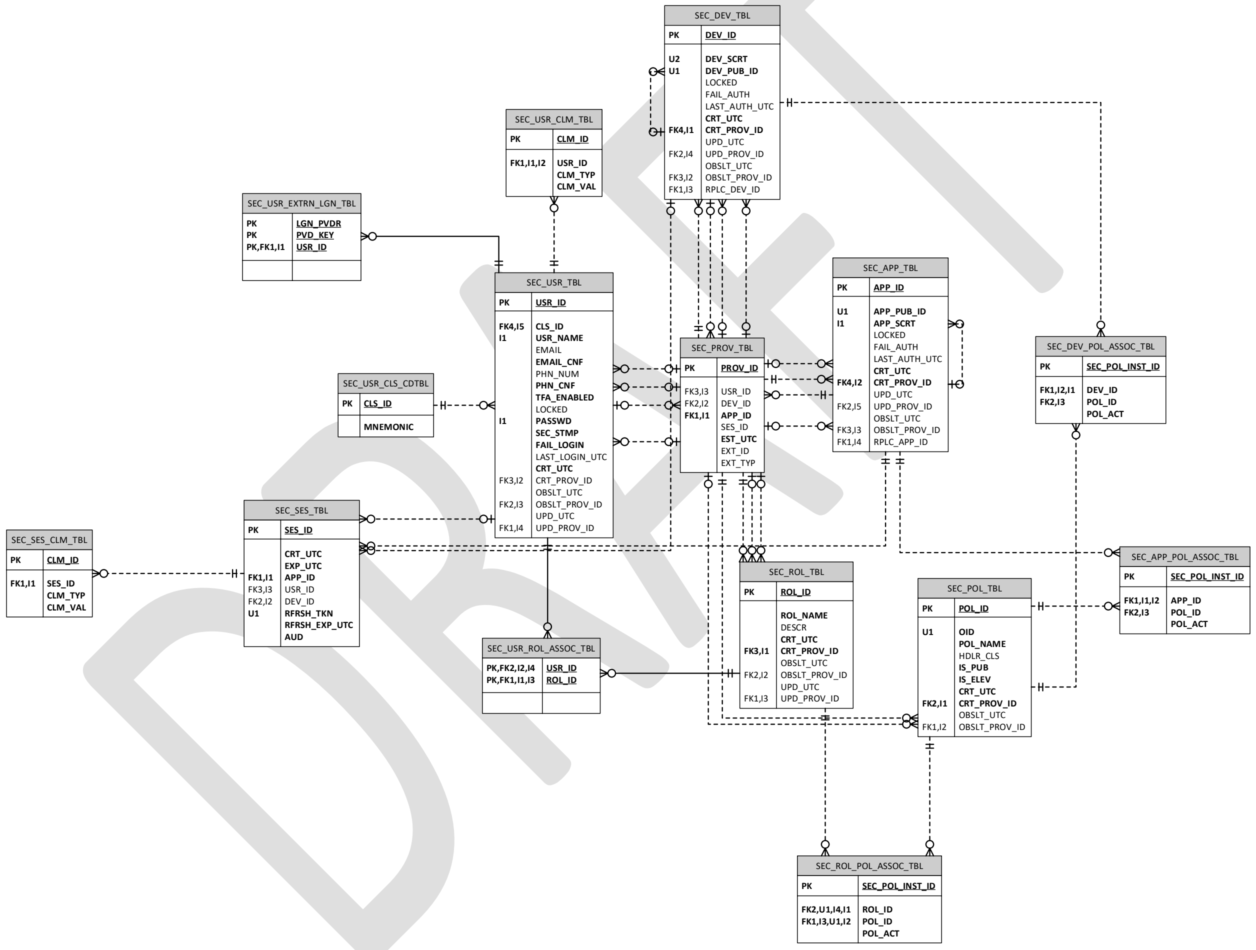


Table	Implements	Description
sec_usr_tbl	Security User	Stores security information related to users such as login, timestamps, etc..
sec_app_tbl	Security Application	Security information related to applications such as client identifiers, and application secrets.

sec_dev_tbl	Security Device	Security information related to devices such as device secrets and claims related to devices.
sec_rol_tbl	Security Role	Security information related to roles which are used as the basis for access control.
sec_pol_tbl	Security Policy	Security information related to policies such as action and data policies used by the ADO.NET security policy information point.
sec_app_pol_assoc_tbl	Security Application <> Policy	Associates security applications to policies to which the application has granted or explicit deny access.
sec_dev_pol_assoc_tbl	Security Device <> Policy	Associates security device to policies to which the device has granted or explicit deny.
sec_usr_clm_tbl	Security User Claims	Associates security users to additional (external) claims made about that user which form the basis of claims when a new session is created for that user.
sec_usr_extrn_lgn_tbl	Security User External Login	Associates security users to the external login provider/information. This allows for mapping of user credentials to external identity providers.
sec_usr_rol_assoc_tbl	Security User Role Association	Associates security users to the roles to which they are a member.
sec_rol_pol_assoc_tbl	Security Role Policy Association	Associates security roles to the policies which members of thayt role are granted or denied access. This forms the basis of an access control list for that user whenever a session is created.
sec_usr_cls_tbl	Security User Classes	Classifies the type of security user as either a human user, a system user or a device or application user.
sec_ses_tbl	Security Session	A complete list of user sessions which have been recorded. Note that this table is only

		populated when and if the ADO.NET session provider is registered in the application configuration.
sec_ses_clm_tbl	Security Session Claims	Stores the session specific claims for a particular session. This includes the purpose of use of the session, any elevations granted to the user and (if applicable) the issuer of the session.
sec_prov_tbl	Security Provenance	Stores information about the provenance of data. A provenance object is created whenever data is updated, created, or obsoleted. One provenance object is created per transaction and batch.

8.3.3.4. ADO Concept Tables

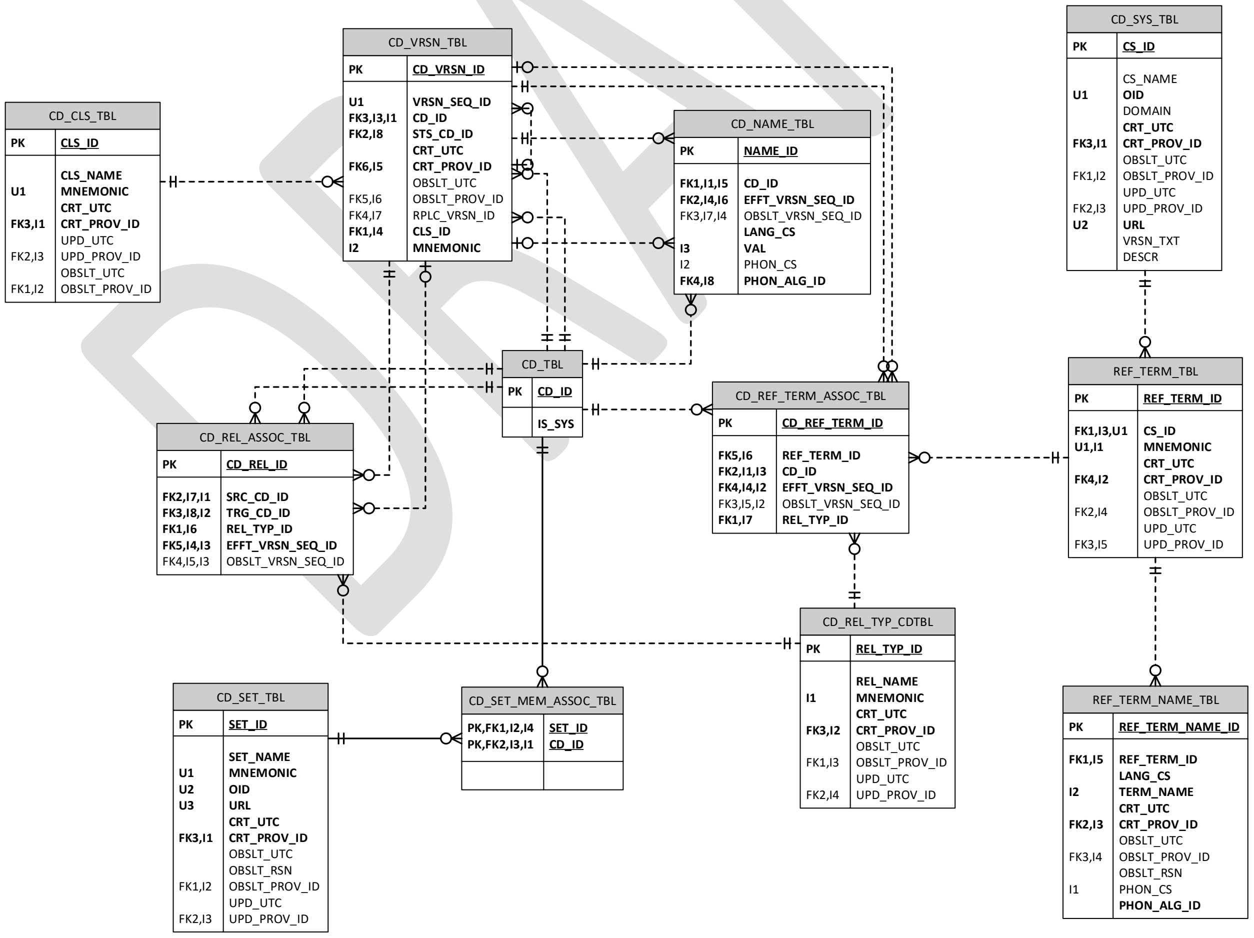


Table	Implements	Description
cd_tbl	Concept	Stores the master list and non-versioned properties of the SanteDB concept dictionary.
cd_vrsn_tbl	Concept Version	Stores the versioned attributes of the SanteDB concept dictionary.
cd_cls_tbl	Concept Classifier	Classifications for the concepts stored within the SanteDB concept dictionary.
cd_name_tbl	Concept Name	Canonical/Internal names for each of the concepts stored in the SanteDB concept dictionary. These are localized names for the concept.
cd_rel_assoc_tbl	Concept Relationship	Associations between concepts which dictate relationships between concepts such as SAME AS, NARROWER THAN
cd_rel_typ_cdtbl	Concept Relationship Type	Identifies the types of relationships that one or more concepts may have between each other.
cd_sys_tbl	Code System	Standardized reference terminologies (code systems) such as LOINC, SNOMED, etc.
cd_ref_term_assoc_tbl	Code Reference Term Association	Associates concepts to reference terms with a strength.
cd_set_tbl	Concept Set	A logical grouping of concepts to a particular use/set.
cd_set_mem_tbl	Concept Set Members	Association between the member concepts of a concept set and the set itself.
ref_term_tbl	Reference Terms	Stores the reference terms (standardized or wire level codes).
ref_term_name_tbl	Reference Term Name	Standardized display names fo the reference terms.

8.4. Business Data Model

The business data model represents a series of classes (rather than database entities) which are used by all application code within the SanteDB HDS. The business data model is also exposed via the HDSI (Immunization Management Service Interface).

Like the logical data model, the business data model is loosely based on the HL7 RIM. The major difference is that the business data model uses an object hierarchy rather than using relationships. The IDataPersistenceService implementations manage the translation of this paradigm.

8.4.1. Business Data Model Queries

Queries on against the business model are exposed via IDataPersistenceService implementations' Query() method. The Query() method in .NET accepts an expression tree parameter (lambda predicate) which is translated via the ModelMapper class.

In order to map the business model classes to LINQ to SQL classes, it is recommended that third party storage plugins use the ModelMapper class and provide an appropriate mapping file.

In order to execute a query, a developer should use the following pattern:

```
var IDataPersistenceService<SecurityUser> persistenceService =
ApplicationContext.Current.GetService<IDataPersistenceService<SecurityUser>>();
var queryResults = persistenceService.Query(u => u.Email.EndsWith("test.com") && u.Roles.Any(r => r.Name
== "Administrators"), null);
```

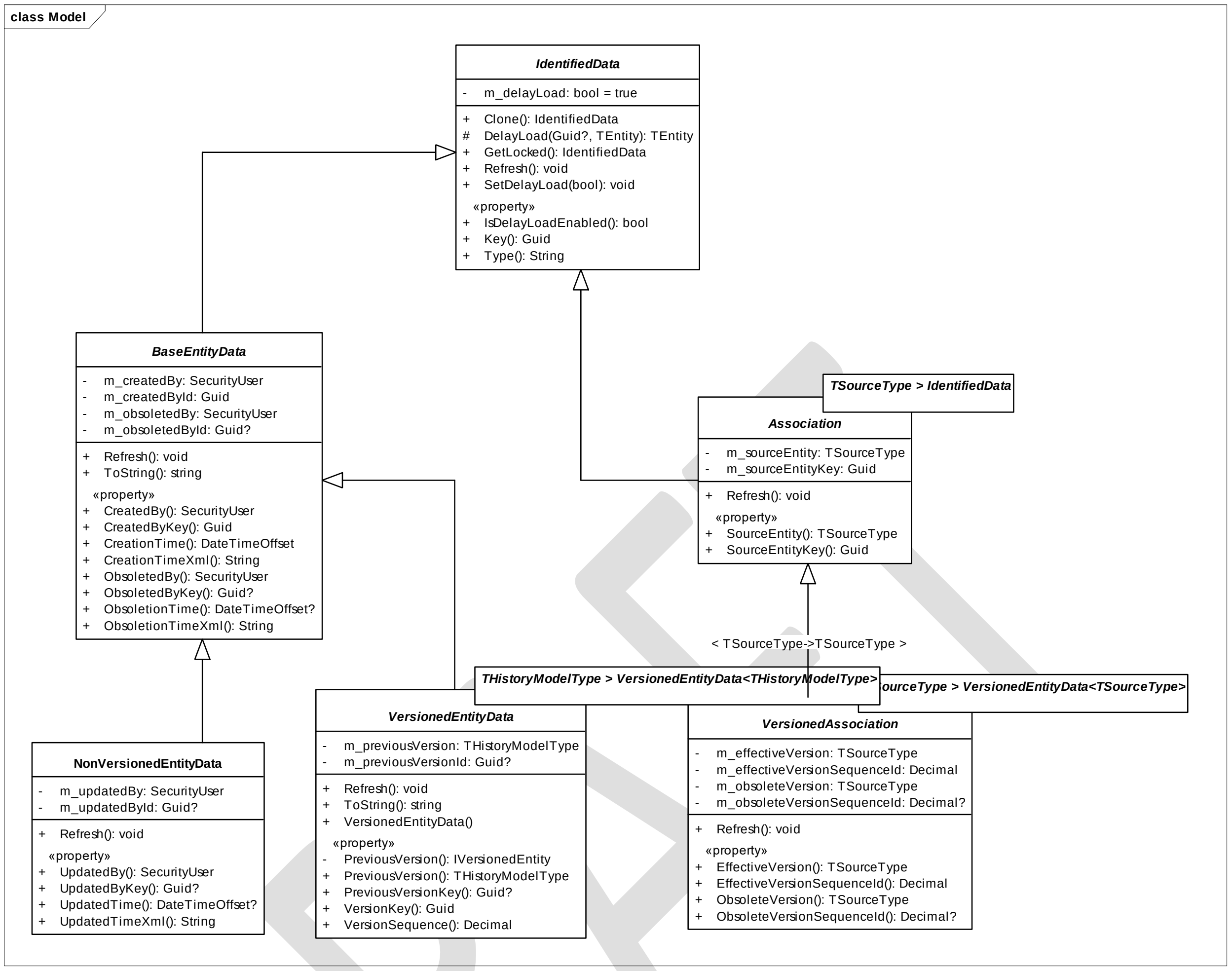
Furthermore, additional LINQ expressions such as Count(), Any(), All(), Take(), Skip() can be used to control execution against the datamodel. It is important to note that LINQ expressions are translated by the persistence layer directly into SQL, therefore advanced operations performed against objects cannot be used in these expressions.

By default, most data persistence implementations (ADO and MSSQL) will support the following methods.

Method	**Equivalent SQL**	**Description / Example**
.Where(***x***)	WITH cte SELECT ***x***	Indicates the provider should perform a sub-query. This is used for guard conditions
.Any(***x***)	EXISTS (SELECT *x*)	Indicates that the provider should count the existence of a sub-element as a condition.
x.EndsWith(***y***)	***x*** LIKE '%' || ***y***	Evaluates to true when a field ends with the specified value.
x.StartsWith(***y***)	***x*** LIKE ***y*** || '%'	Evaluates to true when a field starts with the specified value.
x.Contains(***y***)	***x*** LIKE '%' || ***y*** || '%'	Evaluates to true when the parameter appears anywhere in the field.
x.ToUpper()	UPPER(***x***)	Instructs that a field should be converted to upper case.
x.ToLower()	LOWER(***x***)	Instructs that a field should be converted to lower case.
x.HasValue	***x*** IS NOT NULL	When used on a nullable field, evaluates to true when the nullable field has a value.

8.4.2. Foundational Classes

There are several foundational classes which are used in the business data model. Class associations in the business model are delay loaded meaning that they are loaded upon first access. For example, when loading a *SecurityUser* the *Groups* property is only loaded via the data persistence layer when accessed by a consumer application. This reduces hits to the underlying database.

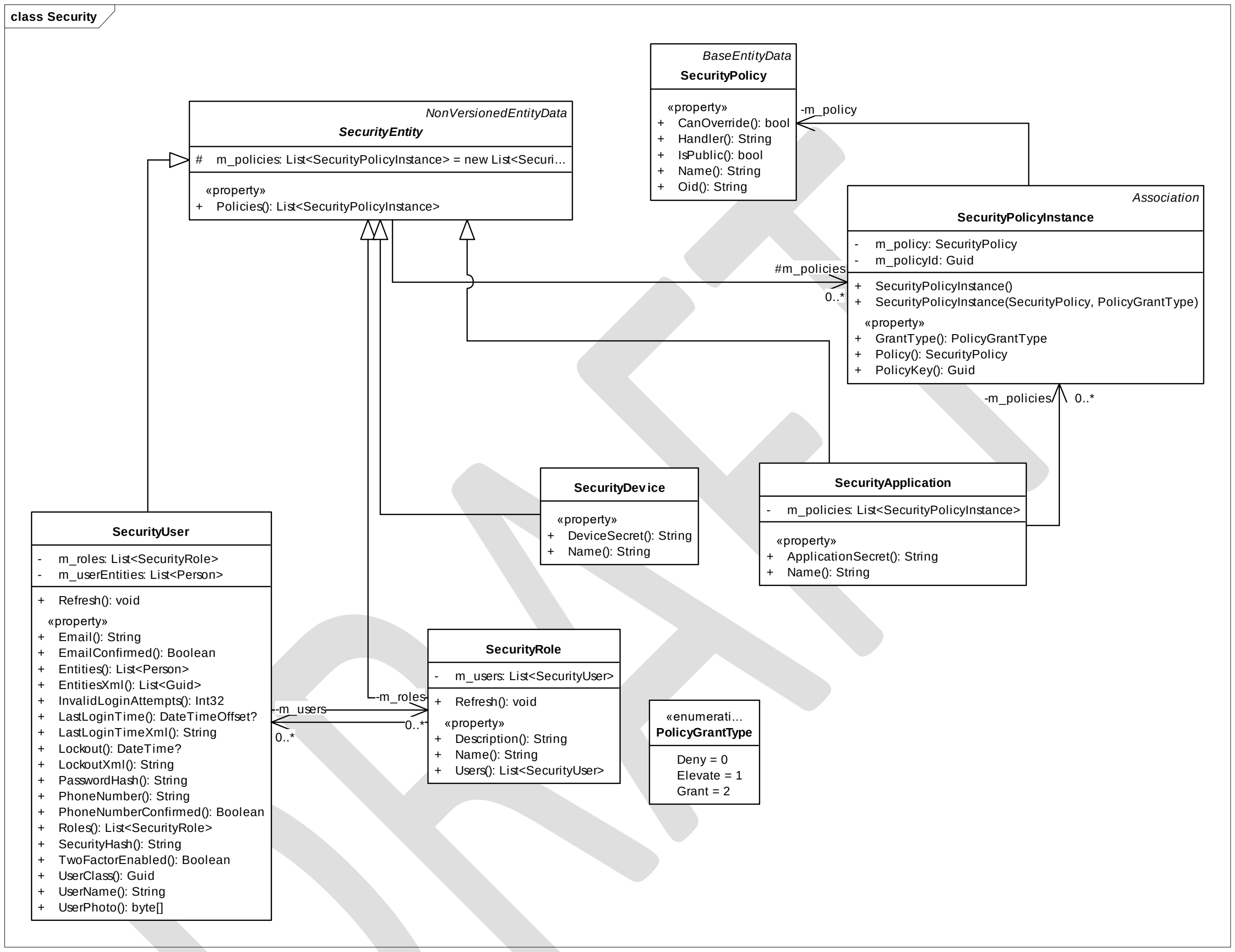

class Model
IdentifiedData
- m_delayLoad: bool = true
+ Clone(): IdentifiedData
# DelayLoad(Guid?, TEntity): TEntity
+ GetLocked(): IdentifiedData
+ Refresh(): void
+ SetDelayLoad(bool): void
«property»
+ IsDelayLoadEnabled(): bool
+ Key(): Guid
+ Type(): String
BaseEntityData
- m_createdBy: SecurityUser
- m_createdById: Guid
- m_obsoletedBy: SecurityUser
- m_obsoletedById: Guid?
+ Refresh(): void
+ ToString(): string
«property»
+ CreatedBy(): SecurityUser
+ CreatedByKey(): Guid
+ CreationTime(): DateTimeOffset
+ CreationTimeXml(): String
+ ObsoletedBy(): SecurityUser
+ ObsoletedByKey(): Guid?
+ ObsoletionTime(): DateTimeOffset?
+ ObsoletionTimeXml(): String
TSourceType > IdentifiedData
Association
- m_sourceEntity: TSourceType
- m_sourceEntityKey: Guid
+ Refresh(): void
«property»
+ SourceEntity(): TSourceType
+ SourceEntityKey(): Guid
< TSourceType->TSourceType >
THistoryModelType > VersionedEntityData<THistoryModelType>
ourceType > VersionedEntityData<TSourceType>
NonVersionedEntityData
- m_updatedBy: SecurityUser
- m_updatedById: Guid?
+ Refresh(): void
«property»
+ UpdatedBy(): SecurityUser
+ UpdatedByKey(): Guid?
+ UpdatedTime(): DateTimeOffset?
+ UpdatedTimeXml(): String
VersionedEntityData
- m_previousVersion: THistoryModelType
- m_previousVersionId: Guid?
+ Refresh(): void
+ ToString(): string
+ VersionedEntityData()
«property»
- PreviousVersion(): IVersionedEntity
+ PreviousVersion(): THistoryModelType
+ PreviousVersionKey(): Guid?
+ VersionKey(): Guid
+ VersionSequence(): Decimal
VersionedAssociation
- m_effectiveVersion: TSourceType
- m_effectiveVersionSequenceId: Decimal
- m_obsoleteVersion: TSourceType
- m_obsoleteVersionSequenceId: Decimal?
+ Refresh(): void
«property»
+ EffectiveVersion(): TSourceType
+ EffectiveVersionSequenceId(): Decimal
+ ObsoleteVersion(): TSourceType
+ ObsoleteVersionSequenceId(): Decimal?

The foundational classes are listed in more detail in .

Class	Purpose
IdentitifedData (abstract)	Encapsulates all business model classes which have a primary key. Controls the delay loading behavior of implementing classes.
BaseEntityData (abstract)	Encapsulates entity data which is audited with a creation/obsoletion time and user.
VersionedEntityData<THistoryModelType> (abstract)	All entities which are versioned are derived from this class. The class contains a property of type THistoryModelType which is a pointer to the previous version of the entity instance.
VersionBoundRelationData<TTargetType> (abstract)	All entities which are associated with a version of an entity (those associated with the VersionedEntityData instance) will derive from this class. It contains information such as effective and obsolete version sequence numbers.

8.4.3. Security Classes

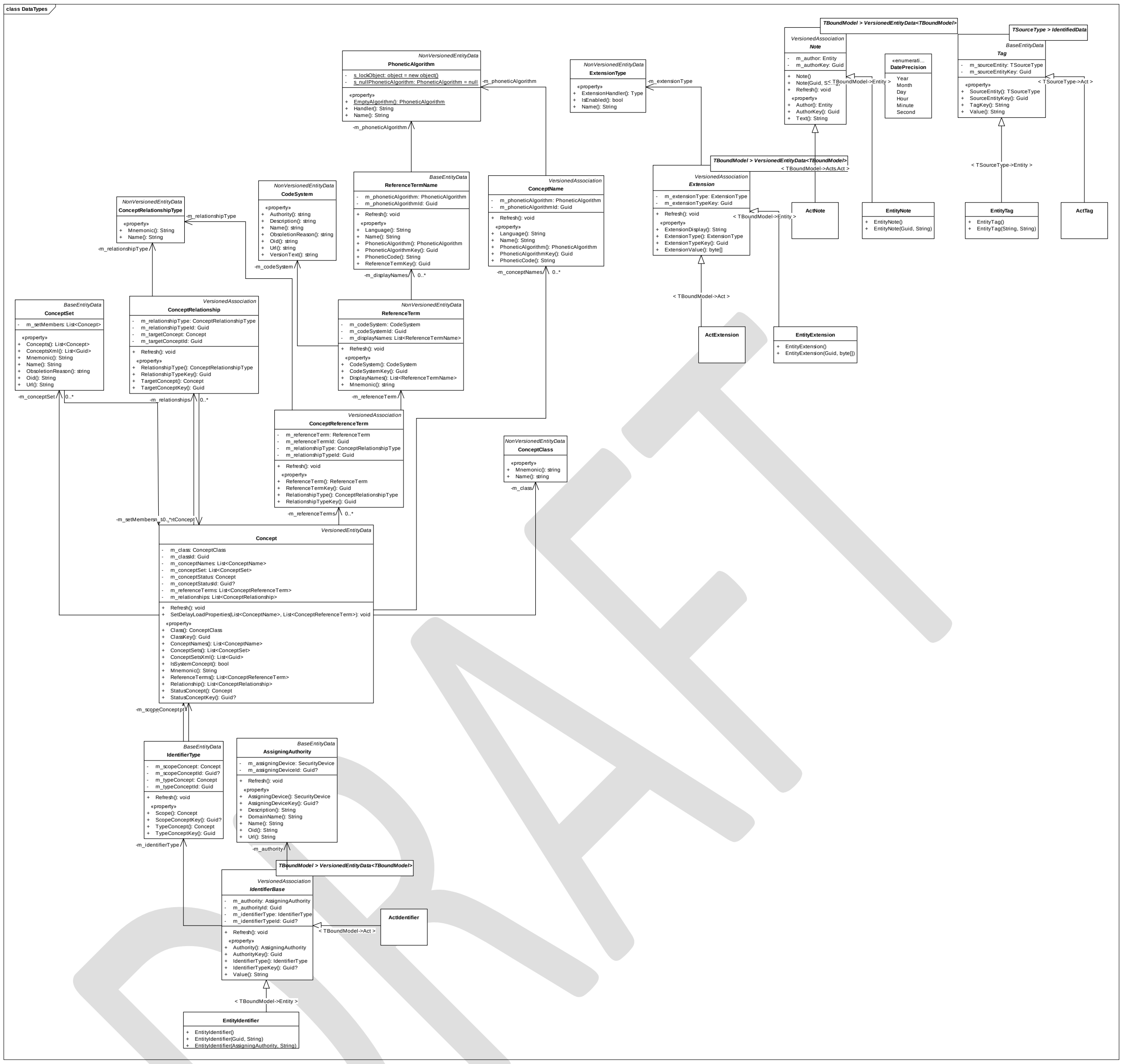
The security business model data classes are not necessarily intended to be used directly by consumers, rather they serve as a series of classes used by the security infrastructure in SanteDB such as the IPolicyInformationService, IPolicyDecisionService, IIdentityProviderService and IRoleProviderService implementations.



The *SecurityUser* class is used to create *IIdentity* and *IPrincipal* during the course of an authentication request. *SecurityDevice* and *SecurityApplication* are used to populate claims data which indicates the device and application the user is using.

8.4.4. Data Types

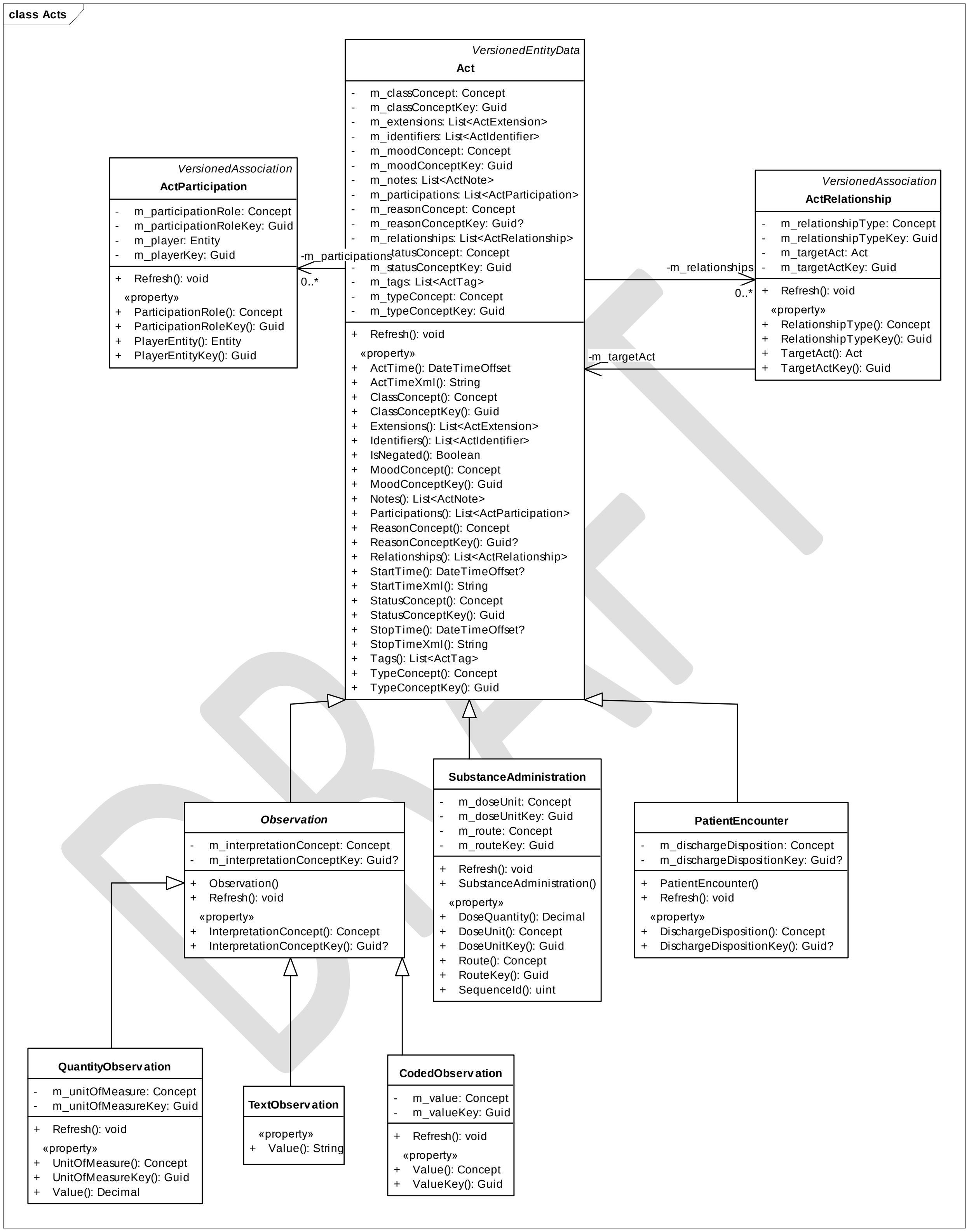
The data type business model classes are used to store reusable, common data elements. These include: The concept system, Assigning Authority System, Identifiers, etc.



class DataTypes
NonVersionedEntityData
PhoneticAlgorithm
- s_lockObject: object = new object()
- s_nullPhoneticAlgorithm: PhoneticAlgorithm = null
«property»
+ EmptyAlgorithm(): PhoneticAlgorithm
+ Handler(): String
+ Name(): String
-m_phoneticAlgorithm
-m_phoneticAlgorithm
NonVersionedEntityData
ExtensionType
«property»
+ ExtensionHandler(): Type
+ IsEnabled(): bool
+ Name(): String
-m_extensionType
TBoundModel > VersionedEntityData<TBoundModel>
VersionedAssociation
Note
- m_author: Entity
- m_authorKey: Guid
+ Note()
+ Note(Guid, String)
+ Refresh(): void
«property»
+ Author(): Entity
+ AuthorKey(): Guid
+ Text(): String
< TBoundModel->Entity >
«enumeration»
DatePrecision
Year
Month
Day
Hour
Minute
Second
TSourceType > IdentifiedData
BaseEntityData
Tag
- m_sourceEntity: TSourceType
- m_sourceEntityKey: Guid
«property»
+ SourceEntity(): TSourceType
+ SourceEntityKey(): Guid
+ TagKey(): String
+ Value(): String
< TSourceType->Act >
< TSourceType->Entity >
TBoundModel > VersionedEntityData<TBoundModel>
< TBoundModel->Acts.Act >
VersionedAssociation
Extension
- m_extensionType: ExtensionType
- m_extensionTypeKey: Guid
+ Refresh(): void
«property»
+ ExtensionDisplay(): String
+ ExtensionType(): ExtensionType
+ ExtensionTypeKey(): Guid
+ ExtensionValue(): byte[]
< TBoundModel->Entity >
ActNote
EntityNote
+ EntityNote()
+ EntityNote(Guid, String)
EntityTag
+ EntityTag()
+ EntityTag(String, String)
ActTag
< TBoundModel->Act >
ActExtension
EntityExtension
+ EntityExtension()
+ EntityExtension(Guid, byte[])
NonVersionedEntityData
ConceptRelationshipType
«property»
+ Mnemonic(): String
+ Name(): String
-m_relationshipType
-m_relationshipType
NonVersionedEntityData
CodeSystem
«property»
+ Authority(): string
+ Description(): string
+ Name(): string
+ ObsoletionReason(): string
+ Oid(): string
+ Url(): string
+ VersionText(): string
-m_codeSystem
BaseEntityData
ReferenceTermName
- m_phoneticAlgorithm: PhoneticAlgorithm
- m_phoneticAlgorithmId: Guid
+ Refresh(): void
«property»
+ Language(): String
+ Name(): String
+ PhoneticAlgorithm(): PhoneticAlgorithm
+ PhoneticAlgorithmKey(): Guid
+ PhoneticCode(): String
+ ReferenceTermKey(): Guid
-m_displayNames
0..*
VersionedAssociation
ConceptName
- m_phoneticAlgorithm: PhoneticAlgorithm
- m_phoneticAlgorithmId: Guid
+ Refresh(): void
«property»
+ Language(): String
+ Name(): String
+ PhoneticAlgorithm(): PhoneticAlgorithm
+ PhoneticAlgorithmKey(): Guid
+ PhoneticCode(): String
-m_conceptNames
0..*
BaseEntityData
ConceptSet
- m_setMembers: List<Concept>
«property»
+ Concepts(): List<Concept>
+ ConceptsXml(): List<Guid>
+ Mnemonic(): String
+ Name(): String
+ ObsoletionReason(): string
+ Oid(): String
+ Url(): String
-m_conceptSet
0..*
VersionedAssociation
ConceptRelationship
- m_relationshipType: ConceptRelationshipType
- m_relationshipTypeId: Guid
- m_targetConcept: Concept
- m_targetConceptId: Guid
+ Refresh(): void
«property»
+ RelationshipType(): ConceptRelationshipType
+ RelationshipTypeKey(): Guid
+ TargetConcept(): Concept
+ TargetConceptKey(): Guid
-m_relationships
0..*
NonVersionedEntityData
ReferenceTerm
- m_codeSystem: CodeSystem
- m_codeSystemId: Guid
- m_displayNames: List<ReferenceTermName>
+ Refresh(): void
«property»
+ CodeSystem(): CodeSystem
+ CodeSystemKey(): Guid
+ DisplayNames(): List<ReferenceTermName>
+ Mnemonic(): string
-m_referenceTerm
VersionedAssociation
ConceptReferenceTerm
- m_referenceTerm: ReferenceTerm
- m_referenceTermId: Guid
- m_relationshipType: ConceptRelationshipType
- m_relationshipTypeId: Guid
+ Refresh(): void
«property»
+ ReferenceTerm(): ReferenceTerm
+ ReferenceTermKey(): Guid
+ RelationshipType(): ConceptRelationshipType
+ RelationshipTypeKey(): Guid
-m_referenceTerms
0..*
NonVersionedEntityData
ConceptClass
«property»
+ Mnemonic(): string
+ Name(): string
-m_class
-m_setMembersm_t0..*etConcept
VersionedEntityData
Concept
- m_class: ConceptClass
- m_classId: Guid
- m_conceptNames: List<ConceptName>
- m_conceptSet: List<ConceptSet>
- m_conceptStatus: Concept
- m_conceptStatusId: Guid?
- m_referenceTerms: List<ConceptReferenceTerm>
- m_relationships: List<ConceptRelationship>
+ Refresh(): void
+ SetDelayLoadProperties(List<ConceptName>, List<ConceptReferenceTerm>): void
«property»
+ Class(): ConceptClass
+ ClassKey(): Guid
+ ConceptNames(): List<ConceptName>
+ ConceptSets(): List<ConceptSet>
+ ConceptSetsXml(): List<Guid>
+ IsSystemConcept(): bool
+ Mnemonic(): String
+ ReferenceTerms(): List<ConceptReferenceTerm>
+ Relationship(): List<ConceptRelationship>
+ StatusConcept(): Concept
+ StatusConceptKey(): Guid?
-m_scopeConceptpt
BaseEntityData
IdentifierType
- m_scopeConcept: Concept
- m_scopeConceptId: Guid?
- m_typeConcept: Concept
- m_typeConceptId: Guid
+ Refresh(): void
«property»
+ Scope(): Concept
+ ScopeConceptKey(): Guid?
+ TypeConcept(): Concept
+ TypeConceptKey(): Guid
-m_identifierType
BaseEntityData
AssigningAuthority
- m_assigningDevice: SecurityDevice
- m_assigningDeviceId: Guid?
+ Refresh(): void
«property»
+ AssigningDevice(): SecurityDevice
+ AssigningDeviceKey(): Guid?
+ Description(): String
+ DomainName(): String
+ Name(): String
+ Oid(): String
+ Url(): String
-m_authority
TBoundModel > VersionedEntityData<TBoundModel>
VersionedAssociation
IdentifierBase
- m_authority: AssigningAuthority
- m_authorityId: Guid
- m_identifierType: IdentifierType
- m_identifierTypeId: Guid?
+ Refresh(): void
«property»
+ Authority(): AssigningAuthority
+ AuthorityKey(): Guid
+ IdentifierType(): IdentifierType
+ IdentifierTypeKey(): Guid?
+ Value(): String
< TBoundModel->Act >
ActIdentifier
< TBoundModel->Entity >
EntityIdentifier
+ EntityIdentifier()
+ EntityIdentifier(Guid, String)
+ EntityIdentifier(AssigningAuthority, String)

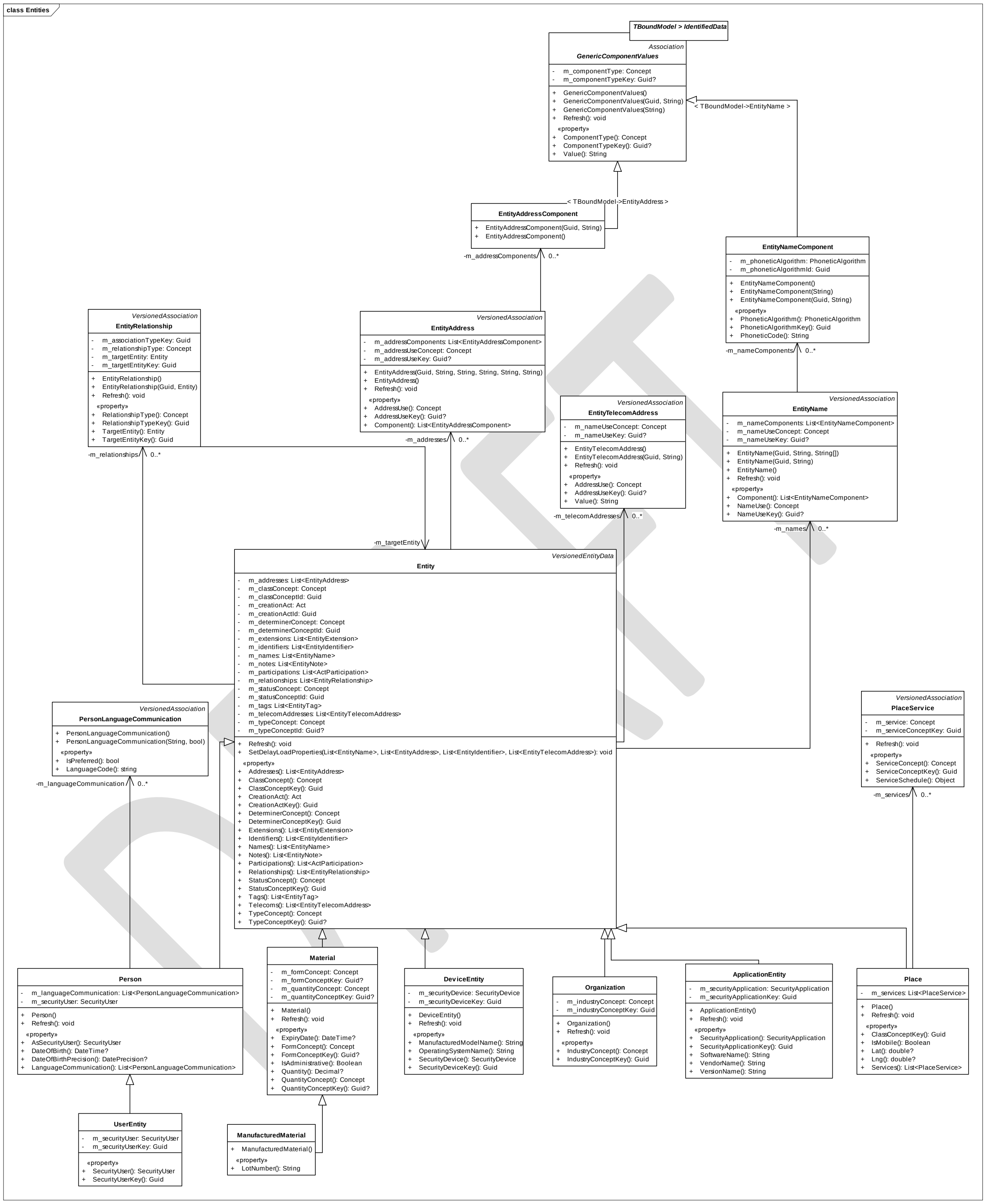
8.4.5. Acts

TODO:


class Acts
VersionedEntityData
Act
- m_classConcept: Concept
- m_classConceptKey: Guid
- m_extensions: List<ActExtension>
- m_identifiers: List<ActIdentifier>
- m_moodConcept: Concept
- m_moodConceptKey: Guid
- m_notes: List<ActNote>
- m_participations: List<ActParticipation>
- m_reasonConcept: Concept
- m_reasonConceptKey: Guid?
- m_relationships: List<ActRelationship>
- m_statusConcept: Concept
- m_statusConceptKey: Guid
- m_tags: List<ActTag>
- m_typeConcept: Concept
- m_typeConceptKey: Guid
+ Refresh(): void
«property»
+ ActTime(): DateTimeOffset
+ ActTimeXml(): String
+ ClassConcept(): Concept
+ ClassConceptKey(): Guid
+ Extensions(): List<ActExtension>
+ Identifiers(): List<ActIdentifier>
+ IsNegated(): Boolean
+ MoodConcept(): Concept
+ MoodConceptKey(): Guid
+ Notes(): List<ActNote>
+ Participations(): List<ActParticipation>
+ ReasonConcept(): Concept
+ ReasonConceptKey(): Guid?
+ Relationships(): List<ActRelationship>
+ StartTime(): DateTimeOffset?
+ StartTimeXml(): String
+ StatusConcept(): Concept
+ StatusConceptKey(): Guid
+ StopTime(): DateTimeOffset?
+ StopTimeXml(): String
+ Tags(): List<ActTag>
+ TypeConcept(): Concept
+ TypeConceptKey(): Guid
VersionedAssociation
ActParticipation
- m_participationRole: Concept
- m_participationRoleKey: Guid
- m_player: Entity
- m_playerKey: Guid
+ Refresh(): void
«property»
+ ParticipationRole(): Concept
+ ParticipationRoleKey(): Guid
+ PlayerEntity(): Entity
+ PlayerEntityKey(): Guid
-m_participations
0..*
VersionedAssociation
ActRelationship
- m_relationshipType: Concept
- m_relationshipTypeKey: Guid
- m_targetAct: Act
- m_targetActKey: Guid
+ Refresh(): void
«property»
+ RelationshipType(): Concept
+ RelationshipTypeKey(): Guid
+ TargetAct(): Act
+ TargetActKey(): Guid
-m_relationships
0..*
-m_targetAct
Observation
- m_interpretationConcept: Concept
- m_interpretationConceptKey: Guid?
+ Observation()
+ Refresh(): void
«property»
+ InterpretationConcept(): Concept
+ InterpretationConceptKey(): Guid?
SubstanceAdministration
- m_doseUnit: Concept
- m_doseUnitKey: Guid
- m_route: Concept
- m_routeKey: Guid
+ Refresh(): void
+ SubstanceAdministration()
«property»
+ DoseQuantity(): Decimal
+ DoseUnit(): Concept
+ DoseUnitKey(): Guid
+ Route(): Concept
+ RouteKey(): Guid
+ SequenceId(): uint
PatientEncounter
- m_dischargeDisposition: Concept
- m_dischargeDispositionKey: Guid?
+ PatientEncounter()
+ Refresh(): void
«property»
+ DischargeDisposition(): Concept
+ DischargeDispositionKey(): Guid?
QuantityObservation
- m_unitOfMeasure: Concept
- m_unitOfMeasureKey: Guid
+ Refresh(): void
«property»
+ UnitOfMeasure(): Concept
+ UnitOfMeasureKey(): Guid
+ Value(): Decimal
TextObservation
«property»
+ Value(): String
CodedObservation
- m_value: Concept
- m_valueKey: Guid
+ Refresh(): void
«property»
+ Value(): Concept
+ ValueKey(): Guid

8.4.6. Entities

TODO


class Entities
TBoundModel > IdentifiedData
Association
GenericComponentValues
- m_componentType: Concept
- m_componentTypeKey: Guid?
+ GenericComponentValues()
+ GenericComponentValues(Guid, String)
+ GenericComponentValues(String)
+ Refresh(): void
«property»
+ ComponentType(): Concept
+ ComponentTypeKey(): Guid?
+ Value(): String
< TBoundModel->EntityName >
< TBoundModel->EntityAddress >
EntityAddressComponent
+ EntityAddressComponent(Guid, String)
+ EntityAddressComponent()
-m_addressComponents 0..*
EntityNameComponent
- m_phoneticAlgorithm: PhoneticAlgorithm
- m_phoneticAlgorithmId: Guid
+ EntityNameComponent()
+ EntityNameComponent(String)
+ EntityNameComponent(Guid, String)
«property»
+ PhoneticAlgorithm(): PhoneticAlgorithm
+ PhoneticAlgorithmKey(): Guid
+ PhoneticCode(): String
-m_nameComponents 0..*
VersionedAssociation
EntityRelationship
- m_associationTypeKey: Guid
- m_relationshipType: Concept
- m_targetEntity: Entity
- m_targetEntityKey: Guid
+ EntityRelationship()
+ EntityRelationship(Guid, Entity)
+ Refresh(): void
«property»
+ RelationshipType(): Concept
+ RelationshipTypeKey(): Guid
+ TargetEntity(): Entity
+ TargetEntityKey(): Guid
-m_relationships 0..*
VersionedAssociation
EntityAddress
- m_addressComponents: List<EntityAddressComponent>
- m_addressUseConcept: Concept
- m_addressUseKey: Guid?
+ EntityAddress(Guid, String, String, String, String, String)
+ EntityAddress()
+ Refresh(): void
«property»
+ AddressUse(): Concept
+ AddressUseKey(): Guid?
+ Component(): List<EntityAddressComponent>
-m_addresses 0..*
VersionedAssociation
EntityTelecomAddress
- m_nameUseConcept: Concept
- m_nameUseKey: Guid?
+ EntityTelecomAddress()
+ EntityTelecomAddress(Guid, String)
+ Refresh(): void
«property»
+ AddressUse(): Concept
+ AddressUseKey(): Guid?
+ Value(): String
-m_telecomAddresses 0..*
VersionedAssociation
EntityName
- m_nameComponents: List<EntityNameComponent>
- m_nameUseConcept: Concept
- m_nameUseKey: Guid?
+ EntityName(Guid, String, String[])
+ EntityName(Guid, String)
+ EntityName()
+ Refresh(): void
«property»
+ Component(): List<EntityNameComponent>
+ NameUse(): Concept
+ NameUseKey(): Guid?
-m_names 0..*
-m_targetEntity
VersionedEntityData
Entity
- m_addresses: List<EntityAddress>
- m_classConcept: Concept
- m_classConceptId: Guid
- m_creationAct: Act
- m_creationActId: Guid
- m_determinerConcept: Concept
- m_determinerConceptId: Guid
- m_extensions: List<EntityExtension>
- m_identifiers: List<EntityIdentifier>
- m_names: List<EntityName>
- m_notes: List<EntityNote>
- m_participations: List<ActParticipation>
- m_relationships: List<EntityRelationship>
- m_statusConcept: Concept
- m_statusConceptId: Guid
- m_tags: List<EntityTag>
- m_telecomAddresses: List<EntityTelecomAddress>
- m_typeConcept: Concept
- m_typeConceptId: Guid?
+ Refresh(): void
+ SetDelayLoadProperties(List<EntityName>, List<EntityAddress>, List<EntityIdentifier>, List<EntityTelecomAddress>): void
«property»
+ Addresses(): List<EntityAddress>
+ ClassConcept(): Concept
+ ClassConceptKey(): Guid
+ CreationAct(): Act
+ CreationActKey(): Guid
+ DeterminerConcept(): Concept
+ DeterminerConceptKey(): Guid
+ Extensions(): List<EntityExtension>
+ Identifiers(): List<EntityIdentifier>
+ Names(): List<EntityName>
+ Notes(): List<EntityNote>
+ Participations(): List<ActParticipation>
+ Relationships(): List<EntityRelationship>
+ StatusConcept(): Concept
+ StatusConceptKey(): Guid
+ Tags(): List<EntityTag>
+ Telecoms(): List<EntityTelecomAddress>
+ TypeConcept(): Concept
+ TypeConceptKey(): Guid?
VersionedAssociation
PersonLanguageCommunication
+ PersonLanguageCommunication()
+ PersonLanguageCommunication(String, bool)
«property»
+ IsPreferred(): bool
+ LanguageCode(): string
-m_languageCommunication 0..*
VersionedAssociation
PlaceService
- m_service: Concept
- m_serviceConceptKey: Guid
+ Refresh(): void
«property»
+ ServiceConcept(): Concept
+ ServiceConceptKey(): Guid
+ ServiceSchedule(): Object
-m_services 0..*
Person
- m_languageCommunication: List<PersonLanguageCommunication>
- m_securityUser: SecurityUser
+ Person()
+ Refresh(): void
«property»
+ AsSecurityUser(): SecurityUser
+ DateOfBirth(): DateTime?
+ DateOfBirthPrecision(): DatePrecision?
+ LanguageCommunication(): List<PersonLanguageCommunication>
Material
- m_formConcept: Concept
- m_formConceptKey: Guid?
- m_quantityConcept: Concept
- m_quantityConceptKey: Guid?
+ Material()
+ Refresh(): void
«property»
+ ExpiryDate(): DateTime?
+ FormConcept(): Concept
+ FormConceptKey(): Guid?
+ IsAdministrative(): Boolean
+ Quantity(): Decimal?
+ QuantityConcept(): Concept
+ QuantityConceptKey(): Guid?
DeviceEntity
- m_securityDevice: SecurityDevice
- m_securityDeviceKey: Guid
+ DeviceEntity()
+ Refresh(): void
«property»
+ ManufacturedModelName(): String
+ OperatingSystemName(): String
+ SecurityDevice(): SecurityDevice
+ SecurityDeviceKey(): Guid
Organization
- m_industryConcept: Concept
- m_industryConceptKey: Guid
+ Organization()
+ Refresh(): void
«property»
+ IndustryConcept(): Concept
+ IndustryConceptKey(): Guid
ApplicationEntity
- m_securityApplication: SecurityApplication
- m_securityApplicationKey: Guid
+ ApplicationEntity()
+ Refresh(): void
«property»
+ SecurityApplication(): SecurityApplication
+ SecurityApplicationKey(): Guid
+ SoftwareName(): String
+ VendorName(): String
+ VersionName(): String
Place
- m_services: List<PlaceService>
+ Place()
+ Refresh(): void
«property»
+ ClassConceptKey(): Guid
+ IsMobile(): Boolean
+ Lat(): double?
+ Lng(): double?
+ Services(): List<PlaceService>
UserEntity
- m_securityUser: SecurityUser
- m_securityUserKey: Guid
«property»
+ SecurityUser(): SecurityUser
+ SecurityUserKey(): Guid
ManufacturedMaterial
+ ManufacturedMaterial()
«property»
+ LotNumber(): String

8.5. Pre-Configured Data Reference

SanteDB's data model is expected to be populated with a minimum set of data which the core functionality will use. If a data persistence store does not have the required data elements described in this section, key functionality of the SanteDB system may not function as expected.

8.5.1. Object Identifier (OID) Reference

The SanteDB project namespace (OID) is located at : 1.3.6.1.4.1.33349.3.1.5.9 [iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprise(1) SanteSuite (33349.3.1.5.9)

Root	Child	Description
1.3.6.1.4.1.33349.3.1.5.9		SanteSuite
1.3.6.1.4.1.33349.3.1.5.9	0	Services
1.3.6.1.4.1.33349.3.1.5.9	1	Applications
1.3.6.1.4.1.33349.3.1.5.9	2	Privacy & Security
1.3.6.1.4.1.33349.3.1.5.9	3	Concepts / Vocabulary
1.3.6.1.4.1.33349.3.1.5.9	4	Templates
1.3.6.1.4.1.33349.3.1.5.9	5	Deployments

8.5.1.1. Template References

SanteDB project contains a series of the templates which are used to pre-populate and provide form entry using SanteDB client components. These templates are identified in .

Template	Template ID	Description
	1.3.6.1.4.1.33349.3.1.5.9.4.1	Root template namespace for all acts related to administration of substances.
Immunization	1.3.6.1.4.1.33349.3.1.5.9.4.1.0 (act.substanceadmin.immunization)	Immunization template containing information related to a routine immunization given to a patient.
Drug Therapy	1.3.6.1.4.1.33349.3.1.5.9.4.1.1 (act.substanceadmin.supplement)	Drug template related to a routine drug therapy such as a supplement schedule.
	1.3.6.1.4.1.33349.3.1.5.9.4.2	Root template namespace for all acts related to patient encounters.
Patient Appointment	1.3.6.1.4.1.33349.3.1.5.9.4.2.0 (act.patientencounter.appointment)	Represents a generic patient encounter appointment.
	1.3.6.1.4.1.33349.3.1.5.9.4.3	Root template namespace for all acts related to stock events.
Stock Transfer	1.3.6.1.4.1.33349.3.1.5.9.4.3.0 (act.stock.xfer)	Act template which is used to indicate the physical transfer of objects from one holder to another.
Stock Adjustment	1.3.6.1.4.1.33349.3.1.5.9.4.3.1 (act.stock.adjust)	Act template which is used to indicate the adjustment of stock in a holder
	1.3.6.1.4.1.33349.3.1.5.9.4.4	Root template namespace for templates related to observations
Weight	1.3.6.1.4.1.33349.3.1.5.9.4.4.0 (act.observation.weight)	A template which is used for recording of patient weight.
Height or Length	1.3.6.1.4.1.33349.3.1.5.9.4.4.1 (act.observation.height)	A template which is used for recording of patient height / length.
Temperature	1.3.6.1.4.1.33349.3.1.5.9.4.4.2 (act.observation.temp)	A template which is used for recording of patient temperature.
Blood Pressure	1.3.6.1.4.1.33349.3.1.5.9.4.4.3 (act.observation.bp)	A template which is used for recording of patient blood pressures.

	1.3.6.1.4.1.33349.3.1.5.9.4.5	Root template namespace of all acts related to problem concerns
Adverse Event Following Immunization	1.3.6.1.4.1.33349.3.1.5.9.4.5.0	Represents the Adverse Event Following Immunization problem event template
Problem Statement	1.3.6.1.4.1.33349.3.1.5.9.4.5.1	Problem Statement – Represents a simply stated problem that the patient is or has experienced with a severity.
Patient allergy / intolerance	1.3.6.1.4.1.33349.3.1.5.9.4.5.2	Represents a patient allergy or known intolerance.
Patient Functional Limitations	1.3.6.1.4.1.33349.3.1.5.9.4.5.3	Represents a description of a functional limitation the patient has.

8.5.1.2. Built-in Concept Sets

Key to the construction and validation of data in the SanteDB HDS and Disconnected Client, is the concept of concept sets. A concept set is described in further detail in 8.2.4 SanteDB Concept Model on page 112.

OID	**Concept Set**	**Description**
1.3.6.1.4.1.33349.3.1.5.9.3.39	Allergies and Intolerance Types	Used to group together those concepts which represent allergies and/or intolerance types.
1.3.6.1.4.1.33349.3.1.5.9.3.41	Act Types	Identifies the types of acts which can occur and is usually the basis for validation on Act.TypeConcept property.
1.3.6.1.4.1.33349.3.1.5.9.3.0	Concept Status	Status codes which map the status of concepts.
1.3.6.1.4.1.33349.3.1.5.9.3.1	Act Class	Codes which are used for classifying the types of Acts which occur in the system.
1.3.6.1.4.1.33349.3.1.5.9.3.10	Address Component Type	Codes which are used to classify the types or parts of an address.
1.3.6.1.4.1.33349.3.1.5.9.3.11	Name Use	Codes which classify how a particular representation of a name are to be used.
1.3.6.1.4.1.33349.3.1.5.9.3.12	Telecom Address Use	Codes which classify the use of telecommunications addresses.
1.3.6.1.4.1.33349.3.1.5.9.3.13	Telecom Address Type	Codes which identify the type of telecommunications address represented..
1.3.6.1.4.1.33349.3.1.5.9.3.14	Service Code	Codes which identify the services offered by a facility.

1.3.6.1.4.1.33349.3.1.5.9.3.15	Industry Code	Codes which classify the type of industry a particular organization operates within.
1.3.6.1.4.1.33349.3.1.5.9.3.17	Role Status	Codes which represents the status of an entity or role.
1.3.6.1.4.1.33349.3.1.5.9.3.18	Name Component Type	Codes which classify parts of a name.
1.3.6.1.4.1.33349.3.1.5.9.3.19	Reason Codes	Codes which represent general reasoning behind performing an action.
1.3.6.1.4.1.33349.3.1.5.9.3.2	Act Mood	Codes which are used to represent the mode of a particular act.
1.3.6.1.4.1.33349.3.1.5.9.3.23	Family members of children	Relationship type codes which are used to represent family relations.
1.3.6.1.4.1.33349.3.1.5.9.3.24	Spousal Family Members	Relationship type codes which ware used to represent spouses
1.3.6.1.4.1.33349.3.1.5.9.3.25	Vaccines	Type codes which are used to represent vaccines
1.3.6.1.4.1.33349.3.1.5.9.3.26	Administration Act Type Codes	Type codes which are used to further classify types of substance administrations (immunization, booster, etc.)
1.3.6.1.4.1.33349.3.1.5.9.3.28	AdministrableDrugForm	Codes which classify the form of a drug (injection, capsule, etc.)
1.3.6.1.4.1.33349.3.1.5.9.3.29	Route of administration	Codes which are used to identify routes of administration (oral, intradermal injection, etc.)
1.3.6.1.4.1.33349.3.1.5.9.3.3	Act Status	Codes which track the current status of an act.
1.3.6.1.4.1.33349.3.1.5.9.3.30	Subset of Discharge Disposition (HL7)	Codes which represent the discharge dispositions supported in SanteDB.
1.3.6.1.4.1.33349.3.1.5.9.3.32	AdministrationSite	Codes which are use to identify the location where an injection takes place.
1.3.6.1.4.1.33349.3.1.5.9.3.34	Vital Signs	Codes which are used to classify observations of vital signs.
1.3.6.1.4.1.33349.3.1.5.9.3.35	Units of Measure	Codes which are used to capture the units of measure.
1.3.6.1.4.1.33349.3.1.5.9.3.36	Units of Measure for Weight	Valid units of measure for weight.
1.3.6.1.4.1.33349.3.1.5.9.3.37	Protocol Violation Reason	Reason codes which are specifically used for “why” an action varies from the clinical protocol.
1.3.6.1.4.1.33349.3.1.5.9.3.38	Stock Reason Codes	Reason codes which are used for stock adjustments.
1.3.6.1.4.1.33349.3.1.5.9.3.4	Act Relationship Type	Codes which are used to classify relationships between acts.
1.3.6.1.4.1.33349.3.1.5.9.3.42	Null Reason	Codes which are used to describe why a value is not present.

1.3.6.1.4.1.33349.3.1.5.9.3.43	Severity Observation Values	Codes which are used to describe the severity of something.
1.3.6.1.4.1.33349.3.1.5.9.3.44	Causes of Death	Codes which are used to describe causes of death.
1.3.6.1.4.1.33349.3.1.5.9.3.45	Observation Act Types	Codes which are used to classify the types of observations.
1.3.6.1.4.1.33349.3.1.5.9.3.46	Reaction Observations	Codes which are used to classify the types of reactions that can occur.
1.3.6.1.4.1.33349.3.1.5.9.3.47	Subset of Disposition (HL7)	
1.3.6.1.4.1.33349.3.1.5.9.3.48	Security Audit Codes	Codes which are used for security audits
1.3.6.1.4.1.33349.3.1.5.9.3.49	Material Type Codes	Codes which are used to describe the type of material.
1.3.6.1.4.1.33349.3.1.5.9.3.5	Act Interpretation	Codes which are used to describe how an act is interpreted.
1.3.6.1.4.1.33349.3.1.5.9.3.50	Protocol Violation - Don't Reschedule	
1.3.6.1.4.1.33349.3.1.5.9.3.51	Diagnosis Codes	Codes which are used for diagnosis of problems.
1.3.6.1.4.1.33349.3.1.5.9.3.52	Problem Observation Types	Used to distinguish regular observations made with observations of problems
1.3.6.1.4.1.33349.3.1.5.9.3.53	Adverse Event Types	Used to distinguish acts which represent adverse events of various types
1.3.6.1.4.1.33349.3.1.5.9.3.59		
1.3.6.1.4.1.33349.3.1.5.9.3.6	Entity Class	
1.3.6.1.4.1.33349.3.1.5.9.3.7	Entity Status	
1.3.6.1.4.1.33349.3.1.5.9.3.8	Entity Relationship Type	
1.3.6.1.4.1.33349.3.1.5.9.3.9	Address Use	
1.3.6.1.4.1.33349.3.1.5.9.3.100	Base for Custom Code Systems	All custom code systems use this reserved root as a base
1.3.6.1.4.1.33349.3.1.5.9.3.200	HL7 Version 2 root tables	Table numbers are used after this OID as a root

8.5.1.3. Custom Code Systems

OID	**Concept Set**	**Description**
1.3.6.1.4.1.33349.3.1.5.9.3.100.1	GS1 Stock Codes	Stock status codes for GS1
1.3.6.1.4.1.33349.3.1.5.9.3.100.2	HL7v2 Administrative Gender	Administrative Gender codes from HL7v2