1-26-2016

# Software Interoperability and the Pods OpenHDS System

Benjamin S. Heasly MS
*University of Southern Maine*

# SOFTWARE INTEROPERABILITY

# AND THE PODS OPENHDS SYSTEM

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE

UNIVERSITY OF SOUTHERN MAINE

DEPARTMENT OF COMPUTER SCIENCE

BY

Benjamin S. Heasly

2017

THE UNIVERSITY OF SOUTHERN MAINE

DEPARTMENT OF COMPUTER SCIENCE

22 January 2016

We hereby recommend that the thesis of Benjamin Heasly entitled

SOFTWARE INTEROPERABILITY

AND THE PODS OPENHDS SYSTEM

be accepted in partial fulfillment of the requirements for the Degree of Master of Science

[illegible] Advisor

[illegible]

[illegible]

Accepted

# Acknowledgements

Thanks to Bruce MacLeod, my thesis advisor, for providing the workspace where I learned a great deal about software, development, and collaboration.

Thanks to the Grameen Foundation and Maine Economic Improvement Fund for supporting my graduate assistantship and coursework.

Thanks to the members of the USM Computer Science faculty with whom I worked, for their excellent and enjoyable instruction in computer science theory and practice: Suad Alagic, David Bantz, Clare Bates Congdon, Bob Boothe, David Briggs, Bruce MacLeod, and Stephen Houser.

Thanks to my fellow students who made learning and working a lot of fun.  Thanks in particular to my friends with whom I worked on the OpenHDS project: Andrew Dorman Taylor and Nathan Wolfe Lienhardt.

# Abstract

This work addressed challenges of software system interoperability faced by the Open Health and Demographics Surveillance System (OpenHDS). OpenHDS is a distributed application for demographic data collection which was used during a public health intervention in Equatorial Guinea. Specific challenges faced during this intervention included offline data collection and synchronization, changing data collection and software requirements, data size and system performance, and correction of software and data collection errors. This work produced in a new system, the PODS OpenHDS System, which applied four design themes in order to address these challenges: Polymorphism, developer Operations, Declarative style, and Self-description.

The OpenHDS system is described, including its demographic data model and applicability to the public health intervention in Equatorial Guinea. Key technologies employed by the PODS OpenHDS System are summarized. The architecture of the PODS OpenHDS System is described, including a central server and multiple tablet computer clients. The four PODS design themes are described, including their realized and anticipated benefits for OpenHDS. Future work for the PODS OpenHDS System is discussed, including applicability to existing systems, known shortcomings, and natural extensions.

# Table of Contents

## List of Tables

## List of Listings

# List of Figures

# Introduction

## OpenHDS

The Open Health and Demographics Surveillance System (OpenHDS) is a software project aimed at collecting and maintaining population demographic data for use in community health studies and interventions.  Demographic records are critical for many studies and interventions in order to provide denominators for reporting and analysis. For example, clinics in a study area may be able to report the number of patients seen over time.  But how does this number compare to the whole population -- what is the rate of visits?

Collecting and maintaining demographic records in developing countries, like Equatorial Guinea, has traditionally been done using paper forms and manual data aggregation (Thriemer et al., 2012).  This method is labor-intensive and error-prone.  The OpenHDS software offers an alternative method based on electronic forms and web-based data aggregation.  The software offers significant advantages over paper-based systems including automated validations at collection time and aggregation time, and faster data entry and search.

In addition, the OpenHDS provides a data model for representing locations and individuals within a study area, and demographic events that occur over the duration of a study along with their effects on the individuals in the population.

A distinguishing feature of the OpenHDS is the use of Android tablet computers to facilitate data entry by surveyors (Google, n.d.-a). The application represents geographic and individual data visually, allowing surveyors to select relevant records of interest by touch. In turn, the application is able to automatically enter relevant data into electronic form documents.

## Interoperability Challenges

The OpenHDS system is a distributed application, with a central server to collect data and various clients including Android tablets and web browsers. The system is complicated by the fact that surveyors operating in the field may have limited or no internet connectivity. This presents interoperability challenges: how to how to support concurrent data collection by multiple offline clients, how to maintain consistency and avoid conflicts among records distributed across clients, and when and how to transfer data between server and clients. Much of the work described here was focused on these interoperability challenges.

Additionally, this work has explored how client and server interoperation can be maintained over time, as software is revised and and project requirements change:

- What design choices and types of metadata can allow the system to respond gracefully to changes, instead of breaking?
- How to minimize the effort of development and redistribution required when implementing system changes?
- How to facilitate correction of inevitable data entry and programming errors?

## Design Themes

This work followed four design themes which proved valuable in addressing the challenges of interoperability and adaptability to change. These themes were also useful in general during the software development process. The themes are:

1. Applications should take advantage of *polymorphic* design.
2. Developer *operations* should be prioritized so that tasks like deployment, updates, and maintenance do not overwhelm an otherwise functional system.
3. Applications should be written and configured favoring a *declarative* style.
4. Applications should be *self-descriptive*.

For mnemonic purposes, these themes may summarized as, "Polymorphic, Operations, Declarative, and Self-Descriptive", or simply PODS. Several appearances of the PODS themes in this work will be discussed, as well as their realized and anticipated advantages. The PODS acronym itself will be used to refer to the specific version of the OpenHDS system developed for this work.

## The PODS OpenHDS System

The OpenHDS project was originated by Bruce MacLeod and students in the Computer Science Department at the University of Southern Maine (USM). The software system includes a server application which provides a REST interface to a database, and an Android tablet application which supports offline data collection.

The work discussed here is a continuation of previous OpenHDS development which applies the PODS themes to the OpenHDS system. The new PODS OpenHDS System includes three areas of work:

1. A complete reimplementation of the server application, now called OpenHDS-Rest (Heasly, Lienhardt, & Others, n.d.).

2. Extensive revisions to the existing OpenHDS-Tablet application (Heasly, Lienhardt, Taylor, & Others, n.d.).

3. A revised architecture through which server and clients communicate.

The PODS OpenHDS system is intended to support the future development of browser-based web clients, although these clients were not included in this work.

The data model, architecture, and operation of the PODS OpenHDS system will be described in detail, with particular focus on interoperability challenges that were faced and how they were addressed in this work. Some of these challenges were encountered during the software development process. Others were encountered by real-world users during a public health intervention in Equatorial Guinea.

# Use in Equatorial Guinea

## CIMS

The OpenHDS system was used  for the Campaign Information Management System (CIMS) deployed in Bioko Island, Equatorial Guinea by Medical Care Development International and USM.  The CIMS supported the Equatorial Guinea Malaria Vaccine Initiative which aimed to combat endemic Malaria using a newly developed vaccine, as well as conventional interventions like spraying of insecticide and distribution of bed nets.  The CIMS project began in 2014 and continued through 2016, at least.

CIMS used OpenHDS to conduct an initial census of Bioko island, and may continue to use OpenHDS throughout the project duration to record demographic updates.  In addition to the demographics, CIMS has extended OpenHDS to track rounds of insecticide spraying and bed net distribution.  Eventually, the project aims to use OpenHDS to track a series of malaria vaccine doses as they are administered to every individual on the island.

The OpenHDS deployment for CIMS may be typical of future OpenHDS deployments.

The project used a central server to host the project database, carry out data processing and validation, and make data available to various clients.  For this purpose, a remote virtual server was deployed using Amazon Web Services (Amazon.com, n.d.).  An alternative would have been to deploy a physical server located at the study site. However, the use of a remote server had advantages: the server itself was not susceptible to power failures or other logistical concerns at the study site, and the server was easily administered by the development team using the internet connection at USM, which was likely faster and more reliable than the connection available at the study site.

Multiple Android tablet computers were deployed with the OpenHDS-Tablet software. Before each round of data collection, each tablet downloaded a large subset demographic data from the central server. These tablets were taken into the field for offline data collection. Following data collection, each tablet uploaded filled-out forms to the server for processing and aggregation. Ideally, each client would have completed a download-upload cycle for each day of data collection. In practice, this cycle was often hindered by intermittent or slow internet connectivity at the study site, large volumes of data being transferred, data collection errors, and updates to the server and tablet software.

## CIMS Challenges Addressed by the PODS OpenHDS system

The CIMS project highlights several interoperability and robustness challenges faced by systems like OpenHDS:

- Most of Bioko island is rural, requiring concurrent data collection using many offline devices.

- The island has a large population -- about 265,000 in 2001 -- requiring large data transfers between server and clients (Beziz & Directorate General of Statistics and National Accounts, 2005).

- The project has unique data tracking requirements, like bed net tracking, which required extensions to the OpenHDS system.

- The project has long duration, requiring OpenHDS to continue functioning as changes are introduced over time.

The PODS OpenHDS system began as part of the CIMS project and has since diverged and evolved independently in order to explore significant architectural and implementation alternatives motivated by the PODS themes.  The PODS OpenHDS system is designed to address challenges faced by CIMS.

## OpenHDS Data Model

The PODS OpenHDS system defines a data model suitable for demographic surveillance in a well-defined geographical area. The model may be informally divided into three parts: Census, Update, and Operations. Census entities and relationships model an initial baseline data collection including individuals, places of interest, and a geographic partitioning of the study area itself. Update entities and relationships model longitudinal data collection -- changes over time to the individuals in the study area. Finally, Operations entities and relationships support field operations, application logic, and auditing by modeling the system users and surveyors.

In addition to specific entities and relationships, the data model prescribes processing side effects that must accompany new records in order to maintain semantic consistency. For example, a person's moving out of the study area would be recorded as an `OutMigration` event. The prescribed side effect of recording this event would be to modify the moving person's `Residency` record with a status of `terminated`. That way, the person would no longer be listed among those living in the study area.

Prescribed side effects also allow the data model to express complex patterns which are built up from multiple entities and relationships. For example, this work introduces the Household pattern, which combines `Individual`, `SocialGroup`, and other records to represent a family.

## Census

Eight Census entities represent people and places in the study area. These would be collected during initial baseline data collection and would be updated and supplemented during subsequent data collection. These entities appear in green in Figure 1, below. They are:

- **`Individual`**. These are people who live in the study area, including those participating in a study and those who are not participating.
- **`Relationship`**. Each `Relationship` forms a one-way connection from one `Individual` to another, with a particular type. For example, an `Individual` might be the "niece of" another `Individual`.
- **`SocialGroup`**. These are families and other groups of people.
- **`Membership`**. Each `Membership` indicates the belonging of an `Individual` to a particular `SocialGroup` during a time period.
- **`Location`**. These are dwellings where `Individuals` may reside, and other places of interest in the study area.
- **`Residency`**. Each `Residency` indicates the dwelling of an `Individual` at a particular `Location` during a time period.
- **`LocationHierarchy`**. These form a tree structure which partitions the study area geographically at several levels of detail, in order to facilitate the search for `Location` records and related records.

- **`LocationHierarchyLevel`**. These describe the levels of detail used for the geographic partitioning of the study area. They also indicate the tree depth of each `LocationHierarchy` record. Typical `LocationHierarchyLevel` records might have names like "continent", "country", "province", "district", etc.

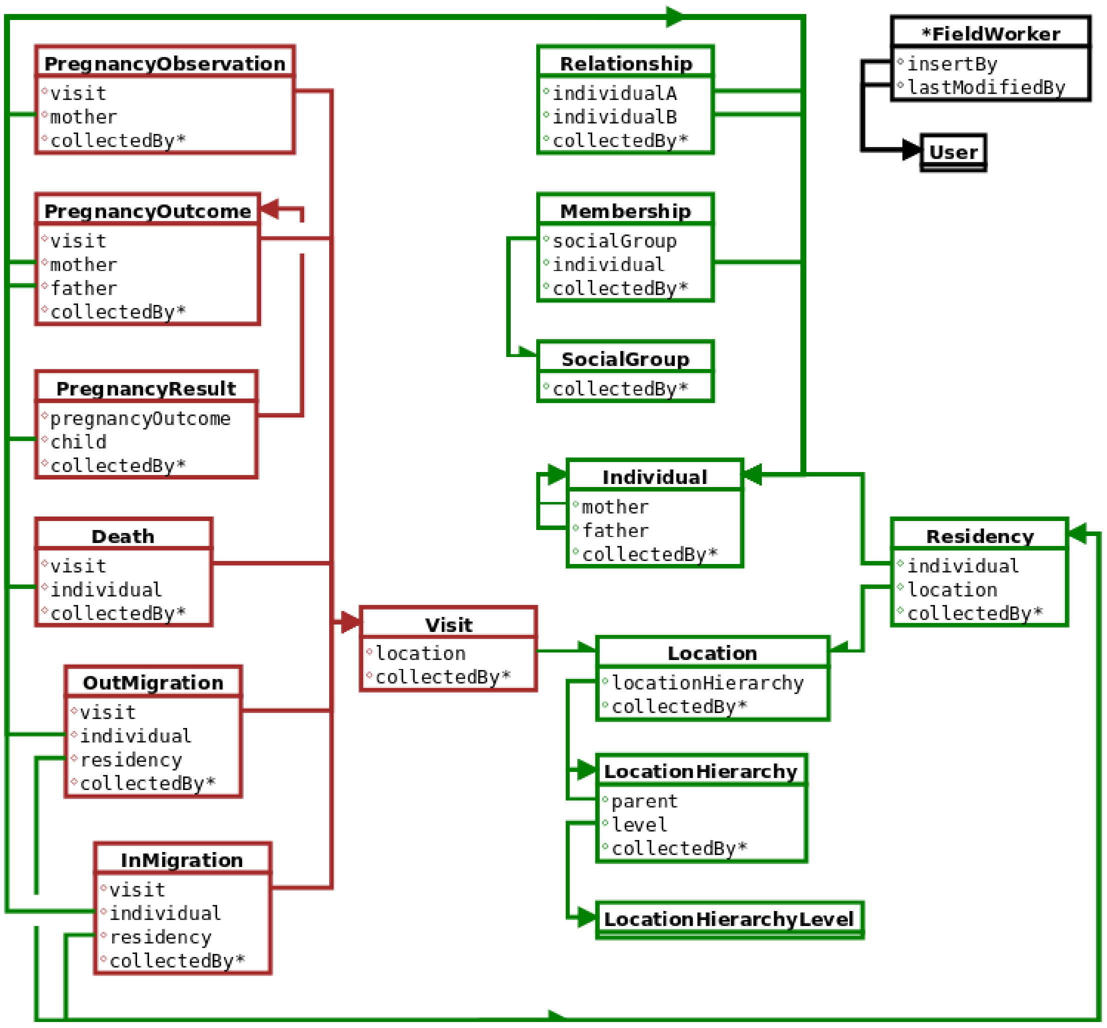


*Figure 1. Data Model Used in the PODS OpenHDS System. Black boxes and arrows represent entities used for application logic and field Operations. Green boxes and arrows represent entities that make up the Census or baseline part of the OpenHDS data model. Red boxes and arrows represent entities that make up the Update or longitudinal part of the OpenHDS data model. Most entities have a* `collectedBy` *attribute which refers to the* `FieldWorker` *entity. For clarity, this standard relationship is indicated with stars (*) instead of with arrows. All entities have attributes* `insertBy` *and* `lastModifiedBy` *which refer to the* `User` *entity. For clarity, these standard attributes are depicted once, for* `FieldWorker`*, and omitted elsewhere.*

## Update

Seven Update entities model demographic change events that may occur over time. These support longitudinal data collection following the initial Census. These entities appear in red in Figure 1. Many longitudinal records are associated with side effects, including modifications to existing Census entities or creation of new Census entities. Briefly, these entities and their expected side effects are:

- **`Visit`**. Each `Visit` represents the act of a surveyor visiting a `Location` in the study area, on a particular date. A `Visit` should be recorded whenever a surveyor visits a `Location` during data collection, even if no other demographic events are recorded.

- **`Death`**. Each `Death` record indicates that an `Individual` has died, and when. The side effects of recording a `Death` must be to modify the status of the `Individual` who died, and to mark the `Individual`'s current `Relationship`, `Membership`, and `Residency` records as "terminated".

- **`OutMigration`**. Each `OutMigration` records the fact that an `Individual` who resided within the study area had moved. The side effect of recording an `OutMigration` must be to terminate the `Individual`'s current `Residency`.

- **`InMigration`**. Each `InMigration` records the fact that an `Individual` has moved to a new `Location` within the study area. If the `Individual` already resides in the study area, the `InMigration` must be recorded as `internal`, with the side effect of terminating the `Individual`'s previous `Residency` and creating a new `Residency` for the `Individual` at the new `Location`.

Otherwise, the `InMigration` must be recorded as `external`, with the side effect of creating a new `Residency` for the `Individual` at the new `Location`.

- **`PregnancyObservation`**. Each `PregnancyObservation` records the fact that an `Individual` is pregnant, and the expected date of delivery.

- **`PregnancyOutcome`**. Each `PregnancyOutcome` records the end of a pregnancy, and the date and manner of ending, for example live birth or miscarriage. Each `PregnancyOutcome` may be associated with zero or more `PregnancyResults`.

- **`PregnancyResult`**. Each `PregnancyResult` represents the result of a full-term pregnancy, including still or live births. For a live birth, the side effect of processing each `PregnancyResult` must be to create a new `Individual` who was born, to create a `Residency` for the `Individual` at the Mother's `Residency`, and to create a `Membership` for the `Individual` in the Mother's family `SocialGroup` (if any).

## Application Logic and Field Operations

Two entities assist with application logic, field Operations, and auditing, but are not themselves subjects of study. These entities appear in black in Figure 1. These are:

- **`User`**. Each `User` represents a person who may access the OpenHDS-Rest server. `Users` must authenticate with a username and password, and may have authorization to perform certain actions like searching for existing records, or creating new records. All Census and longitudinal records are associated with the `Users` who created and last modified them.

- **`FieldWorker`**. Each `FieldWorker` record represents a surveyor who goes into the field to collect Census and Update data. Surveyors also must authenticate with a "fieldWorkerId" and password. Most Census records and all Update records are associated with the `FieldWorker` record of the surveyor who recorded them.

## Households

In addition to the entities of the OpenHDS data model, this work introduces the notion of a Household. A Household is not an entity, but a pattern for combining Census entities in order to represent a family or extended family group. This organizing pattern was introduced as a requirement for CIMS project field operations, and may be useful to future projects.

Each Household has a single `SocialGroup` representing the family itself. Each Household has a distinguished `Individual` who is the head of Household. This `Individual` has `Membership` in the family `SocialGroup`. In addition to the head, each Household may have a number of `Individuals` who are Household members. Like the head of Household, each member has a `Membership` in the family `SocialGroup`. Each member also has a `Relationship` to the head of Household indicating their family or other relationship. These `Relationships` have types like "spouse", "nephew", or "non-family".

The PODS OpenHDS system implements this Household pattern.  It includes Household Registrations and resources as part of its REST API, and performs Registration side effects and data processing consistent to maintain Household entities and relationships. See Input Registrations and Side Effects below for an example Household Registration.

## Changes Made for PODS OpenHDS

The data model used in this work was adapted from a previous version of the OpenHDS, and in turn from a data model published by Bruce MacLeod and others (Benzler, Herbst, & MacLeod, 1998).  This work introduces several modest changes to the data model which fall into the following categories:

- **Add Attributes**.  The CIMS project required the addition of many new entity attributes in order to support field operations and reporting.  Many of these attributes that seem likely to be useful for future projects have been retained, for example a general "description" attribute for the Location entity.

- **Rename**.  A few entities and attributes were renamed to avoid confusion.  For example, an existing `Outcome` entity was renamed to `PregnancyResult` to avoid confusion with the related but distinct `PregnancyOutcome` entity.

- **Refactor**.  To facilitate implementation and documentation, related entities were grouped and implemented by extending common superclasses (see Instructive Inheritance, below).  This resulted in some attribute changes.  For example, the `FieldWorker extId` attribute was renamed to `fieldWorkerId` in order to avoid the misleading suggestion that the `FieldWorker` entity is similar to other entities that have an `extId` attribute.

## Key Technologies

This work employed many software frameworks, software standards, and architectural styles.  For the convenience of the reader, the main technologies are introduced here.

### Spring Framework

The OpenHDS-Rest server application was written using the Spring Framework.  Spring Framework is an open-source application framework which facilitates creation of Enterprise Java applications through features including an inversion of control container, dependency injection, and declarative application configuration (Pivotal, n.d.-d).

Use of Spring allowed OpenHDS-Rest to be developed relatively quickly--a few person-months.  This was largely due to robust facilities provided by the framework which could be included in OpenHDS-Rest, rather than re-implemented:

- Spring Boot allows application configuration via Java annotations, automates selection and resolution of library dependencies and versions, and allows application startup as a one-liner (Pivotal, n.d.-e).

- Spring Data and Java Persistence API (JPA)  declare, implement, and encapsulate database interactions and the mappings between Java objects and database records (Pivotal, n.d.-f)(Oracle, n.d.-a).

- Spring MVC provides HTTP/REST request handling, request mapping to custom handler methods, and marshalling between Java objects and XML and JSON request content (Pivotal, n.d.-b).

- Spring AOP provided Aspect-Oriented Programming, which allowed certain cross-cutting concerns like exception handling and logging to be implemented in one place, then applied automatically and dynamically throughout the application (Pivotal, n.d.-a).

## HATEOAS and HAL

Of central importance for the PODS OpenHDS system, Spring HATEOAS supports Hypermedia As The Engine Of Application State.  This architectural style informed the self-descriptive REST API used by OpenHDS-Rest.  In this style, REST responses include record data as well as hypermedia links that place the data in the context of related records, collections, etc.  Spring HATEOAS automatically renders these links using the Hypermedia Application Language (HAL) of JSON or XML (Fowler, 2010) (Pivotal, n.d.-g)(Kelly, 2013).

HAL links have two essential parts: a link relation name (sometimes shortened to "rel") that indicates the meaning of the link, and the URL where related data may be requested.  Spring HATEOAS supports many of the IANA (Internet Assigned Numbers Authority)  standard relations, such as `self` to indicate the canonical address of a record itself, or `collection` to indicate the canonical collection to which a record belongs (Nottingham, Reschke, & Algermissen, 19 April, 2016).

In addition to standard relations, OpenHDS-Rest introduces several domain-specific relations.  For example, a record that represents an individual will have the standard `self` link, as well as links to to the Individual's `mother` and `residency`, which are specific to the OpenHDS data model.  In order to generate these domain-specific relations, OpenHDS-Rest uses the Java Reflection API to determine the ids of related records and build links to them (Oracle, n.d.-b).

Link relations provide essential textual clues to the client application about how to interpret requested data.  But relations are only clues -- words or short phrases -- and the receiving application still must have logic in place to choose and enact an appropriate behavior to processing of the requested data.  The OpenHDS-Tablet Android application defines a `RelInterpretation` class and instances that provide a mapping from link relations to appropriate data transfer classes and data processing gateways.

## Android

The OpenHDS-Tablet mobile application was written for the Android platform, using the Android Java APIs.  Key parts of the Android APIs include Activities, which model user interactions, Tasks which do background I/O and data processing, data Providers, which implement and encapsulate database interactions, and Intents, which express commands and loose couplings between applications and application components (Google, n.d.-b).

The OpenHDS-Tablet application uses Intents and data Providers to collaborate with an existing Android application called Open Data Kit (ODK) Collect.  ODK Collect provides a user interface for reading and filling out data forms and data Providers for managing form definitions and filled-out form instances.  ODK Collect is also able to send filled-out form instances to a companion server application called ODK Aggregate, although use of ODK Aggregate is optional for this version of OpenHDS (Borriello et al., n.d.-b) (Borriello et al., n.d.-a).

The OpenHDS and ODK Collect use XForm data forms. XForm is an an XML-based standard for representing data forms and user interactions. The standard provides for custom data schemas, binding of data elements to user interface elements, and declarative and device-independent specifications of user interface elements, computations, and data validations (Pemberton & Klotz, n.d.).

XForms of a given type are represented by an XForm definition XML document, which specifies all of the form aspects above. During data entry, users edit XForm instance documents which contain only the data elements specified by the data schema. These instance documents must be interpreted with respect to their respective definition documents.

## Client-Server Interaction

Both the OpenHDS-Rest server application and the OpenHDS-Tablet mobile application use Universally Unique IDentifiers (UUIDs) as the primary key to identify database records. The UUID standard allows any host to independently generate an id which is almost certain to be unique across all records, and indeed, unique across hosts and time (Oracle, n.d.-c). This feature of UUIDs is important for correct interoperation of the OpenHDS-Rest server and clients. Because of off-line data collection, the server and clients must be able to create new records concurrently, independently, and while offline. Yet, when all records have been submitted to the server, the ids of newly created records must not collide. This requirement would be difficult to satisfy with traditional sequential record ids, which are likely to be duplicated across hosts.

In addition to OpenHDS-Tablet clients, the OpenHDS-Rest server is expected to interoperate with web browser based clients. Spring Boot provides a facility for adding Cross-Site Resource Sharing (CORS) headers to HTTP responses. These headers will allow OpenHDS-Rest to interoperate with browser-based clients that are not distributed with OpenHDS-Rest itself (Van Kesteren & Others, 2010).

# System Architecture and Operation

## Architectural Overview

The PODS OpenHDS system is a distributed application made up of the two subsystems: the OpenHDS-Rest server and the OpenHDS-Tablet client. A browser-based web client would be a third subsystem, which has not yet been developed. Each subsystem has clear boundaries defined by the server's REST API. Each may be developed independently, so long as it remains consistent with the API. Figure 2 illustrates the arrangement of these subsystems and their communication.

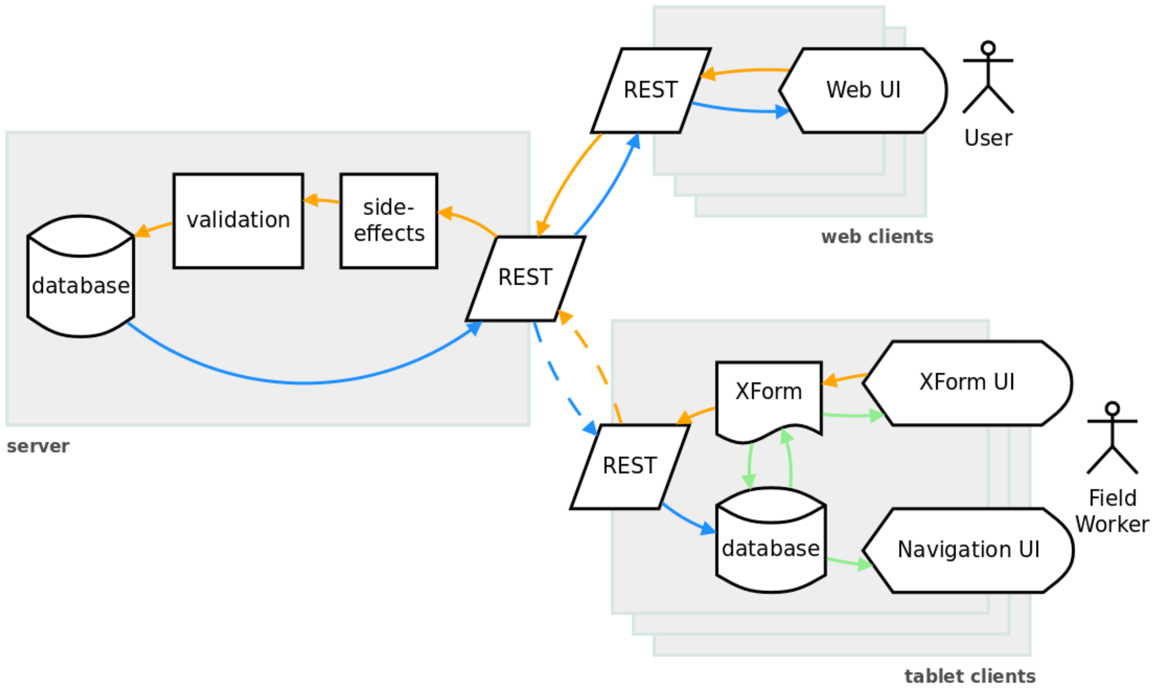


*Figure 2. Distributed Application Architecture. An instance of the OpenHDS-Rest server interacts with multiple clients of different types via its REST API. Orange arrows represent input data flowing towards the server, which may produce side effects and require validation before being stored in the server database. Blue arrows represent validated data flowing out of the server. Green arrows represent data flow internal to tablet clients, including validated data from the server as well as input data collected offline, which has not yet been sent to the server. Solid arrows represent continuous connectivity as would be expected for browser-based web clients. Dashed arrows represent intermittent connectivity as experienced for surveyors who are in the field and therefore offline.*

None of the subsystems would serve as a complete application on its own. The server lacks any user interface and does not address the problem of offline data collection. The tablet and web clients do not perform all of the side effects or validation prescribed by the OpenHDS data model, nor do they address the problems of data aggregation and long-term storage.

The design and implementation of the OpenHDS-Rest server was influential on the design of the tablet application, and would also dictate much of the design of a browser-based web client.  Thus, the server will be described first, followed by the OpenHDS-Tablet client and future web clients.

## OpenHDS-Rest Server

### Server Overview

The OpenHDS-Rest server provides basic functions that are required for the OpenHDS system.  These include processing of data uploaded by clients, aggregating data from multiple clients, and exposing aggregated data to clients for queries and retrieval. Uploading data involves three steps:

1. Perform side effects in response to incoming data, such as creating new records and resolving references to existing records.

2. Validate incoming and newly created records for syntactic and semantic errors.

3. Store validated data in a database.

The server provides additional functionality beyond the basic system requirements, in order to facilitate client-server interactions and system testing.  Additional functionality includes the generation of stub records which limit the sizes of result sets sent to clients, inclusion of HATEOAS links along with query responses which allow the server's REST

API to be self-describing, and generation of sample data to facilitate testing and demonstration.

The implementation treats exception handling as a cross-cutting concern.  It uses the Exception Handler mechanism of the Spring MVC to declare a mapping from Java `Exception` classes to HTTP response codes and messages (Pivotal, n.d.-c).

## Layered Implementation of the OpenHDS Data Model

The OpenHDS-Rest application implements the complicated OpenHDS data model as well as complicated behaviors related to the REST API, side effects, and validation. Therefore, the application risks having a code base which is itself complicated and difficult to understand or modify.  The OpenHDS-Rest implementation for this PODS work employs a layered architecture in order to separate application concerns into cohesive sections and facilitate understanding of and modifications to the code base (Fowler, 2002).  Table 1 illustrates the four layers of the OpenHDS-Rest implementation: Domain, Repository, Service, and Resource.

| Layer | Implementation supertypes | Representative entity classes |
|---|---|---|
| *Resource* | **`UuidIdentifiable RestController`**<br>**`Auditable RestController`**<br>**`AuditableCollected RestController`**<br>**`AuditableExtId RestController`** | **`FieldWorker RestController`**<br>**`User RestController`**<br>**`Individual RestController`**<br>**`InMigration RestController`** |
| *Service* | **`AbstractUuid Service`**<br>**`AbstractAuditable Service`**<br>**`AbstractAuditableCollected Service`**<br>**`AbstractAuditableExtId Service`** | **`FieldWorker Service`**<br>**`User Service`**<br>**`Individual Service`**<br>**`InMigration Service`** |
| *Repository* | **`UuidIdentifiable Repository`**<br>**`Auditable Repository`**<br>**`AuditableCollected Repository`**<br>**`AuditableExtId Repository`** | **`FieldWorker Repository`**<br>**`User Repository`**<br>**`Individual Repository`**<br>**`InMigration Repository`** |
| *Domain* | **`UuidIdentifiable`**<br>**`AuditableEntity`**<br>**`AuditableCollectedEntity`**<br>**`AuditableExtIdEntity`** | **`FieldWorker`**<br>**`User`**<br>**`Individual`**<br>**`InMigration`** |

*Table 1. Layered Application Architecture and Representative Classes. The four layers of the OpenHDS-Rest implementation are shown as rows. From bottom to top: the Domain layer is shown in blue, the Repository layer is shown in green, the Service layer is shown in orange, and the Resource layer is shown in red. The middle column shows class names of supertypes that define common behaviors for each layer. The right column shows representative concrete, entity classes that extend the layer supertypes.*

The Domain layer of the application models each OpenHDS entity as a Java class, with fields corresponding to entity attributes.  Many fields are annotated with standard JPA and Javax Validation annotations in order to express data constraints and entity relationships (Bernard & Peterson, 2009).

The annotated classes allow the Spring Data framework to automatically generate new classes that implement and encapsulate database interactions and the mappings between database records and entity class instances.  These generated classes represent the Repository layer of the application.

To process side effects required by the data model and to validate new data the implementation includes a Service layer which includes one Service class for each entity.  These classes use standard Javax Validation utilities to perform syntactic data validations, like null checks, that were declared as annotations in the Domain layer. These classes also use hand-written Java code to carry out semantic validations and to carry out side effects for longitudinal record processing.  For example, it is invalid for an `Individual`'s death date to precede her birth date.

Finally, to expose the implementation to clients, the server includes a Resource layer, with one Resource class per entity.  These classes use Spring MVC annotations to declare REST resources and resource paths which the Spring Framework should make available as HTTP request endpoints.  These classes associate custom Java code with each resource path, in order to perform a mapping from HTTP request to appropriate Service methods, and in turn to HTTP responses.

## Input Registrations

The OpenHDS-Rest server accepts input data for any entity in the OpenHDS data model, leading to a potentially large number of input data representations that the REST API must support. Furthermore, input representations must be able to specify simple record attributes as well as relationships between records, potentially complicating the input data representation. The PODS OpenHDS-Rest server introduces a unit of input submission called a Registration which is consistent across entities and simplifies the representation of record attributes and relationships.

Each Resource class in the implementation Resource layer defines a concrete Registration type with fields that are specific to one entity. Although the specific fields differ between entities, all Registrations obey the same general form and divide data into three parts:

- **Registration Metadata**. Registrations include a set of metadata fields common for all entities including the name, version number, and version name of the client system, and a timestamp.

- **Simple Attributes**. Each Registration must include a single complex field to represent the record being registered. This field must be named for the record's entity type, for example, `location` or `individual`. Typical attributes include the `uuid` and `name` of the record.

- **Related Record Ids**. Each Registration may include additional fields whose names indicate relationships between entities and whose values contain the UUIDs of specific records. For example, the `IndividualHouseholdRegistration` includes the field `motherUuid` to specify the UUID of the `Individual` record that should be referenced from the `mother` field of the `Individual` being registered.

Clients submitting data to the OpenHDS-Rest server must make an HTTP POST or PUT request at the resource path which corresponds to an entity, such as `/locations`, or `/individuals`. The body of each request must contain an XML or JSON representation of the concrete Registration type defined by in the corresponding Resource class. Listing 1 shows an `IndividualHouseholdRegistration` which might be POSTED to the `/individuals/household` resource path.

```
POST /individuals/household
```

```
{
   registrationSystemName: "systemName",
   registrationVersion: 1,
   registrationVersionName: "versionName",
   registrationDateTime: "2016-03-19T17:33:19.069-04:00[America/New_York]",
   individual: {
      uuid: "my-id",
      firstName: "my-name",
      extId: "my-name",
      gender: "FEMALE"
   },
   collectedByUuid: "f61997d8-...",
   motherUuid: "UNKNOWN",
   fatherUuid: "UNKNOWN",
   headOfHouseholdUuid: "UNKNOWN",
   relationshipUuid: "UNKNOWN",
   locationUuid: "UNKNOWN",
   residencyUuid: "UNKNOWN"
   socialGroupUuid: "f6199a62-...",
   membershipUuid: "UNKNOWN",
}
```

*Listing 1. Sample Household Registration. Clients may obtain an `IndividualHouseholdRegistration` template by making a GET requests at resource path `/individuals/household/sampleRegistration`. The server will respond with a well-formed `IndividualHouseholdRegistration` represented as JSON (as it is here) or XML. Data fields will initially have placeholder values, for example, "systemName" for the name of the client system, or "UNKNOWN" for the ids of related entities (highlighted in cyan), such as the `SocialGroup` of the Household. Clients may modify the data fields with real data, such as a `FieldWorker's` `collectedByUuid` and ids of Household records like the Household's `socialGroupUuid`. After filling in data, the client may POST the modified Registration to resource path `/individuals/household`.*

## Sample Registrations

Although Registrations provide an input data representation that is consistent across entities, clients still must discover the entity-specific field names that are required for each concrete Registration type. To assist clients in creating well-formed Registrations, all entity resources provide a sample Registration which clients may use as a template for data entry.

Clients may request sample Registrations by making an HTTP GET request at the resource path which corresponds to an entity, for example `/individuals/sampleRegistration` or `/inMigrations/sampleRegistration`. Listing 1 shows a sample `IndividualHouseholdRegistration` obtained from the `/individuals/household/sampleRegistration` resource path.

Sample Registrations may be particularly useful for long or complicated Registrations, whose valid syntax may not be obvious to the client. The `IndividualHouseholdRegistration` is particularly long, because it contains the UUIDs of many entities related to the `Individual`'s Household.

## Processing Side Effects

The OpenHDS data model prescribes processing side effects that must accompany the recording of certain records. In order to maintain coherency and clarity of the code base, the OpenHDS-Rest implementation divides this processing into two parts:

1. The Resource layer is responsible for parsing the input Registration, which may be JSON or XML, and representing the Registration data using standard Java classes and classes from the OpenHDS-Rest Domain layer.

2. The Service layer is responsible for actually processing Registration data provided by the Resource layer. This includes looking up related entities by id,

and updating and creating new records.  Notably, the Service layer is independent of the Registration classes defined at the Resource layer.

Figure 3 summarizes the sequence of Resource and Service method calls during processing of an `IndividualHouseholdRegistration`.

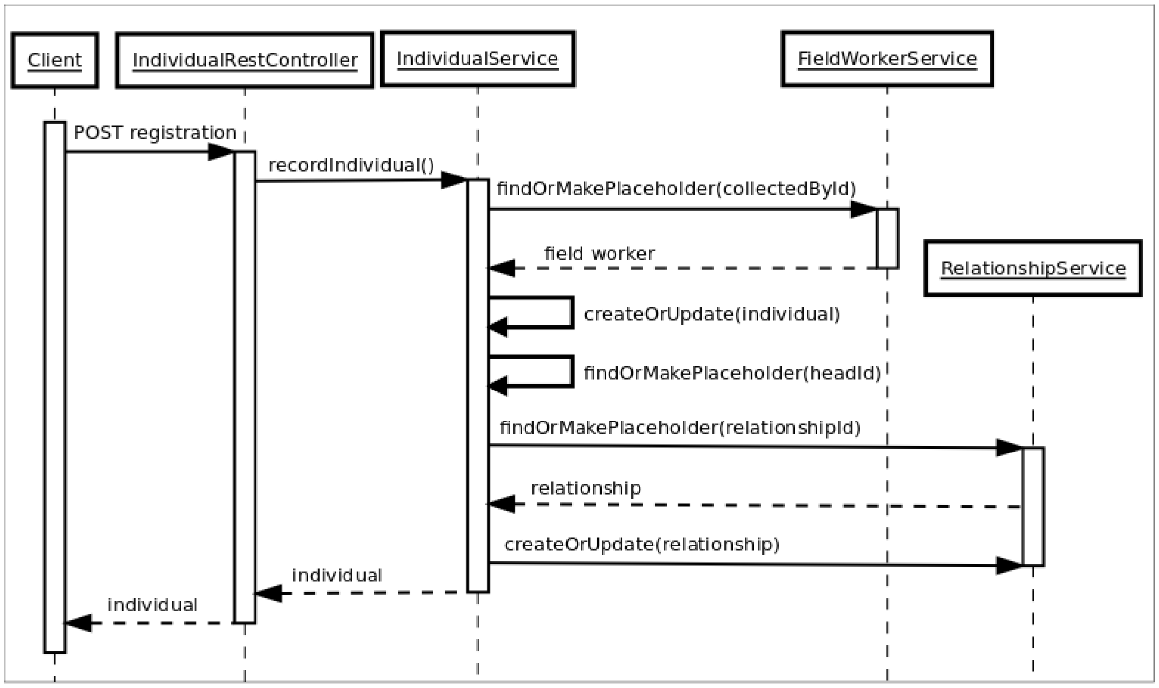


*Figure 3. Sequence Diagram of Registration Processing. A client POSTS an `IndividualHouseholdRegistration` to the OpenHDS-Rest Individual Resource. The `IndividialRestController` handles the POST request by unmarshalling the request body and passing the resulting `Individual` object and ids of related records to the `recordIndividual()` method of the `IndividualService`. The `IndividualService` resolves the the Registration's `collectedBy` `FieldWorker` reference by passing this id to the `findOrMakePlaceholder()` method of the `FieldWorkerService`. The resolved `FieldWorker` is assigned to the `Individual` record. The `IndividualService` stores the `Individual` record by calling its own `createOrUpdate()` method. By similar method calls to the `IndividualService` and `RelationshipService`, a head of Household `Individual` is resolved from the Registration's `headId`, and a `Relationship` is created and stored between the new `Individual` and this head of Household. For clarity, calls to additional Services are*

*omitted, which are similar to the calls to the `RelationshipService`. Calls to the the `MembershipService` and `SocialGroupService` create and store a `Membership` for the `Individual` in the Household `SocialGroup`. Calls to the `ResidencyService` and `LocationService` create and store a `Residency` for the `Individual` at the `Location` specified in the Registration. Finally, the updated `Individual` is returned to the `IndividualRestController`, which in turn responds to the client with a representation of the registered `Individual`.*

The `IndividualHouseholdRegistration` in particular requires numerous side effects because Household is a complex pattern build up from several entities and relationships. In order to process a `IndividualHouseholdRegistration` like the one in Listing 1, the server must:

- Record the `Individual` herself.

- Look up related entities by id, including the `FieldWorker` record for the surveyor who collected the Registration, the `Individuals` who are the mother and father of the new `Individual`, the `Individual` who is the head of the Household to which the new `Individual` belongs, the `SocialGroup` that represents the Household family, and the `Location` that represents the family home.

- Update attributes of the new `Individual`, including the `collectedBy` `FieldWorker`, and the mother and father `Individuals`.

- Create a `Relationship` record between the new `Individual` and the head of the Household.
- Create a `Residency` record for the `Individual` at the `Location` of the family home.
- Create a `Membership` record for the `Individual` in the `SocialGroup` of the family.

OpenHDS Update records may be submitted using Registrations similar to the Household Registration. Processing of these records requires side effects in order to keep the records in the database consistent. For example, after an `Individual` migrates out of the study area, she should no longer be listed as resident within the study area. Entities that require side effects like this include: `Death`, `InMigration`, `OutMigration`, and `PregnancyResult`.

## Unknown or Missing Attribute Values

Surveyors may be unable to obtain enough information to submit complete Registrations. For example, an `Individual`'s mother or father might be in fact not known to the `Individual`, or may not yet have been recorded in the study database. In order to facilitate continued operation despite incomplete data, the PODS OpenHDS-Rest implementation defines a distinguished Unknown record for each entity type.

Unknown records are real records, stored in the database, with the well-known id "UNKNOWN" in place of the UUID that would be used for normal records. In cases where complete information is unavailable, it is preferable to submit the Unknown record as a positive statement of ignorance. Alternatives would have been to submit a null record or no record at all, but these would be ambiguous and might be confused with missing or corrupt data.

The `IndividualHouseholdRegistration` in Listing 1 shows several instances of the distinguished "UNKNOWN" record id. These ids and the Registration are valid, if not particularly informative about the `Individual` being registered.

## Placeholder Records

A challenge for offline data collection is how to determine the order of submission for interdependent records. Consider for example two records: a daughter `Individual`, and her mother `Individual`. The daughter depends on the mother, by way of the `Individual` entitie's `mother` attribute. It would be natural to to submit the mother first, before daughter, so that the mother record may be resolved during processing of the daughter. However, clients might not maintain a proper order for collected records. Moreover, network errors and data validation errors may prevent records that were submitted in the correct order from being processed and stored in the correct order.

The PODS OpenHDS-Rest Service layer employs a placeholder strategy, in order to relieve clients from having to sort records by dependency, or from dealing with cascading failures caused by network or validation errors.

For example, in case a daughter record is submitted before her mother record, the `IndividualService` will create and store a placeholder record to represent the missing mother. This placeholder will use the mother's id provided with the daughter record, but otherwise use default attribute values and Unknown entity relationships. This placeholder will allow the daughter record to be recorded successfully, instead of failing because of the missing mother. Later, when the mother record is submitted, the server will treat the submission as an update to the placeholder, resulting in data consistency.

The OpenHDS-Rest Service layer uses this placeholder strategy throughout, for all entity relationships, not only for `Individuals` and mothers.

## Input Validation

The OpenHDS-Rest server must validate a wide variety of entities, attributes, and relationships defined in the OpenHDS data model, and report validation errors to clients in a way that allows the errors to be corrected. In order to simplify the validation code, the OpenHDS PODS implementation uses a declarative style where possible. In order to provide consistent error messages to clients, the implementation relies on cross-cutting exception handling.

For syntactic validation of single data fields, like "the field must be non-null" or "the string must have a minimum length", the implementation uses a declarative style based on standard Javax Validation annotations. These annotations are applied to fields of OpenHDS-Rest Domain classes and include error messages that are field-specific, such as "Location cannot have a null name". Following side effect processing, the Service layer calls standard Javax validation methods to validate records against the declared annotations, and to throw runtime errors in case of invalid field values.

The Service layer also performs semantic validations that span multiple records. For example, an `Individual` may not have a birth date in the future. Semantic validations like this are implemented explicitly with Java code in the Service class for each entity. In case of a validation error, these Service classes throw runtime errors that include specific messages like "Individual can not have a birth date in the future."

In any case, validation runtime errors are reported to clients by the Resource layer, using a consistent mechanism which includes the specific error message and an appropriate HTTP status code. See Cross-Cutting Exception Handling, below.

## Output Shallow Copies and Stubs

All records stored in the OpenHDS-Rest database are exposed to clients via the Repository, Service, and Resource layers and REST query API. At the Resource layer, records that should be returned to clients are represented as Java objects which refer to one another by standard Java field references. This presents a problem when marshalling results to JSON or XML for transmission to clients. A naive marshaller might follow memory references indefinitely, producing a very large representation that includes reachable but unwanted entities nested within. Another naive marshaller might not follow any references, producing efficient representations of single objects, but providing no means for clients to discover the relationships between records.

The OpenHDS-Rest implementation makes a compromise by producing shallow copies of records in response to queries. Each shallow copy contains all of the primitive attributes of a given object, but in place of related objects, the shallow copy refers to a stub.  Each stub is an object of the same type as the original, with the uniquely identifying `uuid` value of the original, but all other fields left with blank or default values.

The shallow copying implementation uses Java's Reflection API and depends on the OpenHDS-Rest `UUIDIdentifyable` interface.  Original objects are copied verbatim except for fields of type `UUIDIdentifyable`, which are replaced with stubs that use only the `uuid` value declared in the `UUIDIdentifyable` interface.  Use of the Reflection API allowed shallow copies to be implemented once, in a generic and cross-cutting way.

The shallow copier is invoked just before objects are sent to the marshaller on the way to the client.  Listing 2 shows the JSON representation of a shallow-copied `Individual`.

```
GET http://localhost:8080/individuals/43d59e4f-be6f-4c2e-8454-9eddca557798
```

```
{
   uuid: "43d59e4f-be6f-4c2e-8454-9eddca557798",
   insertBy: {
      uuid: "5a2e20b0-..."
   },
   insertDate: 1458421052.792,
   lastModifiedBy: {
      uuid: "5a2e20b0-..."
   },
   lastModifiedDate: 1458421052.792,
   collectedBy: {
      uuid: "5a2e27cc-"
   },
   collectionDateTime: 1458421052.792,
   extId: "location-3-head",
   firstName: "location-3-head",
   lastName: "location-3-head",
   gender: "FEMALE",
   ...
}
```

*Listing 2. A Shallow Copy. To avoid large result sets with many nested records, the OpenHDS-Rest* `ShallowCopier` *generates stub records. These records may be marshalled automatically to JSON or XML without runaway graph retrieval. Yet, they retain the* `uuid` *fields for related records (highlighted in cyan) allowing reconstruction of the full object graph by clients, if needed. The JSON representation of an arbitrary, stub* `Individual` *record is shown. This is typical of results obtained by GET request from the* `Individual` *resource.*

## Output HATEOAS and HAL

Clients that wish to request data from the server must somehow discover the resource path where appropriate data can be found. One approach would be to hard-code the necessary paths as part of the client implementation. This approach would be brittle with respect to application changes because changes to resource paths on the server side would require rebuilding and redistribution of all clients. This problem would be compounded by the shallow copies and stubs produced by the server. Clients that wish to resolve a stub and obtain the corresponding complete record would have to construct the resource path for each stub.  This would require a brittle client *behavior* in addition to the brittle hard-coded data.  The OpenHDS-Rest REST API addresses both problems by implementing the principle of Hypermedia As The Engine Of Application State (HATEOAS).

HATEOAS makes the OpenHDS-Rest REST API self-describing and interactive by way of hyperlinks that are sent to clients along with requested data.  These links describe the data and also allow the client to navigate to related data or other available resources. Each link is preceded with a relation name that gives a hint to the meaning of the link. This paring of relation names and links is conforms to the Hypertext Application Language (HAL), which is a draft standard for rendering HATEOAS links with content.

Listing 3 shows sample GET output from the OpenHDS-Rest root resource.  This resource returns almost no data of its own, only a greeting and a timestamp.  But it includes numerous links to available resources.  The first link is for the `self` relation, which gives the canonical URL for the root resource itself.  The `self` relation is standard and defined for any REST resource.  It is provided for all resources in the OpenHDS-Rest API.

```
GET http://localhost:8080/
```

```
{
   content: "Welcome.  The Current time UTC is 2016-03-19T23:02:28.133Z",
   _links: {
      self: {
         href: "http://localhost:8080/"
      },
      users: {
         href: "http://localhost:8080/users"
      },
      fieldWorkers: {
         href: "http://localhost:8080/fieldWorkers"
      },
      individuals: {
         href: "http://localhost:8080/individuals"
      },
      locations: {
         href: "http://localhost:8080/locations"
      },
      locationHierarchies: {
         href: "http://localhost:8080/locationHierarchies"
      },
      relationships: {
         href: "http://localhost:8080/relationships"
      },
      projectCodes: {
         href: "http://localhost:8080/projectCodes"
      },
      pregnancyObservations: {
         href: "http://localhost:8080/pregnancyObservations"
      },
      ...
   }
}
```

*Listing 3. Root Resource. The GET response for the OpenHDS-Rest root resource is shown, rendered as HAL and JSON. This resource provides very little data, only a greeting message and a timestamp. Along with the data it provides numerous links to other resources for specific entities of the OpenHDS data mode. The link to the resource for the `FieldWorker` entity is highlighted in cyan. These links make the REST API self-describing, and facilitate client navigation, consistent with the principal of HATEOAS.*

Other links from the root resource use non-standard relations that are specific to the OpenHDS data model. Each of these relations has the plural name of an entity from the OpenHDS data model and links to the specific resource for that entity.

Because the server's root resource provides links to all available resources, clients need only know the URL for this resource, and may discover the locations of all other resources by selecting relations and following the associated links.  Indeed, these links are constructed dynamically from a client's point of view: the base URL for each link is taken from the Host header of the client's request.  In Listing 3, this is "localhost".  In other cases this could be the domain name or numerical IP address by which the client knows the server.

Listing 3 highlights in cyan the link to the REST API's `FieldWorker` resource.  A client might follow this link to obtain more information about registered surveyors.  Listing 4 shows the results of such a GET request.

```
GET http://localhost:8080/fieldWorkers
```

```
{
   _links: {
      self: {
         href: "http://localhost:8080/fieldWorkers{?page,size,sort}",
         templated: true
      },
     ...
   },
   _embedded: {
      fieldWorkers: [
         {
            uuid: "5d649d45-4cb6-4f6c-83e7-4dd5a9e4eeb0",
            insertBy: {
               uuid: "5a2e20b0-..."
            },
            insertDate: 1458421049.685,
            lastModifiedBy: {
               uuid: "5a2e20b0-..."
            },
            lastModifiedDate: 1458421049.685,
            fieldWorkerId: "fieldworker",
            firstName: "default fieldworker",
            lastName: "default fieldworker",
            passwordHash: "...",
            _links: {
               insertby: {
                  href: "http://localhost:8080/users/5a2e20b0-..."
               },
               lastmodifiedby: {
                  href: "http://localhost:8080/users/5a2e20b0-..."
               },
               self: {
                  href: "http://localhost:8080/fieldWorkers/5d649d45-..."
               },
            }
         },
         ...
      ]
   },
   page: {
      size: 20,
      totalElements: 2,
      totalPages: 1,
      number: 0
   }
}
```

*Listing 4. FieldWorker Resource. The GET response for the OpenHDS-Rest `FieldWorker` resource is shown, rendered as HAL and JSON. This resource provides a paged view of the collection of `FieldWorker` records. The `self` link is a templated link, with optional parameters for the collection page number, page size, and sort field. An embedded array of `FieldWorker` records follows the links. Only the first record is*

*shown. It is a shallow copy, with a stub indicating the `uuid` but no other data about the `inserBy` `User` who created the record (highlighted in cyan). Each embedded record is accompanied by its own set of links, including the standard `self` link as well as OpenHDS domain-specific links. The field name `insertBy` appears again as the relation name indicating the URL for the full `User` record (also highlighted in cyan).*

The `FieldWorker` resource and other entity resources present a paged view of the collection of entity records. Each page provides little information of its own: size in records of the page, the total number of elements in the collection, the total number of pages in the collection, and the number of the page itself within the collection. The page's `self` link is a templated link, with optional parameters for the page size, page number, and record sort field.

More information is provided in the page's embedded records. This is an array of records from the entity collection corresponding to the page size and page number. Each of these records appears as a shallow copy, accompanied by its own links. These links include standard relations like the `self` relation and non-standard relations based on record field names.

The use of record field names as link relation names allows the OpenHDS-Rest application to complement the shallow copies and stubs described above. For each field where the implementation identifies a `UUIDIdentifiable` object and replaces it with a stub object, the implementation also generates a link to the resource where the full representation that corresponds to the stub may be found. Indeed, the

`ShallowCopier` and the code that generates these links share most of their implementation.

Link relations named like record fields are non-standard.  In general, non-standard relations may be problematic for clients because the semantics of the link may be undocumented or understandable only to human readers.  But for OpenHDS-Rest, the pairing of non-standard relations with record field names suggests a straightforward interpretation: when a non-standard relation is encountered, it may be mapped to a stub field with the same name.  Since field names must be unique within the scope of a single record, this mapping will be unambiguous.  The GET results from the non-standard link may be substituted for the stub record in the containing record.  Following this interpretation, a client could undo the work of the shallow copier and reconstruct any or all of the object graph stored on the server.  It could do this one record at a time, as needed.

Listing 4 shows an example of the complementary relationship between shallow copies and links.  It highlights in cyan the `insertBy` field of the `FieldWorker` record.  This field contains a stub representation of the `User` who created the `FieldWorker` record. The listing also highlights the `insertBy` relation and link which to the full record of the same `User`.  A client might follow this link to obtain more information about the `User`. Listing 5 shows the results of such a GET request.

```
GET http://localhost:8080/users/5a2e27cc-2b52-11e6-b67b-9e71128cae77
```

```
{
   uuid: "5a2e27cc-2b52-11e6-b67b-9e71128cae77",
   firstName: "User",
   username: "User",
   passwordHash: "***",
   _links: {
      self: {
         href: "http://localhost:8080/users/5a2e20b0..."
      },
      collection: {
         href: "http://localhost:8080/users"
      }
   }
}
```

*Listing 5. User Resource. The GET response for the OpenHDS-Rest `User` resource for a single `User` is shown, rendered as HAL and JSON. This resource provides data and links for a single `User` record, in this case an arbitrary `User` with the name "User". The `self` link, which gives the canonical location of this record, is the same as the requested resource. The `collection` relation indicates the location of the larger entity collection to which this specific `User` belongs.*

## Output Bulk

HATEOAS and HAL provide abundant metadata about the results returned to clients, but in turn this metadata significantly increases the size of the response and imposes a parsing burden on clients in order to distinguish link data from record data. Moreover, in order to obtain the full set of records from an entity collection, a client must perform multiple paged requests. Complicated parsing, and multiple requests may be problematic for portable clients operating on battery power, with slow or intermittent connectivity.

For such clients, the OpenHDS-Rest REST API provides an alternative bulk view of each resource, which may be preferable.  Bulk data are presented as a stream which ranges over the entire result set and omits paging and links.  The streaming presentation allows clients to obtain all results in a single request and to take advantage of the error correction and flow control mechanisms of the TCP protocol.  The simplified representation allows clients to parse records as a single XML document or JSON array which contains shallow-copied records

## Sample Data Generation

In order to populate the server database with records, for testing or production use, a user must submit records that are consistent with server's database schema and the OpenHDS data model.  However, in the case of the OpenHDS-Rest implementation Resource layer, the database schema may be automatically generated and hidden from the user.  In addition, the OpenHDS data model is complex and may be difficult for new users to understand at first.  To address these problems, the OpenHDS-Rest implementation is able to generate valid sample data.

Sample data represent an additional form of self-documentation provided by the OpenHDS-Rest implementation.  The sample records demonstrate a plausible and valid set of operational, Census, and Update records which users may explore interactively. This interactive exploration may be a useful supplement to static analysis of the database schema or other written documentation.

At startup, the OpenHDS-Rest server application may generate sample records suitable for testing and demonstration.  These sample records are used during automatic unit and integration tests that are part of the code distribution.  Sample records may also support end-to-end tests and demonstrations that incorporate connected clients.

Figure 4 shows selected sample data that partition the geography of an imaginary study area.  The partitioning begins with a top-level `LocationHierarchy` node and descendant nodes which fan out to form a complete tree (Figure 4 shows only a representative portion of the complete tree).  Each level of the tree corresponds to a `LocationHierarchyLevel` record, which identifies all hierarchy nodes with the same depth.  Each level represents a partitioning of the study area at a particular level of detail.  The tree of `LocationHierarchy` records is extended to include `Location` records at the lowest level.

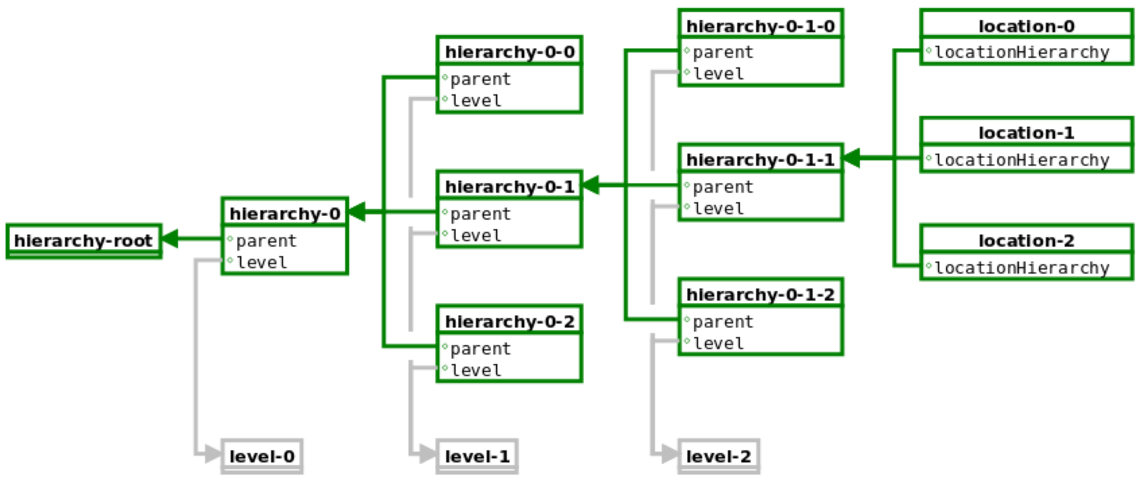


*Figure 4. Sample Geographic Data. Each hierarchy node refers to a* `parent` *node, thus forming a tree structure which can be used to partition geography at several levels of detail. Each hierarchy node also refers to a* `level` *which describes the node's tree depth and may have a mnemonic label. The hierarchy-root is a distinguished node of the hierarchy with no* `parent` *and no* `level`*. Each* `Location` *refers to a* `locationHierarchy` *node, thus extending the tree structure to include data about specific* `Locations`*. In the real sample data set, the number of tree nodes fans out exponentially with increasing tree depth, but this is abbreviated here for clarity. Green boxes and arrows indicate* `Location` *drill-down data that a surveyor might traverse when navigating to a specific* `Location` *or* `Individual`*. Gray boxes and arrows indicate other organizing data.*

The geographic partitioning represented by `LocationHierarchy` records facilitates drill-down navigation by surveyors who may need to identify a single `Location` record from among a large total number of `Location` records. Records involved in in drill-down navigation are shown in green. See Tablet Navigation below.

For the sample data, records have arbitrary names that incorporate numeric suffixes, like "level-2", "hierarchy-0-1-1", or "location-1". For real-world projects, `LocationHierarchyLevel` records might have locally meaningful names, like "state", "county", or "city". Each `LocationHierarchy` node would have a name appropriate for its level, like "Maine", "Cumberland", or "Portland". Finally, each `Location` name would correspond to a structure or other place of interest contained within its nearest `LocationHierarchy` node, for example, "University of Southern Maine".

In addition to geographic partitioning and location data, OpenHDS-Rest may generate sample Household data. As described above, Households are a complex pattern built up from entities in the OpenHDS data model. To facilitate location-based navigation, each Household is associated with a single `Location`. Figure 5 depicts Household data for one `Location` named "location-1". This may be viewed as a continuation of Figure 4, which also depicts "location-1".

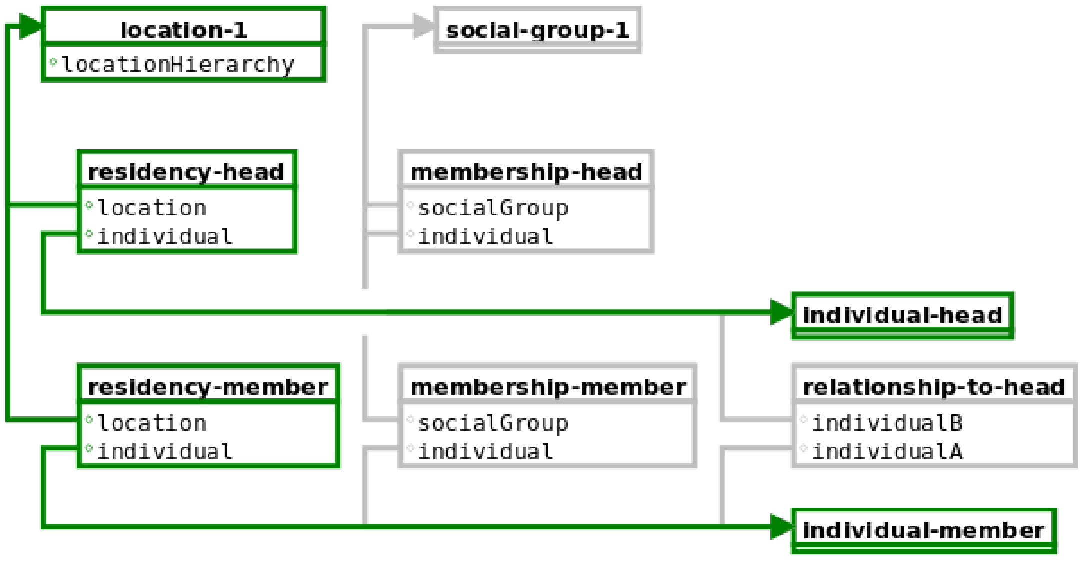


*Figure 5. Sample Household Data. A Household is pattern that groups OpenHDS entities into a typical real-world unit. The same entities could be combined in other ways to represent other situations. A Household comprises a single `Location`, a single `SocialGroup`, an `Individual` who is the head of Household, and additional `Individuals` who are members of the Household. Each `Individual` has a `Residency` at the `Location`, a `Membership` in the `SocialGroup`, and a `Relationship` to the head of Household. Green boxes and arrows indicate `Location` drill-down data that a surveyor might traverse when navigating to a specific `Location` or `Individual`. Gray boxes and arrows indicate other organizing data.*

Drill-down navigation by a surveyor may continue within a Household. Figure 5 shows in green that navigation may proceed from a selected `Location` to the `Individuals` who have `Residencies` at the `Location`. Each sample Household also includes a `SocialGroup`, `Memberships`, and `Relationships`. These complete the Household pattern but are not currently used for navigation.

## Instructive Inheritance

The OpenHDS data model presents challenges for implementation, testing, and documentation because it contains many entities, relationships, and expected behaviors. If each of these were treated separately, the corresponding implementation, test suite, or documentation might become large and difficult to understand. The OpenHDS-Rest implementation addresses this challenge by making extensive use of Java's inheritance mechanism. This approach simplified the system design by factoring out requirements common to some or all entities. This in turn allowed for significant code reuse across implementation layers and test code, and simplified the system documentation.

Inheritance is not a good fit for all design challenges (Gamma, Helm, Johnson, & Vlissides, 1995). For example, Java's single-inheritance mechanism does not allow cross-cutting behaviors to be shared easily across disjoint inheritance trees. But it is a good fit for designs in which all classes share some common behavior, and where variations among classes are strictly specializations of a common superclass.

Such is the case for the OpenHDS-Rest implementation of the OpenHDS data model. All entity classes have in common a `uuid` attribute, which is a unique identifier stable across server and clients. This common attribute is modeled with a `UuidIdentifiable` interface which all entity classes implement. Some entity classes, like `User`, implement only this interface and no further specializing supertype.

Records submitted to the OpenHDS-Rest REST API must be submitted by a `User` in the form of a Registration (see Input Registration above). To enable auditing of these records, the UUID of the submitting `User` is recorded with each new record. This requirement is modeled with an `Auditable` superclass, which extends the `UuidIdentifiable` interface to include an `insertBy` `User` field.

Likewise, records collected by surveyors in the field must include the UUID of the `FieldWorker` who did the data collection. This requirement is modeled with an `AuditableCollected` superclass, which extends the `Auditable` superclass to include a `collectedBy` `FieldWorker` field.

Finally, many entities must be identified with an external id which is human-readable and meaningful to surveyors. This requirement is modeled with an `ExtIdIdentifiable` supertype which extends the `AuditableCollected` to include an `extId` field.

The superclasses `UuidIdentifiable`, `Auditable`, `AuditableCollected`, and `ExtIdIdentifiable` form a chain. At each level, the preceding supertype is specialized by adding a new field which represents a new requirement. The superclasses are not otherwise modified, so that each subtype may stand in as a substitute for its superclass, with no unexpected behavior.

This pattern of chained, strictly specializing inheritance is used in parallel by all layers of the OpenHDS-Rest implementation. The `UuidIdentifiable`, `Auditable`, `AuditableCollected`, and `ExtIdIdentifiable` Repository classes successively add database columns, queries, and indexes appropriate for the corresponding entity classes.  The parallel chain of Service classes successively add appropriate processing side effects and validations.  The parallel chain of Resource classes successively add appropriate resource paths for querying entities.

In addition to the implementation layers, the OpenHDS-Rest code distribution includes unit and integration test superclasses that follow the same inheritance chain.  Thus, tests written for `ExtIdIdentifiable` Services and Resources automatically carry with them the tests written for for `AuditableCollected`, `Auditable`, and `UuidIdentifiable` Services.  This use of inheritance for tests facilitated the development of thorough test coverage.

Table 2 summarizes the OpenHDS-Rest entity classes and groups them by supertype. For each supertype, it shows the REST API resource paths defined by the the corresponding Resource class.

| Entity | Supertype | UuidIdentifiable | Auditable Auditable Collected | ExtId Identifiable |
|---|---|---|---|---|
| | REST API paths available | / /{id} /bulk /sampleRegistraion | /bydate /bydate/bulk /bylocationhierarchy /bylocationhierarchy /bulk /voided | /external/{id} |
| Individual Location Location Hierarchy SocialGroup Visit | | ✔ | ✔ | ✔ |
| ErrorLog Event FieldWorker Location Hierarchy Level Membership Relationship Residency Death InMigration OutMigration Pregnancy Observation Pregnancy Outcome Pregnancy Result | | ✔ | ✔ | |
| ProjectCode User | | ✔ | | |

*Table 2. Instructive Inheritance. Summary of OpenHDS-Rest REST API Entities, Resource Paths, and Polymorphism. Black entity names represent entities used for application logic and field Operations. Green names represent entities that make up the Census or baseline part of the OpenHDS Domain. Red names represent entities that make up the Update or longitudinal part of the OpenHDS Domain. Extensive use of inheritance subtyping simplified the design, implementation, testing, and documentation of the OpenHDS-Rest REST API, including this table.*

For the OpenHDS-Rest implementation, inheritance proved to be expedient and instructive. It allowed for significant code reuse across all implementation layers, and testing. Moreover, it resulted in a simplification of the project design and documentation. Indeed, Table 2 itself owes its compact appearance and simple diagonal structure to the chain-like inheritance used throughout the implementation.

## Cross-Cutting Exception Handling

Previous versions of the OpenHDS server application treated Exception handling and error reporting in an ad-hoc fashion, with separate Exception handling code for each Resource and each Service. This approach was sometimes problematic. It required repetitive code to be written for each Resource class and Service class, making this part of the code base difficult to read. Moreover, there were subtle differences between repetitions, causing unexplained differences in behavior despite similar inputs. From a client's point of view, this implementation approach manifested as an inconsistent error handling and reporting policy.

The PODS OpenHDS-Rest implementation addresses this problem by treating Exception handling as a cross-cutting concern, implemented in a single class in the Resource layer. This Exception Advice class uses Spring MVC annotations to declare a mapping from Java Exception classes to HTTP status codes and methods which produce detailed error messages. For example, authentication and validation Exceptions are mapped to HTTP client errors in the 400 range, while other exceptions are mapped to HTTP server errors in the 500 range.

The Spring Framework catches `Exceptions` from the OpenHDS-Rest Repository, Service, or Resource layers and passes them to the mapped handler methods and response codes. This results in a simple and consistent error reporting policy. For example, `DataIntegrityViolationExceptions` will *always* be reported to the client with HTTP status 409, "Conflict", regardless of which entity Registration or Service class produced the exception.

## OpenHDS-Tablet Client

### Tablet Client Overview

The OpenHDS-Tablet application is an Android mobile application which facilitates the filling out of forms by surveyors during data collection. In general, electronic forms have advantages over paper forms, including input field validation, cross referencing among related forms, and dynamic question sequences based on previous inputs. The OpenHDS-Tablet application supplements these advantages by allowing surveyors to navigate to and select relevant records by touch and automatically filling in many form fields based on the selected records. Automatic form filling increases speed of data entry as well as data integrity, by reducing the amount of typing required of surveyors, and the potential for input typos.

Because surveyors often must work in the field without internet connectivity, the OpenHDS-Tablet application must use a database stored locally on the Android tablet to cache records obtained from the OpenHDS-Rest server. Before data collection, each OpenHDS-Tablet client must download the latest records stored on the server. These records are presented to the surveyor for navigation and form filling during data collection.  Collected forms may be consumed by the tablet application and new records added to the tablet database to facilitate subsequent data collection.  After data collection is completed, each tablet client must upload newly collected data for processing and validation by the OpenHDS-Rest server.

The OpenHDS-Tablet application is responsible for maintaining its local database, and keeping this database synchronized with the OpenHDS-Rest server when connected to the internet.  It is also responsible for creating new XForm instance documents with some elements automatically filled in and for consuming XForm document data after the surveyor has completed data entry.  The application delegates to a separate Android application, ODK Collect, for actual entry of data into XForm instances.

This work on the PODS OpenHDS-Tablet application builds on previous development of the OpenHDS-Rest tablet application.  This work adds new features intended to enhance the communication among the OpenHDS-Tablet clients, the OpenHDS-Rest server, and the ODK Collect application:

- The OpenHDS-Tablet application interprets HATEOAS / HAL links provided by the server in order to select appropriate resources for data download and upload.
- The OpenHDS-Tablet application may request incremental data when downloading records, instead of requesting entity collections in an all-or-none fashion.
- The OpenHDS-Tablet application communicates with the ODK Collection application in order to insert, update, and query form definitions and filled-out form instances.

- Form definitions bundled with the application are designed to resemble Registrations expected by the OpenHDS-Rest server, allowing direct form submissions.
- Form definitions bundled with the application declare Form Behavior metadata which the OpenHDS-Tablet application uses to determine dynamically how to handle each form.
- Form data are mapped between the form documents themselves and the application using flexible rules.

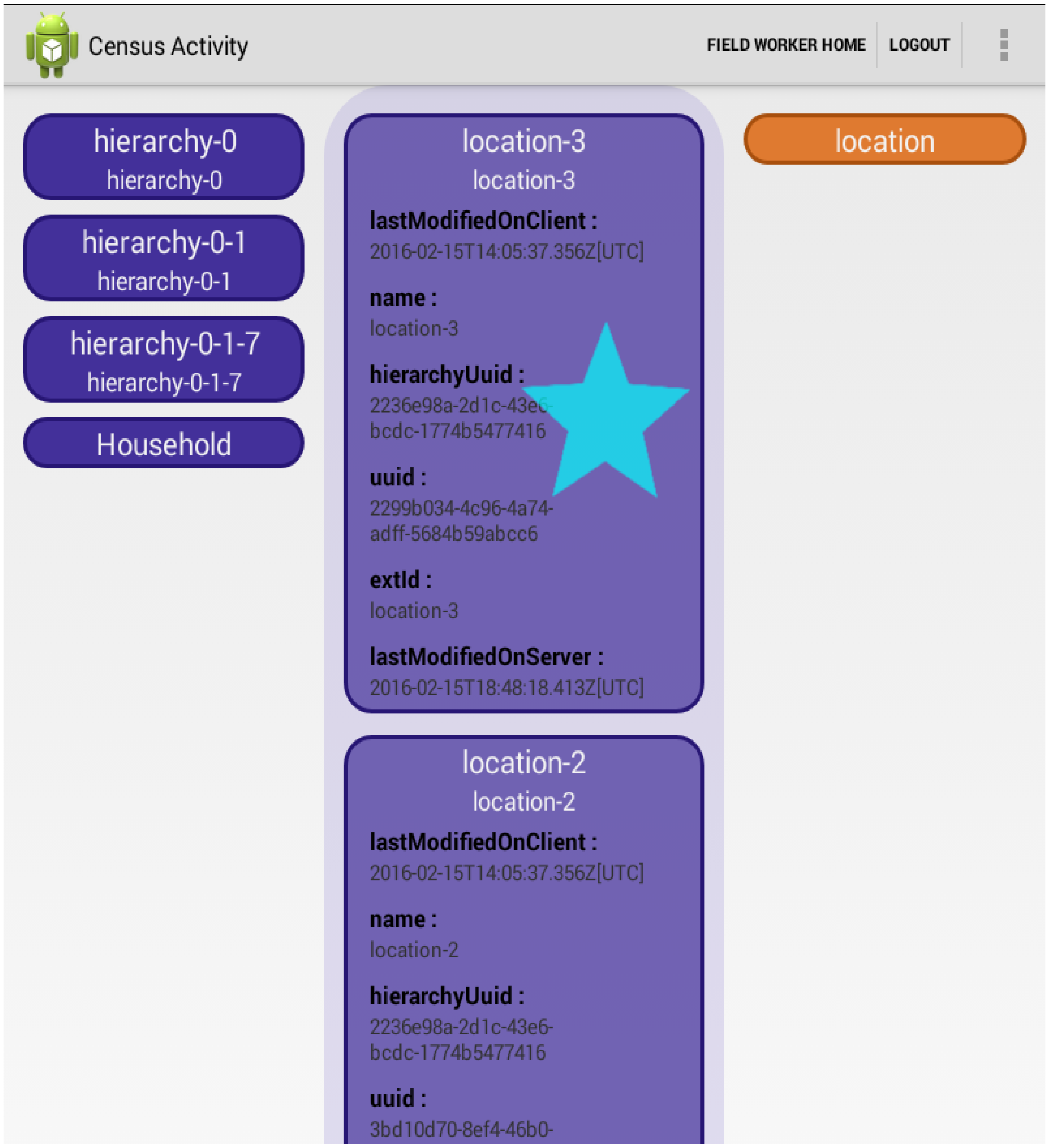
These new features of the OpenHDS-Tablet application, and the general usage of the application, are described in greater detail below.

## Navigation Overview

Logged-in surveyors may navigate through records stored in the OpenHDS-Tablet database. This navigation follows a drill-down path through the `LocationHierarchy`, Household `Locations`, and Household `Individuals`. Figures 6 and 7 show sample data for these drill-down entities, highlighted in green.

During navigation, the surveyor may select various records of interest. For example, a surveyor collecting data at the University of Southern Maine might begin by selecting `LocationHierarchy` records of increasing specificity, like "USA", followed by "Maine", followed by "Portland". These selections represent a single drill-down path through the `LocationHierarchy`. Figure 6 shows a screen capture of the OpenHDS-Tablet drill-down navigation user interface, with a similar drill-down path using arbitrary

`LocationHierarchy` names based on numeric suffixes, instead of meaningful names like "USA", "Maine", and "Portland".



location-3
location-3

**lastModifiedOnClient :**
2016-02-15T14:05:37.356Z[UTC]

**name :**
location-3

**hierarchyUuid :**
2236e98a-2d1c-43e6-bcdc-1774b5477416

**uuid :**
2299b034-4c96-4a74-adff-5684b59abcc6

**extId :**
location-3

**lastModifiedOnServer :**
2016-02-15T18:48:18.413Z[UTC]

location-2
location-2

**lastModifiedOnClient :**
2016-02-15T14:05:37.356Z[UTC]

**name :**
location-2

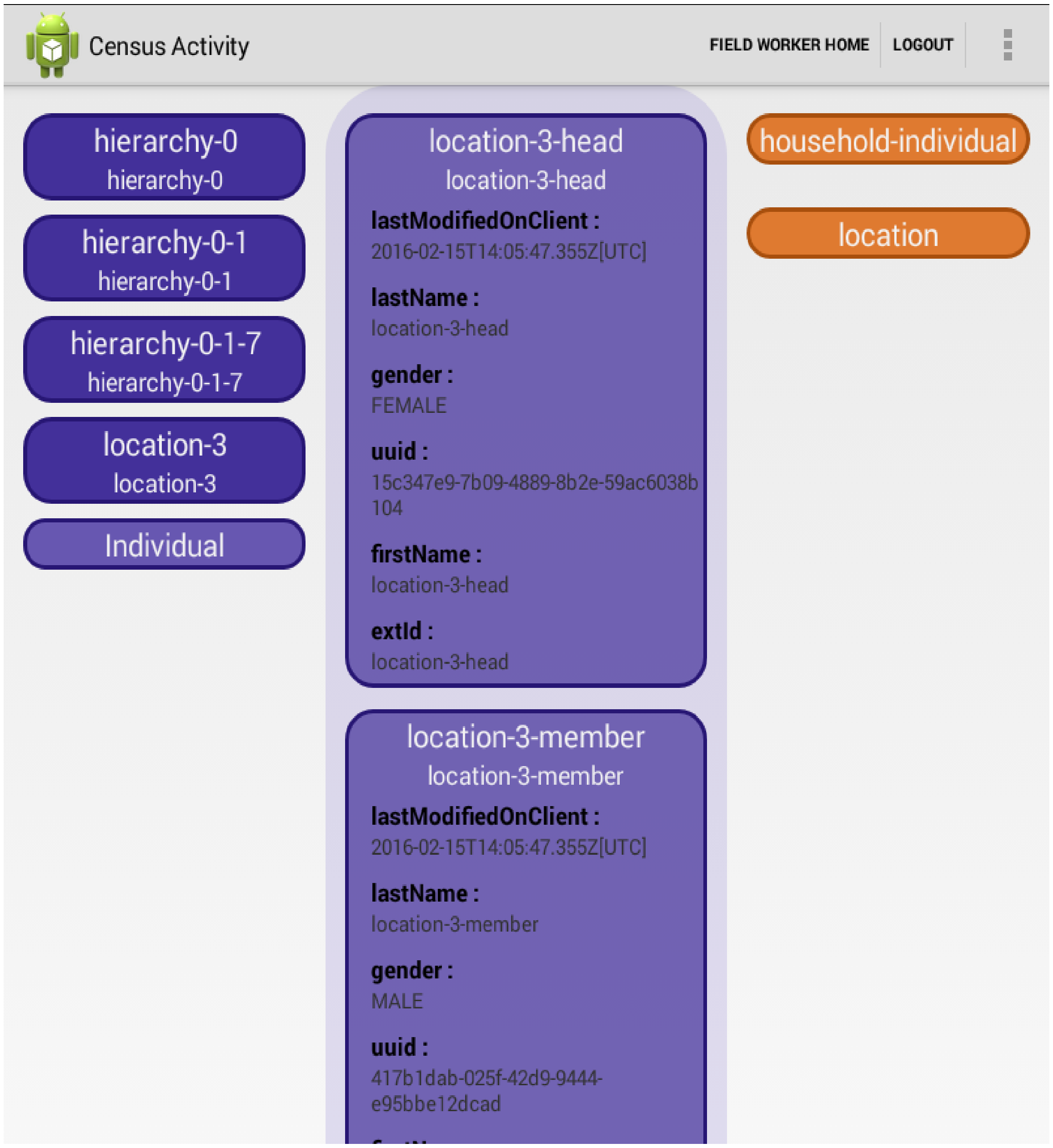
**hierarchyUuid :**
2236e98a-2d1c-43e6-bcdc-1774b5477416

**uuid :**
3bd10d70-8ef4-46b0-

*Figure 6. OpenHDS-Tablet Navigation and Location Selection. A screen capture is show from the OpenHDS-Tablet application's drill-down navigation interface which is used by surveyors during data collection. This example shows automatically generated values with numeric suffixes, like "location-3". During real data collection, surveyors would see meaningful place names like "Portland". The left column displays records that were previously selected during navigation. In this example, three* `LocationHierarchy` *records have been selected, "hierarchy-0", "hierarchy-0-1", and*

*"hierarchy-0-1-7". These indicate that the surveyor has already drilled down through three levels of the location hierarchy. Next, the surveyor may select a Household. Various Household `Locations` are displayed in the middle column. These are the `Locations` associated within the region of hierarchy-0-1-7, at the finest level of geographic partitioning. The blue star on the "location-3" record is not part of the user interface, but indicates the surveyor's selecting of "location-3" as a relevant record. See Figure 9 for the results of this selection. The right column displays the names of XForms which are available for data entry, given the previously selected records. In this case, only the form named "location" is available.*

Following the drill-down through the `LocationHierarchy`, the surveyor may proceed to select a specific `Location`, like "University of Southern Maine". The `Locations` available for selection would be only those `Locations` that belong to the last-selected `LocationHierarchy` record. For example, since the surveyor last selected "Portland", no `Locations` in "Gorham" nor "Lewiston" would be displayed. Figure 9 shows the result of a similar selection for the `Location` with the arbitrary name "location-3".



location-3-head
location-3-head
**lastModifiedOnClient :**
2016-02-15T14:05:47.355Z[UTC]
**lastName :**
location-3-head
**gender :**
FEMALE
**uuid :**
15c347e9-7b09-4889-8b2e-59ac6038b
104
**firstName :**
location-3-head
**extId :**
location-3-head

location-3-member
location-3-member
**lastModifiedOnClient :**
2016-02-15T14:05:47.355Z[UTC]
**lastName :**
location-3-member
**gender :**
MALE
**uuid :**
417b1dab-025f-42d9-9444-
e95bbe12dcad

*Figure 7. OpenHDS-Tablet Navigation and Individual Selection. A screen capture is shown from the OpenHDS-Tablet application's drill-down navigation interface which is used by surveyors during data collection. The user interface is the same as in Figure 6. In this case, the surveyor has just selected "location-3", which now appears in the left column as a previous selection. Next, the surveyor may select an* `Individual`*. Various* `Individuals` *who have a* `Residency` *at "location-3" are displayed in the*

*middle column. Given the previously selected records, the forms named "location" and "household-individual" are available for data entry.*

The progression from `LocationHierarchy` records to `Location` records is natural because each `Location` record has a `locationHierarchy` attribute to associate it with a one `LocationHierarchy` record. Indeed, the progression may proceed further, to include `Individuals` who live at the selected `Location`. This last step is the least direct, because it requires a `Location` as well as `Residency` information in order to determine which `Individuals` should be available for selection -- in relational database terms, it requires a join query.

The result of drill-down navigation is a pathway from Hierarchy root through relevant records, to a `Location` or `Individual`. These relevant records provide the data necessary for pre-filling of XForm instances before presentation to the surveyor for manual data entry.

## Form Pre-Filling and Data Entry

The drill-down user interface presents the surveyor with the option to fill out forms that are relevant given the previously selected records. As shown in Figures 8 and 9, these appear as orange buttons in the right column of the interface. When the surveyor selects one of these buttons, the OpenHDS-Tablet application will create a new XForm instance document of the chosen type.

As far as possible, data elements of the new document will be automatically filled with available data. Sources of available data include the current date and time, the `fieldWorkerId` of the surveyor who logged into the application, and records selected previously during navigation.  See Form Data Mapping below, for details of the mechanism for mapping selected records to XForm document elements.

Automatic filling of form data elements based on selected records is a key feature of the OpenHDS-Tablet application and a significant advantage over paper-based data collection.  Data entry for automatically filled elements is practically instantaneous, speeding the overall data collection process.  At the same time, the potential for data entry errors and omissions is practically reduced to zero.  Rather than a speed-accuracy trade-off, surveyors may enjoy both speed and accuracy.

In addition to the quantitative advantage of speed and accuracy, automatic filling of elements enables a qualitative change in the kinds data elements and forms available. Consider for example an `IndividualHouseholdRegistration` form.  Such forms would have content similar to the `IndividualHouseholdRegistration` shown in Listing 1.  This form contains numerous id fields for records related to the `Individual` being registered.  Each of these ids would correspond to a 32-character UUID which is machine-readable, but decidedly not human readable.  It would be unreasonable or impossible to expect human surveyors to enter multiple such ids manually into a form. The data entry be process would become too slow and tedious.  Worse, the string format of UUID is dense and intolerant of errors, so that even single-character errors could result in unrecoverable corruption of data.

Automatic filling of elements side-steps these human data entry concerns, enabling the OpenHDS-Tablet application to support complex ids like UUIDs, and complex forms like the `IndividualHouseholdRegistration`.

Once the filled-in document is created, the application will use the XForm Instance Gateway to register the new document with the ODK Collect application. The application will use the Android Framework's Intent mechanism to invoke ODK Collect for data entry into the new document, and to process the form after data entry is complete. This processing may include launching of subsequent related forms, creation of new records in the tablet's local database, both of these, or neither. These choice of processing behaviors would be declared in the XForm definition. See Form Behaviors, below.

## Link Relation Interpretations

The PODS OpenHDS-Tablet application developed in this work makes use of HATEOAS / HAL links provided by the PODS OpenHDS-Rest server. These links remove the need for the tablet application to hard-code or construct server resource paths. In addition, the link relation name (sometimes shortened to "rel") provided with each link gives a hint to the tablet about how to interpret the linked resource. However, a relation name is only a hint, and the tablet application still must define behaviors to associate with each rel. To this end, the PODS OpenHDS-Tablet application introduces a `RelInterpretation` class which declares tablet-side behaviors to associate with each server-side relation name.

Each `RelInterpretation` is an object whose "rel" field matches the relation name of a link provided by the server.  Each `RelInterpretation` also declares a parser suitable for reading streaming bulk data from the linked resource, and a database gateway suitable for storing the parsed data in the OpenHDS-Tablet application database.  Figure 8 summarizes these classes associated with the `RelInterpretation` class.

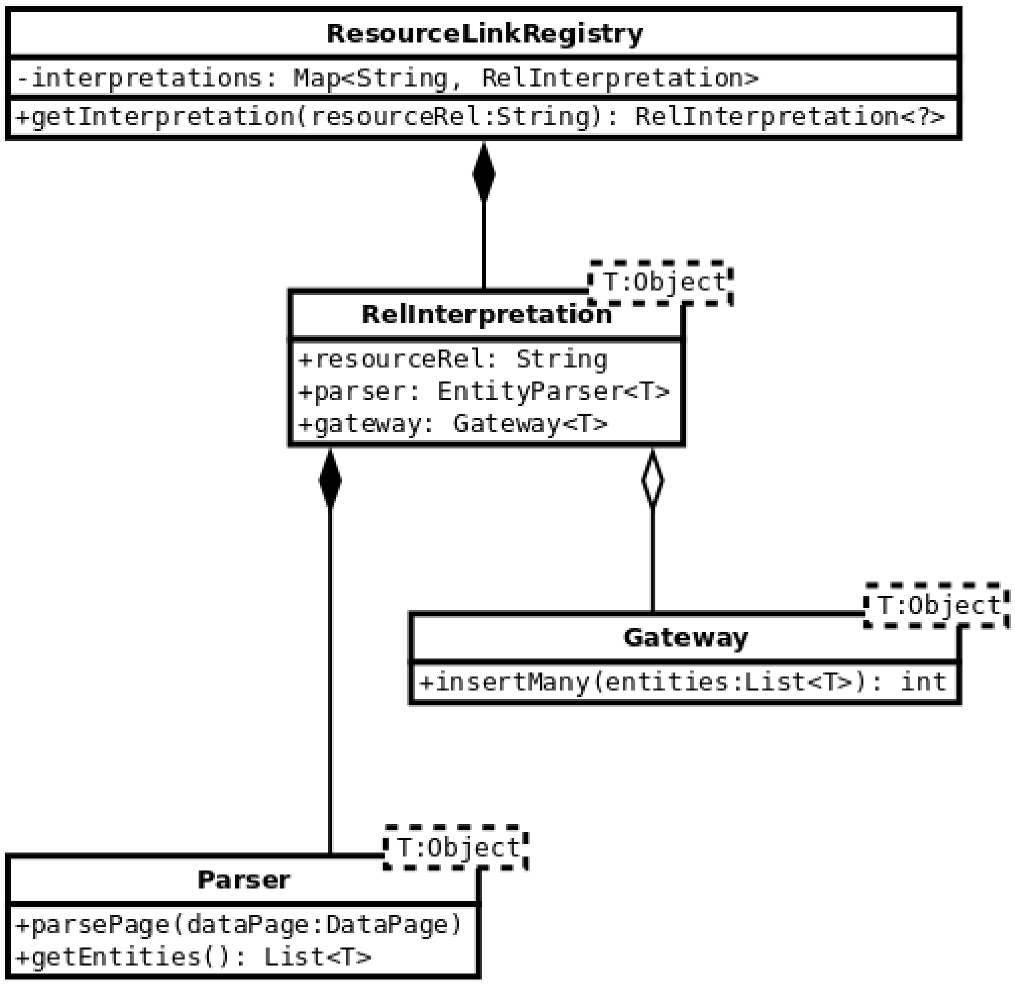

ResourceLinkRegistry
-interpretations: Map<String, RelInterpretation>
+getInterpretation(resourceRel:String): RelInterpretation<?>
T:Object
RelInterpretation
+resourceRel: String
+parser: EntityParser<T>
+gateway: Gateway<T>
T:Object
Gateway
+insertMany(entities:List<T>): int
T:Object
Parser
+parsePage(dataPage:DataPage)
+getEntities(): List<T>

*Figure 8. Class Diagram for Link Relation Interpretation. The Resource Link Registry class acts as a well-known, statically defined holder for RelInterpretations and exposes the getInterpretation() method which allows calling code to obtain the RelInterpretation associated with a particular relation name provided by the OpenHDS-Rest server. Each RelInterpretation is parameterized for an entity type T and holds references to a Parser and the Gateway that are parameterized for the same entity type. These unmarshall and store records from the server, respectively.*

Several tablet `RelInterpretation`s are defined statically, using a declarative style, in a registry class.  This coding style has the advantage of concision and clarity, compared to ad-hoc parsing code distributed about the application.

At runtime, when a User logs into the tablet application, the static `RelInterpretation` declarations are supplemented with actual link URLs read from the server's root resource.  Thus, each client need only know the URL to OpenHDS server root.  The URLs of specific server resources are determined dynamically.

## Downloading Records

Before doing data collection, an administrator User must log in to the OpenHDS-Tablet application and download a set of records from the server for these entities.  In previous versions of the OpenHDS-Tablet application, this step required time-consuming, all-or-none transfer of all records for each entity type.  In addition, the tablet-side parsing of these records was dependent on the specific XML representation and ordering of record fields, and brittle with respect to server-side changes.  These two problems could interact: a parsing error for the first record could disrupt the parsing of all subsequent records, leaving the client with no way to access the subsequent records.

The PODS OpenHDS-Tablet application addresses these download problems in two ways.  First, the application allows incremental downloading of records based on a date selected by a logged-in User.  Only records dated on or after the selected date are send to to the client for parsing.

Second, the PODS application introduces a robust parsing implementation based on a new Page Parser class.

The Page Parser receives a stream of bulk data from one of the OpenHDS-Rest entity resources.  Based on delimiters for XML elements or JSON array elements, the Page Parser breaks the stream into separate pages.  One page at a time is held in memory and passed to an Entity Parser which is appropriate for the stream's entity type.  The Entity Parser uses the page to populate a Domain object of the same entity type.  In turn, each Domain objects is sent to the appropriate database gateway to be stored. Figure 9 summarizes the parsing process for the Location records.

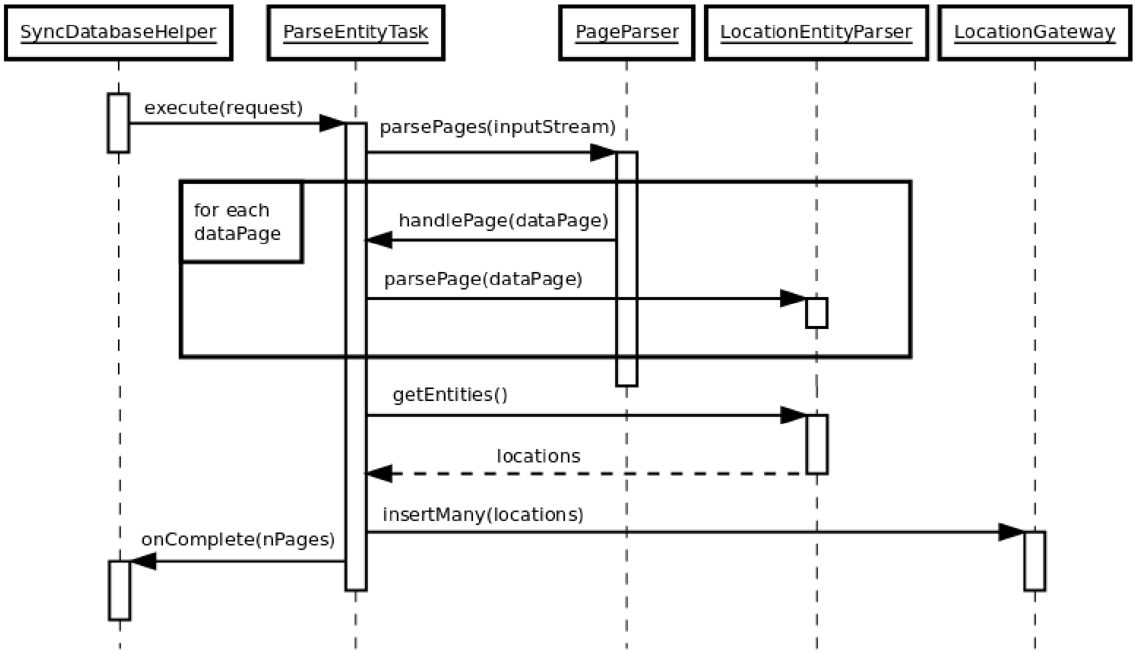


*Figure 9. Sequence Diagram for Download Page Parsing. The SyncDatabaseHelper submits a parsing request to an instance of ParseEntity task. The request includes an inputStream over JSON or XML data, as well as references to a JSON or XML PageParser, a LocationEntityParser, and the LocationGateway. The PageParser parses the inputStream into dataPages and passes each page to the handleData() callback method of the ParseEntityTask. The ParseEntityTask forwards each dataPage to the LocationEntityParser which does Location-specific parsing and maintains a collection of parsed Locations. Parsing continues until the PageParser has exhausted the inputStream. The ParseEntityTask requests the collection of Locations from the LocationEntityParser and forwards the collection to the LocationGateway to be stored. Finally, the ParseEntityTask notifies the SyncDatabaseHelper that parsing is complete, via the onComplete() callback method.*

Breaking the bulk data stream into pages before entity parsing has the advantage of isolating records from one another, and containing errors. The Page Parser minimizes possibility of disrupted stream processing because identifying XML or JSON element markers is simpler than parsing all of the details from XML elements or JSON objects.

## Form Definition and Instance Gateways

The OpenHDS-Tablet application delegates to a separate Android application called ODK Collect to provide two important features:

1. The user interface for XForm data entry.
2. Databases which keep track of XForm definitions and filled-out XForm instances.

Delegating to ODK Collect presents a challenge to the OpenHDS-Tablet implementation. Since multiple parts of the application must interact with ODK Collect, where should the communication logic be implemented? How should communication resources be managed? In previous versions of the OpenHDS-Tablet application, logic and resources were distributed across the code base.

The PODS OpenHDS-Tablet application introduces gateway classes which encapsulate the details of communicating with ODK Collect and present a convenient interface to the rest of the application (Fowler, 2002). Figure 10 summarizes the classes associated with these gateways.

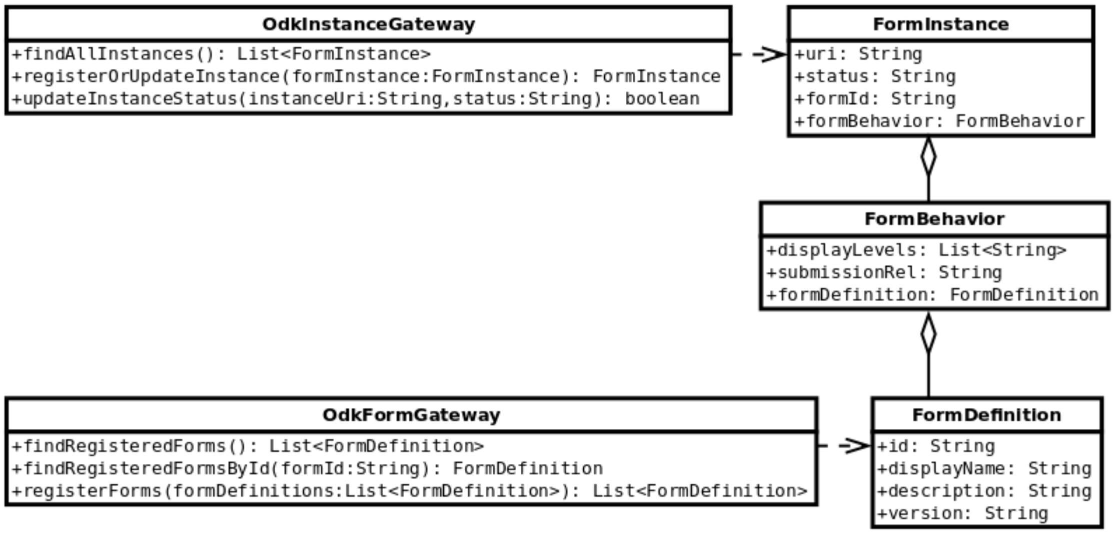

OdkInstanceGateway
+findAllInstances(): List<FormInstance>
+registerOrUpdateInstance(formInstance:FormInstance): FormInstance
+updateInstanceStatus(instanceUri:String,status:String): boolean
FormInstance
+uri: String
+status: String
+formId: String
+formBehavior: FormBehavior
FormBehavior
+displayLevels: List<String>
+submissionRel: String
+formDefinition: FormDefinition
OdkFormGateway
+findRegisteredForms(): List<FormDefinition>
+findRegisteredFormsById(formId:String): FormDefinition
+registerForms(formDefinitions:List<FormDefinition>): List<FormDefinition>
FormDefinition
+id: String
+displayName: String
+description: String
+version: String

*Figure 10. Class Diagram for Form Definition and Instance Gateways. The `OdkInstanceGateway` exposes convenient methods for finding, registering, and updating `FormInstance` records. Likewise, the `OdkFormGateway` exposes convenient methods for finding and registering `FormDefinition` records. Each gateway class encapsulates the details of interacting with the corresponding ODK Collect `ContentProvider` and database. `FormInstance` records represent data entry performed by a surveyor into a specific type of form, at a specific time, and track metadata, such as the data entry completion status. Each `FormInstance` is associated with a `FormBehavior` record which declares behaviors that the OpenHDS-Tablet application should carry out for the form, such as when to display the form during drill-down navigation at which OpenHDS-Rest resource to submit associated form instances. In turn, each `FormBehavior` is associated with a `FormDefinition` record which identifies the type of form and tracks metadata such as the display name, description, and version of the XForm definition document.*

Internally, the gateways use the Android Framework's `ContentResolver` and `ContentProvider` mechanisms in order to insert, update, and query records in the ODK Collect databases of form definitions and instances. These details are hidden from the rest of the application, which sees only convenient interfaces for accomplishing XForm-related tasks.

For XForm definitions these tasks include inserting into the ODK form definition database XForm definitions that were bundled with OpenHDS-Tablet application, and detecting any form definitions that were added to ODK Collect by other means. Figure 11 summarizes the process of querying ODK Collect for available form definitions.

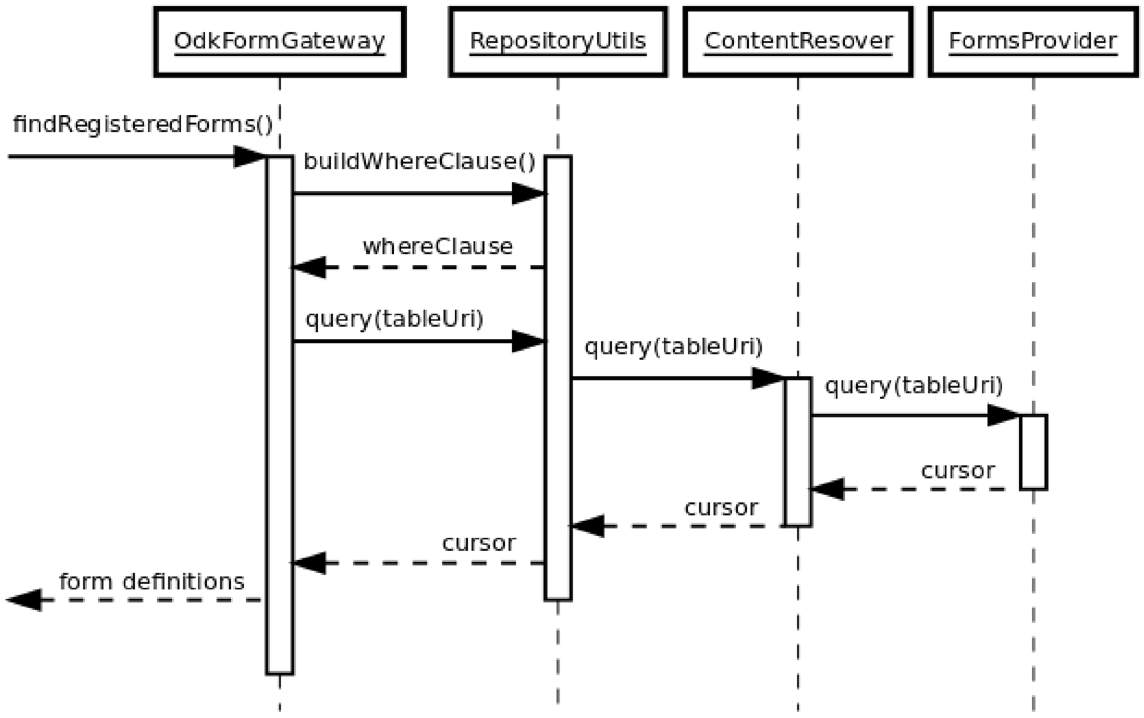


*Figure 11. Sequence Diagram for Form Definition Query. A caller invokes the* `findRegisteredForms()` *method of the OpenHDS-Rest* `OdkFormGateway`*. This gateway collaborates with the OpenHDS-Rest* `RepositoryUtils` *class to construct appropriate query syntax, select a table resource appropriate for the query, and to initiate the query itself. The Android Framework's* `ContentResolver` *forwards the query to the selected table resource, which resides within the ODK Collect application. The ODK Collect* `FormsProvider` *actually carries out the query on the selected table resource and returns results in the form of a cursor. The* `ContentResolver` *returns the cursor back to the OpenHDS-Rest* `RepositoryUtils`*. The* `OdkFormGateway` *reads through the returned cursor to produce a collection of form definitions, and closes the cursor. Finally, the* `OdkFormGateway` *returns the collection of form definitions to the original caller.*

For XForm instances, these tasks include creating new instance documents based on form definitions, pre-filling elements of new documents with data from selected records, registering new instances with ODK Collect to allow data entry by surveyors, and querying and updating the status of each form instances in order to keep track of work in progress, submissions to the server, and errors.

## Form Data Mapping

Two distinguishing features of the OpenHDS-Tablet application are

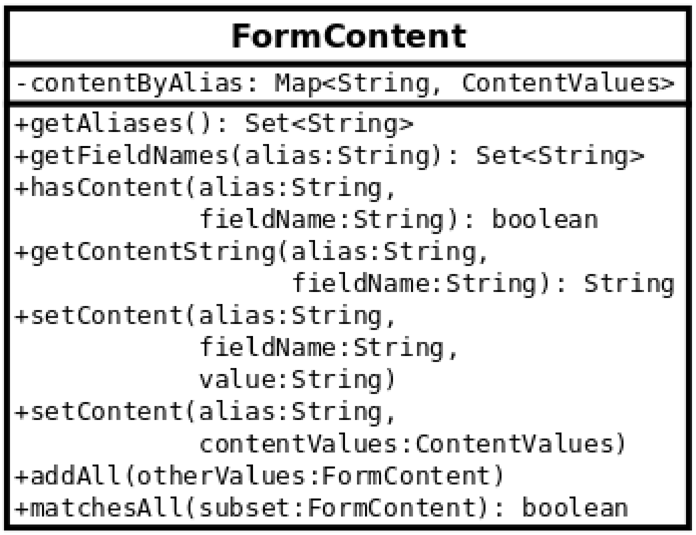
1. Pre-filling of form data fields based on records that were selected by touch.

2. Consuming of data from filled-out forms, in order to update the local database and present new records to the user immediately.

In order to carry out pre-filling and consuming, the application must perform a data mapping between database records and XForm document elements. Previous versions of the OpenHDS-Tablet application relied on explicit, ad-hoc Java code to perform these mappings.  This approach was brittle because the Java code depended on specific XForm definitions and database record fields.  Changes to form definitions or database fields would require modifications to the application source code and distribution of new a new version of the application.

The PODS OpenHDS-Tablet application addresses this mapping problem by introducing a flexible data mapping implementation based on a new `FormContent` class.  This implementation should permit dynamic data mapping which is robust to changes in XForm definitions and database fields.

The flexible data mapping relies on a common representation of data modeled by the new `FormContent` class. `FormContent` is a reusable container which holds string values. Each value is given an identifying field name. Related values may be grouped into records, with each record identified by a name or alias. This grouping yields a two-level structure which allows values to be identified by two coordinates: (record alias, field name). The `FormContent` class provides utility methods for adding, querying, and reading values using these coordinates. Figure 12 summarizes the fields and methods of the `FormContent` class.



```
FormContent
-contentByAlias: Map<String, ContentValues>
+getAliases(): Set<String>
+getFieldNames(alias:String): Set<String>
+hasContent(alias:String,
            fieldName:String): boolean
+getContentString(alias:String,
                  fieldName:String): String
+setContent(alias:String,
            fieldName:String,
            value:String)
+setContent(alias:String,
            contentValues:ContentValues)
+addAll(otherValues:FormContent)
+matchesAll(subset:FormContent): boolean
```

*Figure 12. Class Diagram for Form Content Class. Each `FormContent` instance holds string data, backed internally by a `Map` which allows grouping strings into records that are identified by alias, and by instances of the Android Framework's `ContentValues` class, which allows each string value to be associated with a string field name. The `FormContent` class defines utility methods for working with the two-level container structure. Callers may view stored aliases and field names, query for the values associated with given aliases and field names, and set values by alias and field name. In addition, callers may combine and compare whole `FormContent` instances.*

The two-level `FormContent` structure is sufficient to accommodate data moving to and from the various representations used by the application, including records selected during drill-down navigation, XForm data elements, and the `ContentValues` used by the Android Framework during database insertions. Figure 13 illustrates the two-level structure of `FormContent` and how this fits other representations used by the application.

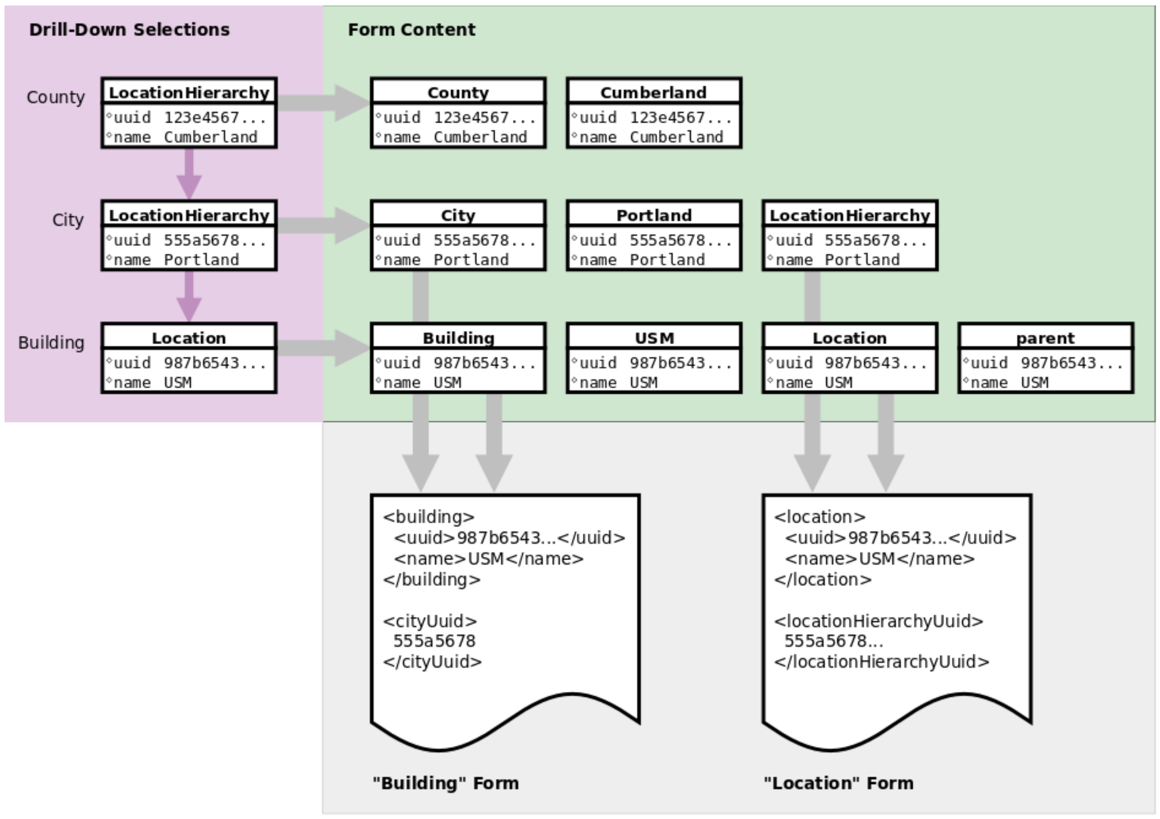


*Figure 13. Data Mapping by Form Content Aliases. Three drill-down selections that might be made by a surveyor are shown in the purple area at the top left. The surveyor might select a County named "Cumberland", a City named "Portland", and a Building named "USM". Selections are added to a `FormContent` container shown in the green area at the top right. Each selection is added using multiple aliases including the literal string "parent", the class name of the selection, the name of the associated `LocationHierarchyLevel`, and the value of the `name` field of each record. When an alias is added more than once, as with the string "parent", the last record added takes the alias, in this case the `Location` record with the name "USM". Data elements for two possible XForm instance documents are shown in the gray area at the bottom. A Building Form names records using uses names of navigation hierarchy levels, such as "city" and "building". A similar Location Form names records using entity names like*

*"locationHierarchy" and "location". Because of multiple aliasing of selected records, data mapping would succeed for either form.*

The mapping process begins by matching top-level fields of the `ContentValues` with any top-level primitive fields of the source or target representation. The matching is simple case-insensitive string comparison. When a match is found, a string value is copied from the source or to the target representation. For drill-down navigation, top-level primitive fields would be the date and time of data collection and the id of the surveyor who is logged in. For XForm instances, top-level primitive fields would correspond to elements that are direct children of the root `data` element. Other representations may not have top-level primitive fields.

The mapping process continues by matching alias names with top-level complex fields of the source or target representation. As with primitive fields, alias matching is simple case-insensitive string comparison. When an alias match is found, primitive matching process is repeated for each field within the scope of the alias. For drill-down navigation, complex fields would correspond to records selected by the surveyor at each level of navigation. For XForm instances, complex fields would correspond to elements that are contained by the children of the root `data` element. Each instance of Android `ContentValues` would itself correspond to a complex field.

Registrations expected by OpenHDS-Rest resources frequently contain top-level fields with names that end with the string "uuid" and values that are the ids of related records, for example `locationUuid` or `motherUuid`. In order to facilitate mapping to and from these fields, the `FormContent` mapping process includes a special case: top-level field names that end with the string "uuid" are treated as equivalent to nested records that contain a single `uuid` field. For example, a top-level field named `locationUuid` would be treated as equivalent to a `location` record which contains a single `uuid` field. Figure 13 illustrates two instances of this special case, for the top-level form fields named `locationHierarchyUuid` and `cityUuid`.

Selection of appropriate aliases is a central problem for the data mapping because there may be multiple appropriate names for a given record. Moreover, various forms and data representations may disagree on the correct name for a record. For example, consider a `LocationHierarchy` record selected by a surveyor during drill-down navigation. In relation to subsequent `LocationHierarchy` records, an appropriate alias for this record would be the literal string "parent", corresponding to the `LocationHierarchy` `parent` attribute which is used to construct the `LocationHierarchy` tree. In relation to subsequent `Location` records, an appropriate alias for the selected `LocationHierarchy` record would be "locationHierarchy", corresponding to the `LocationHierarchy` class name and the `Location` entity's `locationHierarchy` attribute which is used to extend the hierarchy tree to include `Location` records. From a surveyor's point of view, an appropriate alias might be the name of the associated `LocationHierarchyLevel`, such as "state" or "county", to indicate the surveyor's navigation choice at this level. None of these aliases is obviously correct. Indeed, each alias may be the correct one depending on context.

The application resolves this conflict by including each record multiple times, under multiple aliases, within the same `FormContent` instance. A target representation may match any of these aliases for data mapping to succeed, and may be unconcerned with other aliases.

The following aliases are used for each record selected by a surveyor during drill-down navigation:

- The literal string "parent", which is appropriate for the tree structure formed by `LocationHierarchy` records, and by extension, any records presented for navigation in drill-down fashion.
- The name of the associated `LocationHierarchyLevel`, or other level of drill-down navigation, for example "state", "county", or "household".
- The class name of the record, for example `LocationHierarchy`, `Location`, or `Individual`.
- The value of the `name` field for the selected record, if such a field exists, for example, "Maine", or "Cumberland".

Figure 13 illustrates the mapping flexibility afforded by this approach. One set of drill-down selections is applied to two different forms, each written with a different style. In one form, records are named for their entity or class names. In the other form, records are named for their associated `LocationHierarchyLevel`. In both cases, data mapping may proceed automatically.

## Declarative Form Behaviors

Previous versions of the OpenHDS-Tablet application implemented form-related behaviors similar to those described above using ad-hoc Java code. This code depended on specific form definitions and was therefore inflexible -- introducing new forms to the application would require writing new Java code and redistributing the application to surveyors. The PODS OpenHDS system introduces declarative form behaviors as a more flexible alternative. It should be possible to introduce new forms to the application at run time, and the application should respond dynamically by carrying out any new behaviors declared in each form.

The OpenHDS-Tablet application can present a variety of forms to surveyors for data entry. As described above, each form instance would have data elements automatically pre-filled with available data by way of a flexible data mapping process. In addition to pre-filling data elements, the application may carry out a variety of other behaviors for each form instance, for example selecting when to display the form during drill-down navigation, or deciding which records to insert into the tablet database following data entry.

These `FormBehaviors` must be declared in the corresponding XForm definition document, as part of the `meta` form element. This `meta` element is part of the standard XForm schema and used to describe the form definition itself. Listing 6 shows the `meta` child elements introduced in this work which declare the `FormBehaviors` that the OpenHDS-Tablet application should carry out.

```xml
<meta>
    <consumer>individual,residency</consumer>
    <displayLevel>individual,bottom</displayLevel>
    <followUp>
        <formId>pregnancy-observation</formId>
        <filter>
            <isPregnant>YES</isPregnant>
        </filter>
    </followUp>
    <search>
        <IndividualGateway>
            <mother>Individual's Mother</mother>
            <father>Individual's Father</father>
        </IndividualGateway>
    </search>
    <submissionRel>individuals</submissionRel>
    <submissionSubpath>household</submissionSubpath>
</meta>
```
*Listing 6. XForm Form Behavior Metadata. An excerpt from the `IndividualHouseholdRegistration` XForm definition is shown. The `meta` element contains data to describe the form definition itself, in this case by declaring behaviors that the OpenHDS-Tablet application should carry out for forms instance of this type. Six types of behavior are declared, see text.*

At the start of drill-down navigation, the application will use the `FormDefinitionGateway` to query ODK Collect for known form definition documents. As described above, such documents may be bundled with the OpenHDS-Tablet application and registered with ODK using the `FormDefinitionGateway`, or they may be added to ODK Collect by other means. The application will parse each form definition's `meta` element to determine which `FormBehaviors` have been declared, if any. It will cache these behaviors in memory and use them during subsequent navigation. Six types of `FormBehavior` may be declared as child elements of the `meta` element:

1. **`consumer`** -- The `consumer` element may contain a comma-separated list of OpenHDS entity names. Following data entry, the application will map form instance data to a `ContentValues` instance, using the `FormContent`

mechanism described above. It will submit the `ContentValues` to the appropriate database gateway class for insertion into the application's local database. This allows newly collected records to appear immediately in the tablet's drill-down interface, and to be available immediately for pre-filling into subsequent forms.

2. **`displayLevel`** -- The `displayLevel` element may contain a comma-separated list of `LocationHierarchyLevel` names, or names of other drill-down navigation levels at which the form should be presented (as an orange button) to the surveyor for data entry.

3. **`followUp`** -- The `followUp` element may contain the `formId` of a form that should be presented to the surveyor following data entry for the current form. In addition to the `formId`, the `followUp` element may contain `filter` criteria which are compared to form instance data following data entry. When these criteria are present, the follow-up form will be launched only if all of the declared filter elements match the filled-out data elements. Listing 6 shows an example where a "pregnancy-observation" form would be launched, but only if the `isPregnant` data element is filled out with the value of "YES".

4. **`search`** -- The `search` element may contain the names of database gateway classes which should be used for text-based searching of records that will supplement the records selected during drill-down navigation. Each gateway element must contain one or more elements that specify the name of a data element that should receive the search result, and a string label to present to the surveyor during the search. When these elements are present, the application will display a separate search user interface to the surveyor before invoking ODK

Collect for data entry.

5. **`submissionRel`** -- The `submissionRel` element may contain the name of a HATEOAS / HAL link relation name reported by the OpenHDS-Rest server from its root resource. When this element is present, the form instance document may be submitted as a Registration to the corresponding OpenHDS-Rest server resource. XForms intended for submission to server in this way must be designed so that their data schema resembles the XML representation of the Registration expected by the resource. See Input Registrations, above.

6. **`submissionSubpath`** -- The `submissionSubpath` may contain an additional resource path to append to the URL of the server resource that corresponds to the `submissionRel` element. This is usually unnecessary. For the example in Listing 6, the `submissionSubpath` is necessary to distinguish the complex `IndividualHouseholdRegistration`, which is accepted at the `household` sub-path, from the simpler `IndividualRegistration` which is accepted at the root of the `Individual` resource.

All, any, or none of these elements may be declared within the `meta` element of an XForm definition document.

## Form Review and Submission

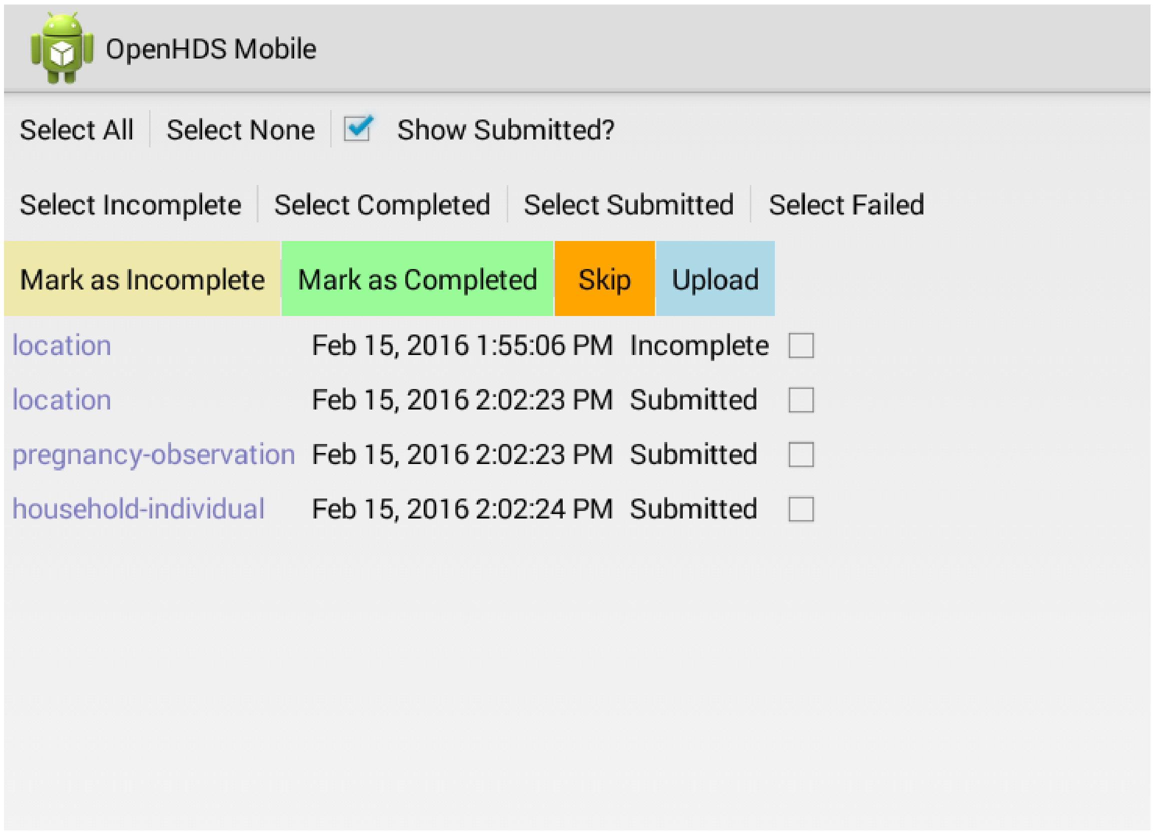
Previous versions of the OpenHDS system used an indirect mechanism for submitting form instances to the server. Rather than direct submissions from the OpenHDS-Tablet application to the server application, the system used an intermediate application, ODK

Aggregate, to transmit instances from the ODK Collect application to the ODK Aggregate server application.

This work introduces as an alternative, the ability to submit form instances directly from the OpenHDS-Tablet application to the OpenHDS-Rest server application.  This direct approach has the advantage of immediate feedback from the server and the ability to review errors and correct them directly on the tablet, using the same ODK Collect application that was used for initial data entry.  This direct mechanism could function as a replacement for the the indirect ODK Aggregate mechanism.  Or, the two mechanisms could work in parallel, depending on project requirements.

Following data collection, an administrator User may log in to the OpenHDS-Tablet application to review the collected XForm instance documents and submit them to the OpenHDS-Rest server.  This highlights a separation of roles between surveyors who perform data collection offline, and administrator Users who are permitted to submit data to the OpenHDS-Rest server when connected to the internet.

Figure 14 shows a screen capture of the OpenHDS-Rest form review and submission user interface, as displayed to a logged-in User.  A table lists all the form instances collected by surveyors and shows the type of each form, the date and time of data collection, and form status as reported by ODK Collect via the application's `FormInstanceGateway`.  Using checkboxes and utility buttons, the User may select a subset of form instances and perform operations on them, such as changing their ODK Collect status or uploading them to the OpenHDS-Rest server.



| | | | |
|---|---|---|---|
| location | Feb 15, 2016 1:55:06 PM | Incomplete | ☐ |
| location | Feb 15, 2016 2:02:23 PM | Submitted | ☐ |
| pregnancy-observation | Feb 15, 2016 2:02:23 PM | Submitted | ☐ |
| household-individual | Feb 15, 2016 2:02:24 PM | Submitted | ☐ |

*Figure 14. Form Review and Submission User Interface. A screen capture of the OpenHDS-Tablet form review and submission user interface is shown. A table at the bottom lists four form instances: two for the form named "location", one for "pregnancy-observation" and one for "household-individual". Form type labels appear in blue and act like hyperlinks: pressing the allows the `User` to view and edit the form using ODK Collect. The date of data collection is displayed next to the type of each form. The status of each form as reported by ODK Collect is shown next to the date: one "Incomplete" and three "Submitted". A check box on the far right of each form listing allows the `User` to select and deselect specific form instances. Gray utility buttons at the top allow the `User` to select multiple forms at a time including "Select All", "Select None", "Select Incomplete", "Select Completed", "Select Submitted", and "Select Failed". A check box at the top allows the `User` to choose whether or not forms with the "Submitted" status should be displayed in the table. Colored buttons allow the user to*

*set the ODK Collect status of all selected forms. The yellow button sets for statuses to "Incomplete". The green button sets form statuses to "Completed". The Orange button sets form status immediately to "Submitted" but does not perform actual form submission, effectively skipping submission for selected forms. Finally, the blue button uploads selected forms to the appropriate resources of the OpenHDS-Rest server. For each uploaded form, if the upload was successful, the ODK Collect form status is set to "Submitted"; if the upload was unsuccessful, the ODK Collect form status is set to "Failed". In either case, after an attempted upload, each form instance's status label may be pressed to display the success or error message from the server.*

In order for upload to the server to succeed, the definition document associated with a form instance must declare a `submissionRel` metadata element which indicates which OpenHDS-Rest server resource should receive the instance document. See Form Behaviors, above. Following an attempted form submission, the application will capture the response message from the server and make it available for display to the `User` upon pressing the status label for the form instance.

In case of a submission error, the user may be able to read the error message from the server and learn the cause of the failure. The application provides a simple mechanism by which `Users` may be able to correct such errors: upon pressing the form id label for a particular form instance, the application will use an Android Intent to invoke the ODK Collect application for editing of the same form instance. The `User` may be able to update the instance data, return to the form review and submission interface, and submit the form instance again.

## Web Clients

In addition to the OpenHDS-Rest server application and OpenHDS-Tablet application, this work on the PODS OpenHDS system should support the future development of browser-based web clients. These clients would operate online with internet connectivity and could therefore take advantage of the OpenHDS-Rest server's self-descriptive features in ways that are not possible for the offline tablet client.

The HATEOAS / HAL links supplied by OpenHDS-Rest resources should facilitate the development of web clients by others. As shown in Listings 2, 3, 4, and 5 clients that know how to reach the server's root resource may navigate from there to all other resources, simply by following links. Indeed, libraries like angular-hateoas provide the ability to automatically follow HATEOAS / HAL links for user clients developed with the AngularJS framework (Marquis, n.d.)(Google & Others, n.d.).

An advantage for connected, browser-based web clients is the ability to interact with the `User` between requests. This affords the `User` a chance to read and interpret the HATEOAS / HAL relation names in order to decide which links to follow.

In addition to HATEOAS / HAL links, the server also supplies Cross-Origin Resource Sharing (CORS) headers with responses from each resource. CORS headers will allow the development of clients that are developed and hosted separately from the server itself. Without these headers, standard-compliant web browsers would disallow the necessary cross-origin requests. That is, requests from a client that originates from one host, to a resource like OpenHDS-Rest which originates from a different host.

Sample Registrations provided by each OpenHDS-Rest resource should also facilitate development of client applications.  These applications may dynamically request sample Registrations and other sample data and present these as templates for the `User` to edit.  This approach would be preferable to alternatives like guessing at the input format expected by the server, or hard-coding static input templates into the client application.

## Design Themes

The four PODS design themes guided the design and implementation of the PODS OpenHDS system. The themes are:

1. Applications should take advantage of *polymorphic* design.

2. Developer *operations* should be prioritized so that tasks like deployment, updates, and maintenance do not overwhelm an otherwise functional system.

3. Applications should be written and configured favoring a *declarative* style.

4. Applications should be *self-descriptive*.

Each theme warrants some consideration for its own sake. Examples will be enumerated of how each theme was applied to this work and the expected advantages.

### Polymorphism

Polymorphism is a fundamental programming concept, allowing various program components or data that express a common interface to be treated interchangeably by the program (Cardelli & Wegner, 1985). In general, polymorphism may yield advantages like code reuse and shortened development time. In addition to these practical advantages, embracing polymorphism may prove instructive to program designers, implementers, and developers.

The OpenHDS-Rest application implemented for this work makes extensive use of Java's mechanism for polymorphism by class inheritance. Each implementation layer of the application uses a similar chain of superclasses for entities that are `UuidIdentifiable`, `Auditable`, `AuditableCollected`, and `ExtIdIdentifiable`. For each layer, this permitted significant code reuse since common behaviors could be implemented once, at the superclass level, instead of multiple times, at each concrete subclass. This code reuse likely saved development time relative to independent implementation without code reuse.

Use of inheritance simplified not only the source code but also the application's REST API, its testing, and its documentation. Test reuse resulted in broad test coverage of the API, which should support future application development and robustness. Simplification of the REST API and its documentation should facilitate client development and learning of the API by users.

The OpenHDS-Tablet application developed for this work uses an adaptor mechanism to achieve a form of ad-hoc polymorphism(Cardelli & Wegner, 1985). To simplify mapping of data between various representations, the application uses a common `FormContent` representation. This polymorphism is beyond the reach of Java language inheritance mechanism because the polymorphic data representations extend beyond the Java language to XForm definition documents, OpenHDS-Rest Registrations, and database records. Instead, the common interface exists at the design level, where each representation has a simple two-level structure with named primitive data fields and complex records or objects which may be identified by aliases. The polymorphism is realized by Java code that maps each representation to and from instances of the `FormContent` class.

This polymorphic mapping mechanism eliminated a significant amount of explicit Java mapping code from the tablet application.  This code cleanup should make the application easier for future developers to understand and modify.  More importantly, this mapping mechanism is flexible and will apply to new XForm definitions added to the system, without incurring Java development or distribution of a new version of the application.

## Developer Operations

Application development projects should prioritize developer operations in addition to specific domain or implementation concerns.  Operations concerns include things like software building, testing and debugging, deployment and startup requirements, and the number and arrangement of applications that must interact.  These operational concerns cut across other domain- or implementation-specific concerns because developers must address them first, and repeatedly, in order to address other concerns.

This work on the PODS OpenHDS-Rest server application and OpenHDS-Tablet client application introduces several changes to the OpenHDS system aimed at smoothing developer operations:

- **Build with Spring Boot** -- The OpenHDS-Rest application uses the Spring Boot feature of the Spring Framework to simplify application configuration, selection of dependency versions, building, and packaging.  This eliminates most of project's build configuration files in favor of conventions selected by the Boot project.

- **Simple Startup** -- The OpenHDS-Rest application is easy to start up, since initial configuration and creation of necessary records, like the default User, are handled by the application itself.  Startup does not require database initialization with special scripts, nor user interaction through a graphical interface. This simplicity allows automated systems to start the application in order to run tests or deploy to the web. Such systems include Travis CI for automated testing, and Heroku for automated web deployment.

- **Configure by REST** -- As with initial startup, configuration of the application, such as the specification of `ProjectCode` constant values and creation of Users and Field Workers, may be done entirely through the OpenHDS-Rest REST API. Neither special database scripts nor command line server access are required.

- **Bundle Basic Forms** -- XForm definition documents may be bundled with the OpenHDS-Tablet application and registered with the ODK Collect application by a logged in User. The ODK Aggregate server application is not required for XForm registration, although this is also possible.

- **Direct Duplex** -- The OpenHDS-Tablet application may communicate directly with the OpenHDS-Rest REST API. This direct communication allows `Users` to see immediate status messages from the server in response to form submissions, and to attempt to correct errors immediately. This communication should facilitate application debugging, project configuration, and actual data collection. No additional, intermediate applications are required to submit data to the OpenHDS-Rest server, although submission by indirect means is also possible.

A result of all these changes is that the PODS OpenHDS system is straightforward to build, deploy, and test on a single development machine without special client or server requirements.  In particular, a server with a public domain name or IP address is not required.

This theme, that developer operations matter, is of high importance for software development projects because operational concerns may impede or enhance other concerns.  In the worst case, an application that is well-designed and implemented from a domain point of view may nevertheless be unusable because the application is too difficult to build, install, or configure.  In the best case, an application that is easy to modify, build, test, and debug may be able to help developers and domain experts to identify, express, and implement application requirements in an iterative fashion.  In either case, the operational concerns are unavoidable and influential.

## Declarative Style

Applications should support a declarative style for control, programming, and configuration, especially for parts of an application which will be exposed to users or will make up part of the application's external API.  As compared to the imperative style, the declarative style offers an explicit separation of intention or goal from implementation.  For users who are concerned with what an application will do, and who are unconcerned with the steps taken internally do it, the declarative style should yield advantages of readability and understandability (Fahland et al., 2009).  Furthermore, this separation allows the same intention or goal to be declared across components or versions of an application, and across applications.

Facilitating control and configuration of the OpenHDS system is important for projects like CIMS, which must extend the system to meet project-specific requirements, and must make changes to the system over the duration of the project.

The PODS OpenHDS-Rest and OpenHDS-Tablet applications developed for this work use this theme of declarative style in three ways:

1. Input Registrations like the `IndividualHouseholdRegistration` demonstrate a declarative style of application control.

2. `FormBehaviors` declared as XForm metadata demonstrate declarative configuration of the tablet application.

3. The mapping of cross-cutting mapping of Java `Exception` class to HTTP response status demonstrates a declarative style of programming.

Input Registrations like the `IndividualHouseholdRegistration` represent commands that clients may send to the OpenHDS-Rest server in order to control its behavior. These commands use a declarative style. They specify data to be recorded, along with appropriate side effects and validation, but they do not specify the manner in which these actions should be carried out. Carrying out the intention of the command is left as an implementation detail of the server. This separation of command from implementation is not merely academic (Dijkstra, 1982). One practical advantage is that multiple clients may express the same commands to the server, and allow the server to respond to each client in a uniform manner.

The use of Unknown and placeholder records by the server represents a further separation of command from implementation. These special types of record allow clients to submit related records in any order, with eventual consistency and without errors from unsatisfied dependencies. Clients have only to express the goal that several records be recorded, and need not implement the details of dependency sorting and resolution.

`FormBehaviors` declared in XForm definition documents represent declarative configuration of the tablet application. The application itself implements various behaviors that it may perform with forms in general. It is left to each form definition to declare whether and how these behaviors should be applied to its form instances. Again, the separation of behavior implementation from behavior declaration should make the application easier to configure for new and evolving projects. For new projects, system administrators must be able to read, understand and write behavior declarations in XForm definition documents. This task is modest compared to the task of reading, understanding, and modifying the corresponding implementations in Java code.

Declarative form behaviors extend beyond the quality of the tablet source code to reach dynamic extensibility of the PODS OpenHDS system. It would be possible for the OpenHDS-Rest server to be updated with a new resource, and a new XForm definition added to the tablet which declares this new resource as its destination. The tablet application should detect both of these changes and send instances of this new form to the correct server resource. This should require no development or redistribution of the tablet application.

Cross-cutting `Exception` handling by the OpenHDS-Rest server's Resource layer represents a declarative style of programming. Instead of writing ad-hoc `Exception` handling code throughout the application, the OpenHDS-Rest server uses Java annotations to declare a mapping from Java `Exception` class names to HTTP response codes and handler methods. The mapping occurs in a single class file, which should make the application's exception handling policy easy to read, understand, and modify when necessary. In addition, cross-cutting approach should make the applications `Exception` handling behavior uniform and easy for clients to understand. The implementation details of catching `Exceptions` and invoking declared handler methods are rote, uninteresting to most users and developers, and implemented separately by the supporting Spring Framework.

## Self-Description

Applications should be self-descriptive for the same reason that applications require separate written documentation -- to instruct users in their use. Separate documentation may be problematic because it may be incomplete or incorrect to begin with, and may go out of date if it is not maintained in parallel with the application during development. Tools like Doxygen aim to facilitate documentation updates by incorporating documentation directly in source code (van Heesch, n.d.). But these tools only facilitate and do not guarantee that documentation will be updated, or that updated documentation will be distributed to users. Indeed, some users may be required to use an application without having access to or knowledge of the separate documentation.

The theme of self-description is one way to address these problems by integrating instructive features into the application itself. At a minimum, this might include bundling written documentation as part of the application installation process. This theme may be extended by allowing applications to expose their workings and data to users at runtime. This may require application features that are separate from the immediate business requirements of the application. Nevertheless, these features may provide valuable support to business features, since they will help users understand how to use the business features.

The OpenHDS-Rest application provides three such features which are self-descriptive and instructive but not strictly business requirements. These are the HATEOAS / HAL links provided to clients along with records from each REST resource, sample Registrations which clients may request from each resource, and sample geographic and Household data which the server may generate at startup and expose to REST clients.

The inclusion of HATEOAS / HAL links with responses to clients should facilitate the development of browser-based web clients by removing the burden from clients of recording which resources are available on the server, or what is the path to each resource. Instead, the server's root resource may report all of this information on demand. This benefit also applies to the OpenHDS-Tablet client. During this work, several hard-coded resource paths were deleted from the source code and replaced with a single class which parses the relations and links reported by the server.

Sample Registrations provided by each OpenHDS-Rest resource should facilitate development of browser-based web clients by providing templates which users may fill out in order to add data to the system. Client developers would then be freed from the burden of finding and understanding documentation about expected Registrations, and realizing this understanding in client application code. Instead, the task of client development would be reduced to presenting Registration templates, allowing users to edit them, and submitting the edited templates to the server. This benefit would extend to client maintenance as well as initial development. If the OpenHDS-Rest server were updated and changes made to expected Registrations, browser-based web clients could detect these changes dynamically as changes to the sample Registration templates that they receive from the server.  All such clients would see these changes at the same time. A given client could continue to interoperate with multiple versions of the OpenHDS-Rest server without requiring development and redeployment effort, or conditional logic based on server version.

Finally, sample data generated at startup by the OpenHDS-Rest server should prove instructive to users unfamiliar with the OpenHDS data model.  A server deployed with sample data represents an interactive instruction tool which users might use to replace or reinforce instruction based on written documentation about the OpenHDS data model. For example, it may not be obvious based on the schema how to construct a valid data set that satisfies all constraints on record attributes and relationships.  Since the sample data provided to the user are the same data used for the application's internal integration tests, the sample data represent an interactive existence proof of a well-constructed data set.

Sample data generation may combine with non-standard HATEOAS / HAL relations to provide an especially instructive interactive tool.  The application provides non-standard link relations which indicate the relationships among records.  A user may observe not only that an `Individual` has a `mother` attribute or a `residency` attribute, but may also follow the "mother" or "residency" link to explore those related records.

# Future Work

This work makes significant changes to the OpenHDS system, with a view to improving the system design, development, and operational characteristics.  Development may continue for the PODS OpenHDS system, and also for other versions of OpenHDS.

Several opportunities for future work have become apparent during this work, and would be natural continuations of this work.

## Application of PODS to Existing Systems

The PODS OpenHDS system represents a self-consistent distributed application, which may be suitable for future public health projects.  Because of the significant architectural changes, this new system is not suitable as an in-place substitute for existing projects that use older or site-specific OpenHDS implementations.

Nevertheless, ongoing projects which were begun with other versions of the OpenHDS might be able to incorporate some aspects of this work.  The incorporation process would require system configuration and deployment effort, as well as Java development effort.

Portions of the code may be useful for existing implementations. It might be possible for ongoing projects to incorporate the self-descriptive HATEOAS / HAL link generation into the REST API of the OpenHDS-Rest server.  The essential components for this are the Spring Framework's HATEOAS package and annotations, the several classes developed in this work including the `ShallowCopier` `LinkRegistry`, and

`EntityLinkAssembler`. These components use the Java Reflection API in order to function in a generic, reusable way and avoid intrusion into the rest of the project code.

Changes required to incorporate these components would include adding Spring HATEOAS annotations to Resource classes, adding `EntityLinkAssembler` method calls to rest endpoint handler methods, and refactoring Domain classes to use the `UuidIdentifiable` interface which the `ShallowCopier` uses to identify related records and generation stub objects.

Other aspects of the OpenHDS-Rest server implementation are thoroughly integrated and would be difficult to incorporate in other projects.

The OpenHDS-Tablet application developed for this work assumes that its server counterpart will provide HATEOAS / HAL descriptions of available resources. Aside from this assumption, the tablet application should be available for incorporation into ongoing projects. This might allow ongoing projects to take advantage of declarative form behaviors and flexible data mapping.

Incorporating the new OpenHDS-Tablet application into an existing project based on a previous version of OpenHDS would require reimplementing the client-server communications, to use fixed, known resource paths instead of paths provided as HATEOAS / HAL links. Because older versions of the OpenHDS server application don't expect direct form submissions from the client, it would be necessary to use an indirect mechanism for form submission, perhaps including the ODK Aggregate and other server-side applications.

## Web Clients and Security

As discussed above in Web Clients, the OpenHDS-Rest server developed in this work is intended to support multiple browser-based web clients. Development of the these clients should be facilitated by the self-descriptive features of the OpenHDS-Rest server, including HATEOAS / HAL links, sample Registration templates, and sample data generation. These clients could be developed and hosted separately from the OpenHDS-Rest server because the server provides CORS response headers to satisfy web browser security standards related to cross-origin requests. Web clients would be natural extensions of this work.

Development of multiple browser-based web clients might be accompanied by application `Users` with various project `Roles` and `Privileges`. The current OpenHDS-Rest implementation defines a security model which requires `Users` to authenticate with username and password, and which associates `Users` with flexible sets of `Roles` and `Privileges`. However, the current implementation does not demonstrate a consistent `User` authorization policy and user permissions may not be checked consistently. A natural and necessary continuation of this work would be to revisit the Resource and Service implementation layers to apply a consistent authorization policy based on `Roles` and `Privileges`.

## Queries and Optimizations

For auditing and reporting of project data collection, it might be necessary to report which `User` or which surveyor initially inserted, modified, or collected a record. The current OpenHDS-Rest implementation provides this information as attributes of

`Auditable` and `AuditableCollected` entities, which includes most of the entities in the system. However, the current implementation does not support queries for records based on these attributes. For example, it it is not possible to request the `Individual` records, and only those `Individual` records, that were collected by a particular surveyor, or last modified by a particular `User`. A natural extension of this work would be to add `User`-based and `FieldWorker`-based queries to the Service and Resource layers.

The current implementation supports location-based queries, for example, all the `Location` records, and only those `Location` records, fall within the partition region of a given `LocationHierarchy` record. While functional, these query implementations are inefficient, requiring multiple database queries and manipulations of results sets per request. The central difficulty is that the location-based queries require recursive traversal of the `LocationHierarchy` tree structure, in addition to database queries and joins. Recursive searches are not generally expressible with typical query languages like SQL. Supporting efficient location-based queries would be a natural extension of this work, and would require some additional research and database design (Celko, 2012).

All resources of the OpenHDS-Rest REST API may send and receive data represented as JSON or XML. These data representations are ubiquitous and should be appropriate for many clients. However, creating responses in these representations requires significant computational work: The database must be queried for results, the result set must be converted via object-relational-mapping process to a set of Java objects, then the Java objects must converted again to JSON or XML via a marshalling process. For large results sets, such as the downloaded record sets required by the OpenHDS-Tablet application, the conversion may be a rate limiting and impractical step. The OpenHDS-Tablet application developed in this work offers a partial solution to this problem, by allowing incremental downloading of only the most recently updated records.

A complete solution would avoid unnecessary conversion altogether. For example, resources of the OpenHDS-Rest server might express result sets as binary database files, rather than JSON or XML text. These data files could be transferred to clients and applied directly to the local database. This approach has been used with a separate version of OpenHDS for the CIMS project. A similar feature would be a valuable extension the OpenHDS system developed in this work.

## Declarative Link Relation Interpretations

The PODS implementation of the OpenHDS-Tablet application uses `RelInterpretation` objects to associate OpenHDS-Rest resources with appropriate client-side parsing and database objects. These `RelInterpretations` are declared in a static, declarative style which has advantages of concision readability over the imperative style.

Future work might replace the static Java declarations of `RelInterpretations` with dynamic, data-based declarations. For example, the `RelInterpretations` might be declared using XML or JSON documents which contain HATEOAS / HAL link relation names along with the names of appropriate `EntityParser` and database gateway classes. The tablet application could parse these declarations at runtime in order to populate its registry of `RelInterpretations`.

Declaring `RelInterpretations` in this dynamic, data-based way would carry advantages similar to the declarative `FormBehaviors` introduced in this work. For example, the behavior of the tablet application could be changed and adapted to server-side changes, without incurring Java development and redistribution of the tablet application itself. Indeed, the relation interpretation documents could be published from a central server and detected and downloaded automatically by OpenHDS-Tablet clients.

# Conclusion

This work made significant architectural and implementation changes to the OpenHDS system, producing a new PODS OpenHDS system.  The PODS system was guided by four design themes which are represented by the PODS acronym: *polymorphism*, developer *operations*, *declarative* style, and *self-description*.

Application of these themes to the OpenHDS system assisted in addressing challenges of interoperability and adaptability to change which accompany the OpenHDS distributed application, offline data collection, the OpenHDS data model, and projects such as the CIMS project in Equatorial Guinea which must extend the OpenHDS system and evolve over time.

This work yielded a new server application called OpenHDS-Rest and a significantly revised version of the OpenHDS-Tablet client application.  It may aid future development of these applications, as well as the development of browser-based web client applications which would supplement the tablet client.

THE UNIVERSITY OF SOUTHERN MAINE

DEPARTMENT OF COMPUTER SCIENCE

22 January 2016

We hereby recommend that the thesis of Benjamin Heasly entitled

SOFTWARE INTEROPERABILITY

AND THE PODS OPENHDS SYSTEM

be accepted in partial fulfillment of the requirements for the Degree of Master of Science

____________________ Advisor

____________________

____________________

Accepted

# References

Amazon.com. (n.d.). Amazon Web Services. Retrieved 4 June, 2016, from https://aws.amazon.com/

Benzler, J., Herbst, K., & MacLeod, B. (1998). A Data Model for Demographic Surveillance Systems. Retrieved from https://www.researchgate.net/profile/Abraham_Herbst/publication/228705686_A_data_model_for_demographic_surveillance_systems/links/09e41507fa27b71c7f000000.pdf

Bernard, E., & Peterson, S. (2009). JSR 303: Bean validation. Retrieved June 4, 2016, from http://beanvalidation.org/1.0/spec/

Beziz, P., & Directorate General of Statistics and National Accounts. (2005, May 25). Evolution and Distribution of the total population by sex and region. Retrieved May 15, 2016, from http://www.dgecnstat-ge.org/

Borriello, G., Brunette, W., Dell, N., Larson, C., Sudar, S., Sundt, M., & Others. (n.d.-a). Open Data Kit » Aggregate. Retrieved June 4, 2016, from https://opendatakit.org/use/aggregate/

Borriello, G., Brunette, W., Dell, N., Larson, C., Sudar, S., Sundt, M., & Others. (n.d.-b). Open Data Kit » Collect. Retrieved June 4, 2016, from https://opendatakit.org/use/collect/

Cardelli, L., & Wegner, P. (1985). On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, *17*(4), 471–523.

Celko, J. (2012). *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Elsevier/Morgan Kaufmann.

Dijkstra, E. W. (1982). On the Role of Scientific Thought. In *Selected Writings on*

*Computing: A personal Perspective* (pp. 60–66). Springer New York.

Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., & Zugal, S. (2009). Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. In T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, & R. Ukor (Eds.), *Enterprise, Business-Process and Information Systems Modeling* (pp. 353–366). Springer Berlin Heidelberg.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Fowler, M. (2010). Richardson Maturity Model. Retrieved June 4, 2016, from http://martinfowler.com/articles/richardsonMaturityModel.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). 1. Introduction. In *Design patterns: elements of reusable object-oriented software* (pp. 20–21). Pearson Education India.

Google. (n.d.-a). Android. Retrieved June 4, 2016, from https://www.android.com/

Google. (n.d.-b). Application Fundamentals | Android Developers. Retrieved June 4, 2016, from https://developer.android.com/guide/components/fundamentals.html

Google, & Others. (n.d.). AngularJS — Superheroic JavaScript MVW Framework. Retrieved May 15, 2016, from https://angularjs.org/

Heasly, B., Lienhardt, N. W., & Others. (n.d.). OpenHDS REST. Retrieved May 15, 2016, from https://github.com/benjamin-heasly/openhds-rest

Heasly, B., Lienhardt, N. W., Taylor, A., & Others. (n.d.). OpenHDS Tablet. Retrieved May 15, 2016, from https://github.com/benjamin-heasly/openhds-tablet

Kelly, M. (2013, September 18). Hypertext Application Language. Retrieved June 4, 2016, from http://stateless.co/hal_specification.html

Marquis, J. (n.d.). Angular Hateoas. Retrieved May 15, 2016, from https://github.com/jmarquis/angular-hateoas

Nottingham, M., Reschke, J., & Algermissen, J. (19 April, 2016). Link Relations. Retrieved June 4, 2016, from http://www.iana.org/assignments/link-relations/link-relations.xhtml

Oracle. (n.d.-a). Java Persistence API. Retrieved June 4, 2016, from http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

Oracle. (n.d.-b). Trail: The Reflection API (The Java$^{TM}$ Tutorials). Retrieved June 4, 2016, from https://docs.oracle.com/javase/tutorial/reflect/

Oracle. (n.d.-c). UUID (Java Platform SE 7 ). Retrieved June 4, 2016, from https://docs.oracle.com/javase/7/docs/api/java/util/UUID.html

Pemberton, S., & Klotz, L. (n.d.). The Forms Working Group. Retrieved June 4, 2016, from https://www.w3.org/MarkUp/Forms/

Pivotal. (n.d.-a). 10. Aspect Oriented Programming with Spring. Retrieved June 4, 2016, from http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html

Pivotal. (n.d.-b). 21. Web MVC framework. Retrieved June 4, 2016, from http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

Pivotal. (n.d.-c). ExceptionHandler (Spring Framework 4.2.6.RELEASE API). Retrieved June 4, 2016, from http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html

Pivotal. (n.d.-d). Spring. Retrieved June 4, 2016, from https://spring.io/

Pivotal. (n.d.-e). Spring Boot. Retrieved June 4, 2016, from http://projects.spring.io/spring-boot/

Pivotal. (n.d.-f). Spring Data. Retrieved June 4, 2016, from http://projects.spring.io/spring-data/

Pivotal. (n.d.-g). Spring HATEOAS. Retrieved June 4, 2016, from http://projects.spring.io/spring-hateoas/

Thriemer, K., Ley, B., Ame, S. M., Puri, M. K., Hashim, R., Chang, N. Y., … Ali, M. (2012). Replacing paper data collection forms with electronic data entry in the field: findings from a study of community-acquired bloodstream infections in Pemba, Zanzibar. *BMC Research Notes*, *5*, 113.

van Heesch, D. (n.d.). Doxygen: Main Page. Retrieved June 4, 2016, from http://www.stack.nl/~dimitri/doxygen/

Van Kesteren, A., & Others. (2010). Cross-Origin Resource Sharing. Retrieved June 4, 2016, from https://www.w3.org/TR/cors/