Copyright

by

James Peter Giannoules

2014

HIDDEN TEXT: Optional—If you do not include a copyright page, delete entire page and the following page break.

The Report Committee for James Peter Giannoules

Certifies that this is the approved version of the following report:

ProxStor: Flexible Scalable Proximity Data Storage & Analysis

APPROVED BY

SUPERVISING COMMITTEE:

**Supervisor:**

|  |
| --- |
| Suzanne Barber |
| Adnan Aziz |

**ProxStor: Flexible Scalable Proximity Data Storage & Analysis**

by

James Peter Giannoules, B.S.C.S

Hidden text: identical to name in UT transcript. Abbreviate previous degrees earned.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2014

Dedication

For

Breanna, Peter, Robert, and Lucas

Acknowledgements

I would like to acknowledge the support of my professors at the University of Texas who have given their time and support over these past two years.

Abstract

ProxStor: Flexible Scalable Proximity Data Storage & Analysis

James Peter Giannoules, M.S.E.

The University of Texas at Austin, 2014

Supervisor: Suzanne Barber

Hidden text: The abstract may be included at the discretion of the supervisor.

ProxStor is a cloud-based human proximity storage and query informational system taking advantage of both the near ubiquity of mobile devices and the growing digital infrastructure in our everyday physical world (i.e. the Internet of Things). The combination provides the ability for mobile devices to detect when they enter and when they leave the proximity of a space. ProxStor provides a low-overhead interface for storing these proximity events while additionally offering search and query capabilities to enable a richer class of location aware applications.

This report includes the motivation and requirements as well as the details on the design and implementation of ProxStor.

Table of Contents

HIDDENT TEXT: If you choose to place the chapter number (“Chapter 1”) and the chapter title (“Introduction”) on different lines, the automatically generated table of contents will reflect that format. After creating a new table of contents, set them on the same line by deleting the page number and paragraph marker at the end of each chapter number line.

List of Tables xii

List of Figures xiii

List of Illustrations xiv

Section 1: Introduction 1

1.1 Vision 1

1.2 ProxStor 1

1.3 User Stories 2

1.3.1 Story 1 - Friend Locator 2

1.3.2 Story 2 - Restaurant Food Review 3

1.3.2 Story 3 - Event Planning 4

1.4 Contributions 5

1.5 Outline 6

Chapter 2: Requirements and Specifications 7

2.1 Requirements 7

2.1.1 Functional Requirements 7

2.1.2 Non-Functional Requirements 9

2.2 Specifications 10

Chapter 3: System Design 12

3.1 Technology Stack 12

3.2 ProxStor Design 12

3.3 ProxStor Client Components 14

3.3.1 ProxStor Connector 14

3.3.1.1 Simple Connector API 14

3.3.1.2 Class Definition API 14

3.4 ProxStor Cloud Component 15

3.4.1 ProxStor JAX-RS Resources 15

3.4.2 ProxStor Data Access Layers 15

3.4.3 ProxStor Graph Interface 15

3.4 Graph Relational Database 15

Chapter 4: REST API 17

4.1 HTTP Methods 17

4.1.1 GET 18

4.1.2 POST 18

4.1.3 PUT 18

4.1.4 DELETE 18

4.2 HTTP Responses 18

4.2.1 OK (200) 19

4.2.2 Created (201) 19

4.2.3 No Content (204) 19

4.2.4 Forbidden (403) 19

4.2.5 Not Found (404) 19

4.2.6 Server Error (500) 20

4.2.7 Service Unavailable (503) 20

4.3 URIs 20

4.4 Web API 21

4.5 Fixed Object Web Services Interfaces 21

4.5.1 User URI 22

4.5.1.2 Create User 23

4.5.1.2 Retrieve User 23

4.5.1.3 Update User 24

4.5.1.4 Delete User 24

4.5.2 Knows URI 24

4.5.2.1 Create Knows 25

4.5.2.2 Retrieve Knows 26

4.5.2.3 Retrieve Knows Reverse 26

4.5.2.4 Update Knows 26

4.5.2.5 Delete Knows 27

4.5.3 Device URI 27

4.5.3.1 Create Device 28

4.5.3.2 Retrieve User’s Devices 29

4.5.3.3 Retrieve Device 29

4.5.3.4 Update Device 29

4.5.3.5 Delete Device 29

4.5.4 Location URI 30

4.5.4.2 Create Location 30

4.5.4.2 Retrieve Location 31

4.5.4.3 Update Location 31

4.5.4.4 Delete Location 32

4.5.5 Within URI 32

4.5.5.1 Create Within 33

4.5.5.2 Retrieve Within 33

4.5.5.3 Retrieve Within Reverse 33

4.5.2.4 Test Within 34

4.5.2.5 Delete Within 34

4.5.6 Nearby URI 34

4.5.6.1 Create Nearby 35

4.5.6.2 Retrieve Nearby 36

4.5.6.3 Test Nearby 36

4.5.6.4 Update Nearby 36

4.5.6.5 Delete Nearby 36

4.5.7 Sensor URI 37

4.5.7.1 Create Sensor 38

4.5.7.2 Retrieve Location’s Sensors 38

4.5.7.3 Retrieve Sensor 39

4.5.7.4 Update Sensor 39

4.5.7.5 Delete Sensor 39

4.5.8 Locality URI 40

4.5.8.2 Create Locality 41

4.5.8.2 Retrieve Locality 41

4.5.8.3 Retrieve User’s Localities 42

4.5.8.4 Update Locality 42

4.5.8.5 Delete Locality 42

4.5.9 Search URI 42

4.5.9.1 Submitting Search 43

4.5.10 Administration URI 44

4.5.10.1 Create Database Instance 45

4.5.10.2 Retrieve Database Instance 45

4.5.10.3 Shutdown Database Instance 45

4.6 Dynamic Web Services Interfaces 46

4.6.1 Device Check-in URI 46

4.6.1.1 Device Check-in (Environmental) 47

4.6.1.2 Device Check-out (Environmental) 47

4.6.1.3 Device Check-in (Sensor) 48

4.6.1.4 Device Check-out (Sensor) 48

4.6.2 User Check-in URI 48

4.6.2.1 User Check-in 49

4.6.2.2 User Check-out 49

4.6.2.3 Retrieve User Locality 50

4.6.3 Query: Fixed Format URI 50

4.6.3.1 Submit Query 51

4.6.4 Query: Complex URI 51

Chapter 5: Results 52

Software Engineering Metrics 52

Lessons Learned 52

What Worked 52

What Didn’t Work 52

Chapter 6: Conclusion 53

Relationship to Existing Work [?] 53

Relationship to Prior Work [?] 53

Summary 53

Future Work 53

Obtaining ProxStor 53

References 54

## List of Tables

Table X: User URI Methods 23

Table X: Knows URI Methods 25

Table X: Device URI Methods 28

Table X: Location URI Methods 30

Table X: Within URI Methods 32

Table X: Nearby URI Methods 35

Table X: Sensor URI Methods 37

Table X: Locality URI Methods 40

Table X: Search URI Methods 43

Table X: Admin URI Methods 44

Table X: Device Check-in URI Methods 47

Table X: User Check-in URI Methods 49

Table X: Query URI Methods 50

## List of Figures

Figure 1: Specified system 11

Figure 2: ProxStor Components 13

Figure X: Base URI 20

Figure X: User URI 22

Figure X: Knows URI 24

Figure X: Device URI 27

Figure X: Location URI 30

Figure X: Within URI 32

Figure X: Nearby URI 34

Figure X: Sensor URI 37

Figure X: Locality URI 40

Figure X: Search URI 42

Figure X: Admin URI 44

Figure X: Device Check-in URI 46

Figure X: User Check-in URI 49

Figure X: Query URI 50

## List of Illustrations

Illustration n: Title of Illustration: (Heading 9,h9 style: TOC 9) nn

Illustration n: (This list is automatically generated if the paragraph style Heading 9,h9 is used. Optional: If you do not include a List of Illustrations, delete the entire page. **Do not delete the section break below.** It is needed to initiate the printing of Arabic page numbers from the next page onwards.) nn

## Section 1: Introduction

### Vision

Motivated by the proliferation of mobile devices and the expansion of digital fingerprints residing in fixed infrastructure [IoT reference] it should be possible to maintain a repository of these digital fingerprints as well as the places (locations) associated with them. These fingerprints may include unique identifying information from a wireless access point, a Bluetooth device, Bluetooth Low Energy beacons, Near-Field Communication devices, or another emerging / future technology. With these being (relatively) fixed to a location it is possible to infer the location of a mobile device based on these small observations. A database to maintain this information would potentially need to support tens of millions of locations with many times the number of fingerprints. On top of this layer would reside information on hundreds of millions of users and their devices. Each encounter with a location should be stored persistently, thus the database needs to maintain billions of such encounters. These digital fingerprints do not necessarily need to provide fine-grained location information to be useful.

### ProxStor

ProxStor is an instantiation of the above envisioned system beginning to answer the question of how such as system should be built as well as exploring how the system should be used. The system described in this report uses a client-server model. The clients are mobile devices detecting these fingerprints with the assumption that the owner of the device can be assumed to be in a location if the device senses the fingerprint. The server portion of ProxStor is a web application managing both a database repository as well and servicing the client queries. ~~The base concept is that if the fingerprint is detectable by a mobile device (i.e. the device includes a sensor for that fingerprint type), ProxStor can determine the user’s location.~~

*Along the journey from conceptualization to actual implementation several questions have been answered. What type of system best solves this space? How should the data be stored and queried? What is the optimal interface for these imagined mobile devices? What sort of questions can be asked of such as system?[remove?]*

ProxStor leverages the power of an emerging class of NoSQL databases knows as Graph Relational Databases to scale up and out to support the envisioned billions of data points while still rapidly satisfying user queries. The data models used for various objects in ProxStor are designed to enable this evaluation, but themselves are not as data rich as a real-world application would require. This does not impact the feasibility assessment, but should be noted before using ProxStor in a real-world deployment.

### User Stories

To demonstrate how ProxStor can enable a richer class of location aware applications the following user stories are presented. These stories introduce Bob, a fictitious user of various imagined mobile applications. These stories assume that a running instance of ProxStor is available on the internet and furthermore that the database behind ProxStor has been populated with necessary data to satisfy the particular story.

The functionality of ProxStor described in each story exists today.

#### Story 1 - Friend Locator

On a whim Bob decides to head into a busy downtown district to spend his evening with his friends. Bob isn’t entirely certain where his friends are currently at – they have the habit of roaming about. He opens his “friend locator” application to view which of his friends might be nearby. As Bob has hoped the application displays a nearby close friend who is within walking distance. Using an on-screen walking map Bob heads off in the friend’s direction while the application automatically initiates a text message to his friend notifying that Bob is on his way.

This imagined “friend locator” application used ProxStor in three important ways:

1. Upon launching the application ProxStor was used to “check in” and therefore to determine Bob’s current location. Bob’s location is also now recorded within ProxStor for current and future queries.
2. The application sent a query asking *which of Bob’s closest friends are here (or nearby)?* The query may have been limited to only a radius the application interpreted as ‘walking distance’. A second query could have been used to extend the radius beyond walking distance. The application then applied its logic for best fit, taking into account the degree of friendship and the distance Bob specified he was willing to travel. Note that although ProxStor doesn’t derive the strength of the friendship relationship between users it does store this, and can leverage the value in queries.
3. Once the target friend was identified, ProxStor provided the “friend locator” application with his/her current location, which combined with Bob’s current location, provided the starting and ending points used to generate the on-screen walking directions map.

#### 1.3.2 Story 2 - Restaurant Food Review

A restaurant Bob only occasionally frequents has recently changed its menu, to his dismay. He opens his favorite “food review” application to check what his friends who have recently dined here ordered, and what they thought of the food. The nice thing about this application is that even if his friends didn’t leave behind a written review Bob has the option of contacting them directly to ask his questions. Bob is pleased to see several of his friends have been here just in the past month. Since Bob knows which of these friends have compatible palates he is better able to make his dining selection.

This imagined food review application used ProxStor in two important ways:

1. Upon entering the restaurant ProxStor was used to “check in” and therefore to determine and store Bob’s location. This provided the “food review” application with the necessary physical context. Note that this “check in” also allows other users of ProxStor, even independent from the users of the food review application, to know the whereabouts of Bob.
2. ProxStor answered the proximity aware query *which of Bob’s friends has been here recently?* This information was used in turn by the imagined application to reference its food review specific database to extract relevant reviews. To extend the power of such a query the application could have further requested ProxStor to return friends of Bob’s friends who have also dined here recently. Such a potential enhancement operates under the assumption that even people one further degree separated from Bob will still have similar food tastes.

#### 1.3.2 Story 3 - Event Planning

Bob is planning a week long business trip to Austin, Tx and he knows he will have some free time in the evenings. He opens his favorite “event planning” application and informs it that he will be heading to Austin. The application then shows him the top activities his friends in the Austin area typically are up to, broken down by day of the week. With a tap Bob extends the scope of the displayed information to include his friend’s closest friends as well. Using the application’s recommendations as a guide Bob schedules activities for several of his evenings in Austin.

This imagined event planning application used ProxStor in two important ways:

1. ProxStor answered the query *where were my friends on the evening of day?* ProxStor already knows who Bob’s friends are, and further filters results to provide only the listing of locations for the requested time period. Note that in this example the query is repeated for each day of the week because the range of interest is limited to evenings. Each day breaks the continuity and disallows one large date range query.
2. The “event planning” application also used to answer whether the query result location(s) were within an acceptable radius of where Bob planned to be (e.g. downtown). This acceptable area was either defined by Bob or calculated by the application.

### Contributions

The primary contributions contained in this report are:

* A complete coherent client application programming interface (API) for implementing the envisioned system is provided. Consideration is given to both low- and high-powered devices.
* A working example of the envisioned system, ProxStor, is documented. This includes documenting the system design features, both macro and micro in scale. In many cases the rationale for decisions is also provided.
* The feasibility of the envisioned system, based on experimentation with ProxStor, is provided.

It is hoped that the work contained herein can be a starting point for future work in this area.

### Outline

This remainder of this report describes ProxStor’s system design, its application programming interface. The rationale for key design points is provided.

Chapter 2 of the report contains the requirements and specifications necessary to successful implement the envisioned system. This includes functional and non-functional requirements as well as high-level design specifications.

Chapter 3 explores the internal deign of the ProxStor implementation including listing all the technologies brought together to build a successful system. The chapter then describes the overall system design, and the user-exposed object types, the ProxStor connector used to easily enable Java client applications. The JAX-RS web interface is explored including how the system accesses the back-end database. Finally, the chapter concludes with a discussion of the rationale for electing to use a graph relational database including the data model for some common usage scenarios.

Chapter 4 documents all of the possible HTTP response statuses, how the HTTP methods map into the RESTful web API, and all exposed URIs and their supported operations. This chapter highlights the consistent and simple ProxStor API, which is a key contribution of this report.

Chapter 5 includes the results of building and testing the prototype. This includes standard software engineering codebase metrics as well as lessons learned.

Chapter 6 summarizes what was learned from this exercise as well as outlining planned future work.

## Chapter 2: Requirements and Specifications

This Chapter covers both what ProxStor should do and how it should do it. These are presented as both functional and non-functional requirements as well as design specifications. The internal of the actual implementation are described in Chapter 4.

### 2.1 Requirements

To achieve the goals set forth for ProxStor the following functional and non-functional requirements must be met.

#### 2.1.1 Functional Requirements

The following functional requirements must be met to enable the necessary basic operations of ProxStor:

* Data Representation: the system shall represent the following data types and provide the capability to create, retrieve, update, and delete each. Additionally each object shall be uniquely addressable:
  + Users
  + Locations
  + Location Environmental (“sensible” aspects of a Location)
  + Devices
  + Sensors within Devices
* Data Relationship: the system shall provide the capability to create and dissolve the following associations:
  + A user *uses* a particular device
  + A user *knows* another user
  + A devices *contains* sensor(s)
  + A location is *contained within* another location
  + A location is *within* a specified distance of another location
  + A location is *identified* by a unique environmental element contained within
* Data Query: the system shall provide the capability to query based on the following relationships between objects:
  + Retrieve users known by specified user
  + Retrieve users who know specified user (the reverse direction of the above)
  + Retrieve devices owned by specified user
  + Retrieve locations within specified location
  + Retrieve locations which contain specified location (the inverse of above)
  + Retrieve locations within specified distance of another location
  + Retrieve all sensors within specified location
* Administration: the system shall expose basic administrative operations:
  + Instantiate a new database
  + Connect to existing database
  + Report usage (administrative) statistics
* Persistence: All data objects remain persistently within the system until a delete operation is performed, regardless of the age or inactivity of the object. A full history of a user’s movements is retained. No pruning of older data shall be performed.
* Current Location: the system shall support a concept of each user’s current location, implying a temporal processing aspect to the system.
* Environmental-based Movements: the system shall support the concept of accepting client reported check-in and check-out operations based on a device sensing an environmental:
  + A check-in indicates that a device has just detected the environmental and thus the associated user’s current location shall be updated accordingly.
  + A check-out indicates that a device has just ceased detection of the environmental and thus the associated user’s current location shall become unknown.
* Manual Movements: the system shall support the user manually specifying his or her current location.
* Data Processing: The user’s current location shall be used for a class of queries
  + Retrieve user’s current location
  + Retrieve those users in the user’s current location
  + Retrieve
* Data Set: the system shall handle arbitrarily large data sets with no pre-defined upper limit. Data sets of billions of entries shall be possible.
* Concurrency: the system shall gracefully and coherently handle multiple concurrent client accesses. If more than one client attempts access to the same database object the system must ensure ordering and atomicity of operations.

#### 2.1.2 Non-Functional Requirements

The following are the non-functional requirements goals for ProxStor. Striving for these goals is done as a best effort.

* Scalability: The system should be designed to easily scale both vertically and horizontally to support increased user load.
* Extensibility: The system should permit extending capabilities with minimal impact to the existing code. These extensions should be possible in a manner such that the external interface remains coherent.
* Interoperability: The system should support a wide range of both client devices and of data center deployments. This applies to both the machine running ProxStor as well as the persistent storage solution.
* Availability: Because of the nature of check-in and check-out events the system should be available continuously with no downtime.
* Capacity: The system should permit billions of records accessed by millions of users concurrently.
* Speed: The client devices accessing ProxStor will vary widely in their computing power, battery life, and network connectivity. The system should accept, process, and respond to requests as quickly as possible. Note that efficiency is not a requirement.
* Privacy: This is non-requirement. Privacy is not a concern of the ProxStor project. It is acknowledged that a real world deployment of a ProxStor-like system would require stringent privacy measures.

### 2.2 Specifications

The above requirements combined with the current state of mainstream computing lead to the following specifications.

* The system should be accessible over the internet to provide ubiquitous client accessibility.
* The web presence component of the system should be flexible enough in nature to be deployed in one of the many web container architectures in use today. This will permit one who is deploying ProxStor to use either an in-house datacenter or a hosting service.
* The web presence component should be minimalistic and coherent.
* The web presence components should reduce potential downtime by providing redundancy.
* The web presence component should be based on HTTP.
* The persistent data store should be accessible by the web presence component.
* The persistent data store should be isolated from clients.
* The web presence and persistent data store components should be de-coupled.
* A client should only need to comprehend basic HTTP communications and basic object [de]serialization.

The following is a high-level diagram of the specified system.



Figure 1: Specified system

## Chapter 3: System Design

This chapter describes the internal design and implementation of ProxStor.

### 3.1 Technology Stack

The implementation of ProxStor leveraged several technologies and built upon existing freely available software. Principle among these software offerings are:

* Jersey – provides a framework for development of RESTful web services providing support for the JAX-RS API. See [Jersey] and [JAX-RS].
* Tinkerpop Blueprints – provides a common interface for developing applications on top of graph databases. If the blueprints project has enabled back-end support for a given database then ProxStor supports it as well. The project website describes it as “analogous to the JDBC, but for graph databases”. See [blueprints].
* Gson – Java library used to convert objects into their JSON representation and to convert strings back into Java objects.
* Postman REST Client – provides an easy to use graphical interface to test and refine a REST interface, including the ability to save and restore saved commands and adapt to different networking environments. See [postman].
* Winstone Servlet Container – provides a fast minimalist servlet container for running ProxStor. See [winstone].

All ProxStor code was written in the Java programming language. The NetBeans IDE was used. Maven was leveraged for managing project dependencies.

### 3.2 ProxStor Design

The ProxStor project consists of two components:

1. The ProxStor Connector and API for client application consumption
2. The ProxStor Cloud Service serving as the web interface as well as providing back-end database access

Figure X shows the components of ProxStor and their relative positions within the stack. The ProxStor components are shaded grey.

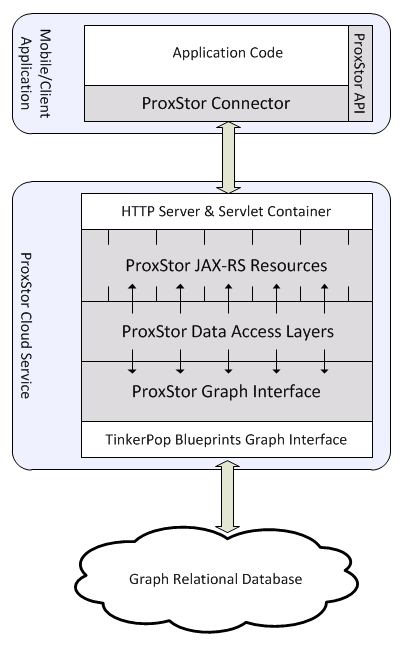


Figure 2: ProxStor Components

***[TODO Section needs filling in + a transition to the next section…]***

### 3.3 ProxStor Client Components

#### 3.3.1 ProxStor Connector

The ProxStor Connector package provides a Java client with:

* Simplified API for communicating with ProxStor
* Class definitions for objects which are exchanged between client and server.

##### 3.3.1.1 Simple Connector API

The ProxStor Connector class enables a client to:

* Connect to a running ProxStor instance
* Manage the lifecycle of all exposed object types
* Perform device or user check-ins and check-out

All the client invocations of these methods involve handling of the ProxStor object types. The client does not need to be fluent in network communication design. Additionally, the connector returns simple Boolean status for operations. The client need not perform exception handling.

Please refer to the ProxStorConnector.java source file and the respective JavaDoc for more details.

##### 3.3.1.2 Class Definition API

The ProxStor Connector API package defines the following object types (as Java classes):

* **User** – Uniquely identifying a user of the ProxStor system.
* **Device** – Uniquely identifying a user’s device. ProxStor enforces the relationship that a device is used my one and only one user.
* **Location** and **LocationType** – A location in the physical world as well as the type of location. Locations can be related to each other in either a nesting fashion or by defining the distance in between.
* **Sensor** and **SensorType** – A sense-able environmental element within a location. When a device detects a sensor the system can infer location. TODO: *The term Sensor still feels backwards.*
* **Locality** – Created whenever a user is within location. A Locality persistently records the user, device, sensor, location, and time information on each check in.

Please refer to the proxstor.api source package and the respective JavaDoc for more details.

### 3.4 ProxStor Cloud Component

ProxStor’s cloud component is implemented as a Java servlet. This component provides both the RESTful HTTP interface as well it coordinates access to the back-end database and enforces the desired data model.

#### 3.4.1 ProxStor JAX-RS Resources

TODO: jersey, design of resources, file organization, separation of concerns, resource and sub-resource locators…

#### 3.4.2 ProxStor Data Access Layers

TODO: role of DAOs within the servlet.

#### 3.4.3 ProxStor Graph Interface

TODO: blueprints, [non] transactional, neo4j/orientdb/tinkergraph/etc…

### 3.4 Graph Relational Database

TODO: Rationale for using graph relational database

TODO: Background on NOSQL?

TODO: ProxStor Graph Model w/Data Model diagrams

## Chapter 4: REST API

ProxStor’s services are exposed to the world as a web service using meaningful URIs and returning consistent HTTP responses. Applications consuming the Java ProxStor Connector (see <*Connector Section>*) do not need to comprehend the below details as the connector provides a more simplified interface. For those directly accessing the web interface, for example implementing their own connector, this section provides a guide to the interface.

ProxStor’s web interface is modeled as a RESTful interface. REST stands for Representational State Transfer and is a dominant design model for web services [IBM DW]. In brief, a REST service is:

* Stateless
* Uses standard HTTP methods
* URIs are used in a directory-like structure to access resources
* Uses JSON to transfer objects

Through this RESTful interface ProxStor exposes CRUD (Create, Retrieve, Update, and Delete) [] operations for all static object types. Note that this matches well with the function requirements for data creation, update, persistence, and deletion.

The reader who is unfamiliar with REST is encouraged to read [] or []. This paper does not further elaborate on REST or CRUD.

### 4.1 HTTP Methods

ProxStor uses the following standard HTTP methods for all interactions:

* GET
* POST
* PUT
* DELETE

#### 4.1.1 GET

For all retrieval operations the GET method is used. Examples include retrieval of a user’s current location or to perform a static object search.

#### 4.1.2 POST

For all create operations the POST method is used. Examples include addition of a new user into the system or a new location check-in.

#### 4.1.3 PUT

For all update operations the PUT method is used. Examples include updates to the profile for a location or to modify the knows (friendship) relationship between two users.

#### 4.1.4 DELETE

For all delete operations the DELETE method used. Examples include removing the within relationship between two locations or removing a user from the database.

In addition to the traditional deletion of an object from the persistent data store, DELETE also is used in ProxStor to:

* Check-out from a location
* Shutdown the running database instance

The following sections clarify the usage of all exposed URIs.

### 4.2 HTTP Responses

ProxStor strives to implement consistent HTTP responses in all situations simplifying client (and client library) development. The below sections document the various HTTP responses and under what circumstances they are returned.

#### 4.2.1 OK (200)

HTTP status 200 (OK) is returned when the requested operation completed successfully without error. If the request was for the retrieval of information the body of the response will contain JSON representation of such data.

#### 4.2.2 Created (201)

HTTP status 201 (Created) is returned when a request to create new content inside ProxStor completes successfully. In the header of the 201 response the *Location* field will indicate the URI of the new content. As fitting the specific request the HTTP response body may also include a JSON representation of the newly created content.

#### 4.2.3 No Content (204)

HTTP status 204 (No Content) is returned when a request to perform an operation is successfully completed and the context of the request does not involve the transfer of data. In ProxStor status 204 is returned for successful updating and deleting of objects as the status code is all the information a client requires.

#### 4.2.4 Forbidden (403)

HTTP status 403 (Forbidden) is currently only returned in situation, when the administrator attempts to create a new graph database instance while one already exists.

#### 4.2.5 Not Found (404)

HTTP status 404 (Not Found) is returned in both the case of a malformed URI and an invalid request. In the bad URI case the status 404 is returned by the servlet container, not ProxStor per se.

ProxStor return status 404 if the client request references an unknown or nonexistent object. For example, if the request was to retrieve a user object with a non-existent userId.

#### 4.2.6 Server Error (500)

HTTP status 500 (Server Error) typically indicates an unhandled exception within ProxStor which reached up to the servlet container; however ProxStor does intentionally return 500 in two cases.

The first is if an URISyntaxException is thrown while preparing the *Location* header field in a 201 (Created) response.

The second is if an attempted instantiation of a new back-end Graph instance fails.

#### 4.2.7 Service Unavailable (503)

HTTP status 503 (Service Unavailable) is returned when an attempt is made to retrieve the database instance information, but no running database instance exists.

### 4.3 URIs

A uniform resource identifier (URI) is a string used to uniquely identify the name of a resource and uniform resource locators (URL) used in HTTP requests (i.e. a web address) are in fact URIs. All communications with ProxStor is initiated with an HTTP request to *some* URI. Thus it is necessary to review the URI layout to actually comprehend the ProxStor web interface.

All URIs in ProxStor are relative to the base URI, which is actually a factor of the servlet container in use. For example, when running ProxStor within Winstone the default base URI becomes:



Figure X: Base URI

The exact base URI will depend on the specifics of your ProxStor deployment. The following sections list all exposed URIs and describes their use. The key concept to grasp is that all the URIs documented herein are actually relative (appended) to the system base URI.

When designing the web interface care was taken to ensure all ProxStor URIs were meaningful and expressive. Combining the HTTP request type and the URI should be sufficient to describe the operation being attempted.

### 4.4 Web API

The complete ProxStor Web API is documented in the following sections. For each URI the following is described:

* Supported HTTP requests
* Operations performed within ProxStor
* Success and Failure HTTP response status codes

The Web Services interface can roughly be broke into fixed-object and dynamic categories. In either case all URIs are relative to the same base.

### 4.5 Fixed Object Web Services Interfaces

The (relatively) fixed components of the Web Interface provide CRUD interfaces for lifecycle management of the following ProxStor object types:

* Users
  + *Knows* relationships between users
  + Devices *owned* by Users
* Locations
  + Locations connected by *Within* relationship
  + Locations connected by *Nearby* relationship
  + Sensors *inside* Locations
* Locality object representing a period of time User was in a Location
* Searching these fixed components
* Administration of ProxStor

Search actions are siloed to a specific object type (User, Devices, ...) while fixed component queries are formed with URI constructs. The static data returned has context assumed from the clients query. For example, querying ProxStor for all users a user with a specified *userId* knows with a certain strength *minStrength* will return a list of User JSON object without supporting metadata. The client must maintain the context of whose friends list is represented.

#### 4.5.1 User URI

All operations related to manipulation of user objects within the database happen relative to the user URI:



Figure X: User URI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | / | POST | Content-Type: application/json  Accept: application/json | Create user |
| 2 | /{userid} | GET | Accept: application/json | Retrieve user |
| 3 | /{userid} | PUT | Content-Type: application/json | Update user |
| 4 | /{userid} | DELETE |  | Delete user |

Table X: User URI Methods

The URI /{userid} is referred to as the base user + userId URI for convenience. The {userid} notation is meant to signify that the numeric database-specific user id is to be inserted in place of the {userid} string.

##### 4.5.1.2 Create User

To create a new user the client should prepare a proxstor.API.User object containing the user information. Note that at this time the userId field is null because the system has not yet assigned an id. This User object is then converted into JSON and sent via a HTTP POST request to the user URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the user is successfully added ProxStor will return an HTTP status 201 (Created) with the new userId in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the User object and can be directly used in a GET request. If the userId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full User object, including the userId, is returned to the client in JSON. This is the location used by the ProxStor Connector.

##### 4.5.1.2 Retrieve User

To retrieve a user from the database the client must send a GET HTTP request to the base user + userId URI path with the header field *Accept* set to *application/json*. If the specified userId is valid ProxStor will respond with an HTTP status of 200 (OK) with the JSON representation of the User object in the response body.

##### 4.5.1.3 Update User

To update a user the requestor must send the JSON representation of the updated User object in an HTTP PUT request to the base user + userId URI. Note that the userId in the URI and the userId in the JSON representation of User must be identical in addition to being a valid user id in the database. If the user update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

##### 4.5.1.4 Delete User

To delete a user from the database the client must send an HTTP DELETE request to the base user + userId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

#### 4.5.2 Knows URI

All operations relating to the *knows* relationship between users are performed relative to the knows URI:



Figure X: Knows URI

Note that all knows operations are in the context of a specific base user + userId URI, and thus the context of a user. A knows relationship does not exist without at least specifying the user who is asserting the level to which they know someone else and the URI structure is representing this constraint.

The strength value, {s}, is to degree to which the knows relationship is established. The supported values are 1 through 100, with higher values representing stronger relationships. It is envisioned that an application building upon ProxStor will categorize the value ranges into terms understood to the user. For example, friendship could be anything greater than 50. For the purposes of this report keep in mind that the strength value is a required component of the URI.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | /user/{userid2} | POST |  | Create know relationship |
| 2 | / | GET | Accept: application/json | Retrieve users who userid knows |
| 3 | /reverse | GET | Accept: application/json | Retrive users who know userid |
| 4 | /user/{userid2} | PUT |  | Update knows relationship |
| 5 | /user/{userid2} | DELETE |  | Delete knows relationship |

Table X: Knows URI Methods

The URI /user/{userid2} is referred to as the userId2 URI for convenience. The {userid2} notation is meant to signify that the numeric database-specific user id is to be inserted in place of the {userid2} string.

##### 4.5.2.1 Create Knows

To create a new knows relationship between two users the client must send an HTTP POST to the knows + userId2 URI. The URI encodes all the information ProxStor needs to establish the relationship, therefore no JSON representation is sent by the client. If both the userId and userId2 are valid users (and not the same value), the strength value is in the valid range (1-100), and a knows relationship does not already exist, then ProxStor will respond with an HTTP status 201 (Created).

##### 4.5.2.2 Retrieve Knows

To retrieve all the users whom a specific user knows with at least a minimum strength the client must send an HTTP GET to the knows URI with the HTTP header field *Accept* set to *application/json*. ProxStor will confirm the validity of the user id in the URI and find all the users who the user knows with at least strength from the URI. If the user id is valid ProxStor will respond with HTTP status 200 (OK) with the body of the response containing a JSON representation of a list of proxstor.api.User objects.

##### 4.5.2.3 Retrieve Knows Reverse

To retrieve the users who know a specific user with at least a minimum strength the client appends /reverse to the knows URI. Note that this retrieval is the opposite direction of that in section 4.3.2.2. In other words, these are the users who know the specified user id with minimum strength s – not users who user id knows. The remainder of the interface is identical to 4.3.2.2.

##### 4.5.2.4 Update Knows

To update the strength value in an established knows relationship the client must issue an HTTP PUT request to the knows + userId2 URI with the updated strength value encoded in the URI. If a knows relationship already exists from userid to userid2 and the strength value is valid, then ProxStor will update the relationship and respond with HTTP status 204 (No Content).

##### 4.5.2.5 Delete Knows

To remove the knows relationship between two users the client must send an HTTP DELETE request to the knows + userId2 URI. Note that the strength value must be a valid value in the range 1 to 100, but the actual value is ignored in this operation. If a knows relationship exists between userid and userid2, then ProxStor will delete this relationship and respond with HTTP status 204 (No Content).

If either of the user ids was invalid or the knows relationship was not already established ProxStor will respond with HTTP status 404 (Not Found).

#### 4.5.3 Device URI

All operations related to manipulation of device objects within the database happen relative to the device URI:



Figure X: Device URI

Note that all device related operations are in the context of a specific base user + userId URI, and thus a single specific user. A device does not exist in ProxStor without being associated with a user and so the URI naturally expresses this.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | / | POST | Content-Type: application/json  Accept: application/json | Create device |
| 2 | / | GET | Accept: application-json | Retrieve all user’s devices |
| 3 | /{devid} | GET | Accept: application/json | Retrieve device |
| 4 | /{devid} | PUT | Content-Type: application/json | Update device |
| 5 | /{devid} | DELETE |  | Delete device |

Table X: Device URI Methods

The URI /{devid} is referred to as the base device + devId URI for convenience. The {devid} notation is meant to signify that the numeric database-specific device id is to be inserted in place of the {devid} string.

##### 4.5.3.1 Create Device

To create a new device the client should prepare a proxstor.API.Device object containing the device information. Note that at this time the devId field is null because the system has not yet assigned an id. This Device object is then converted into JSON and sent via a HTTP POST request to the desired user’s device URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the device is successfully added ProxStor will return an HTTP status 201 (Created) with the new devId in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the Device object and can be directly used in a GET request. If the devId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full Device object, including the devId, is returned to the client in JSON. This is the location used by the ProxStor Connector.

##### 4.5.3.2 Retrieve User’s Devices

To retrieve all devices owned by a specific user the client must sent an HTTP GET request to the base device URI for the owning user. Do not specify a devId specific portion to the URI. The request header field *Accept* should be set to *application/json*. If the user specified in the URI is valid and owns at least one device ProxStor will respond with an HTTP status of 200 (OK) and the body of the response will contain a JSON representation of a list of Device objects. If the user is valid, but owns no devices, the response status will be 204 (No Content) with no contents in the body.

##### 4.5.3.3 Retrieve Device

To retrieve a single device from the database the client must send an HTTP GET request to the base device + devId URI path for the owning user with the header field *Accept* set to *application/json*. If the specified devId is valid and owned by the userId in the URI ProxStor will respond with an HTTP status of 200 (OK) with the JSON of the of the Device object in the response body.

##### 4.5.3.4 Update Device

To update a device the requestor must send the JSON representation of the updated Device object in an HTTP PUT request to the owning user’s base device + devId URI. Note that the devId in the URI and the devId in the JSON representation of Device must be identical in addition to being a valid device id in the database and be owned by the userId in the URI. If the device update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

##### 4.5.3.5 Delete Device

To delete a device from the database the client must send an HTTP DELETE request to the owning user’s base device + devId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

#### 4.5.4 Location URI

All operations related to manipulation of location objects within the database happen relative to the location URI:



Figure X: Location URI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | / | POST | Content-Type: application/json  Accept: application/json | Create location |
| 2 | /{locid} | GET | Accept: application/json | Retrieve location |
| 3 | /{locid} | PUT | Content-Type: application/json | Update location |
| 4 | /{locid} | DELETE |  | Delete location |

Table X: Location URI Methods

The URI /{locid} is referred to as the base location + locId URI for convenience. The {locid} notation is meant to signify that the numeric database-specific location id is to be inserted in place of the {locid} string.

##### 4.5.4.2 Create Location

To create a new location the client should prepare a proxstor.API.Location object containing the location information. Note that at this time the locId field is null because the system has not yet assigned an id. This Location object is then converted into JSON and sent via a HTTP POST request to the location URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the location is successfully added ProxStor will return an HTTP status 201 (Created) with the new locId available in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the Location object and can be directly used in a GET request. If the locId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full Location object, including the locId, is returned to the client in JSON. This is the method used by the ProxStor Connector.

##### 4.5.4.2 Retrieve Location

To retrieve a location from the database the client must send a GET HTTP request to the base location + locId URI path with the header field *Accept* set to *application/json*. If the specified locId is valid ProxStor will respond with an HTTP status of 200 (OK) with the JSON representation of the Location object in the response body.

##### 4.5.4.3 Update Location

To update a location the requestor must send the JSON representation of the updated Location object in an HTTP PUT request to the base location + locId URI. Note that the locId in the URI and the locId in the JSON representation of Location must be identical in addition to being a valid location id in the database. If the location update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

##### 4.5.4.4 Delete Location

To delete a location from the database the client must send an HTTP DELETE request to the base location + locId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

#### 4.5.5 Within URI

All operations relating to the *within* relationship between location are performed relative to the within URI:



Figure X: Within URI

Note that all within operations are in the context of a specific base location + locId URI, and thus the context of a location. A within relationship does not exist without at least specifying the location which is within another location.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | /{locid2} | POST |  | Create within relationship |
| 2 | / | GET | Accept: application/json | Get locations within |
| 3 | /reverse | GET | Accept: application/json | Get location containing |
| 4 | /{locid2} | GET |  | Test location within |
| 5 | /{locid2} | DELETE |  | Delete within relationship |

Table X: Within URI Methods

The URI /{locid2} is referred to as the locId2 URI for convenience. The {locid2} notation is meant to signify that the numeric database-specific location id is to be inserted in place of the {locid2} string.

##### 4.5.5.1 Create Within

To create a new within relationship between two locations the client must send an HTTP POST to the within + locId2 URI. The URI encodes all the information ProxStor needs to establish the relationship, therefore no JSON representation is sent by the client. If both the location identifiers are valid (and not the same value) and a within relationship does not already exist, then ProxStor will respond with an HTTP status 201 (Created).

##### 4.5.5.2 Retrieve Within

To retrieve all the locations within a specific location the client must send an HTTP GET to the within URI with the HTTP header field *Accept* set to *application/json*. ProxStor will confirm the validity of the location id in the URI and find all the locations within the specified location. If the locsation id is valid ProxStor will respond with HTTP status 200 (OK) with the body of the response containing a JSON representation of a list of proxstor.api.Location objects.

##### 4.5.5.3 Retrieve Within Reverse

To retrieve the locations which contains a specific location the client appends /reverse to the within URI. Note that this retrieval is the opposite direction of that in section 4.3.5.2. The remainder of the interface is identical to 4.3.5.2.

##### 4.5.2.4 Test Within

To test whether a location is within another location the client may issue an HTTP GET request to the within + locId2 URI. If a within relationship exists between locid and locid2 then ProxStor will respond with HTTP status 204 (No Content).

##### 4.5.2.5 Delete Within

To remove the within relationship between two locations the client must send an HTTP DELETE request to the within + locId URI. If a within relationship exists between locid and locid2, then ProxStor will delete this relationship and respond with HTTP status 204 (No Content).

If either of the locations ids was invalid or the within relationship was not already established ProxStor will respond with HTTP status 404 (Not Found).

#### 4.5.6 Nearby URI

All operations relating to the *nearby* relationship between locations are performed relative to the nearby URI:



Figure X: Nearby URI

Note that all nearby operations are in the context of a specific base location + locId URI, and thus the context of a location. A nearby relationship does not exist without at least specifying the location who is asserting the distance to which they are nearby some other location. The URI structure is representing this relationship.

The distance value, {d}, is to distance in meters between locations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | /{locid2} | POST |  | Create nearby relationship |
| 2 | / | GET | Accept: application/json | Retrieve locations within distance |
| 3 | /{locid2} | GET |  | Tests location distance |
| 4 | /{locid2} | PUT |  | Update nearby relationship |
| 5 | /{locid2} | DELETE |  | Delete nearby relationship |

Table X: Nearby URI Methods

The URI /{locid2} is referred to as the locId2 URI for convenience. The {locid2} notation is meant to signify that the numeric database-specific location id is to be inserted in place of the {locid2} string.

##### 4.5.6.1 Create Nearby

To create a new nearby relationship between two users the client must send an HTTP POST to the nearby + locId2 URI. The URI encodes all the information ProxStor needs to establish the relationship, therefore no JSON representation is sent by the client. If both the locid and locid2 are valid locations (and not the same value) and a nearby relationship does not already exist, then ProxStor will respond with an HTTP status 201 (Created).

##### 4.5.6.2 Retrieve Nearby

To retrieve all the locations within a specified distance of a specific location the client must send an HTTP GET to the nearby URI with the HTTP header field *Accept* set to *application/json*. ProxStor will confirm the validity of the location id in the URI and find all the locations nearby within the distance encoded in the URI. If the location id is valid ProxStor will respond with HTTP status 200 (OK) with the body of the response containing a JSON representation of a list of proxstor.api.Location objects.

##### 4.5.6.3 Test Nearby

To test whether a location is nearby within a specific distance to another location the client may issue an HTTP GET request to the nearby + locId2 URI. If a nearby relationship exists between locid and locid2 and the distance is less than or equal to d then ProxStor will respond with HTTP status 204 (No Content).

##### 4.5.6.4 Update Nearby

To update the distance value in an established nearby relationship the client must issue an HTTP PUT request to the nearby + userId2 URI with the updated distance value encoded in the URI. If a nearby relationship already exists from locid to locid2 then ProxStor will update the relationship and respond with HTTP status 204 (No Content).

##### 4.5.6.5 Delete Nearby

To remove the nearby relationship between two locations the client must send an HTTP DELETE request to the nearby + locId2 URI. Note that distance must be included in the URI, but the value is ignored in this operation. If a nearby relationship exists between locid and locid2, then ProxStor will delete this relationship and respond with HTTP status 204 (No Content).

If either of the location ids was invalid or the nearby relationship was not already established ProxStor will respond with HTTP status 404 (Not Found).

#### 4.5.7 Sensor URI

All operations related to manipulation of sensor objects within the database happen relative to the sensor URI:



Figure X: Sensor URI

Note that all sensor related operations are in the context of a specific base location + locId URI, and thus a single specific location. A sensor does not exist in ProxStor without being associated with a location and so the URI naturally expresses this.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | / | POST | Content-Type: application/json  Accept: application/json | Create sensor |
| 2 | / | GET | Accept: application-json | Retrieve all location’s sensors |
| 3 | /{sensorid} | GET | Accept: application/json | Retrieve sensor |
| 4 | /{sensorid} | PUT | Content-Type: application/json | Update sensor |
| 5 | /{sensorid} | DELETE |  | Delete sensor |

Table X: Sensor URI Methods

The URI /{sensorid} is referred to as the base sensor + sensodId URI for convenience. The {sensorid} notation is meant to signify that the numeric database-specific sensor id is to be inserted in place of the {sensorid} string.

##### 4.5.7.1 Create Sensor

To create a new sensor the client should prepare a proxstor.API.Sensor object containing the sensor information. Note that at this time the sensorId field is null because the system has not yet assigned an id. This Sensor object is then converted into JSON and sent via a HTTP POST request to the desired location’s sensor URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the sensor is successfully added ProxStor will return an HTTP status 201 (Created) with the new sensorId in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the Sensor object and can be directly used in a GET request. If the sensorId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full Sensor object, including the sensorId, is returned to the client in JSON. This is the method used by the ProxStor Connector.

##### 4.5.7.2 Retrieve Location’s Sensors

To retrieve all sensors inside a specific location the client must sent an HTTP GET request to the base sensor URI for the owning location. Do not specify a sensorId specific portion to the URI. The request header field *Accept* should be set to *application/json*. If the location specified in the URI is valid and contains at least one sensor ProxStor will respond with an HTTP status of 200 (OK) and the body of the response will contain a JSON representation of a list of Sensor objects. If the location is valid, but contains no sensors, the response status will be 204 (No Content) with no contents in the body.

##### 4.5.7.3 Retrieve Sensor

To retrieve a single sensor from the database the client must send an HTTP GET request to the base sensor + sensorId URI path for the owning location with the header field *Accept* set to *application/json*. If the specified sensorId is valid and contained within the locId in the URI ProxStor will respond with an HTTP status of 200 (OK) with the JSON of the of the Sensor object in the response body.

##### 4.5.7.4 Update Sensor

To update a sensor the requestor must send the JSON representation of the updated Sensor object in an HTTP PUT request to the owning location’s base sensor + sensorId URI. Note that the sensorId in the URI and the sensorId in the JSON representation of Sensor must be identical in addition to being a valid sensod id in the database and be inside the location specified in the URI. If the sensor update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

##### 4.5.7.5 Delete Sensor

To delete a sensor from the database the client must send an HTTP DELETE request to the owning location’s base sensor + sensorId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

#### 4.5.8 Locality URI

A Locality represents the bringing together of a device and a sensor (or a user and a location). The ProxStor system collects these localities through its lifetime. These are used to answer questions about a user's current and historic locations.

Manipulating a Locality is not very useful by itself (that's what checkin is for). These operations are provided mainly for development and testing purposes.

All operations related to manipulation of Locality objects within the database happen relative to the locality URI:



Figure X: Locality URI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | / | POST | Content-Type: application/json  Accept: application/json | Create locality |
| 2 | /{localityid} | GET | Accept: application/json | Retrieve locality |
| 3 | /user/{userid} | GET | Accept: application/json | Retrieve localities for user |
| 4 | /{localityid} | PUT | Content-Type: application/json | Update locality |
| 5 | /{localityid} | DELETE |  | Delete locality |

Table X: Locality URI Methods

The URI /{localityid} is referred to as the base locality + localityId URI for convenience. The {localityId} notation is meant to signify that the numeric database-specific locality id is to be inserted in place of the {localityid} string.

##### 4.5.8.2 Create Locality

To create a new locality the client should prepare a proxstor.API.Locality object containing the locality information. Note that at this time the localityId field is null because the system has not yet assigned an id. This Locality object is then converted into JSON and sent via a HTTP POST request to the locality URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the locality is successfully added ProxStor will return an HTTP status 201 (Created) with the new localityId in two locations. The client is free to choose whichever extraction method it wishes.

The first location is with the Location field of the response header. This Location contains the full URI to the Locality object and can be directly used in a GET request. If the localityId alone is needed the client must process the field to retain only the id portion after the final forward slash.

The second location is in the body of the response. The full Locality object, including the localityId, is returned to the client in JSON.

##### 4.5.8.2 Retrieve Locality

To retrieve a locality from the database the client must send a GET HTTP request to the base locality + localityId URI path with the header field *Accept* set to *application/json*. If the specified localityId is valid ProxStor will respond with an HTTP status of 200 (OK) with the JSON representation of the Locality object in the response body.

##### 4.5.8.3 Retrieve User’s Localities

To retrieve previous localities for a specified user the client must send an HTTP GET request to the base locality + user + userId URI with header field *Accept* set to *application/json*. If the specified userId is valid ProxStor will respond with HTTP status 200 (OK) and the body containing the JSON list representation of the previous proximity objects associated with the specified user (up to a max of 1024).

##### 4.5.8.4 Update Locality

To update a locality the requestor must send the JSON representation of the updated Locality object in an HTTP PUT request to the base locality + locd URI. Note that the localityId in the URI and the localityId in the JSON representation of Locality must be identical in addition to being a valid locality id in the database. If the locality update was successful ProxStor will respond with an HTTP status of 204 (No Content) with no content in the response body.

##### 4.5.8.5 Delete Locality

To delete a locality from the database the client must send an HTTP DELETE request to the base locality + localityId URI. No special header fields need be specified. ProxStor will respond with an HTTP status of 204 (No Content) if the deletion was successful.

#### 4.5.9 Search URI

All operations related to the searching are relative to the search URI:



Figure X: Search URI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | /users | POST | Content-Type: application/json  Accept: application/json | Search users |
| 2 | /devices | POST | Content-Type: application/json  Accept: application/json | Search devices |
| 3 | /locations | POST | Content-Type: application/json  Accept: application/json | Search locations |
| 4 | /sensors | POST | Content-Type: application/json  Accept: application/json | Search sensors |

Table X: Search URI Methods

The URI for the respective object type is referred to as the object search URI for convenience.

##### 4.5.9.1 Submitting Search

All four URIs for searching are used very similarly. To return search results the client determines which object type to search through and sends an HTTP POST request to appropriate object search URI. The request header fields *Content-type* and *Accept* must both set to *application/json.* The body of the request shall contain a *partially* specified JSON representation of the corresponding object type. For example, to search through users the client might send a partial proxstor.api.User with only the email address specified. This causes ProxStor to find all matching users – in this case the single user with the specified email address.

If ProxStor find one or more matches to the search then it responds with HTTP status 200 (OK) and the JSON list representation of the appropriate object types is contained within the body.

If no matches are found ProxStor responds with HTTP status 204 (No Content).

Note that using wildcards or regular expressions inside the fields of the objects is not currently supported.

#### 4.5.10 Administration URI

All operations related to the administration of ProxStor are relative to the admin URI:



Figure X: Admin URI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | /graph | POST | Content-Type: multipart/form-data | Create/connect to database instance |
| 2 | /graph | GET |  | Retrieve database instance |
| 3 | /graph | DELETE |  | Shutdown running database instance |

Table X: Admin URI Methods

The URI /graph is referred to as the base graph admin URI for convenience.

##### 4.5.10.1 Create Database Instance

To instruct ProxStor to create or connect to a backend database instance the administrator must send an HTTP POST to the admin + graph URI containing a multipart/form-data encoded in the URL. These form data elements will be converted into a Map<String, String> and passed to GraphFactory. This allows the administrator to connect ProxStor to any Graph instance supported by Blueprints, whether it’s a new instance of a database or a reconnection to an existing one. For more information on GraphFactory see [].

If the Graph instance is successfully created ProxStor will return an HTTP response of 200 (OK).

If a Graph instance is already running ProxStor will return an HTTP status of 403 (Forbidden).

If a Graph instance cannot be created from the form parameters provided an HTTP status of 500 (Internal Server Error) will be returned.

##### 4.5.10.2 Retrieve Database Instance

To retrieve information on the running database instance the administrator must send an HTTP GET to the admin + graph URI.

If a running database instance exists ProxStor will return an HTTP status of 200 (OK) and the body of the response will contain the plain text status.

If no running database instance exists ProxStor will return an HTTP status of 503 (Service Temporarily Unavailable).

##### 4.5.10.3 Shutdown Database Instance

To stop (shutdown) a running database instance the administrator must sent an HTTP DELETE request to the admin + graph URI.

If a running database instance exists ProxStor will stop that running (including committing all transactions to disk) and return HTTP status 200 (OK).

If a running database instance does not exist ProxStor will return HTTP status 404 (Not Found).

### 4.6 Dynamic Web Services Interfaces

The dynamic components of the web interface provide the following operations:

* Device check-in
* User check-in
* Query
  + Fixed Format
  + Complex

#### 4.6.1 Device Check-in URI

All Device check-in actions are relative to the URI:



Figure X: Device Check-in URI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | / | POST | Content-Type: application/json  Accept: application/json | Check-in (Environmental) |
| 2 | / | DELETE | Accept: application/json | Check-out  (Environmental) |
| 3 | /{sensorid} | POST |  | Check-in (Sensor) |
| 4 | /{sensorid} | DELETE |  | Check-out (Sensor) |

Table X: Device Check-in URI Methods

Here a device (devid) reports detecting a sensor or environmental artifact. The device is used by a User, and the sensor is in a Location. ProxStor will create a Locality instance associated with the User referencing the Location.

The URI /{sensorid} is referred to as the sensorId URI for convenience. The {sensorid} notation is meant to signify that the numeric database-specific sensor id is to be inserted in place of the {sensorid} string.

##### 4.6.1.1 Device Check-in (Environmental)

When a device detects a new environmental element it should report the discovery to the ProxStor service by creating a new proxstor.api.Sensor object and filling in the known data, such as type and identifier. This Sensor object must then converted into JSON and sent via a HTTP POST request to the device check-in URI with the header fields *Content-type* and *Accept* both set to *application/json*. If a locality is successfully created from the request ProxStor will return an HTTP status 201 (Created) with the new Locality available in the body of the response as well as indicated in the *Location* field in the header.

##### 4.6.1.2 Device Check-out (Environmental)

After a device successfully checks into a location using the above interface it must also monitor the environmental and notify ProxStor when the device no longer senses it. To report this check-out (no longer sensing environmental) the client must send an HTTP DELETE request to the base device check-in URI with the header field *Content-type* set to *application/json*. The body of the request should contain the partial proxstor.api.Sensor object used to check-in, or optionally the complete sensor object retrieved based on the sensorId from the locality object. ProxStor will respond with an HTTP status of 204 (No Content) if the check-out was successful.

##### 4.6.1.3 Device Check-in (Sensor)

If the client already knows the precise sensorId corresponding to the environmental being sensed it may use a more optimized non-JSON POSTing interface. The client sends an HTTP POST request to the device check-in + sensorId URI. The full URI provides ProxStor with the necessary information to associate a device with a sensor. If a locality is successfully created from the request ProxStor will return an HTTP status 201 (Created) with the new Locality available in the body of the response as well as indicated in the *Location* field in the header.

##### 4.6.1.4 Device Check-out (Sensor)

The same non-JSON POSTing interface can be used to check out of a location as well. The client sends an HTTP DELETE request to the device check-in + sensorId URI. The full URI provides ProxStor with the necessary information to dissociate a device from a sensor. ProxStor will respond with an HTTP status of 204 (No Content) if the check-out was successful.

#### 4.6.2 User Check-in URI

All User related check-in are relative to the URI:



Figure X: User Check-in URI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | /location/{locid} | POST | Accept: application/json | Check-in (Manual) |
| 2 | /location/{locid} | DELETE |  | Check-out (Manual) |
| 3 | / | GET | Accept: application/json | Retrieve current locality |

Table X: User Check-in URI Methods

Here a User (userid) reports being in Location {locid}. The request is taken literally. ProxStor will create a Locality instance associated with the User referencing the Location.

##### 4.6.2.1 User Check-in

If a user wishes to manually check into a location this may be achieved by sending an HTTP POST request to the user check-in + location URI. The request header field *Accept* should be set to *application/json*. If a locality is successfully created from the request ProxStor will return an HTTP status 201 (Created) with the new manually created Locality available in the body of the response as well as indicated in the *Location* field in the header.

##### 4.6.2.2 User Check-out

If a user wishes to manually check out of a location this may be achieved similar to the manual check-in process. The client sends an HTTP DELETE request to the user check-in + location URI, but this time there are no requirements on the request header. ProxStor will respond with an HTTP status of 204 (No Content) if the check-out was successful.

##### 4.6.2.3 Retrieve User Locality

To retrieve a specified user’s current locality the client must send an HTTP GET request to the base user check-in URI with the request header field *Accept* set to *application/json*. If the user from the URI has a currently active locality ProxStor will respond with an HTTP status of 200 (OK) containing the JSON representation of the Locality in the body. If the user is not currently in any active locality the response will be 204 (No Content).

#### 4.6.3 Query: Fixed Format URI

All fixed-format query requests are performed relative to the query URI:



Figure X: Query URI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **URI** | **Method** | **HTTP Header** | **Description** |
| 1 | / | POST | Content-Type : application/json  Accept: application/json | Submit Query |

Table X: Query URI Methods

##### 4.6.3.1 Submit Query

To submit a fixed format query to ProxStor the client must first prepare a JSON representation of proxstor.api.Query containing the appropriate defined fields. The client then sends an HTTP POST request to the query URI with the header fields *Content-type* and *Accept* both set to *application/json*. If the query is accepted then ProxStor will respond with an HTTP status of 200 (OK). The body of the response will contain the JSON list representation of the matching Localities. If the query was valid, but returned no results ProxStor will return an HTTP status of 204 (No Content).

#### 4.6.4 Query: Complex URI

TBD

## Chapter 5: Results

### Software Engineering Metrics

### Lessons Learned

#### What Worked

#### What Didn’t Work

## Chapter 6: Conclusion

### Relationship to Existing Work [?]

### Relationship to Prior Work [?]

### Summary

### Future Work

Gremlin

Optimization

Test scale out horizontally

Mobile application

### Obtaining ProxStor

Pointer to wiki…

Pointer to git repo…

Pointer to javadocs…

## References

**(citations not ready, complete, nor in order)**

[ ] Blueprints. <https://github.com/tinkerpop/blueprints/wiki>

[ ] Tinkerpop. <http://www.tinkerpop.com/>

[ ] RESTful Web services: The basics. <http://www.ibm.com/developerworks/library/ws-restful/index.html>

[ ] Jersey – RESTful Web Services in Java.

<https://jersey.java.net/>

[ ] HTTP Status Codes

<http://www.restapitutorial.com/httpstatuscodes.html>

[ ] Postman REST Client

<https://twitter.com/postmanclient>

[ ] orientdb

<http://www.orientechnologies.com/orientdb/>

[ ] Get started with Bluetooth Low Energy`

<http://www.jaredwolff.com/blog/get-started-with-bluetooth-low-energy/>

[ ] Winstone Servlet Container

<http://winstone.sourceforge.net/>

[ ] mongoDB

<http://www.mongodb.org/>

[ ] Neo4j

<http://neo4j.com/>

**[ ]** Ian Robinson , Jim Webber , Emil Eifrem, Graph Databases, O'Reilly Media, Inc., 2013

[ ] ProxStor Wiki

<https://github.com/jgiannoules/proxstor/wiki>

[ ] ProxStor Source Code

<https://github.com/jgiannoules/proxstor>

[ ] Atzori, Luigi, Antonio Iera, and Giacomo Morabito. “The internet of things: A survey.” *Computer networks* 54.15 (2010): 2787-2805.

[ ] Apache Maven

<http://maven.apache.org>

[ ] ArangoDB

<https://www.arangodb.com/>