National Institute of Standards and Technology
U.S. Department of Commerce

NIST Special Publication 500-288
Revision 0
Draft 1

Specification for WS-Biometric Devices (WS-BD)

Revision 0
Draft 1

Ross J. Micheals
Matt Aronoff
Kayee Kwong
Kevin Mangold
Karen Marshall

NIST Special Publication 500-288

Specification for WS-Biometric Devices (WS-BD)

Revision 0
Draft 1

Recommendations of the National Institute of Standards and Technology

Ross J. Micheals
Matt Aronoff
Kayee Kwong
Kevin Mangold
Karen Marshall

INFORMATION TECHNOLOGY

Biometric Clients Lab
Image Group
Information Access Division
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-8940

Spring 2011

US Department of Commerce
Gary Locke, Secretary

National Institute of Standards and Technology
Patrick D. Gallagher, Director

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analysis to advance the development and productive use of information technology.

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

Table of Contents

1 Introduction 1
1.1 Terminology 1
1.2 Documentation Conventions 2
1.2.1 Quotations 2
1.2.2 Machine-Readable Code 2
1.2.3 Sequence Diagrams 2
1.3 Normative References 2
2 Design Concepts and Architecture 1
2.1 Tiered Specifications 1
2.2 Interoperability 1
2.3 Architectural Components 1
2.3.1 Clients 2
2.3.2 Sensor 2
2.3.3 Sensor Service 2
2.4 Intended Use 2
2.5 General Service Behavior 4
2.5.1 Security Model 4
2.5.2 HTTP Request-Response Usage 4
2.5.3 Client Identity 5
2.5.4 Sensor Identity 5
2.5.5 Locking 6
2.5.6 Operations Summary 7
2.5.7 Idempotency 8
2.5.8 Service Lifecycle Behavior 9
3 Data Dictionary 10
3.1.1 Namespaces 10
3.1.2 UUID 10
3.1.3 WsbdDictionary 10
3.1.4 WsbdStringArray 11
3.1.5 WsbdUuidArray 11

3.1.6 WsbdStatus 11
3.2 WsbdResult 13
3.2.1 Terminology Shorthand 14
3.2.2 WsbdResult Required Elements 14
3.2.3 WsbdResult Element Summary 14
3.3 Validation 15
4 Operations 16
4.1 General Usage Notes 16
4.1.1 Precedence of Status Enumerations 16
4.1.2 Parameter Failures 17
4.1.3 Visual Summaries 18
4.2 Documentation Conventions 20
4.2.1 General Information 20
4.2.2 WsbdResult Summary 21
4.2.3 Usage Notes 21
4.2.4 Unique Knowledge 22
4.2.5 Return Values Detail 22
4.3 Register 23
4.3.1 WsbdResult Summary 23
4.3.2 Usage Notes 23
4.3.3 Unique Knowledge 23
4.3.4 Return Values Detail 23
4.4 Unregister 25
4.4.1 WsbdResult Summary 25
4.4.2 Usage Notes 25
4.4.3 Unique Knowledge 25
4.4.4 Return Values Detail 26
4.5 Try Lock 28
4.5.1 WsbdResult Summary 28
4.5.2 Usage Notes 28
4.5.3 Unique Knowledge 28
4.5.4 Return Values Detail 28
4.6 Steal Lock 31

4.6.1 WsbdResult Summary 31
4.6.2 Usage Notes 31
4.6.3 Unique Knowledge 32
4.6.4 Return Values Detail 32
4.7 Unlock 34
4.7.1 WsbdResult Summary 34
4.7.2 Usage Notes 34
4.7.3 Unique Knowledge 34
4.7.4 Return Values Detail 34
4.8 Get Common Info 36
4.8.1 WsbdResult Summary 36
4.8.2 Usage Notes 36
4.8.3 Unique Knowledge 36
4.8.4 Return Values Detail 36
4.9 Initialize 38
4.9.1 WsbdResult Summary 38
4.9.2 Usage Notes 38
4.9.3 Unique Knowledge 38
4.9.4 Return Values Detail 39
4.10 Get Configuration 42
4.10.1 WsbdResult Summary 42
4.10.2 Usage Notes 42
4.10.3 Unique Knowledge 42
4.10.4 Return Values Detail 42
4.11 Set Configuration 46
4.11.1 WsbdResult Summary 46
4.11.2 Usage Notes 46
4.11.3 Unique Knowledge 47
4.11.4 Return Values Detail 47
4.12 Capture 51
4.12.1 WsbdResult Summary 51
4.12.2 Usage Notes 51
4.12.3 Unique Knowledge 52

106 4.12.4 Return Values Detail 52
107 4.13 Download 56
108 4.13.1 WsbdResult Summary 56
109 4.13.2 Usage Notes 56
110 4.13.3 Unique Knowledge 60
111 4.13.4 Return Values Detail 60
112 4.14 Get Download Info 62
113 4.14.1 WsbdResult Summary 62
114 4.14.2 Usage Notes 62
115 4.14.3 Unique Knowledge 62
116 4.14.4 Return Values Detail 62
117 4.15 Thrifty Download 65
118 4.15.1 WsbdResult Summary 65
119 4.15.2 Usage Notes 65
120 4.15.3 Unique Knowledge 66
121 4.15.4 Return Values Detail 66
122 4.16 Cancel 68
123 4.16.1 WsbdResult Summary 68
124 4.16.2 Usage Notes 68
125 4.16.3 Unique Knowledge 69
126 4.16.4 Return Values Detail 69
127 Appendix A Common Info 71
128 A.1 Data Dictionary 71
129 A.1.1 H3 71
130 Appendix B Parameter Details 72
131 B.1 Connections 72
132 B.1.1 Last Updated 72
133 B.1.2 Inactivity Timeout 72
134 B.1.3 Maximum Concurrent Sessions 72
135 B.1.4 Least Recently Used (LRU) Sessions Automatically Dropped 73
136 B.2 Timeouts 73
137 B.2.1 Detailed Info Timeout 73
138 B.2.2 Initialization Timeout 73

B.2.3 Get Configuration Timeout 73
B.2.4 Set Configuration Timeout 74
B.2.5 Capture Timeout 74
B.2.6 Post-Acquisition Processing Time 74
B.2.7 Lock Stealing Prevention Period 74
B.3 Storage 74
B.3.1 Maximum Storage Capacity 74
B.3.2 Least-Recently Used Capture Data Automatically Dropped 75
Appendix C Configuration 76
C.1 Parameter Description 76
Appendix D Content Type Data 77
D.1 Content Type 77
D.2 Image Formats 77
D.3 Video Formats 77
D.4 General Biometric Formats 77
D.5 Modality-Specific Formats 77
Appendix E Security Profiles 78
E.1 None 78
E.2 HTTPS 78
E.3 OpenID & OAuth 78
Appendix F Live Preview 79
Appendix G Pending Issues 80
Appendix H Acknowledgments 81
Appendix I Revision History 82

1 Introduction

Text written in this style is reserved for editorial comments that have no design decisions.

Text written in this style is reserved for comments that represent open design decisions.

Imagine an intelligent biometric device that is secure, tamper-proof, and spoof resistant. Such a device would enable biometrics as a viable option for remote authentication. Imagine a new generation of fingerprint scanner, small enough and thin enough to clip onto a police officer's uniform—enabling law enforcement to quickly identify suspects. These envisioned devices require a communications protocol that is secure, globally connected, and free from requirements on operating systems, device drivers, form factors, and low-level communications protocols. WS-Biometric Devices is a protocol designed in the interest of furthering this goal, with a specific focus on the single process shared by all biometric systems—*acquisition*.

1.1 Terminology

This section contains terminology commonly used throughout this document. First time readers are encouraged to skip this section and revisit it as needed.

biometric capture device
: a system component capable of capturing biometric data in digital form

client
: a logical endpoint that originates operation requests

HTTP
: Hypertext Transfer Protocol. Unless specified, the term HTTP may refer to either HTTP as defined in [RFC2616] or HTTPS as defined in [RFC2660].

payload
: the content of an HTTP request or response. An **input payload** refers to the XML content of an HTTP *request*. An **output payload** refers to the XML content of an HTTP *response.*

payload parameter
: an operation parameter that is passed to a service within an input payload

REST
: Representational State Transfer

sensor or **biometric sensor**
: a single biometric capture device or a logical collection of biometric capture devices

submodality
: a distinct category or subtype within a biometric modality

target sensor or **target biometric sensor**
: the biometric sensor exposed by a particular service

URL parameter

a parameter passed to a web service by embedding it to the URL

Web service or **service**

a software system designed to support interoperable machine-to-machine interaction over a network [WSGloss]

XML

Extensible Markup Language [XML]

1.2 Documentation Conventions

The following documentation conventions are used throughout this document.

1.2.1 Quotations

If the inclusion of a period within a quotation might lead to ambiguity as to whether or not the period should be included in the quoted material, the period will be placed outside the trailing quotation mark. For example, a sentence that ends in a quotation would have the trailing period "inside the quotation, like this quotation punctuated like this." However, a sentence that ends in a URL would have the trailing period outside the quotation mark, such as "`http://example.com`".

1.2.2 Machine-Readable Code

With the exception of some reference URLs, Machine readable information will typically be depicted with a `mono-spaced font, such as this`.

1.2.3 Sequence Diagrams

Throughout this document, sequence diagrams are used to help explain various scenarios. These diagrams are informative simplifications and are intended to help explain core specification concepts. Operations are depicted in a functional, remote procedure call style. The level of abstraction presented in the diagrams, and the details that are shown (or not shown) will vary according to the particular information being illustrated.

1.3 Normative References

[RFC2045] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, *http://www.ietf.org/rfc/rfc2045.txt,* IETF RFC 2045, November 1996.

[RFC2046] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, *http://www.ietf.org/rfc/rfc2045.txt,* IETF RFC 2045, November 1996.

[RFC2119] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, *http://www.ietf.org/rfc/rfc2119.txt*, IETF RFC 2119, March 1997.

[RFC2616] R. Fielding, et al. Hypertext Tranfer Protocol—HTTP/1.1, *http://www.ietf.org/rfc/rfc2616.txt*, IETF RFC 2616, June 1999.

[RFC2660] E. Rescorla, et al. *The Secure HyperText Transfer Protocol*, *http://www.ietf.org/rfc/rfc2660.txt*, IETF RFC 2660, August 1999.

[RFC4122] P. Leach, M. Mealling, and R. Salz, *A Universally Unique Identifier (UUID) URN Namespace, http://www.ietf.org/rfc/rfc4122.txt*, IETF RFC 4122, July 2005.

[SP500-288A] *NIST Special Publication 500-288 A. Conformance Profiles for WS-Biometric Devices*.

[WSGloss] H. Haas, A. Brown, *Web Services Glossary*, *http://www.w3.org/TR/ws-gloss/,* February 11, 2004.

[XML] Tim Bray, et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, *http://www.w3.org/TR/xml/* W3C Recommendation. November 26, 2008.

[XSDPart1] *XML Schema Part 1: Structures Second Edition*, *http://www.w3.org/TR/xmlschema-1*, W3C Recommendation. 28 October 2004.

[XSDPart2] *XML Schema Part 2: Datatypes Second Edition*, *http://www.w3.org/TR/xmlschema-2*, W3C Recommendation. 28 October 2004.

2 Design Concepts and Architecture

This section describes the major design concepts and overall architecture of WS-BD. The main purpose of a WS-BD service is to expose a target biometric sensor to clients via web services.

2.1 Tiered Specifications

This specification represents the first in what is envisioned as a series of specifications. Each individual specification describes a "level"—a numbered collection of capabilities and features. To help preserve compatibility, additional capabilities will introduced in higher specification levels. Each level should fully contain the capabilities of its previous level. As the first level, Level 1, this specification provides the foundation from which other specifications will be derived.

The following is a brief description of the general capabilities expected to be addressed in the planned levels of the specification.

- **Level 1**. Core synchronous acquisition operations via lightweight web service protocols (REST).
- **Level 2**. Synchronous and asynchronous acquisition operations. Asynchronous messaging might be achieved via (SOAP-based) web services running on both the client and server or via polling. Live preview capabilities will be an inherent feature of the specification. Fine-grained authorization management might be included at this level.
- **Level 3**. Device discovery and workflow management. The functionality at this level may be better described by profiling existing web service standards.

It is expected that this "roadmap" will change and accommodate feedback and evolving requirements. Levels are intended to facilitate backwards compatibility. As a design philosophy, higher levels specifications should, as much as possible, provide *additional*, rather *substitute* functionality.

2.2 Interoperability

ISO/IEC 2382-1 (1993) defines *interoperability* as "*the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little to no knowledge of the unique characteristics of those units.*"

Conformance to a standard does not necessarily guarantee interoperability. *There should be an example here of where this happens by design.*

A major design goal of WS-BD is to *maximize* interoperability, by *minimizing* the required "knowledge of the unique characteristics" of a component that supports WS-BD.

There should be an additional note regarding conformance profiles.

2.3 Architectural Components

Before discussing the envisioned use of WS-BD, it may be useful to distinguish between the various components that might comprise a WS-BD implementation. These are *logical* components, and may or may

not correspond to particular *physical* boundaries. This distinction becomes vital in understanding WS-BD's operational models.

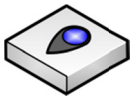
2.3.1 Clients

A *client* is any software component that originates requests for biometric acquisition. Note that a client might be one of many hosted in a parent (logical or physical) component, and that a client might originate requests to a variety of destinations.

This icon is used to depict an arbitrary WS-BD client. A personal digital assistant (PDA) is used to serve as a reminder that a client might be hosted on a non-traditional computer.

2.3.2 Sensor

A biometric *sensor* is any component that is capable of acquiring, i.e., digitally sampling, a biometric. Most sensor components are hosted within a dedicated hardware component, but this is not necessarily globally true. For example, a keyboard is a general input device, but might also be used for a keystroke dynamics biometric.



This icon is used to depict a biometric device. The icon has a vague similarity to a fingerprint scanner, but should be thought of as an arbitrary biometric sensor.

As discussed in §2.1, the term "sensor" in used in this document in a singular sense, but may in fact be referring to multiple biometric capture devices.

2.3.3 Sensor Service

The *sensor service* is the "middleware" software component that exposes a biometric sensor to a client through web services. The sensor service adapts HTTP request-response operations to biometric sensor command & control.

This icon is used to depict a sensor service. The icon is abstract and has no meaningful form, just as a sensor service is a piece of software that has no physical form.

2.4 Intended Use

Each implementation of WS-BD will be realized via a mapping of logical to physical components. A distinguishing characteristic of an implementation will be the physical location of the sensor service component. WS-BD is designed to support two scenarios.

- The sensor service and biometric sensor are hosted by different physical components. A *physically separated service* is one where there is both a physical and logical separation between the biometric sensor and the service that provides access to it.

- The sensor service and biometric sensor are hosted within the same physical component. A *physically integrated service* is one where the biometric sensor, and the service that provides access to it, reside within the same physical component.

Figure 1 depicts a physically separated service. In this scenario, a biometric sensor is tethered to a personal computer, workstation, or server. The web service, hosted on the computer, listens for communication requests from clients. An example of such an implementation would be a USB fingerprint scanner attached to a personal computer. A lightweight web service, running on that computer could listen to requests from local (or remote) clients—translating WS-BD requests to and from biometric sensor commands.

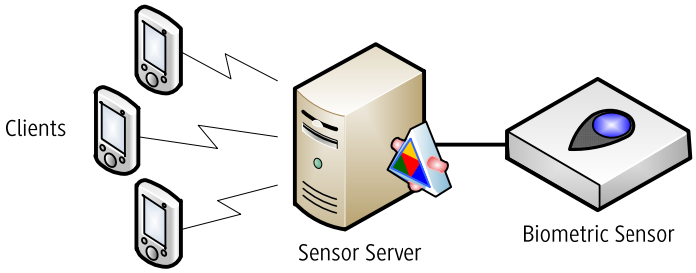


Figure 1. A physically separated WS-BD implementation.

Figure 2 depicts a physically integrated service. In this scenario, a single hardware device has an embedded biometric sensor, as well as a web service. Similar functionality is seen in many network printers; it is possible to point a web browser to a local network address, and obtain a web page that displays information about the state of the printer, such as toner and paper levels. Clients make requests directly to the integrated device; and a web service running within an embedded system translates the WS-BD requests to and from biometric sensor commands.

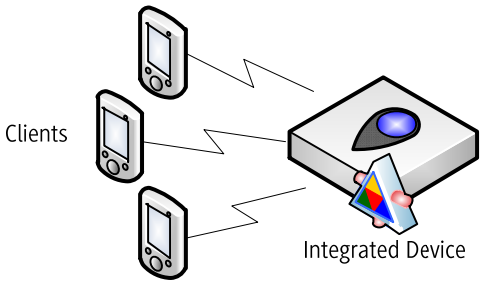


Figure 2. A physically integrated WS-BD implementation.

Finally, it should be admitted that from a systems engineering viewpoint, the "separate" versus "integrated" distinction is indeed a simplification with a high degree of fuzziness. For example, one might imagine putting a hardware shell around a USB fingerprint scanner connected to a small form-factor computer. Inside the shell, the sensor service and sensor are on different physical components. Outside the shell, the sensor service and sensor appear integrated. The definition of what constitutes the "same" physical component depends on the particular implementation, and the intended level of abstraction. Regardless, it is a useful

distinction in that it illustrates the flexibility afforded by leveraging highly interoperable communications protocols.

2.5 General Service Behavior

The following section describes the general behavior of WS-BD clients and services.

2.5.1 Security Model

In this version of the specification, it is assumed that if a client is able to establish a HTTP (or HTTPS) communication with the sensor service, then the client is fully authorized to use the service. This implies that all successfully connected clients which have equivalent authority may be considered *peers*.

Clients might be prevented by connecting through various HTTP protocols, such as HTTPS with client-side certificates, or a more sophisticated protocol such as OpenId (http://openid.net/) and/or OAuth.

Recommended security profiles are discussed in Appendix E.

2.5.2 HTTP Request-Response Usage

Most biometrics devices are inherently *single user*—i.e., they are designed to sample the biometrics from a single user at a given time. Web services, on the other hand, are inherently designed to be *stateless* and *multiuser.* A biometric device exposed via web services must therefore provide a mechanism to reconcile these competing designs.

Notwithstanding the native limits of the underlying web server, WS-BD services *must* be capable of handling multiple, concurrent requests. Services *must* respond to requests for operations that do not require exclusive control of the biometric sensor and must do so without waiting until the biometric sensor is in a particular state.

Because there is no well-accepted mechanism for providing asynchronous notification via REST, each individual operation *must* block until completion. Clients are not expected to poll—rather make a single HTTP request and block for the corresponding result. Because of this, it is expected that a client would perform WS-BD operations on an independent thread, so not to interfere with the general responsiveness of the client application. WS-BD clients therefore *must* be configured in such a manner such that individual HTTP operations have timeouts that are compatible with a particular implementation.

WS-BD operations may be longer than typical REST services. Consequently, there is a clear need to differentiate between service level errors and HTTP communication errors. WS-BD services *must* pass-through the status codes underlying a particular request. In other words, services *must not* use (or otherwise 'piggyback') HTTP return codes to indicate service-level failure. If a service receives a well-formed request, then the service MUST return an 'OK' response code (200). Service failures are described within the contents of the XML data returned to the client for any given operation. *There may need to be a clarification about how the set configuration can return 400 when it receives malformed (as in malformed XML) requests.*

This is deliberately different from REST services that override HTTP return codes to provide service-specific error messages. This design pattern facilitates clearer communication of failures and errors than overriding

particular HTTP failure codes. It also provides better support for HTTP libraries that do not easily support HTTP return code overrides.

2.5.3 Client Identity

Before discussing how WS-BD balance single-user vs. multi-user needs, it is necessary to understand the WS-BD model for how an individual client can easily and consistently identify itself to a service.

HTTP is, by design, a *stateless* protocol. Therefore, any persistence about the originator of a sequence of requests must be built in (somewhat) artificially to the layer of abstraction above HTTP itself. This is accomplished in WS-BD via a *session*—a collection of operations that originate from the same logical endpoint. To initiate a session, a client performs a *registration* operation and obtains a *session identifier* (or "session id"). During subsequent operations, a client uses this identifier as a parameter to uniquely identify itself to a server. When the client is finished, it is expected to close a session with an *unregistration* operation. To conserve resources, services may automatically unregister clients that do not explicitly unregister after a period of inactivity (see §4.4.2.1).

This use of a session id directly implies that the particular sequences that constitute a session are entirely the responsibility of the *client*. A client might opt to create a single session for its entire lifetime, or, might open (and close) a session for a limited sequence of operations. WS-BD supports both scenarios.

It is not recommended, but it is possible, that a client might maintain multiple sessions with the same service simultaneously. This can be assumed without loss of generality, since a client with multiple sessions to a service could be decomposed into to "sub-clients"—one (sub-) client per session id.

Just as a client might maintain multiple session ids, a single session id might be shared among a collection of clients. By sharing the session id, a biometric sensor may then be put in a particular state by one client, and then handed-off from another. Such a sophisticated implementation is outside the scope of this specification. However, session id sharing is certainly permitted, and a deliberate artifact of the convention of using of the session id as the client identifier. Many-to-many relationships (i.e., multiple session ids being shared among multiple clients) are also possible, but should be avoided.

For simplicity, unless otherwise stated, this specification is written in a manner that assumes that a single client maintains a single session id.

2.5.4 Sensor Identity

In general, implementers *should* map each target biometric sensor to a single URL. However, just as it is possible for a client to communicate with multiple services, a host might be responsible for controlling multiple target biometric sensors.

A service that controls independent sensor should expose each device via different URLs. A service that controls multiple biometric devices simultaneously, however, *should* be exposed via the same endpoint. The following figures and example illustrates this distinction.

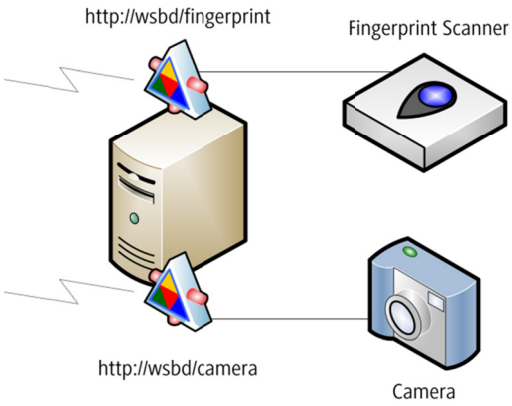


Figure 3. Independent sensors controlled by separate services

EXAMPLE: Figure 3 shows a physically separate implementation where a single host machine controls two biometric sensors—one fingerprint scanner, and one digital camera. The devices act independently and are therefore exposed via two different services—one at the URL `http://wsbd/fingerprint`, and one at `http://wsbd/camera`.

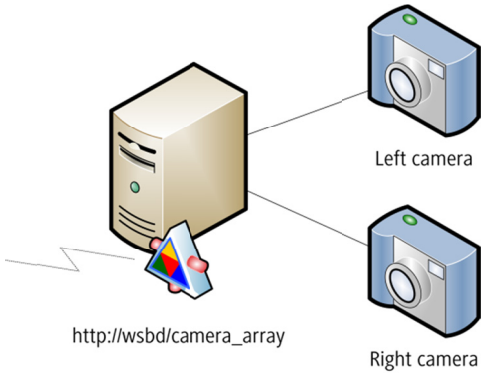


Figure 4. A sensor array controlled by a single service

EXAMPLE: Figure 4 shows a physically separate implementation where a single host machine controls a pair of cameras used for stereo vision. The cameras act together as a single logical sensor, and are both exposed via the same service, `http://wsbd/camera_array`.

2.5.5 Locking

WS-Biometric Devices (WS-BD) uses a *lock* to satisfy two complimentary requirements:

1. A service must have exclusive, sovereign control over biometric sensor hardware to perform a particular *sensor operation* such as initialization, configuration, or capture.
2. A client needs to perform an uninterrupted sequence of sensor operations.

Each WS-BD service exposes a *single* lock (one per service) that controls access to the sensor. Clients obtain the lock in order to perform a sequence of operations that should not be interrupted. Obtaining the lock is an

indication to the server (and indirectly to peer clients) that (1) a series of sensor operations is about to be initiated and (2) that server may assume sovereign control of the biometric sensor.

A client releases the lock upon completion of its desired sequence of tasks. This indicates to the server (and indirectly to peer clients) that the uninterruptable sequence of operations is finished. A client might obtain and release the lock many times within the same session or a client might open and close a session for each pair of lock/unlock operations. This decision is entirely dependent on a particular client.

The statement that a client might "own" or "hold" a lock is a convenient simplification that makes it easier to understand the client-server interaction. In reality, each sensor service maintains a unique global variable that contains a session id. The originator of that session id can be thought of as the client that "holds" the lock to the service. Clients are expected to release the lock after completing their required sensor operations, but there is lock *stealing*—a mechanism for forcefully releasing locks. This feature is necessary to ensure that one client cannot hold a lock indefinitely, denying its peers access to the biometric sensor.

As stated previously (see §2.5.3), it is implied that all successfully connected clients enjoy the same access privileges. Each client is treated the same and are expected to work cooperatively with each other. This is critically important, because it is this implied equivalence of "trust" that affords a lock *stealing* operation.

2.5.5.1 Pending Operations

Changing the state of the lock *must* have no pending (i.e., currently running) sensor operations. That is, a client may unlock, steal, or even re-obtain the service lock even if the target biometric sensor is busy. When lock ownership is transferred during a sensor operation, overlapping sensor operations are prevented by sensor operations returning `sensorBusy`.

2.5.6 Operations Summary

All WS-BD operations fall into one of eight categories:

1. Registration
2. Locking
3. Information ('Info')
4. Initialization
5. Configuration
6. Capture
7. Download
8. Cancelation

Of these, the initialization, configuration, capture, and cancelation operations are all sensor operations (i.e., they require exclusive sensor control) and require locking. Registration, locking, and download are all non-sensor operations. They do not require locking and (as stated earlier) must be available to clients regardless of the status of the biometric sensor.

There are two information operations, one that is a sensor operation, one that is not. This allows a client to obtain information that must be obtained from the sensor hardware itself, such as a firmware version. *Download* is not a sensor operation as this allows for a collection of clients to dynamically share acquired biometric data. One client might perform the capture and hand off the download responsibility to a peer.

The following is a brief summary of each type of operation:

- *Registration* operations open and close (unregister) a session.
- *Locking* operations are used by a client to obtain the lock, release the lock, and *steal* the lock.
- *Information* operations query the service for information about the service itself, such as the supported biometric modalities, and common service configuration parameters.
- The *initialization* operation prepares the biometric sensor for operation.
- *Configuration* operations get or set sensor parameters.
- The *capture* operation signals to the sensor to acquire a present biometric
- *Download* operations transfer the captured biometric data from the service to the client.
- Sensor operations can be stopped by the *cancellation* operation.

2.5.7 Idempotency

The W3C Web Services glossary [WSGloss] defines idempotency as:

> *[the] property of an interaction whose results and side-effects are the same whether it is done one or multiple times.*

When regarding an operation's idempotence, it should be assumed no *other* operations occur in between successive operations, and that each operation is successful. Notice that idempotent operations *may* have side-effects—but the final state of the service must be the same over multiple (uninterrupted) invocations.

The following example illustrates idempotency using an imaginary web service.

> **EXAMPLE**: A REST-based web service allows clients to create, read, update, and delete customer records from a database. A client executes an operation to update a customer's address from "123 Main St" to "100 Broad Way."
>
> Suppose the operation is idempotent. Before the operation, the address is "123 Main St". After one execution of the update, the server returns "success", and the address is "100 Broad Way". If the operation is executed a second time, the server again returns "success," and the address remains "100 Broad Way".
>
> Now suppose that when the operation is executed a second time, instead of returning "success", the server returns "no update made", since the address was already "100 Broad Way." Such an operation is *not* idempotent, because executing the operation a second time yielded a different result than the first execution.

The following is an example in the context of WS-BD.

> **EXAMPLE**: A service has an available lock. A client invokes the lock operation and obtains a "success" result. A subsequent invocation of the operation also returns a "success" result. The operation being idempotent means that the results ("success") and side-effects (a locked service) of the two sequential operations are identical.

To best support robust communications, WS-BD is designed to offer idempotent services whenever possible.

2.5.8 Service Lifecycle Behavior

The lifecycle of a service (i.e, when the service starts responding to requests, stops, or is otherwise unavailable) *must* be modeled after an integrated implementation. This is because it is significantly easier for a physically separated implementation to emulate the behavior of a fully integrated implementation than it is the other way around. This requirement has a direct effect on the expected behavior of how a physically separated service would handle a change in the target biometric sensor.

Specifically, on a desktop computer, hot-swapping the target biometric sensor is possible through an operating system's plug-and-play architecture. By design, this specification does not assume that it is possible to replacing a biometric sensor within an integrated device. Therefore, having a physically separated implementation emulate an integrated implementation provides a simple means of providing a common level of functionality.

By virtue of the stateless nature of the HTTP protocol, a client has no simple means of detecting if a web service has been restarted. For most web communications, a client should not require this—it is a core capability that comprises the robustness of the web. Between successive web requests, a web server might be restarted on its host any number of times. In the case of WS-BD, replacing an integrated device with another (configured to respond on the same endpoint) is an *effective* restart of the service. Therefore, by the emulation requirement, replacing the device within a physically separated implementation must behave similarly.

A client may not be directly affected by a service restart, if the service is written in a robust manner. For example, upon detecting a new target biometric sensor, a robust server could *quiesce* (refusing all new request until any pending requests are completed) and automatically restart.

Upon restarting, services *may* return to a fully reset state—i.e., all sessions should be dropped, and the lock *should not* have an owner. However, a high-availability service *may* have a mechanism to preserve state across restarts, but is significantly more complex to implement (particularly when using integrated implementations!). A client that communicated with a service that was restarted would lose both its session and the service lock (if held). With the exception of the *get common info* operation, through various fault statuses a client would receive indirect notification of a service restart. If needed, a client could use the service's common info timestamp (§B.1.1) to detect potential changes in the *get common info* operation.

3 Data Dictionary

This section contains descriptions of the data elements that are contained within the WS-BD data model. Each data type is described via an accompanying XML Schema type definition [XSDPart1, XSDPart2].

3.1.1 Namespaces

The following namespaces, and corresponding namespace prefixes are used throughout this document.

Prefix	Namespace	Remarks
xs	`http://www.w3.org/2001/XMLSchema`	The `xs` namespace refers to the XML Schema specification. Definitions for the `xs` data types (i.e., those not explicitly defined here) can be found in [XSDPart2].
wsbd	`http://nist.gov/itl/bws/ws-bd/L1/r0/`	The `wsbd` namespace is an URL that provides a globally unique name—it does not resolve to an active website.

It should be assumed that each of the following data types is defined in the target namespace `wsbd`.

The remaining examples should be vetted for correctness, including schema validation.

3.1.2 UUID

A UUID is a unique identifier as defined in [RFC4122]. A service *must* use UUIDs that conform to the following XML Schema type definition.

```
<xs:simpleType name="uuid">
 <xs:restriction base="xs:string">
  <xs:pattern value="[\da-fA-F]{8}-[\da-fA-F]{4}-[\da-fA-F]{4}-[\da-fA-F]{4}-[\da-fA-F]{12}"/>
  </xs:restriction>
</xs:simpleType>
```

EXAMPLE: Each line in the following code fragment contains a well-formed UUID.

```
E47991C3-CA4F-406A-8167-53121C0237BA
10fa0553-9b59-4D9e-bbcd-8D209e8d6818
161FdBf5-047F-456a-8373-D5A410aE4595
```

3.1.3 WsbdDictionary

A WsbdDictionary is a generic container used to hold an arbitrary collection of name-value pairs.

```
<xs:complexType name="WsbdDictionary">
  <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="unbounded" name="item">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="key" nillable="true" type="xs:string" />
```

```
          <xs:element name="value" nillable="true" type="xs:anyType" />
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

EXAMPLE: A query to get the metadata of a sensor return a dictionary of supported settings and their current value.

WsbdDictionary instances are nestable—i.e., the `value` element of one WsbdDictionary can contain another WsbdDictionary . *There should be a note about the use of xs:anyType, and how conformance is (or is not?) affected by the use of types outside of those not documented here.*

3.1.4 WsbdStringArray

A WsbdStringArray is a generic container used to hold a collection of strings.

```
<xs:complexType name="WsbdStringArray">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="element" nillable="true"
type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
```

EXAMPLE: Each line in the following code fragment is an example of a valid WsbdStringArray. Enclosing tags (which may vary) are omitted.

```
<element>sessionId</element>
<element>value1</element><element>value2</element>
<element>leftThumb</element><element>rightThumb</element>
```

3.1.5 WsbdUuidArray

A WsbdUuidArray is a generic container used to hold a collection of UUIDs.

```
<xs:complexType name="WsbdUuidArray">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="element" nillable="true"
type="wsbd:uuid" />
    </xs:sequence>
  </xs:complexType>
```

EXAMPLE: The following code fragment is an example of a *single* WsbdUuidArray with three elements. Enclosing tags (which may vary) are omitted.

```
<element>E47991C3-CA4F-406A-8167-53121C0237BA</element>
<element>10fa0553-9b59-4D9e-bbcd-8D209e8d6818</element>
<element>161FdBf5-047F-456a-8373-D5A410aE4595</element>
```

3.1.6 WsbdStatus

The WsbdStatus represents a common enumeration for communicating state information about a service.

```
<xs:simpleType name="WsbdStatus">
    <xs:restriction base="xs:string">
```

```
      <xs:enumeration value="success" />
      <xs:enumeration value="failure" />
      <xs:enumeration value="invalidId" />
      <xs:enumeration value="canceled" />
      <xs:enumeration value="canceledWithSensorFailure" />
      <xs:enumeration value="sensorFailure" />
      <xs:enumeration value="lockNotHeld" />
      <xs:enumeration value="lockHeldByAnother " />
      <xs:enumeration value="sensorNeedsInitialization" />
      <xs:enumeration value="sensorNeedsConfiguration" />
      <xs:enumeration value="sensorBusy" />
      <xs:enumeration value="sensorTimeout" />
      <xs:enumeration value="unsupported" />
      <xs:enumeration value="badValue" />
      <xs:enumeration value="noSuchParameter" />
      <xs:enumeration value="preparingDownload" />
    </xs:restriction>
  </xs:simpleType>
```

3.1.6.1 Definitions

The following table defines all of the potential values for the WsbdStatus enumeration.

Value	Description
success	The operation completed successfully.
failure	The operation failed. The failure was due to a web service (as opposed to a *sensor* error).
invalidId	The provided id is not valid. This can occur if the client provides a (session or capture) id that is either: (a) unknown to the server (i.e., does not correspond to a known registration or capture result), or (b) the session has been marked inactive because too much time has passed between operations associated with the provided (session) id
canceled	The operation was canceled. NOTE: A sensor service might cancel its own operation if it is taking too long. This can happen if a service maintains its own internal timeout that is shorter than a sensor timeout.
canceledWithSensorFailure	The operation was canceled, but during (and perhaps because of) cancellation, a sensor failure occurred. This particular status accommodates for hardware that may not natively support cancelation.
sensorFailure	The operation could not be performed because of a biometric sensor (as opposed to web service) failure. **NOTE**: Clients that receive a status of *sensorFailure should* assume that the sensor will need to be reinitialized in order to restore normal operation.
lockNotHeld	The operation could not be performed because the client does not hold the lock. **NOTE**: This status implies that at the time the lock was queried, no other client currently held the lock. However, this is not a guarantee that any subsequent attempts to obtain the lock will succeed.
lockHeldByAnother	The operation could not be performed because another client currently holds

	the lock.
initializationNeeded	The operation could not be performed because the sensor requires initialization.
configurationNeeded	The operation could not be performed because the sensor requires configuration. *Do we require that every service provide a default configuration? If so, then would any operation return this?*
sensorBusy	The operation could not be performed because the sensor is currently perf another task. **NOTE**: Services may self-initiate an activity that triggers a *sensorBusy* result. That is, it may not be possible for a client to trace back a *sensorBusy* status to any particular operation. An automated self-check, heartbeat, or other activitity such as a data transfer may place the target biometric sensor into a "busy" mode. (See §4.13.2.2 for information about post-acquisition processing.)
sensorTimeout	The operation was not performed because the biometric sensor experienced a timeout. **NOTE**: The most common cause of a sensor timeout would be a lack of interaction with a sensor within an expected timeframe.
unsupported	The service does not support the requested operation.
badValue	The operation could not be performed because a value provided for a particular parameter was either (a) an incompatible type or (b) outside of an acceptable range.
noSuchParameter	The operation could not be performed because the service did not recognize the name of a provided parameter.
preparingDownload	The operation could not be performed because the service is currently preparing captured data for download. (See §4.13.2.2)

Many of the permitted status values have been designed specifically to support physically separate implementations—a scenario where it is easier to distinguish between failures in the web service and failures in the biometric sensor. This is not to say that within an integrated implementation such a distinction is not possible, only that some of the status values are more relevant for physically separate versions.

For example, a robust service would allow all sensor operations to be canceled with no threat of a failure. Unfortunately, not all commercial, off-the-shelf (COTS) sensors natively support cancellation. Therefore, the *canceledWithSensorFailure* status is offered to accommodate this. Implementers can still offer cancellation, but have a mechanism to communicate back to the client, that sensor initialization might be required.

3.2 WsbdResult

Unless a service returns with an HTTP error, all WS-BD operations *must* reply with an HTTP message that contains a WsbdResult that conforms to the following XML Schema snippet.

```
<xs:complexType name="WsbdResult">
 <xs:sequence>
  <xs:element minOccurs="1" name="status" type="wsbd:WsbdStatus"/>
  <xs:element minOccurs="0" name="badFields " nillable="true" type="wsbd:WsbdStringArray"/>
  <xs:element minOccurs="0" name="captureIds " nillable="true" type="wsbd:WsbdUuidArray"/>
```

```
  <xs:element minOccurs="0" name="commonInfo" nillable="true" type="wsbd:WsbdDictionary"/>
  <xs:element minOccurs="0" name="configuration" nillable="true" type="wsbd:WsbdDictionary"/>
  <xs:element minOccurs="0" name="message" nillable="true" type="xs:string"/>
  <xs:element minOccurs="0" name="contentType" nillable="true" type="xs:string"/>
  <xs:element minOccurs="0" name="submodality" nillable="true" type="xs:string"/>
  <xs:element minOccurs="0" name="sensorData" nillable="true" type="xs:base64Binary"/>
  <xs:element minOccurs="0" name="sessionId" nillable="true" type="wsbd:uuid"/>
</xs:sequence>
</xs:complexType>
```

3.2.1 Terminology Shorthand

Since a WsbdResult is the intended outcome of all requests, this document may state that an operation "returns" a particular status value. This is shorthand for a WsbdResult output payload with a `status` element containing that value.

> **EXAMPLE:** The following output payload "returns `success`". Fill in example.

Likewise, the same shorthand is implied by a client "receiving" a status, or an operation "yielding" a status.

3.2.2 WsbdResult Required Elements

Notice that from a XML Schema validation perspective [XSDPart1], a schema-valid WsbdResult must contain a `status` element, and may or may not contain any of the remaining elements.

The specific permitted elements of a WsbdResult are determined via a combination of (a) the operation, and (b) the result's `status`. That is, different operations will have different requirements on which elements are permitted or forbidden, depending that operation's status.

> **EXAMPLE**: As will be detailed later (§4.3.4.1 and §4.5.4.1), a *register* operation returning a `status` of `success` must also populate the `sessionId` element. However, a *try lock* operation that returns a `status` of `success` cannot populate any element other than `status`.

DESIGN NOTE: An XML inheritance hierarchy could have been used to help enforce which elements are permitted under which circumstances. However, a de-normalized representation (in which all of the possible elements are valid with respect to a *schema*) was used to simplify client and server implementation. Futher, this reduces the burden of managing an object hierarchy for the sake of enforcing simple constraints.

3.2.3 WsbdResult Element Summary

The following is a brief informative description of each WsbdResult element.

Element	Description
`status`	The disposition of the operation. All WsbdResults *must* contain a `status` element. (Used in all operations.)
`badFields`	The list of fields that contain invalid or ill-formed values. (Used in almost all operations.)
`captureIds`	Identifiers that may be used to obtain data acquired from a capture operation (§4.12, §4.13).
`commonInfo`	Service metadata that is independent of session and does not require control of the target biometric sensor (§4.8).
`configuration`	Information about the target biometric sensor (§4.10, §4.11).

message	A string providing *informative* detail regarding the output of an operation. (Used in almost all operations.)
contentType	The format of the biometric data for the corresponding capture identifier (§4.13, §4.14, §4.15).
submodality	The submodality of the biometric data for the corresponding capture identifier (§4.13, §4.14, §4.15). The submodality is used to distinguish among biometrics belonging to the same (major) modality. For example, a multi-finger scanner might use `submodality` to distinguish between "flat left index" and a "rolled right little" fingers.
sensorData	The biometric data corresponding to a particular capture identifier (§4.13, §4.15).
sessionId	A unique session identifier (§4.3).

3.3 Validation

This section should contain information regarding validation of the input and output XML with respect to schemas. It should have guidance on

- *the fact that the XML schema validation is fairly weak, and that additional 'business logic' validation is not encapsulated in the schemas themselves—the biggest example being wsbdresult, a minor example being forbidden session Ids*
- *where/if the full schemas for wsbd are externally available in machine-readable form (i.e., not within this document)*
- *how a service might behave it is receives xml that is not valid w.r.t. the schemas—in particular qualifiying datatypes in the dictionary.*

4 Operations

This section provides detailed information regarding each WS-BD operation.

4.1 General Usage Notes

The following usage notes apply to all operations, unless the detailed documentation for a particular operation conflicts with these general notes, in which case the detailed documentation takes precedence.

1. **Failure messages are informative.** If an operation fails, then the message element *may* contain an informative message regarding the nature of that failure. The message is for informational purposes only—the functionality of a client *must not* depend on the contents of the message.
2. **Results must only contain required and optional elements.** Services *must only* return elements that are either required or optional. All other elements must *not* be contained in the result, even if they are empty elements. Likewise, to maintain robustness in the face of a non-conformant service, clients *should* ignore any element that is not in the list of permitted WsbdResult elements for a particular operation call.
3. **Sensor operations must not occur within a non-sensor operation.** Services *must only* perform any sensor control within the operations *get common info*, *initialize*, *get configuration*, *set configuration*, *capture*, and *cancel*. Similarly, services are permitted (but not required) to perform operations that control the biometric sensor *only* within the following operations:
4. **Sensor operations must require locking.** Even if a service implements a sensor operation without controlling the target biometric sensor.
5. **Content Type.** Clients *must* make HTTP requests using a content type of `application/xml` [RFC2616, §14].

4.1.1 Precedence of Status Enumerations

To maximize the amount of information given to a client when an error is obtained, and to prevent different implementations from exhibiting different behaviors, all WS-BD services *must* return status values according to a fixed priority. In other words, when multiple status messages might apply, a higher-priority status *must* always be returned in favor of a lower-priority status.

The status priority, listed from highest priority ("`invalidId`") to lowest priority ("`success`") is as follows:

1. `invalidId`
2. `noSuchParameter`
3. `badValue`
4. `unsupported`
5. `canceledWithSensorFailure`
6. `canceled`
7. `lockHeldByAnother`
8. `lockNotHeld`
9. `sensorBusy`
10. `sensorFailure`
11. `sensorTimeout`
12. `initializationNeeded`

13. `configurationNeeded`
14. `preparingDownload`
15. `failure`
16. `success`

Notice that success is the *lowest* priority—an operation should only be deemed successful if no *other* kinds of (non-successful) statuses apply.

The following examples illustrates how this ordering effects the status returned in a situation in which multiple

EXAMPLE: Figure 5 illustrates that client cannot receive a "sensorBusy" status if it does not hold the lock, even if a sensor operation is in progress (recall from §2.5.5 that sensor operations require holding the lock).

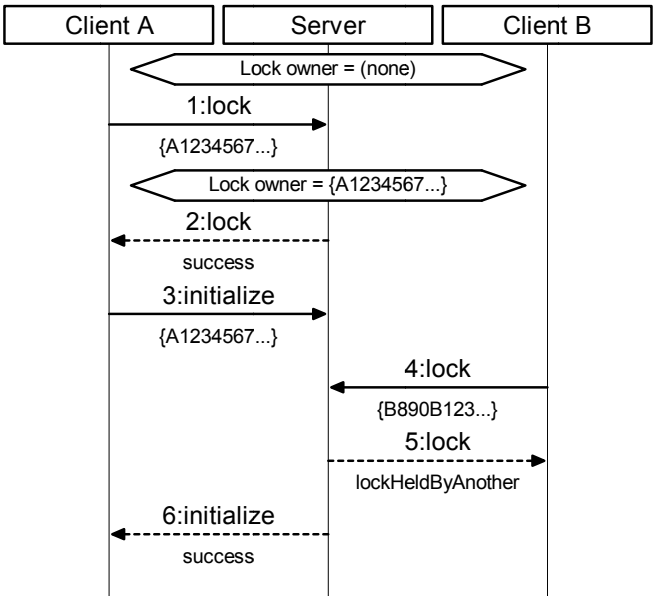


Figure 5. Example illustrating how a client cannot recieve a "sensorBusy" status if it does not hold the lock.

EXAMPLE: Suppose there are two clients; Client A and Client B. Client A holds the lock and starts initialization on (Step 1–3). Immediately after Client A initiates capture, Client B (Step 4) tries to obtain the lock while Client A is still capturing. In this situation, the valid statuses that could be returned to Client B are "sensorBusy" (since the sensor is busy performing a capture) and "lockHeldByAnother" (since Client A holds the lock). In this case, the service returns "lockHeldByAnother" (Step 5) since "lockHeldByAnother" is higher priority than "sensorBusy."

4.1.2 Parameter Failures

Services *must* distinguish among `invalidId`, `noSuchParameter`, `badValue`, and `unsupported` according to the following rules. These rules are presented here in the order of precedence that matches the previous subsection.

1. **Is a recognizable UUID provided?** If the operation requires a UUID as an input URL parameter, and provided value is not an UUID (i.e., the UUID is *not* parseable), then the service *must* return `badValue`.

Additionally, the WsbdResult's `badFields` list must contain the name of the offending parameter (`sessionId` or `captureId`).

…otherwise…

2. **Is the UUID understood?** If an operation requires an UUID as an input URL parameter, and the provided value *is* a UUID, but cannot accept the provided value, then the service *must* return `invalidId`. Additionally, the WsbdResult's `badFields` list must contain the name of the offending parameter (`sessionId` or `captureId`).

   …otherwise…

3. **Are the parameter names understood?** If an operation does not recognize a provided input parameter *name*, then the service *must* return. `invalidId`. This behavior may differ from service to service, as different services *may* recognize (or not recognize) different parameters. The unrecognized parameter(s) *must* be listed in the WsbdResult's `badFields` list.

   …otherwise…

4. **Are the parameter values acceptable?** If an operation recognizes all of the provided parameter names, but cannot accept a provided *value* because it is (a) and inappropriate type, or (b) outside the range advertised by the service (§*Forward reference needed. Are allowable ranges advertised in common info?*), the then service *must* return *badValue*. The parameter names associated with the unacceptable values *must* be listed in the WsbdResult's `badFields` list. Clients are expected to recover the bad values themselves by reconciling the WsbdResult corresponding to the offending request.

   …otherwise…

5. **Is the request supported?** If an operation accepts the parameter names and values, but the particular request is not supported by the service or the target biometric sensor, then the service *must* return *unsupported*. The parameter names that triggered this determination *must* be listed in the WsbdResult's `badFields` list. By returning multiple fields, a service is able to imply that a particular *combination* of provided values is unsupported.

NOTE: It may be helpful to think of *invalidId* as a special case of *badValue* reserved for URL parameters of type UUID.

4.1.3 Visual Summaries

The following two tables provide *informative* visual summaries of WS-BD operations. These visual summaries are an overview; they are not authoritative. (§4.3–4.16 are authoritative.)

4.1.3.1 Input & Output

The following table represents a visual summary of the inputs and outputs corresponding to each operation.

Operation *inputs* are indicated in the "URL Fragment" and "Input Payload" columns. Operation inputs take the form of either (a) a URL parameter, with the parameter name shown in "curly brackets" ("{" and "}") within the URL fragment (first column), and/or, (b) a input payload (defined in §1.1).

Operation *outputs* are provided via WsbdResult, which is contained in the body of an operation's HTTP response.

URL Fragment (Includes inputs)	Method	Operation	Input payload	Idempotent	Sensor Operation	Permitted WsbdResult Elements (within output payload)								Detailed Documentation (§)
						status	badFields	sessionId	commonInfo	configuration	captureIds	contentType	submodality	sensorData	
register	POST	register	none			●		●							4.3
register/{sessionId}	DELETE	unregister	none	◆		●	●								4.4
lock/{sessionId}	POST	try lock	none	◆		●	●								4.5
lock/{sessionId}	PUT	steal lock	none	◆		●	●								4.6
lock/{sessionId}	DELETE	unlock	none	◆		●	●								4.7
info	GET	get common info	none	◆		●			●						4.8
initialize/{sessionId}	POST	initialize	none	◆	■	●	●								4.9
configure/{sessionId}	GET	get configuration	none	◆	■	●	●			●					4.10
	POST	set configuration	config	◆	■	●	●								4.11
capture/{sessionId}	POST	capture	none		■	●	●				●				4.12
download/{captureid}	GET	download	none	◆		●	●			●		●	●	●	4.13
download/{captureid}/info	GET	get download info	none	◆						●		●	●		4.14
download/{captureid}/{maxSize}	GET	thrifty download	none	◆		●	●					●		●	4.15
cancel/{sessionId}	POST	cancel operation	none	◆	■	●	●								4.16

The `message` element is not shown in this table; when it appears it is *always* optional.

Presence of a symbol in a table cell indicates that operation is idempotent (◆), a sensor operation (■), and which elements *may* be present in the operation's WsbdResult (●). Likewise, the lack of a symbol in a table cell indicates the operation is not idempotent, not a sensor operation, and which elements of the operation's WsbdResult are forbidden.

> **EXAMPLE**: The *capture* operation (fifth row from the bottom) is not idempotent, but is a sensor operation. The output may contain the elements `status`, `badFields`, and/or `captureIds` in its WsbdResult. The detailed information regarding the WsbdResult for *capture*, (i.e., which elements are specifically permitted under what circumstances) is found in §4.12.

4.1.3.2 Permitted Status Values

The following table provides a visual summary of the status values permitted.

765

Operation Description	success	failure	invalidId	canceled	canceledWithSensorFailure	sensorFailure	lockNotHeld	lockHeldByAnother	sensorNeedsInitialization	sensorNeedsConfiguration	sensorBusy	sensorTimeout	unsupported	badValue	noSuchParameter	preparingDownload
register	●	●														
unregister	●	●	●								●			●		
try lock	●	●	●					●						●		
steal lock	●	●	●											●		
unlock	●	●	●					●						●		
get common info	●	●														
initialize	●	●	●	●	●	●	●	●			●	●		●		
get configuration	●	●	●	●	●	●	●	●	●	●	●	●		●		
set configuration	●	●	●	●	●	●	●	●	●		●	●	●	●	●	
capture	●	●	●	●	●	●	●	●	●	●	●	●		●		
download	●	●	●											●		●
get download info	●	●	●											●		●
thrifty download	●	●	●										●	●		●
cancel	●	●	●				●	●						●		

766 The presence (absence) symbol in a cell indicates that the respective status *may* (*may not*) be returned by the
767 corresponding operation.

768 **EXAMPLE**: The *register* operation may only return a WsbdResult with a WsbdStatus that contains
769 either `success` or `failure`. The *unregister* operation may only return `success`, `failure`, `invalidId`,
770 `sensorBusy`, or `badValue`.

771 4.2 Documentation Conventions

772 Each WS-BD operation is documented according to the following conventions.

773 4.2.1 General Information

774 Each operation begins with the following tabular summary:

Description	A short description of the operation
HTTP Method	The HTTP method that triggers the operation, i.e., `GET`, `POST`, `PUT`, or `DELETE`

URL Template	The suffix used to access the operation. These take the form
	`resourceName`
	or
	`resourceName/{URL_parameter_1}/…/{URL_parameter_N}`
	Each parameter, `{URL_parameter…}` *must* be replaced, in-line with the value to

	be passed in for that particular parameter.
	
	Parameters have no explicit names, other than defined by this document or reported back to the client within the contents of a `badFields` element.
	
	It is assumed that consumers of the service will prepend the URL to the service endpoint as appropriate.
	
	EXAMPLE: The resource `resourceName` hosted at the endpoint
	`http://example.com/Service`
	would be accessible via
	`http://example.com/Service/resourceName`

URL Parameters	A description of the URL-embedded operation parameters. For each parameter the following details are provided: • the name of the parameter • the expected data type (§3) • a description of the parameter
Input Payload	A description of the content, if any, to be posted to the service as input to an operation.
Idempotent	Yes—the operation is idempotent (§2.5.7). No—the operation is not idempotent.
Sensor Operation (Lock Required)	Yes—the service may require exclusive control over the target biometric sensor. No—this operation does not require a lock. Given the concurrency model (§2.5.5) this value doubles as documentation as to whether or not a lock is required

4.2.2 WsbdResult Summary

The "WsbdResult Summary" section summarizes the various forms of a WsbdResult that may be returned by the operation. Each row represents a distinct combination of permitted values & elements.

`[status value]`	status=status literal [required element name]=description of permitted contents of the element [optional element name]*=description of permitted contents of the element
⋮	⋮

For each row, the left column contains a permitted status value, and the right column contains a summary of the constraints on the WsbdResult when the `status` element takes that specific value.

Element names suffixed with a '*' indicate that the element is *optional*.

There should be an example or two here, either in-line or via a forward reference.

4.2.3 Usage Notes

For each operation, this subsection describes behaviors & requirements that are specific to the operation.

4.2.4 Unique Knowledge

For each operation, there is a brief description of whether or not the operation affords an opportunity for the server or client to exchange information unique to a particular implementation. The term "unique knowledge" is used to reflect the definition of interoperability referenced in §2.2.

4.2.5 Return Values Detail

This subsection details the various return values that the operation may return. For each permitted status value, the following table details the WsbdResult requirements:

Status Value	The particular status value
Condition	The service accepts the registration request
Required Elements	A list of the required elements. Listed for each required element is the element name and its expected contents.
Optional Elements	A list of the required elements. Listed for each optional element is the element name and its expected contents.

Constraints and information unique to the particular operation/status combination may follow the table, but some status values have no trailing explanatory text.

4.3 Register

Description	Open a new client-server session
HTTP Method	`POST`
URL Template	`/register`
URL Parameters	None
Input Payload	None
Idempotent	No
Sensor Operation	No

4.3.1 WsbdResult Summary

`success`	`status="success"` `sessionId`=session id (UUID)
`failure`	`status="failure"` `message*`=informative message describing failure

4.3.2 Usage Notes

Register provides a unique identifier that can be used to associate a particular client with a server. In a sequence of operations with a service, a *register* operation is likely one of the first operations performed by a client (*get common info* being the other). It is expected (but not required) that a client would perform a single registration during that client's lifetime.

DESIGN NOTE: By using an UUID, as opposed to the source IP address, a server can distinguish among clients sharing the same originating IP address (i.e., multiple clients on a single machine, or multiple machines behind a firewall). Additionally, a UUID allows a client (or collection of clients) to determine client identity rather than enforcing a particular model (§2.5.3).

4.3.3 Unique Knowledge

As specified, the *register* operation cannot be used to provide or obtain knowledge about unique characteristics of a client or service.

4.3.4 Return Values Detail

The *register* operation *must* return a WsbdResult according to the following constraints.

4.3.4.1 Success

Status Value	`success`
Condition	The service accepts the registration request
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`" `sessionId` (UUID, §3.1.2) an identifier that can be used to identify a session
Optional Elements	None

The "register" operation *must not* ever provide a `sessionId` of 00000000-0000-0000-0000-000000000000.

4.3.4.2 Failure

Status Value	`failure`
Condition	The service cannot accept the registration request
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

Registration might fail if there are too many sessions already registered with a service. The `message` element *must* only be used for informational purposes. Clients *must not* depend on particular contents of the message element to control client behavior.

See §B.1 for how sensor metadata can be used to determine the maximum number of current sessions a service can support.

4.4 Unregister

Description	Close a client-server session
HTTP Method	`DELETE`
URL Template	`/register/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session to remove
Input Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

4.4.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message`*=informative message describing failure
`sensorBusy`	`status="sensorBusy"`
`badValue`	`status="badValue"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)

4.4.2 Usage Notes

Unregister closes a client-server session. Although not strictly necessary, clients *should* unregister from a service when it is no longer needed. Services *should* support (on the order of) thousands of concurrent sessions, but this cannot be guaranteed, particularly if the service is running within limited computational resources. Conversely, clients *should* assume that the number of concurrent sessions that a service can support is limited. (See §B.1 for details on connection metadata.)

4.4.2.1 Inactivity

A service *may* automatically unregister a client after a period of inactivity, or if demand on the service requires that least-recently used sessions be dropped. This is manifested by a client receiving a status of `invalidId` without a corresponding unregistration. Services *should* set this value on the order of 100s of minutes. (See §B.1 for details on connection metadata.)

4.4.2.2 Sharing Session Ids

A session id is not a secret, but clients that share session ids run the risk of having their session prematurely terminated by a rouge peer client. This behavior is permitted, but discouraged. See §2.5 for more information about client identity and the assumed security models.

4.4.2.3 Locks & Pending Sensor Operations

If a client that holds the service lock unregisters, then a service must also release the service lock, with one exception. If the unregistering client both holds the lock and is responsible for a pending sensor operation, the service must return sensorBusy (See §4.4.4.3).

4.4.3 Unique Knowledge

As specified, the *unregister* operation cannot be used to provide or obtain knowledge about unique characteristics of a client or service.

4.4.4 Return Values Detail

The *unregister* operation *must* return a WsbdResult according to the following constraints.

4.4.4.1 Success

Status Value	`success`
Condition	The service accepted the unregistration request
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`"
Optional Elements	None

If the unregistering client currently holds the service lock, and the requesting client is not responsible for any pending sensor operation, then successful unregistration *must* also release the service lock.

As a consequence of idempotency, a session id does not need to ever have been registered successfully in order to *un*register successfully. Consequently, the *unregister* operation cannot return a status of `invalidId`.

4.4.4.2 Failure

Status Value	`failure`
Condition	The service could not unregister the session.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

In practice, failure to unregister is expected to be a rare occurrence. Failure to unregister might occur if the service experiences a fault with an external system (such as a centralized database used to track session registration and unregistration)

4.4.4.3 Sensor Busy

Status Value	`sensorBusy`
Condition	The service could not unregister the session because the biometric sensor is currently performing a sensor operation within the session being unregistered.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorBusy`"
Optional Elements	None

This status *must only* be returned if (a) the sensor is busy and (b) the client making the request holds the lock (i.e., the session id provided matches that associated with the current service lock). Any client that does not hold the session lock *must not* result in a `sensorBusy` status.

> **EXAMPLE**: The following sequence diagram illustrates a client that cannot unregister (Client A) and a client that can unregister (Client B). After the initialize operation completes (Step 6), Client A can unregister (Steps 7-8).

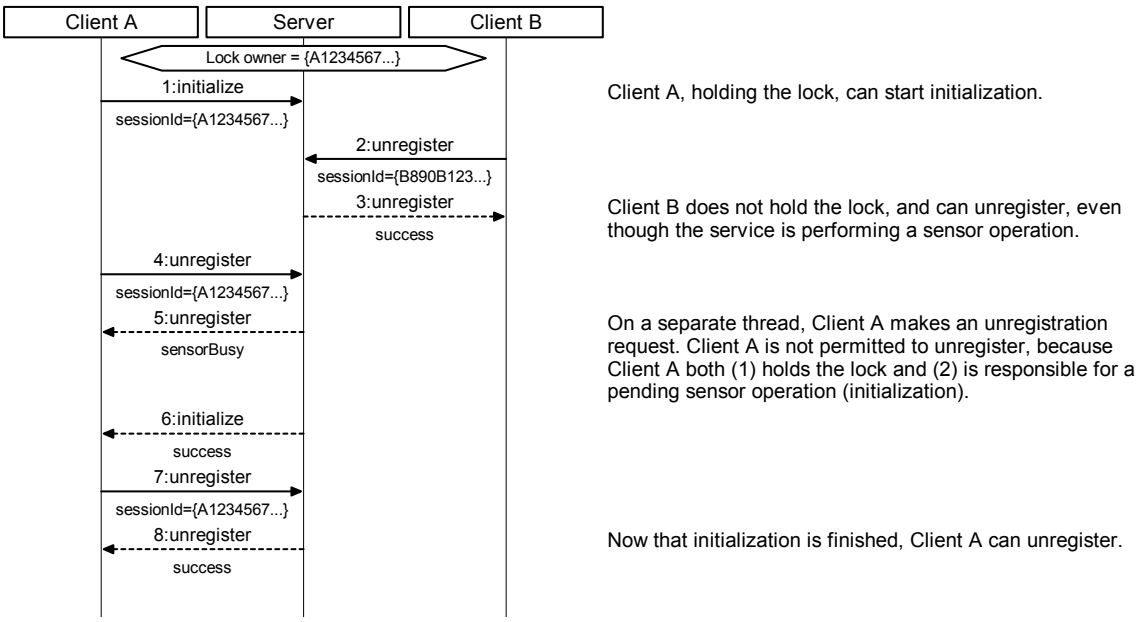


Figure 6. Example of how an *unregister* operation can result in `sensorBusy`.

4.4.4.4 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.5 Try Lock

Description	Try to obtain the service lock
HTTP Method	PUT
URL Template	`/lock/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session requesting the service lock
Input Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

4.5.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message`*=informative message describing failure
`invalidId`	`status="invalidId"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)
`lockHeldByAnother`	`status="lockHeldByAnother"`
`badValue`	`status="badValue"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)

4.5.2 Usage Notes

The *try lock* operation attempts to obtain the service lock. The word "try" is used to indicate that the call returns immediately, and does not block until the lock is obtained. See §2.5.5 for detailed information about the WS-BD concurrency and locking model.

4.5.3 Unique Knowledge

As specified, the *try lock* cannot be used to provide or obtain knowledge about unique characteristics of a client or service.

4.5.4 Return Values Detail

The *try lock* operation *must* return a WsbdResult according to the following constraints.

4.5.4.1 Success

Status Value	`success`
Condition	The service was successfully locked to the provided session id.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`"
Optional Elements	None

Clients that hold the service lock are permitted to perform sensor operations (§2.5.5). By idempotency (§2.5.7), if a client already holds the lock, subsequent *try lock* operations should also return `success`.

4.5.4.2 Failure

Status Value	`failure`
Condition	The service could not be locked to the provided session id.
Required Elements	`status` (WsbdStatus, §3.1.6)

	the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

Services must reserve a `failure` status to report system or internal failures and prevent the acquisition of the lock. Most *try lock* operations that do not succeed will not produce a `failure` status, but more likely a `lockHeldByAnother` status (See §4.5.4.4 for an example).

4.5.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or triggered automatically by the service due to inactivity (§4.4.4.1).

See §4.1.2 for general information on how services must handle parameter failures.

4.5.4.4 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	The service could not be locked to the provided session id because the lock is held by another client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockHeldByAnother`"
Optional Elements	None

EXAMPLE: The following sequence diagram illustrates a client that cannot obtain the lock (Client B) because it is held by another client (Client A).

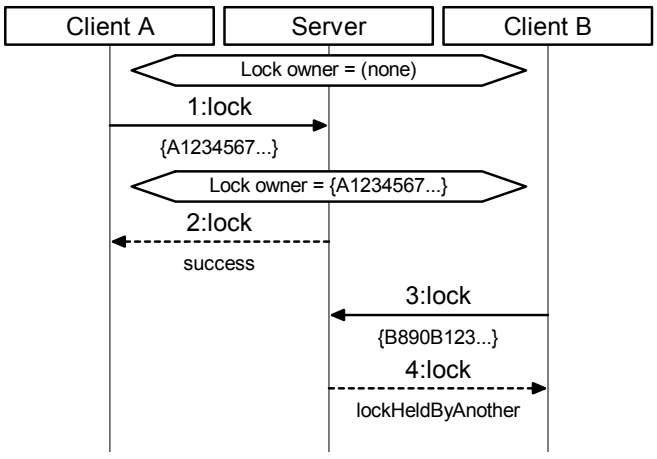


Figure 7. Example of a scenario yielding a `lockHeldByAnother` result.

4.5.4.5 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.6 Steal Lock

Description	Forcibly obtain the lock away from a peer client
HTTP Method	PUT
URL Template	`/lock/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session requesting the service lock
Input Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

4.6.1 WsbdResult Summary

success	`status="success"`
failure	`status="failure"` `message`*=informative message describing failure
invalidId	`status="invalidId"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)
badValue	`status="badValue"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)

4.6.2 Usage Notes

The *steal lock* operation allows a client to forcibly obtain the lock away from another client that already holds the lock. The purpose of this operation is to prevent a client that experiences a fatal error from forever preventing another client access to the service, and therefore, the biometric sensor.

4.6.2.1 Avoid Lock Stealing

Developers and integrators *should* endeavor to reserve lock stealing for exceptional circumstances—such as when a fatal error prevents a client from releasing a lock. Lock stealing *should* not be used as the primary mechanism in which peer clients coordinate biometric sensor use.

4.6.2.2 Lock Stealing Prevention Period (LSPP)

To assist in coordinating access among clients and to prevent excessive lock stealing, a service *may* trigger a time period that forbids lock stealing for each sensor operation. For convenience, this period of time will be referred to as the *lock stealing prevention period (LSPP).*

During the LSPP, all attempts to steal the service lock will fail. Consequently, if a client experiences a fatal failure during a sensor operation, then all peer clients need to wait until the service re-enables lock stealing.

To prevent excessive lock stealing, all services *should* implement a non-zero LSPP. The recommended time for the LSPP is on the order of 100 seconds. Services that enforce an LSPP *must* start the LSPP immediately before sovereign sensor control is required. Conversely, services *should not* enforce an LSPP unless absolutely necessary.

Examples of a request that may proceed without sovereign sensor control are those that return a status of `invalidId`, or `badValue`. In these cases, there is no functional need to prevent lock stealing since the request does not directly involve the biometric sensor. *More specifics and examples might be necessary here. There may be a need to identify that different implementations may or may not trigger the LSPP on borderline status*

results, such as 'failure', and that each sensor operation itself might be a new decision point on whether or not to trigger a LSPP

An LSPP ends after a fixed amount of time has elapsed, unless another sensor operation restarts the LSPP. Services should keep the length of the LSPP fixed throughout the service's lifecycle. It is recognized, however, that there may be use cases in which a variable LSPP timespan is desirable or required . Regardless, when determining the appropriate timespan, implementers *should* carefully consider the tradeoffs between preventing excessive lock stealing, versus forcing all clients to wait until a service re-enables lock stealing.

4.6.2.3 Cancelation & (Lack of) Client Notification

Lock stealing *must* have no effect on any currently running sensor operations. It is possible that a client initiates a sensor operation, has its lock stolen away, yet the operation completes successfully. *Subsequent* sensor operations would yield a `lockNotHeld` status, which a client could use to indicate that their lock was stolen away from them. Services *should* be implemented such that the LSPP is longer than any sensor operation.

4.6.3 Unique Knowledge

As specified, the *steal lock* operation cannot be used to provide or obtain knowledge about unique characteristics of a client or service.

4.6.4 Return Values Detail

The *steal lock* operation *must* return a WsbdResult according to the following constraints.

4.6.4.1 Success

Status Value	`success`
Condition	The service was successfully locked to the provided session id.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`"
Optional Elements	None

See §2.5.5 for detailed information about the WS-BD concurrency and locking model. Cancelation must have no effect on pending sensor operations (§4.6.2.3).

4.6.4.2 Failure

Status Value	`failure`
Condition	The service could not be locked to the provided session id.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

Most *steal lock* operations that yield a `failure` status will do so because the service receives a lock stealing request during a lock stealing prevention period (§4.6.2.2). Services *must* also reserve a `failure` status for other non-LSPP failures that prevent the acquisition of the lock.

Implementers *may* choose to use the optional `message` field to provide more information to an end-user as to the specific reasons for the failure. However (as with all other `failure` status results), clients *must* not depend on any particular content to make this distinction.

4.6.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or triggered automatically by the service due to inactivity (§4.4.4.1).

See §4.1.2 for general information on how services must handle parameter failures.

4.6.4.4 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.7 Unlock

Description	Release the service lock
HTTP Method	`DELETE`
URL Template	`/lock/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session releasing the service lock
Input Payload	None
Idempotent	Yes
Sensor Operation	No

4.7.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message`*=informative message describing failure
`invalidId`	`status="invalidId"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)
`badValue`	`status="badValue"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)

4.7.2 Usage Notes

The *unlock* operation release a service lock, making locking available to other clients.

See §2.5.5 for detailed information about the WS-BD concurrency and locking model.

4.7.3 Unique Knowledge

As specified, the *unlock* operation cannot be used to provide or obtain knowledge about unique characteristics of a client or service.

4.7.4 Return Values Detail

The *steal lock* operation *must* return a WsbdResult according to the following constraints.

4.7.4.1 Success

Status Value	`success`
Condition	The service returned to an unlocked state.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`"
Optional Elements	None

Upon releasing the lock, a client is no longer permitted to perform any sensor operations (§2.5.5). By idempotency (§2.5.7), if a client already has released the lock, subsequent *unlock* operations should also return `success`.

4.7.4.2 Failure

Status Value	`failure`
Condition	The service could not be transitioned into an unlocked state.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"

Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

Services must reserve a `failure` status to report system or internal failures and prevent the release of the service lock. The occurrence of *unlock* operations that fail is expected to be rare.

4.7.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or triggered automatically by the service due to inactivity (§4.4.4.1).

See §4.1.2 for general information on how services must handle parameter failures.

4.7.4.4 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.8 Get Common Info

Description	Retrieve metadata about the service that does not depend on session-specific information, or sovereign control of the target biometric sensor
HTTP Method	`GET`
URL Template	`/info`
URL Parameters	None
Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

4.8.1 WsbdResult Summary

`success`	`status="success"` `commonInfo`=dictionary containing service metadata
`failure`	`status="failure"` `message*`=informative message describing failure

4.8.2 Usage Notes

The *get common info* operation provides information about the service and target biometric sensor. This operation *must* return information that is both (a) independent of session, and (b) does not require sovereign biometric sensor control. In other words, services *must not* control the target biometric sensor during a *get common info* operation itself. Implementers *may* (and are encouraged to) use service startup time to query the biometric sensor directly to create a cache of information and capabilities. The contents of this cache are a basis for *get common info* operations.

The *get common info* operation does *not* require that a client be registered with the service. Unlike other operations, it does *not* take a session id as a URL parameter.

There should be a description or forward reference of where/how a service advertises its (a) available parameters and/or (b) default configuration.

4.8.3 Unique Knowledge

As specified, the *get common info* can be used to obtain knowledge about unique characteristics of a service. Through *get common info*, a service may expose implementation and/or service-specific configuration parameter names and values that are not described in a conformance profile [SP500-288A].

4.8.4 Return Values Detail

The *steal lock* operation *must* return a WsbdResult according to the following constraints.

4.8.4.1 Success

Status Value	`success`
Condition	The service provides service metadata
Required Elements	`status` (WsbdStatus, §3.1.6) the literal `"success"` `commonInfo` (WsbdDictionary, §3.1.3) information about the service metadata

Optional Elements	None

See Appendix A for additional information on common info.

4.8.4.2 Failure

Status Value	`failure`
Condition	The service cannot provide service metadata
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

4.9 Initialize

Description	Initialize the target biometric sensor
HTTP Method	`POST`
URL Template	`/initialize/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session requesting initialization
Input Payload	None
Idempotent	Yes
Sensor Operation	Yes

4.9.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*`=informative message describing failure
`invalidId`	`status="invalidId"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)
`canceled`	`status="canceled"`
`canceledWithSensorFailure`	`status="canceledWithSensorFailure"`
`sensorFailure`	`status="sensorFailure"`
`lockNotHeld`	`status="lockNotHeld"`
`lockHeldByAnother`	`status="lockHeldByAnother"`
`sensorBusy`	`status="sensorBusy"`
`sensorTimeout`	`status="sensorTimeout"`
`badValue`	`status="badValue"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)

4.9.2 Usage Notes

The *initialize* operation prepares the target biometric sensor for (other) sensor operations.

Some biometric sensors have no requirement for explicit initialization. Services exposing such a sensor *should* immediately return a `success` result.

Although not strictly necessary, services *should* directly map this operation to the initialization of the target biometric sensor, unless the service can reliably determine that the target biometric sensor is in a fully operational state. In other words, a service may decide to immediately return `success` if there is a reliable way to detect if the target biometric sensor is currently in an initialized state. This style of "short circuit" evaluation could reduce initialization times. However, a service that always initializes the target biometric sensor would enable the ability of a client to attempt a manual reset of a sensor that has entered a faulty state. This is particularly useful in physically separated service implementations where the connection between the target biometric sensor and the web service host may be less reliable than an integrated implementation.

4.9.3 Unique Knowledge

As specified, the *initialize* operation cannot be used to provide or obtain knowledge about unique characteristics of a client or service.

4.9.4 Return Values Detail

4.9.4.1 Success

Status Value	`success`
Condition	The service successfully initialized the target biometric sensor
Required Elements	`status` *must* be populated with the WsbdStatus literal `"success"`
Optional Elements	None

4.9.4.2 Failure

Status Value	`failure`
Condition	The service experienced a fault that prevented successful initialization.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

A `failure` status *must* only be used to report failures that occurred within the web service, not within the target biometric sensor (§4.9.4.5, §4.9.4.6)

4.9.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or triggered automatically by the service due to inactivity (§4.4.4.1).

See §4.1.2 for general information on how services must handle parameter failures.

4.9.4.4 Canceled

Status Value	`canceled`
Condition	The initialization operation was interrupted by a cancelation request.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`canceled`"
Optional Elements	None

See §4.16.2.2 for information about what may trigger a cancelation.

4.9.4.5 Canceled with Sensor Failure

Status Value	`canceledWithSensorFailure`
Condition	The initialization operation was interrupted by a cancelation request and the target biometric sensor experienced a failure
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`canceledWithSensorFailure`"
Optional Elements	`message` (xs:string, [XSDPart2])

	an informative description of the nature of the failure

Services *must* return a `canceledWithSensorFailure` result if a cancelation request caused a failure within the target biometric sensor. Clients receiving this result may need to reattempt the initialization request to restore full functionality. See §4.16.2.2 for information about what may trigger a cancelation.

4.9.4.6 Sensor Failure

Status Value	`sensorFailure`
Condition	The initialization failed due to a failure within the target biometric sensor
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorFailure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

A `sensorFailure` status *must* only be used to report failures that occurred within the target biometric sensor, not a failure within the web service (§4.9.4.2).

4.9.4.7 Lock Not Held

Status Value	`lockNotHeld`
Condition	Initialization could not be performed because the requesting client does not hold the lock
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockNotHeld`"
Optional Elements	None

Sensor operations require that the requesting client holds the service lock.

4.9.4.8 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	Initialization could not be performed because the lock is held by another client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockHeldByAnother`"
Optional Elements	None

4.9.4.9 Sensor Busy

Status Value	`sensorBusy`
Condition	Initialization could not be performed because the service is already performing a different sensor operation for the requesting client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorBusy`"
Optional Elements	None

4.9.4.10 Sensor Timeout

Status Value	`sensorTimeout`
Condition	Initialization could not be performed because the target biometric sensor took too long to complete the initialization request.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorTimeout`"
Optional Elements	None

A service did not receive a timely response from the target biometric sensor. Note that this condition is distinct from the client's originating HTTP request, which may have its own, independent timeout. (See B.2 for information on how a client might determine timeouts.)

4.9.4.11 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

1046 # 4.10 Get Configuration

Description	Get the target biometric sensor's current configuration
HTTP Method	`GET`
URL Template	`/configure/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session requesting the configuration
Payload	Not applicable
Idempotent	Yes
Sensor Operation	Yes

1047 ## 4.10.1 WsbdResult Summary

success	`status="success"` `configuration`=current configuration of the sensor (WsbdResult, §3.2)
failure	`status="failure"` `message*`=informative message describing failure
invalidId	`status="invalidId"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)
canceled	`status="canceled"`
canceledWithSensorFailure	`status="canceledWithSensorFailure"`
sensorFailure	`status="sensorFailure"`
lockNotHeld	`status="lockNotHeld"`
lockHeldByAnother	`status="canceled"`
sensorNeedsInitialization	`status="sensorNeedsInitialization"`
sensorNeedsConfiguration	`status="sensorNeedsConfiguration"`
sensorBusy	`status="sensorBusy"`
sensorTimeout	`status="sensorTimeout"`
badValue	`status="badValue"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)

1048 ## 4.10.2 Usage Notes

1049 The *get configuration* operation retrieves the service's current configuration.

1050 ## 4.10.3 Unique Knowledge

1051 As specified, the *get configuration* can be used to obtain knowledge about unique characteristics of a service.
1052 Through *get configuration*, a service may expose implementation and/or service-specific configuration
1053 parameter names and values that are not described in a conformance profile [SP500-288A].

1054 ## 4.10.4 Return Values Detail

1055 The *get configuration* operation *must* return a WsbdResult according to the following constraints.

1056 ### 4.10.4.1 Success

Status Value	`success`
Condition	The service provides the current configuration
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`" `configuration` (WsbdDictionary, §3.1.3)

	the target biometric sensor's current configuration
Optional Elements	None

There needs to be a reference with respect to what constitutes a well-known configuration

4.10.4.2 Failure

Status Value	`failure`
Condition	The service cannot provide the current configuration due to service (not target biometric sensor) error.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

Services *must* only use this status to report failures that occur within the web service, not the target biometric sensor (see §4.10.4.5, §4.10.4.6).

4.10.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or triggered automatically by the service due to inactivity (§4.4.4.1).

See §4.1.2 for general information on how services must handle parameter failures.

4.10.4.4 Canceled

Status Value	`canceled`
Condition	The *get configuration* operation was interrupted by a cancelation request.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`canceled`"
Optional Elements	None

See §4.16.2.2 for information about what may trigger a cancelation.

4.10.4.5 Canceled with Sensor Failure

Status Value	`canceledWithSensorFailure`
Condition	The *get configuration* operation was interrupted by a cancelation request during which the target biometric sensor experienced a failure
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`canceledWithSensorFailure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

Services *must* return a `canceledWithSensorFailure` result if a cancelation request caused a failure within the target biometric sensor. Clients receiving this result may need to perform initialization to restore full functionality. See §4.16.2.2 for information about what may trigger a cancelation.

4.10.4.6 Sensor Failure

Status Value	`sensorFailure`
Condition	The configuration could not be queried due to a failure within the target biometric sensor.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorFailure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

A `sensorFailure` status *must* only be used to report failures that occurred within the target biometric sensor, not a failure within the web service (§4.9.4.2).

4.10.4.7 Lock Not Held

Status Value	`lockNotHeld`
Condition	The configuration could not be queried because the requesting client does not hold the lock.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockNotHeld`"
Optional Elements	None

Sensor operations require that the requesting client holds the service lock.

4.10.4.8 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	The configuration could not be queried because the lock is held by another client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockHeldByAnother`"
Optional Elements	None

4.10.4.9 Sensor Needs Initialization

Status Value	`sensorNeedsInitialization`
Condition	The configuration could not be queried because the target biometric sensor has not been initialized.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorNeedsInitialization`"
Optional Elements	None

Services *should* be able to provide the sensors configuration without initialization; however, this is not strictly necessary. Regardless, robust clients *should* assume that configuration will require initialization.

4.10.4.10 Sensor Needs Configuration

Status Value	`sensorNeedsConfiguration`
Condition	The configuration could not be queried because the target biometric sensor has not been initialized.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorNeedsConfiguration`"
Optional Elements	None

Services *may* require configuration to be set before a configuration can be retrieved if a service does not provide a valid default configuration.

4.10.4.11 Sensor Busy

Status Value	`sensorBusy`
Condition	The configuration could not be queried because the service is already performing a different sensor operation for the requesting client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorBusy`"
Optional Elements	None

4.10.4.12 Sensor Timeout

Status Value	`sensorTimeout`
Condition	The configuration could not be queried because the target biometric sensor took too long to complete the request.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorTimeout`"
Optional Elements	None

A service did not receive a timely response from the target biometric sensor. Note that this condition is distinct from the client's originating HTTP request, which may have its own, independent timeout. (See B.2 for information on how a client might determine timeouts.)

4.10.4.13 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.11 Set Configuration

Description	Set the target biometric sensor's configuration
HTTP Method	POST
URL Template	`/configure/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session requesting the configuration
Payload	Desired sensor configuration (WsbdDictionary, §3.1.3)
Idempotent	Yes
Sensor Operation	Yes

4.11.1 WsbdResult Summary

success	`status="success"`
failure	`status="failure"` `message*`=informative message describing failure
invalidId	`status="invalidId"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)
canceled	`status="canceled"`
canceledWithSensorFailure	`status="canceledWithSensorFailure"`
sensorFailure	`status="sensorFailure"`
lockNotHeld	`status="lockNotHeld"`
lockHeldByAnother	`status="canceled"`
sensorNeedsInitialization	`status="sensorNeedsInitialization"`
sensorBusy	`status="sensorBusy"`
sensorTimeout	`status="sensorTimeout"`
unsupported	`status="unsupported"` `badFields={`field names`}` (WsbdStringArray, §3.1.4)
badValue	`status="badValue"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4) *(or)* `status="badValue"` `badFields={`field names`}` (WsbdStringArray, §3.1.4)
noSuchParameter	`status="unsupported"` `badFields={`field names`}` (WsbdStringArray, §3.1.4)

4.11.2 Usage Notes

The *set configuration* operation sets the configuration of a service's target biometric sensor.

4.11.2.1 Input Payload Information

The *set configuration* operation is the only operation that takes input within the body of the HTTP request. The desired configuration *must* be sent as a single, unnamed WsbdDictionary.

EXAMPLE: The following represents a 'raw' request to configure a service at `http://10.0.0.2:7000/Sensor` such that `string1=value` and `integer1=1`. (In this example, each `value` element contains fully qualified namespace information, although this is not necessary.)

```
POST http://10.0.0.2:7000/Sensor/configure/1678e0fa-b578-4234-bb59-6f2f92d7b80c HTTP/1.1
Content-Type: application/xml
Host: 10.0.0.2:7000
```

```
Content-Length: 351
Expect: 100-continue

<WsbdDictionary xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://itl.nist.gov/w
sbd/L1"><item><key>string1</key><value xmlns:d3p1="http://www.w3.org/2001/XMLSchema" i:type="d3p
1:string">value</value></item><item><key>integer1</key><value xmlns:d3p1="http://www.w3.org/2001
/XMLSchema" i:type="d3p1:int">1</value></item></WsbdDictionary>
```

More information regarding the use of the `xmlns` attribute can be found in [XMLNS].

This example should be updated to use the correct namespace

4.11.3 Unique Knowledge

The *set configuration* can be used to provide knowledge about unique characteristics to a service. Through *set configuration*, a client *may* provide implementation and/or service-specific parameter names and values that are not described in a conformance profile [SP500-288A].

4.11.4 Return Values Detail

The *set configuration* operation *must* return a WsbdResult according to the following constraints.

4.11.4.1 Success

Status Value	`success`
Condition	The service was able to successfully set the full configuration
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`"
Optional Elements	None

References needed here

4.11.4.2 Failure

Status Value	`failure`
Condition	The service cannot set the desired configuration due to service (not target biometric sensor) error.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` An informative description of the nature of the failure.

Services *must* only use this status to report failures that occur within the web service, not the target biometric sensor (see §4.11.4.5, §4.11.4.6).

4.11.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or triggered automatically by the service due to inactivity (§4.4.4.1).

4.11.4.4 Canceled

Status Value	`canceled`
Condition	The *set configuration* operation was interrupted by a cancelation request.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`canceled`"
Optional Elements	None

See §4.16.2.2 for information about what may trigger a cancelation.

4.11.4.5 Canceled with Sensor Failure

Status Value	`canceledWithSensorFailure`
Condition	The *set configuration* operation was interrupted by a cancelation request during which the target biometric sensor experienced a failure
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`canceledWithSensorFailure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

Services *must* return a `canceledWithSensorFailure` result if a cancelation request caused a failure within the target biometric sensor. Clients receiving this result may need to perform initialization to restore full functionality. See §4.16.2.2 for information about what may trigger a cancelation.

4.11.4.6 Sensor Failure

Status Value	`sensorFailure`
Condition	The configuration could not be set due to a failure within the target biometric sensor.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorFailure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

A `sensorFailure` status *must* only be used to report failures that occurred within the target biometric sensor, not a failure within the web service (§4.11.4.2). Errors with the configuration itself should be reported via an `unsupported` (§4.11.4.12), `badValue` (§4.11.4.13), or `badValue` status (§4.11.4.14).

4.11.4.7 Lock Not Held

Status Value	`lockNotHeld`
Condition	The configuration could not be queried because the requesting client does not hold the lock.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockNotHeld`"
Optional Elements	None

Sensor operations require that the requesting client holds the service lock.

4.11.4.8 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	The configuration could not be set because the lock is held by another client.

Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockHeldByAnother`"
Optional Elements	None

4.11.4.9 Sensor Needs Initialization

Status Value	`sensorNeedsInitialization`
Condition	The configuration could not be set because the target biometric sensor has not been initialized.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorNeedsInitialization`"
Optional Elements	None

Services *should* be able to set the configuration without initialization; however, this is not strictly necessary. Similarly, clients *should* assume that setting configuration will require initialization.

4.11.4.10 Sensor Busy

Status Value	`sensorBusy`
Condition	The configuration could not be set because the service is already performing a different sensor operation for the requesting client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorBusy`"
Optional Elements	None

4.11.4.11 Sensor Timeout

Status Value	`sensorTimeout`
Condition	The configuration could not be set because the target biometric sensor took too long to complete the request.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorTimeout`"
Optional Elements	None

A service did not receive a timely response from the target biometric sensor. Note that this condition is distinct from the client's originating HTTP request, which may have its own, independent timeout. (See B.2 for information on how a client might determine timeouts.)

4.11.4.12 Unsupported

Status Value	`unsupported`
Condition	The requested configuration contains one or more values that is are syntactically and semantically valid, but not supported by the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`unsupported`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the field name(s) that corresponding to the unsupported value(s)
Optional Elements	None

Returning *multiple* fields allows a service to indicate that a particular *combination* of parameters is not supported by a service. See §4.1.2 for additional information on how services must handle parameter failures.

An example regarding a webcam that only supports particular combinations of resolution & frame rates might be helpful here.

4.11.4.13 Bad Value

Status Value	`badValue`
Condition	Either: (a) The provided session id is not a well-formed UUID, or, (b) The requested configuration contains a parameter value that is either syntactically (e.g., an inappropriate data type) or semantically invalid (e.g., a value outside of an acceptable range).
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains either (a) the single field name, "`sessionId`", or (b) the field name(s) that contain invalid value(s)
Optional Elements	None

Notice that for the *set configuration* operation, an invalid URL parameter *or* one or more invalid input payload parameters can trigger a `badValue` status.

See §4.1.2 for general information on how services must handle parameter failures.

4.11.4.14 No Such Parameter

Status Value	`noSuchParameter`
Condition	The requested configuration contains a parameter name that is not recognized by the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`noSuchParameter`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the field name(s) that are not recognized by the service
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.12 Capture

Description	Capture biometric data
HTTP Method	`POST`
URL Template	`/capture/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session requesting the configuration
Input Payload	None
Idempotent	No
Sensor Operation	Yes

4.12.1 WsbdResult Summary

success	`status="success"` `captureIds`={identifiers of captured data} (WsbdUuidArray, §3.1.5)
failure	`status="failure"` `message*`=informative message describing failure
invalidId	`status="invalidId"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)
canceled	`status="canceled"`
canceledWithSensorFailure	`status="canceledWithSensorFailure"`
sensorFailure	`status="sensorFailure"`
lockNotHeld	`status="lockNotHeld"`
lockHeldByAnother	`status="lockHeldByAnother"`
sensorNeedsInitialization	`status="sensorNeedsInitialization"`
sensorNeedsConfiguration	`status="sensorNeedsConfiguration"`
sensorBusy	`status="sensorBusy"`
sensorTimeout	`status="sensorTimeout"`
badValue	`status="badValue"` `badFields={"sessionId"}` (WsbdStringArray, §3.1.4)

4.12.2 Usage Notes

The *capture* operation triggers biometric acquisition. On success, the operation returns one or more identifiers, or *capture id*s. Naturally, the *capture* operation is *not* idempotent. Each *capture* operation returns unique identifiers—each execution returning references that are particular to that capture. Clients then can retrieve the captured data itself by passing a *capture id* as a URL parameter to the *download* operation.

Multiple *capture ids* are supported to accommodate sensors that return collections of biometric data. For example, a multi-sensor array might save an image per sensor. A mixed-modality sensor might assign a different capture id for each modality.

IMPORTANT NOTE: The *capture* operation *may* include some post-acquisition processing. Although post-acquisition processing is directly tied to the *capture* operation, its effects are primarily on data transfer, and is therefore discussed in detail within the *download* operation documentation (§4.13.2.2)

4.12.2.1 Providing Timing Information

Depending on the sensor, a *capture* operation may take anywhere from milliseconds to tens of seconds to execute. (It is possible to have even longer running capture operations than this, but special accommodations may need to be made on the server and client side to compensate for typical HTTP timeouts.) By design, there

is no explicit mechanism for a client to determine how long a capture operation will take. However, services can provide "hints" to through capture timeout information (B.2.5), and clients can automatically adjust their own timeouts and behavior accordingly.

4.12.3 Unique Knowledge

As specified, the *capture* operation cannot be used to provide or obtain knowledge about unique characteristics of a client or service.

4.12.4 Return Values Detail

The *capture* operation *must* return a WsbdResult according to the following constraints.

4.12.4.1 Success

Status Value	`success`
Condition	The service successfully performed a biometric acquisition
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`" `captureIds` (WsbdUuidArray, §3.1.5) one more UUIDs that uniquely identify the data acquired by the operation
Optional Elements	None

See the usage notes for *capture* (§4.12.2) and *download* (§4.13.2) for full detail.

4.12.4.2 Failure

Status Value	`failure`
Condition	The service cannot perform the capture due to a service (not target biometric sensor) error.
Required Elements	`status` *must* be populated with the WsbdStatus literal `"failure"`
Optional Elements	`message` An informative description of the nature of the failure.

Services *must* only use this status to report failures that occur within the web service, not the target biometric sensor (see §4.12.4.5, §4.12.4.6). A service may fail at capture if there is not enough internal storage available to accommodate the captured data (§B.3).

4.12.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not registered with the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or triggered automatically by the service due to inactivity (§4.4.4.1).

See §4.1.2 for general information on how services must handle parameter failures.

4.12.4.4 Canceled

Status Value	`canceled`
Condition	The *capture* operation was interrupted by a cancelation request.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`canceled`"
Optional Elements	None

See §4.16.2.2 for information about what may trigger a cancelation.

4.12.4.5 Canceled with Sensor Failure

Status Value	`canceledWithSensorFailure`
Condition	The *capture* operation was interrupted by a cancelation request during which the target biometric sensor experienced a failure
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`canceledWithSensorFailure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

Services *must* return a `canceledWithSensorFailure` result if a cancelation request caused a failure within the target biometric sensor. Clients receiving this result *may* need to perform initialization to restore full functionality. See §4.16.2.2 for information about what may trigger a cancelation.

4.12.4.6 Sensor Failure

Status Value	`sensorFailure`
Condition	The service could perform the capture due to a failure within the target biometric sensor.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorFailure`"
Optional Elements	`message` (xs:string, [XSDPart2]) an informative description of the nature of the failure

A `sensorFailure` status *must* only be used to report failures that occurred within the target biometric sensor, not a failure within the web service (§4.12.4.2).

4.12.4.7 Lock Not Held

Status Value	`lockNotHeld`
Condition	The service could not perform a capture because the requesting client does not hold the lock.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockNotHeld`"
Optional Elements	None

Sensor operations require that the requesting client holds the service lock.

4.12.4.8 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	The service could not perform a capture because the lock is held by another client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockHeldByAnother`"
Optional Elements	None

4.12.4.9 Sensor Needs Initialization

Status Value	`sensorNeedsInitialization`
Condition	The service could not perform a capture because the target biometric sensor has not been initialized.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorNeedsInitialization`"
Optional Elements	None

Services *should* be able perform capture without explicit initialization. However, the specification recognizes that this is not always possible, particularly for physically separated implementations. Regardless, for robustness, clients *should* assume that setting configuration will require initialization.

4.12.4.10 Sensor Needs Configuration

Status Value	`sensorNeedsConfiguration`
Condition	The capture could not be set because the target biometric sensor has not been configured.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorNeedsConfiguration`"
Optional Elements	None

A service *should* offer a default configuration to allow capture to be performed without an explicit configuration. Regardless, for robustness, clients *should* assume that capture requires configuration.

4.12.4.11 Sensor Busy

Status Value	`sensorBusy`
Condition	The service could not perform a capture because the service is already performing a different sensor operation for the requesting client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorBusy`"
Optional Elements	None

4.12.4.12 Sensor Timeout

Status Value	`sensorTimeout`
Condition	The service could not perform a capture because the target biometric sensor took too long to complete the request.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`sensorTimeout`"
Optional Elements	None

A service did not receive a timely response from the target biometric sensor. Note that this condition is distinct from the client's originating HTTP request, which may have its own, independent timeout. (See §B.2 for information on how a client might determine timeouts.)

4.12.4.13 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

1222 See §4.1.2 for general information on how services must handle parameter failures.

4.13 Download

Description	Download the captured biometric data
HTTP Method	`GET`
URL Template	`/download/{captureId}`
URL Parameters	`{captureId}` (UUID, §3.1.2) Identity of the captured data to download
Input Payload	None
Idempotent	Yes
Sensor Operation	No

4.13.1 WsbdResult Summary

`success`	`status="success"` `configuration`=sensor configuration at the time of capture `contentType`= content type [RFC2045, RFC2046] describing the captured data `submodality`= the submodality of the captured data (xs:string) `sensorData`=biometric data (xs:base64Binary)
`failure`	`status="failure"` `message*`=informative message describing failure
`invalidId`	`status="invalidId"` `badFields={"captureId"}` (WsbdStringArray, §3.1.4)
`badValue`	`status="badValue"` `badFields={"captureId"}` (WsbdStringArray, §3.1.4)
`preparingDownload`	`status="preparingDownload"`

4.13.2 Usage Notes

The *download* operation allows a client to retrieve biometric data acquired during a particular capture.

4.13.2.1 Capture and Download as Separate Operations

WS-BD decouples the acquisition operation (capture) from the data transfer (*download*) operation. This has two key benefits. First, it is a better fit for services that have post-acquisition processes. Second, it allows multiple clients to download the captured biometric data by exploiting the concurrent nature of HTTP. By making *download* a simple data transfer operation, service can handle multiple, concurrent downloads without requiring locking (§2.5.5).

4.13.2.2 Services with Post-Acquisition Processing

A service does *not* need to make the captured data available immediately after capture; a service may have distinct acquisition and post-acquisition processes. The following are two examples of such services:

> **EXAMPLE**: A service exposing a fingerprint scanner also performs post processing on a fingerprint image—segmentation, quality assessment, and templatization.
>
> **EXAMPLE**: A service exposes a digital camera in which the captured image is not immediately available after a photo is taken; the image may need to be downloaded from to the camera's internal storage or from the camera to the host computer (in a physically separated implementation). If the digital camera was unavailable for an operation due to a data transfer, a client requesting a sensor operation would receive a `sensorBusy` status.

The first method is to perform the post-processing within the *capture* operation itself. I.e., *capture* not only blocks for the acquisition to be performed, but also blocks for the post-processing—returning when the post-processing is complete. This type of capture is the easier of the two to both (a) implement on the client, and (b) use by a client.

EXAMPLE: Figure 8 illustrates an example of a *capture* operation that includes post-processing. Once the post-processing is complete, capture ids are returned to the client.

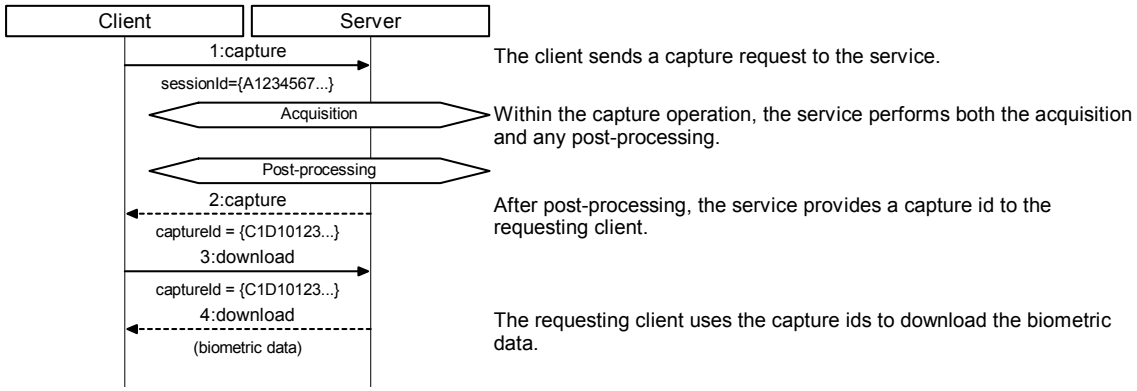


Figure 8. Including post-processing in the capture operation means downloads are immediately available when capture completes.

In the second method, post-processing may be performed by the web service *after* the capture operation returns. Capture ids are still returned to the client, but are in an intermediate state. This exposes a window of time in which the capture is complete, but the biometric data is not yet ready for retrieval or download. Data-related operations (*download*, *get download info*, and *thrifty download*) performed within this window return a `preparingDownload` status to clients to indicate that the captured data is currently in an intermediate state—captured, but not yet ready for retrieval.

EXAMPLE: Figure 9 illustrates an example of a *capture* operation with separate post-processing. Returning to the example of the fingerprint scanner that transforms a raw biometric sample into a template after acquisition, assume that the service performs templitiazation after capture returns. During post-processing, requests for the captured data return `preparingDownload`, but the sensor itself is available for additional operations.

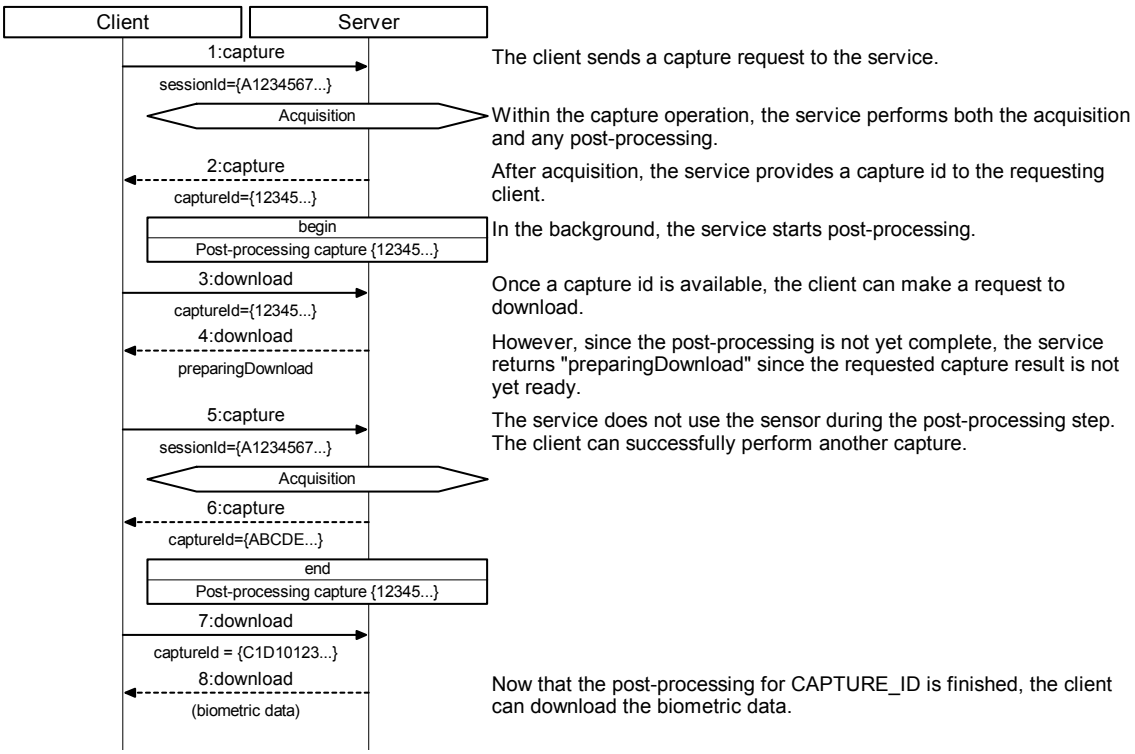


Figure 9. Example of capture with separate post-acquisition processing that does involve the target biometric sensor. Because the post-acquisition processing does not involve the target biometric sensor, it is available for sensor operations.

Services with an independent post-processing step *should* perform the post-processing on an independent unit of execution (e.g., a separate thread, or process). However, post-processing *may* include a sensor operation, which would interfere with incoming sensor requests.

EXAMPLE: Figure 10 illustrates another variation on a *capture* operation with separate post-processing. Return to the digital camera example, but assume that it is a physically separate implementation and capture operation returns immediately after acquisition. The service also has post-acquisition process that downloads the image data from the camera to a computer. Like the previous example, during post-processing, requests for the captured data return `preparingDownload`. However, the sensor is *not* available for additional operations because the post-processing step requires complete control over the camera to transfer the images to the host machine: preparing them for download.

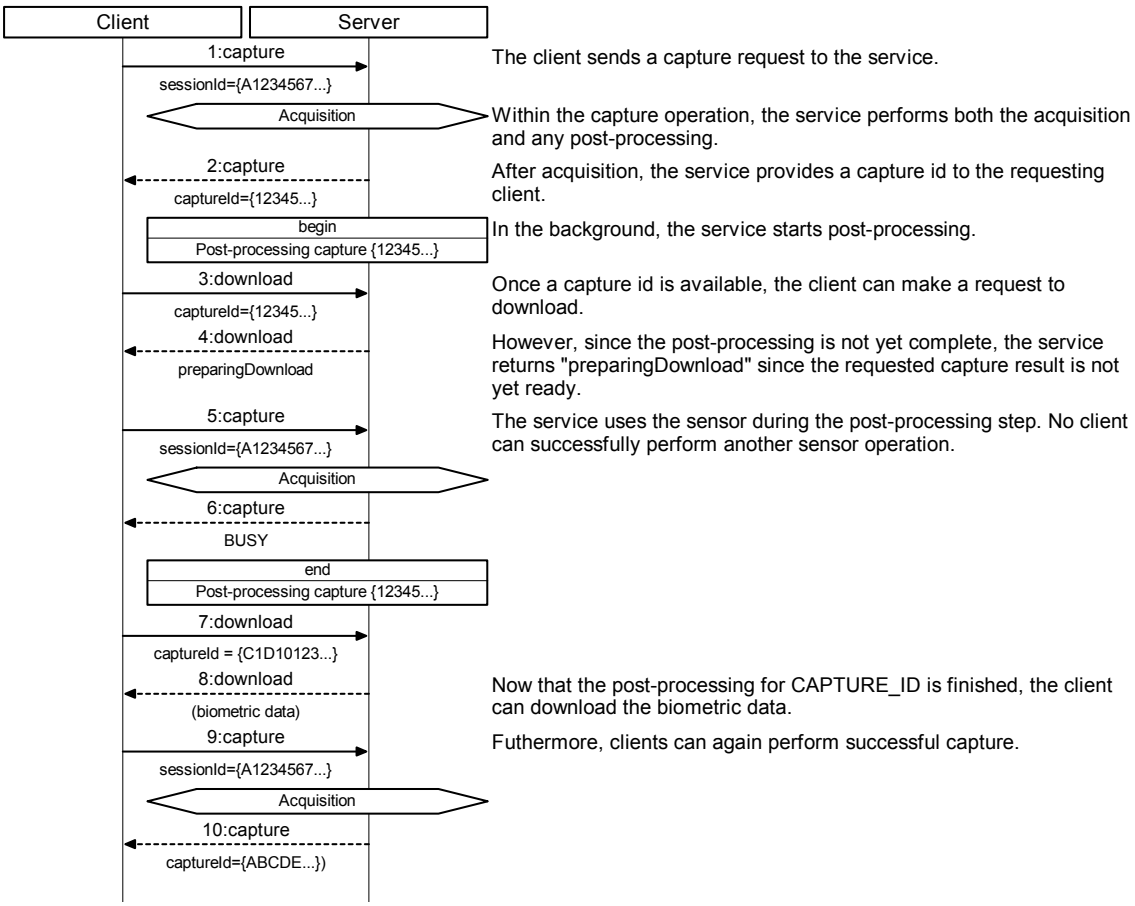


Figure 10. Example of capture with separate post-acquisition processing that does involve the target biometric sensor. Because the post-acquisition processing does not involve the target biometric sensor, it is available for sensor operations.

Unless there is an advantage to doing so, when post-acquisition processing includes a sensor operation, implementers should avoid having a capture operation that returns directly after acquisition. In this case, even when the capture operation finishes, clients cannot perform a sensor operation until the post-acquisition processing is complete.

In general, implementers *should* try to combine both the acquisition and post-acquisition processing into one capture operation—particularly if the delay due to post-acquisition processing is either operationally acceptable or a relatively insignificant contributor to the combined time.

A *download* operation *must* return `failure` if the post-acquisition processing cannot be completed successfully. Such failures cannot be reflected in the originating *capture* operation —that operation has already return successfully with capture ids. Services *must* eventually resolve all `preparingDownload` statuses to `success` or `failure`. Through *get common info*, a service can provide information to a client on how long to wait after capture until a `preparingDownload` is fully resolved.

4.13.2.3 Client Notification

A client that receives a `preparingDownload` must poll the service until the requested data becomes available. However, through *get common info*, a service can provide "hints" to a client on how long to wait after capture until data can be downloaded (§B.2.6)

4.13.3 Unique Knowledge

The *download* operation can be used to provide knowledge about unique characteristics to a service. Through *download*, a client *may* provide a data of a format that does not adhere to any conformance profile [SP500-288A]. *There should be a note regarding where "well accepted" content types & submodalities are defined.*

4.13.4 Return Values Detail

The *download* operation *must* return a WsbdResult according to the following constraints.

4.13.4.1 Success

Status Value	`success`
Condition	The service can provide the requested data
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`" `configuration` (WsbdDictionary, §3.1.3) the sensor's configuration as it was set at the time of capture `contentType` (xs:string, [XSDPart2]) a RFC2045/RFC2046 compliant xs:string [XSDPart2] describing the captured data `submodality` (xs:string, [XSDPart2]) a description of the submodality of the captured data `sensorData` (xs:base64Binary, [XSDPart2]) the biometric data corresponding to the requested capture id, base-64 encoded
Optional Elements	None

A successful download *must* populate the WsbdResult with all of the following information:

1. The `status` element *must* be populated with the WsbdStatus literal "`success`"
2. The `configuration` element *must* be populated with the configuration held by the target biometric sensor at the time of capture. In the case that a capture operation yields multiple capture ids, services *should (or should this be "must"?)* return the same `configuration` contents for each id associated with a particular capture.
3. The `contentType` element must contain an RFC2045/RFC2046 compliant string (xs:string) describing the captured data
4. The `submodality` element must contain a string (xs:string) describing the submodality of the captured data. In the case that the modality does not have an applicable submodality (e.g., "face"), the modality itself *must* be used.
5. The `sensorData` element must contain the biometric data, base-64 encoded (xs:base64Binary), corresponding to the requested capture id

See the usage notes for both *capture* (§4.12.2) and *download* (§4.13.2) for more detail regarding the conditions under which a service is permitted to accept or deny download requests.

4.13.4.2 Failure

Status Value	`failure`
Condition	The service cannot provide the requested data.

Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) the literal "`failure`"

A service might not be able to provide the requested data due to failure in post-acquisition processing, a corrupted data store or other service or storage related failure.

4.13.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided capture id is not recognized by the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`captureId`"
Optional Elements	None

A capture id is invalid if it was not returned by a *capture* operation. A capture id may become unrecognized by the service automatically if the service automatically clears storage space to accommodate new captures (§B.3).

See §4.1.2 for general information on how services must handle parameter failures.

4.13.4.4 Bad Value

Status Value	`badValue`
Condition	The provided capture id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`captureId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.13.4.5 Preparing Download

Status Value	`preparingDownload`
Condition	The requested data cannot be provided because the service is currently performing a post-acquisition process—i.e., preparing it for download
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`preparingDownload`"
Optional Elements	None

See the usage notes for both *capture* (§4.12.2) and *download* (§4.13.2) for full detail.

4.14 Get Download Info

Description	Get only the metadata associated with a particular capture
HTTP Method	`GET`
URL Template	`/download/{captureId}/info`
URL Parameters	`{captureId}` (UUID, §3.1.2) Identity of the captured data to query
Input Payload	Not applicable
Idempotent	Yes
Sensor Operation	No

4.14.1 WsbdResult Summary

`success`	`status="success"` `configuration`=sensor configuration at the time of capture `contentType`= content type [RFC2045, RFC2046] describing the captured data `submodality`= the submodality of the captured data (xs:string)
`failure`	`status="failure"` `message*`=informative message describing failure
`invalidId`	`status="invalidId"` `badFields={"captureId"}` (WsbdStringArray, §3.1.4)
`badValue`	`status="badValue"` `badFields={"captureId"}` (WsbdStringArray, §3.1.4)
`preparingDownload`	`status="preparingDownload"`

4.14.2 Usage Notes

Given the potential large size of some biometric data the *get download info* operation provides clients with a way to get information about the biometric data without needing to transfer the biometric data itself. It is logically equivalent to the *download* operation, but without any sensor data. Therefore, unless detailed otherwise, the usage notes for *download* (§4.14.2) also apply to *get download info*.

4.14.3 Unique Knowledge

The *get download info* operation can be used to provide knowledge about unique characteristics to a service. Through *get download infodownload*, a client *may* provide a data of a format that does not adhere to any conformance profile [SP500-288A].

4.14.4 Return Values Detail

The *get download info* operation *must* return a WsbdResult according to the following constraints.

4.14.4.1 Success

Status Value	`success`
Condition	The service can provide the requested data
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`" `configuration` (WsbdDictionary, §3.1.3) the sensor's configuration as it was set at the time of capture `contentType` (xs:string, [XSDPart2])

	a RFC2045/RFC2046 compliant xs:string [XSDPart2] describing the captured data `submodality` (xs:string, [XSDPart2]) a description of the submodality of the captured data
Optional Elements	None

A successful *get download info* operation returns all of the same information as a successful *download* operation (§4.13.4.1), but without the sensor data.

4.14.4.2 Failure

Status Value	`failure`
Condition	The service cannot provide the requested data.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` (xs:string, [XSDPart2]) the literal "`failure`"

A service might not be able to provide the requested data due to failure in post-acquisition processing, a corrupted data store or other service or storage related failure.

4.14.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided capture id is not recognized by the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`captureId`"
Optional Elements	None

A capture id is invalid if it was not returned by a *capture* operation. A capture id may become unrecognized by the service automatically if the service automatically clears storage space to accommodate new captures (§B.3).

See §4.1.2 for general information on how services must handle parameter failures.

4.14.4.4 Bad Value

Status Value	`badValue`
Condition	The provided capture id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`captureId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.14.4.5 Preparing Download

Status Value	`preparingDownload`
Condition	The requested data cannot be provided because the service is currently performing a post-acquisition process—i.e., preparing it for download
Required Elements	`status` (WsbdStatus, §3.1.6)

	the literal "`preparingDownload`"
Optional Elements	None

See the usage notes for both *capture* (§4.12.2) and *download* (§4.13.2) for full detail.

4.15 Thrifty Download

Description	Download a compact representation of the captured biometric data suitable for preview
HTTP Method	`GET`
URL Template	`/download/{captureId}/{maxSize}`
URL Parameters	`{captureId}` (UUID, §3.1.2) Identity of the captured data to download `{maxSize}` (xsd:integer) Content-type dependent indicator of maximum permitted download size
Input Payload	None
Idempotent	Yes
Sensor Operation	No

4.15.1 WsbdResult Summary

`success`	`status="success"` `contentType=` content type [RFC2045, RFC2046] describing the captured data `sensorData=`biometric data (xs:base64Binary)
`failure`	`status="failure"` `message*=`informative message describing failure
`invalidId`	`status="invalidId"`
`badValue`	`status="badValue"` `badFields=`either `"captureId"`, `"maxSize"`, or both (WsbdStringArray, §3.1.4)
`unsupported`	`status="unsupported"`
`preparingDownload`	`status="preparingDownload"`

4.15.2 Usage Notes

The *thrifty download* operation allows a client to retrieve a compact representation of the biometric data acquired during a particular capture. It is logically equivalent to the *download* operation, but provides a compact version of the sensor data. Therefore, unless detailed otherwise, the usage notes for *download* (§4.14.2) also apply to *get download info*.

The suitability of the *thrifty download* data as a biometric is implementation dependent. For some applications, the compact representation may be suitable for use within a biometric algorithm; for others, it may only serve the purpose of preview.

The content type of *thrifty download* operation *may* be different than that for download. (Notice that the specification does not have a separate operation for querying the content type of *thrifty download* operation sensor data). A service also *may* offer different major media types for its *download* and *thrifty download* operation.

> **EXAMPLE**: A service exposing a fingerprint scanner may provide template data in a *download* operation, but still offer a visual thumbnail of the captured biometric with *thrifty download*.

For images, the `maxSize` parameter describes the maximum image width or height (in pixels) that the service may return; neither dimension may exceed `maxSize`. It is expected that servers will dynamically scale the captured data to fulfill a client request. This is not strictly necessary, however, as long as the maximum size requirements are met.

For non-images, the default behavior is to return `unsupported`. It is *possible* to use URL parameter `maxSize` as general purpose parameter with implementation-dependent semantics. (See the next section for details.)

4.15.3 Unique Knowledge

The *thrifty download* operation can be used to provide knowledge about unique characteristics to a service. Through *thrifty download*, a service *may* (a) redefine the semantics of `maxSize` or (b) provide a data of a format that does not adhere to any conformance profile [SP500-288A].

4.15.4 Return Values Detail

The *thrifty download* operation *must* return a WsbdResult according to the following constraints.

4.15.4.1 Success

Status Value	`success`
Condition	The service can provide the requested data
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`success`" `contentType` (xs:string, [XSDPart2]) a RFC2045/RFC2046 compliant xs:string [XSDPart2] describing the captured data `sensorData` (xs:base64Binary, [XSDPart2]) the biometric data corresponding to the requested capture id, base-64 encoded, scaled appropriately to the `maxSize` parameter.
Optional Elements	None

For increased efficiency, a successful the *thrifty download* operation only returns the sensor data, and its associated content type.

4.15.4.2 Failure

Status Value	`failure`
Condition	The service cannot provide the requested data.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`failure`"
Optional Elements	`message` An informative description of the nature of the failure.

A service might not be able to provide the requested data due to a corrupted data store or other service or storage related failure.

4.15.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided capture id is not recognized by the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`captureId`"
Optional Elements	None

A capture id is invalid if it does not correspond to a *capture* operation. A capture id may become unrecognized by the service automatically if the service automatically clears storage space to accommodate new captures (§B.3).

See §4.1.2 for general information on how services must handle parameter failures.

4.15.4.4 Bad Value

Status Value	`badValue`
Condition	The provided capture id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains one or both of the following fields: - "`captureId`" if the provided session id not well-formed - "`maxSize`" if the provided maxSize parameter is not well-formed
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

4.15.4.5 Unsupported

Status Value	`unsupported`
Condition	The service does not support thrifty download,

Required Elements	`status`
	must be populated with the WsbdStatus literal `"badValue"`
	
	`badFields`
	must be populated with a WsbdStringArray (§3.1.4) that contains the single field name, "`maxSize`"—i.e. "`<element>maxSize</element>`".

Optional Elements	None

Services that capture biometrics that are not image-based *should* return `unsupported`.

4.15.4.6 Preparing Download

Status Value	`preparingDownload`
Condition	The requested data cannot be provided because the service is currently performing a post-acquisition process—i.e., preparing it for download
Required Elements	None
Optional Elements	None

Like *download*, the availability of *thrifty download* data may also be affected by the sequencing of post-acquisition processing. See §4.13.2.2 for detail.

4.16 Cancel

Description	Cancel the current sensor operation
HTTP Method	`POST`
URL Template	`/cancel/{sessionId}`
URL Parameters	`{sessionId}` (UUID, §3.1.2) Identity of the session requesting cancelation
Input Payload	None
Idempotent	Yes
Sensor Operation	Yes

4.16.1 WsbdResult Summary

`success`	`status="success"`
`failure`	`status="failure"` `message*`=informative message describing failure
`invalidId`	`status="invalidId"`
`lockNotHeld`	`status="lockNotHeld"`
`lockHeldByAnother`	`status="lockHeldByAnother"`
`badValue`	`status="badValue"`, `badFields={"sessionId"}`

4.16.2 Usage Notes

The *cancel* operation stops any currently running sensor operation; it has no effect on non-sensor operations. If cancelation of an active sensor operation is successful, *cancel* operation receives `success` result, while the canceled operation receives a `canceled` (or `canceledWithSensorFailure`) result. As long as the operation is canceled, the *cancel* operation itself receives a `success` result, regardless if cancelation caused a sensor failure. In other words, if cancelation caused a fault within the target biometric sensor, as long as the sensor operation has stopped running, the *cancel* operation is considered to be successful.

Some illustrative sequence diagrams should go here.

All services *must* provide cancellation for all sensor operations.

4.16.2.1 Canceling Non-Sensor Operations

Clients are responsible for canceling all non-sensor operations via client-side mechanisms only. Cancellation of sensor operations requires a separate service operation, since a service may need to "manually" interrupt a busy sensor. A service that had its client terminate an operation would have no way to easily determine that a cancellation request.

An example of a client-side cancelation should go here.

4.16.2.2 Cancelation Triggers

Canceling a sensor operation may be triggers by a *client* or by the *service* itself.

Typically, the client that originates the sensor operation to be cancelled also initiates the cancelation request. Because WSBD operations are performed synchronously, cancellations are typically initiated on an separate unit of execution such as an independent thread or process.

Notice that the only requirement to perform cancellation is that the *requesting* client hold the service lock. It is *not* a requirement that the client that originates the sensor operation to be canceled also initiates the cancellation request.

Therefore, as discussed in it is *possible* that a client may cancel the sensor operation initiated by another client. This occurs if a peer client (a) manages to steal the service lock before the sensor operation is completed, or (b) is provided with the originating client's session id.

A service might also *self-initiate* cancelation. In normal operation, a service that does not receive a timely response from a target biometric sensor would return `sensorTimeout`. However, if the service's internal timeout mechanism fails, a service *may* initiate a cancel operation itself. Implementers should use this as a "last resort" compensating action.

4.16.3 Unique Knowledge

As specified, the *cancel* operation cannot be used to provide or obtain knowledge about unique characteristics of a client or service.

4.16.4 Return Values Detail

The *cancel* operation *must* return a WsbdResult according to the following constraints.

4.16.4.1 Success

Status Value	`success`
Condition	The service successfully canceled the sensor operation
Required Elements	`status` *must* be populated with the WsbdStatus literal `"success"`
Optional Elements	None

See the usage notes for *capture* (§4.12.2) and *download* (§4.13.2) for full detail.

4.16.4.2 Failure

Status Value	`failure`
Condition	The service could not cancel the sensor operation
Required Elements	`status` *must* be populated with the WsbdStatus literal `"failure"`
Optional Elements	`message` An informative description of the nature of the failure.

Services *should* try to return `failure` in a timely fashion—there is little advantage to a client if it receives the cancelation failure *after* the sensor operation to be canceled completes.

4.16.4.3 Invalid Id

Status Value	`invalidId`
Condition	The provided session id is not recognized by the service.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`invalidId`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

A session id is invalid if it does not correspond to an active registration. A session id may become unregistered from a service through explicit unregistration or triggered automatically by the service due to inactivity (§4.4.4.1).

See §4.1.2 for general information on how services must handle parameter failures.

4.16.4.4 Lock Not Held

Status Value	`lockNotHeld`
Condition	The service could cancel the operation because the requesting client does not hold the lock.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockNotHeld`"
Optional Elements	None

Sensor operations require that the requesting client holds the service lock.

4.16.4.5 Lock Held by Another

Status Value	`lockHeldByAnother`
Condition	The service could not cancel the operation because the lock is held by another client.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`lockHeldByAnother`"
Optional Elements	None

4.16.4.6 Bad Value

Status Value	`badValue`
Condition	The provided session id is not a well-formed UUID.
Required Elements	`status` (WsbdStatus, §3.1.6) the literal "`badValue`" `badFields` (WsbdStringArray, §3.1.4) an array that contains the single field name, "`sessionId`"
Optional Elements	None

See §4.1.2 for general information on how services must handle parameter failures.

Appendix A Common Info

This contains detailed information on the set of "well-accepted" metadata potentially available from a *get common info* operation.

A.1 Data Dictionary

Data types used here can be moved into the global data dictionary (§3), or, if there are no data types unique to this section, than this subsection can be removed.

A.1.1 H3

A.1.1.1 H4

Appendix B Parameter Details

This subsection details the individual parameters available from a *get common info* operation. For each parameter, the following information is listed:

- The formal parameter name
- The expected data type of the parameter's value
- If a the service is required to implement the parameter

B.1 Connections

The following parameters describe how the service handles session lifetimes and registrations.

B.1.1 Last Updated

Formal Name	`lastUpdated`
Data Type	`xs:dateTime` [XSDPart2]
Required	Yes

This parameter provides a timestamp of when the service last *updated* the common info parameters (this parameter not withstanding). The timestamp *must* include time zone information. Implementers *should* expect clients to use this timestamp to detect if any cached values of the (other) common info parameters may have changed.

B.1.2 Inactivity Timeout

Formal Name	`inactivityTimeout`
Data Type	`xs:nonNegativeInteger` [XSDPart2]
Required	Yes

This parameter describes how long, in *seconds*, a session may be inactive before it may be automatically closed by the service. A value of '0' indicates that the service never drops sessions due to inactivity.

Inactivity time is measured *per session*. Services *must* measure it as the time elapsed between (a) the time at which a client initiated the session's most recent operation and (b) the current time. Services *must* only use the session id to determine a session's inactivity time. For example, a service does not maintain different inactivity timeouts for requests that use the same session id, but originate from two different IP addresses. Services *may* wait longer than the inactivity timeout to drop a session, but *must not* drop inactive sessions any sooner than the `inactivityTimeout` parameter indicates.

B.1.3 Maximum Concurrent Sessions

Formal Name	`maximumConcurrentSessions`
Data Type	`xs:positiveInteger` [XSDPart2]
Required	Yes

This parameter describes the maximum number of concurrent sessions a service can maintain. Upon startup, a service *must* have zero concurrent sessions. When a client registers successfully (§4.3), the service increases

its count of concurrent sessions by one. After successful unregistration (§4.4), the service decreases its count of concurrent sessions by one .

B.1.4 Least Recently Used (LRU) Sessions Automatically Dropped

Formal Name	`autoDropLruSessions`
Data Type	`xsd:boolean` [XSDPart2]
Required	Yes

This parameter describes whether or not the service automatically unregisters the least-recently-used session when the service has reached its maximum number of concurrent session. If *true*, then upon receiving a registration request, the service may drop the least-recently used session if the maximum number of concurrent sessions has already been reached. If *false*, then any registration request that would cause the service to exceed its maximum number of concurrent sessions results in failure.

B.2 Timeouts

Clients *should* not block indefinitely on any operation. However, since different services may differ significantly in the time they require to complete an operation, clients require a means to determine appropriate timeouts. The timeouts in this subsection describe how long a *service* waits until the service either returns `sensorTimeout` or initiates a service-side cancellation (§4.16.2.1). Services *may* wait longer than the times reported here, but, (under normal operations) *must not* report a `sensorTimeout` or initiate a cancelation before the reported time elapses. In other words, a client *should* be able to use these timeouts to help determine a reasonable upper bound on the time required for sensor operations.

Note that these timeouts do not include any round-trip and network delay—clients *should* add an additional window to accommodate delays unique to that particular client-server relationship.

B.2.1 Detailed Info Timeout

Formal Name	`detailedInfoTimeout`
Data Type	`xsd:nonNegativeInteger` [XSDPart2]
Required	Yes

This parameter describes how long, *in milliseconds*, a service will wait for a target biometric sensor to perform a detailed information operation before it returns `sensorTimeout` (§4.12.4.12) or initiates a service-side cancellation (§4.16.2.1).

B.2.2 Initialization Timeout

Formal Name	`initializationTimeout`
Data Type	`xs:positiveInteger` [XSDPart2]
Required	Yes

This parameter describes how long, *in milliseconds*, a service will wait for a target biometric sensor to perform initialization before it returns `sensorTimeout` (§4.9.4.10) or initiates a service-side cancellation (§4.16.2.1).

B.2.3 Get Configuration Timeout

Formal Name	`getConfigurationTimeout`
Data Type	`xs:positiveInteger` [XSDPart2]

Required	Yes

This parameter describes how long, *in milliseconds*, a service will wait for a target biometric sensor to retrieve its configuration before it returns `sensorTimeout` (§4.10.4.12) or initiates a service-side cancellation (§4.16.2.1).

B.2.4 Set Configuration Timeout

Formal Name	`setConfigurationTimeout`
Data Type	`xs:positiveInteger` [XSDPart2]
Required	Yes

This parameter describes how long, *in milliseconds*, a service will wait for a target biometric sensor to set its configuration before it returns `sensorTimeout` (§4.11.4.11) or initiates a service-side cancellation (§4.16.2.1).

B.2.5 Capture Timeout

Formal Name	`captureTimeout`
Data Type	`xs:positiveInteger` [XSDPart2]
Required	Yes

This parameter describes how long, *in milliseconds*, a service will wait for a target biometric sensor to perform biometric acquisition before it returns `sensorTimeout` (§4.11.4.11) or initiates a service-side cancellation (§4.16.2.1).

B.2.6 Post-Acquisition Processing Time

Formal Name	`postAcquisitionProcessingTime`
Data Type	`xsd:nonNegativeInteger` [XSDPart2]
Required	Yes

This parameter describes an upper bound on how long, *in milliseconds*, a service takes to perform post-acquisition processing. A client *should not* expect to be able to download captured data *before* this time has elapsed. Conversely, this time also describes how long after a capture a server is permitted to return `preparingDownload` for the provided capture ids. A value of zero (‘`0`’) indicates that the services includes any post-acquisition processing within the `capture` operation.

B.2.7 Lock Stealing Prevention Period

Formal Name	`postAcquisitionProcessingTime`
Data Type	`xsd:nonNegativeInteger` [XSDPart2]
Required	Yes

This parameter describes the length, in *seconds*, of the lock stealing prevention period (§4.6.2.2).

B.3 Storage

The following parameters describe how the service stores captured biometric data.

B.3.1 Maximum Storage Capacity

Formal Name	`maximumStorageCapacity`
Data Type	`xs:positiveInteger` [XSDPart2]

Required	Yes

This parameter describes how much data, in bytes, the service is capable of storing.

B.3.2 Least-Recently Used Capture Data Automatically Dropped

Formal Name	`lruCaptureDataAutomaticallyDropped`
Data Type	`xsd:boolean` [XSDPart2]
Required	Yes

This parameter describes whether or not the service can automatically deletes the least-recently-used capture to stay within its maximum storage capacity. If *true*, the service may automatically delete the least-recently used biometric data to accommodate for new data. If *false*, then any operation that would require the service to exceed its storage capacity would fail.

Appendix C Configuration

There should be discussion here about how a client might go about determining a configuration—either building it from the "bottom up" based on the parameter descriptions, or, by simple substitution from a default configuration.

C.1 Parameter Description

This section will contain information on how a service describes the configuration parameters that can be set.

Appendix D Content Type Data

D.1 Content Type

This subsection contains a catalog of content types for use in conformance profiles. **Error! Reference source not found.** details which data format are appropriate

D.2 Image Formats

image/bmp	
image/jpeg	
image/png	
image/tiff	

D.3 Video Formats

D.4 General Biometric Formats

biometric/ansi-nist-itl-2000	
biometric/cbeff-xxx	

D.5 Modality-Specific Formats

biometric/iso-19794-	

Appendix E Security Profiles

E.1 None

This section will describe a configuration in which all security is established at a lower network level.
Implementers *should* only use this security profile for testing and development.

E.2 HTTPS

This section will describe a configuration in which HTTPS is used for encryption purposes. There should be a note about the optional use of

- *Basic or digest authentication*
- *Client-side certificates*

E.3 OpenID & OAuth

This section might describe a configuration using OpenID and/or OAuth to control access to the service at a fine level of granularity. This may be out of scope for this revision of the document.

Appendix F Live Preview

This section will contain guidance on how to support live preview.

Appendix G Pending Issues

The following is a list of pending/open issues regarding the WS-BD specification.

1. It is assumed that messages are returned from the service in a single language. Integrated multilingual support might be supported by a special sensor service configuration operation, or, by supporting multiple languages within the messages returned by a service.
2. Should there be a 'try configuring' operation that *tests* to see if a configuration is valid before it executes it? Would the results have any sort of runtime guarantees, or more of a test to determine if a configuration is supported or not? Would this operation require a lock?
3. Should we allow sensor data to also contain XML, rather than just a base64 encoded payload? Is there are role for XML/binary hybrids?
4. Should we allow for content negotiation?
5. Should the LSPP be determined automatically, based on the longest running sensor operation?

Appendix H Acknowledgments

The authors thank the following individuals for their participation in the creation of this specification.

Tod Companion, Department of Homeland Security, Science & Technology Directorate
Bert Coursey, Department of Homeland Security, Science & Technology Directorate
Rhonda Farrell, Booz Allen Hamilton
Michael Garris, National Institute of Standards and Technology, Information Technology Lab
John Manzo, Federal Bureau of Investigation, Biometric Center of Excellence
Scott Swann, Federal Bureau of Investigation
Cathy Tilton, Daon Inc.
Bradford Wing, National Institute of Standards and Technology, Information Technology Lab

Appendix I Revision History

Draft 0—Initial release. Operations and data types are well defined, but detailed documentation is not yet complete. Appendixes (metadata, conformance, and security profiles) are not yet written.

Draft 1—Second release. Made significant improvements based on public comment. Removed 'Detailed Info' and augmented 'Get Content Type' into 'Get Download Info.' Detailed operation documentation is complete, but appendixes still need work.