HOPE Databroker – User Manual Vinter bundle

ADDI Medical

Revision history

Doc. version	Date	Author	Description
v1.0.0	2021-02-26	BS (Björn Strihagen)	First official version

Content

1 Introduction

References

1 HOPE Solution - Security features v1.1.0

About HOPE Databroker

HOPE Databroker is the backbone component in the HOPE suite and serves as a versatile platform for developing and integrating software products within focus on the welfare, healthcare, life science industry and research with a special focus on patient involvement.

It provides features for storing, retrieving and communicating medical data, for patient communication. And all this with strict requirement on security and legislations compliance including authentication, access control and patients consents.

HOPE Databroker can be used in conjunction with the other components in the HOPE suite, HOPA App and HOPE Practitioner to form a complete out of the box patient support product. It can also be used “stand alone” as backbone for third party applications or as a data integration engine.

Notable is that *patient involvement* is not limited to making individually generated information from sensors, mobile devices, e-forms etc. available to healthcare providers. Equally important is the other way around, making information and notifications available to the patient, e.g. activity plans, video-instructions and electronic documentation and that the communication can be performed via the channels and devices preferred by the individual, such as computer, mobile, web, mail, chat and push notifications.

The HOPE suite has been developed together with patients and healthcare providers, for patients and healthcare providers, focusing on simplicity, quality and safety.

About this document

This document describes the HOPE Databroker from a user’s perspective. Since HOPE Databroker is accessed via its Web-service based API, the “user” in this case is typically developers, designers or managers that needs an understanding of the features of HOPE Databroker, beyond what is apparent from the marketing material.

Terminology

term	description
Client	The part that invokes (consumes) the HOPE Databroker services.

	Note: We use this since it has proven more easily understood than *consumer*, which is often confused with data consumer.
Module	A group of features accessible via the HOPE Databroker API. On module may have several operations. Note: We avoid the term *service*, since it is ambiguous.
Operation	On single invocation point. Sometimes referred to as *method*. Note: We avoid the term *service*, since it is ambiguous.
Part	A computer program that is involved in data exchange via HOPE Databroker including the modules themselves. I.e a *client* or a *server*.
Patient	In this context the term *patient* is used equivalent to *individual*. Notable is that a *patient* doesn’t have to be a patient in the sense that it is subject for care. We just use the term patient, in lack of better, to represent the role in the system.
Provider	The modules and data that is private for a specific organization. Also called *account*.
Server	Service producer

2 General

2.1 Background

Healthcare Operability with Patient Engagement - HOPE Solution is a modular solution supporting the **exchange of information** between patients and healthcare providers (including the life science industry). This is achieved by making individually generated information from sensors, mobile devices and e-forms available to existing systems of the healthcare provider. Equally important is the other way around, making information from the healthcare providers available to the patient, e.g. activity plans, video-instructions and electronic documentation.

And all this with strict requirement on security and legislations compliance including the consent from the patients.

HOPE Solution has been developed together with patients and healthcare providers, for patients and healthcare providers; focusing on simplicity, quality and safety.

One obvious requirement (once you give it some thought) for a product that is supporting exchange of information between patients and healthcare providers is that it should be reasonably easy to integrate with the healthcare provider's existing systems. No one wants yet another stand-alone system. In addition, it should be flexible enough to allow for the healthcare provider to select (and pay for) only those features that are actually needed.

The "magic sauce" in this is the modular service structure with open API:s which allows for services to be invoked directly from the customers' systems as well as for the modules to be replaced with custom fitted implementation whenever that is needed. In addition is the data model based on the HL7 FHIR standard, which focuses on interchangeability.

2.2 Module overview

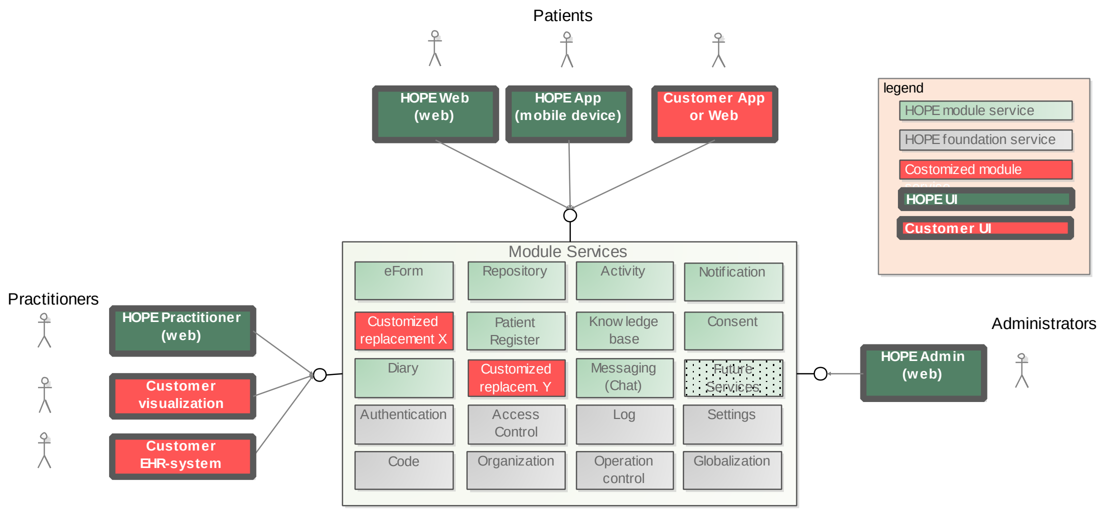

Figure 1. The modular service structure with open API:s is the basis for HOPE Solution.

HOPE Solution provides a wide range of features from which a specific user will possibly have to use for just a few. For this reason, HOPE Solution is divided into ***modules*** – which represent a group of related features from the user's perspective and where the modules can be used independently from each other (at least to some extent). This makes it possible for a user to select (and pay for) only the modules of interest (and possibly to add remaining modules later).

In technical terms, every module is implemented as a **web-service** with a public API (aka *Service contract*) from which it is accessed by user interface components as well as from other server-side systems.

Notable is that the HOPE modules themselves as well as the other products in the HOPE suite all use this public API. This means that whatever HOPE UI features you see, could also be implemented by any user.
This is in contrast to most commercial products where, in the case there is a public API in the first place, it is an add-on that exposes a limited set of features while the vendors own products uses other (secret) access methods, e.g. data sharing via a common database.

But even if all modules *can* be accessed, this doesn't mean that all modules *must* be accessed. At least not directly. Access can also be made indirectly, i.e. from another module. One example is when the modules make use of the security modules (*Authentication, AccessControl, Consent and Log)* to verify that the user is authenticated (logged in) and should be granted access to the operation or data requested.

This, of course, relieves the client developer from that responsibility which both makes life easier and the solution safer.

2.3 Provider model

The ***provider*** (also knowns as an *account)* is an important concept in HOPE Databroker. A *provider* is a piece of storage reserved for one specific organization (customer or health care provider).

On a technical level, several providers can be managed in the same *installation*, all sharing the same URL, and several installations can be hosted on the same machine.

The main thing to keep in mind is that providers are *totally isolated* from each other. This goes down to a physical level where each provider has its own set of database files, its own setting, its own set of automation files etc. You can think of it as the equivalent to virtual machines.

The main goal of this total separation is to guarantee that data from different data controllers (sv. *personuppgiftsansvariga*) are kept separate from each other, even if they are hosted in the same installation. This also meets the requirement from the data controllers to be able to instantly stop or remove a specific provider without affecting other providers.

As an extra benefit, this also makes it possible to easily scale up a provider by moving it to another machine without affecting neither other providers nor the patients using HOPE App.

Even within the same data controller it might be useful to make use of multiple providers. This is the case if HOPE Databroker is used in multiple “business areas” with basically nothing in common. Different patients, different automation rules, different practitioners etc. In this case just keep in mind that data is really separated and thus you cannot e.g. automatically aggregate data for the same patient from multiple providers

2.4 W3C Web Services

The modules are accessed using W3C Web Services (SOAP), or just WS for short. The API is formally described in WSDL and XSD.

This is a technique that is widely used and supported by all major development tools providers, which provides for good transport layer security and complies to national de facto standards and recommendations.

Furthermore, the individual operations are stateless in the sense that is commonly used in the craft of the trade[1].

We know what you are thinking - as a frontend developer you “need” a REST-API.
No problem. Since your application most likely will require some kind of backend layer

[1] On a general level all persistent storage represent state. The same goes for the authentication session. So, there are actually very few purely stateless servers out there.

anyway, you can still use REST between the frontend and the backend and let the backend do the WS-calls.

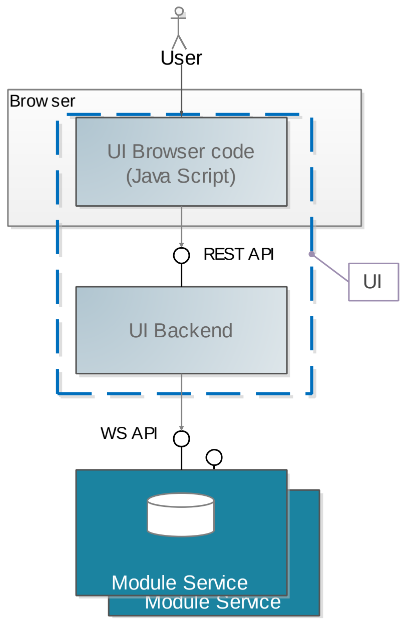

Figure 2. The figure shows a typical scenario with an application (called UI) accessing module services provided by HOPE Databroker. The application uses REST internally and WS for HOPE Databroker access.

2.5 Abstract identifiers

In HOPE Databroker all persistent objects are internally identified by an abstract id, a GUID, which is guaranteed to be universally unique. (*ReferencedObject.guid*)

When an object is created and saved in HOPE Databroker a new GUID is by default created and assigned by HOPE Databroker. After that, the GUID cannot be changed throughout the entire lifetime of that object.

Consequently the GUID:s are used when object reference each other within HOPE Databroker, e.g. when an *Activity* refers to its “parent” *ActivityPlan* or when an *Observation* refers to its corresponding *Activity*.

Notably, this is also the case when referencing *Patient* objects. So even if there is an identification attribute (*Actor.identifier*), this is intended for identifying the object externally to HOPE Databroker. From the point of view from HOPE Databroker, a *ssn* (sv. *personnummer*), *HSA-Id* or another well-known identification token, is just considered as any data attribute. Thus, changing the *ssn* for a patient is totally possible. But changing the GUID is not.

Having said that, there actually is a check in the *PatientRegister* to prevent duplicate identification attributes for patients. But that is more of a special case.

As stated above - the GUID is *by default* created and assigned by HOPE Databroker when the object is created. However, if that would *always* be the case it would become almost impossible to allow objects to be exported and imported since existing cross references must be maintained. So, to allow for this the HOPE Databroker only creates a new GUID if there isn't one already assigned by the client. But do not use this feature if you don't have to. The risk is that the client re-uses a GUID (by mistake) resulting in an existing object being overwritten.

2.6 Security modules

HOPE Databroker provides several modules to meet the high requirements on security for healthcare data (Authentication, AccessControl, Consent, Log).

Each of these is described separately, but it might be handy with a short overview of how they interact.

1. All access of the HOPE Databroker is ***encrypted*** using https (SSL/TLS). This is actually not a feature in HOPE Databroker itself but rather a standard way to deploy any web-based application.
2. All operation calls to HOPE Databroker requires the end user to be ***authenticated*** (logged in). The proof of such an authentication is a session key retrieved from the authentication process. This session key, which has a limited lifetime is passed as part of the *Context*-attribute in every operation.
   The only exceptions to "*All operation calls*" above, are the operations that are involved in the authentication process itself.
3. But just because HOPE Databroker has knowledge about who is attempting an operation call, this does not automatically ***authorize*** the user, i.e. grant access to that operation or to the "resources" requested. This is achieved by the operations themselves calling the ***AccessControl*** module.
4. The output from the access control is defined by a number of access rules that can be quite complex and also include the patients ***consent*** (defined in the Consent module). The access rules are defined "manually" during the configuration phase for that provider.
5. Finally, all access is logged according to the legal requirements (sv. *Socialstyrelsens föreskrifter*) by the operations calling the ***Log*** module.

Notice that the client application has no responsibility to maintain the security in HOPE Databroker, and thus the security is *not* dependent of the client application being trusted.

And from the client developer perspective, all that is required is really just to initiate the authentication and call the appropriate operations.

But what is the meaning of exposing the modules for AccessControl and Log at all, if they are only used from within HOPE Databroker?
The thing is that if even if they do not have to be called by the client doesn't mean it wouldn't be useful.

In a user-friendly application, it is good practice to disable features on a UI-level to which the user is not authorized. So e.g., if the application has a "Delete patient"-button this should most likely be enabled only if the user is allowed to delete patients. To find out if that is the case, the application can call the AccessControl module directly (i.e., without doing actual attempt to delete a patient).
Failing to do such a check in advance will not allow the patient to be deleted, just make the user confused. So, the purpose here is purely *usability*, not security.

Regarding the Log module, this can be convenient for an application that is accessing data sources outside HOPE Databroker (in addition to HOPE Databroker) but for convenience needs to keep all accesses in the same log. For this reason, the Log module can be called directly.

2.7 Connect to HOPE Databroker

Before you can start using the "useful" services in HOPE Databroker, such as store and retrieve data in the repository you must go through the following steps.
It is assumed that If the "owner" of that provider is not yourself, especially if the data is supplied by another data controller than yourself, you will need to have to agree on the legal terms.

1. There must be an existing provider that is configured and customized for by service provider. Notably for this context, it must be configured with the appropriate authentication methods and user roles and access rules.
2. Your application must have access to the HOPE Databroker *URL* and the *provider id* for the specific provider.
3. Your application must be granted access to HOPE Databroker services in the first place. See note 1.
   Technically this is done by the use of a secret *application token* which you get from

the service provider “manually” (e.g. via encrypted mail).

4. **In runtime:** Your application authenticates itself, passing the application token when calling `LoginByToken`.

5. For a human user to login, he/she must also be granted access to do so.
For practitioner, the must be registered in the Organization service (by a back office process) and for patients they must either be registered in the Patient register (by the provider) or be allowed to self registrate during login (by a configuration setting).

6. **In runtime:** To log in a user (human or service) the application makes use of the appropriate authentication operations, depending on login method, e.g. `InitiateLogin/FinalizeLogin`, to “guide” the user through the authentication process. See note 2.

Note 1:
Normally, the application is not granted access to actual data. This is restricted to human actors, i.e., a patient or a practitioner, since an application cannot be held responsible for its use (or misuse) of personal data.
But to facilitate the authentication of the human actors the application needs to be granted some limited access.

Note 2:
The thing to note here is that the application uses its own session (achieved when logging in using the *application token*) in the authentication process, but it is still the end user that is logging in with its own credentials (e.g. via BankId), thus retrieving its own session.
So, there are two session involved here. This might seem a bit confusing if you are not aware of it.
The benefit of this is a kind of two-factor protection in itself; an application cannot (by itself) masquerade as a user, and users cannot login to HOPE Databroker unless via an application that is granted access.

2.8 Relation to FHIR

HOPE Databroker is based on the work of HL7 with the FHIR standard (*Fast Healthcare Interoperability Resources*). FHIR is the result of the joint effort of a large number of companies within the healthcare industry around the world, with goal of making the exchange of electronic health records easier. So, for short, FHIR represents the best commonly accepted Healthcare data model.

In HOPE Databroker we have done our best effort to adopt FHIR. But there are some limitations that makes it both impossible and meaningless to strive for 100% compliance on a binary level.

First of all, the FHIR standard (if there is such a thing, since it is not yet approved by HL7 and ANSI) is huge. There is no way anyone could implement it all. Secondly, FHIR does not fully specify all of the aspects that HOPE Databroker provides. Its focus is to provide for exchanging electronic health records which makes e.g. dealing with patient generated data and patient communication is outside the scope. In addition, FHIR still contains a large portion of extendibility (e.g. extensions) that still requires parties to agree on thing on a point-to-point basis.

So, what HOPE Databroker actually does is to comply to the FHIR resources wherever this is possible but only to the extent that is relevant for the features provided by making the data type names in HOPE Databroker identical to the corresponding FHIR resource names- The same goes for the attributes and the code systems.

This means that you can refer to the FHIR documentation and that HOPE Databroker is compliant to FHIR on a *conceptual* level.

2.9 Communication channels

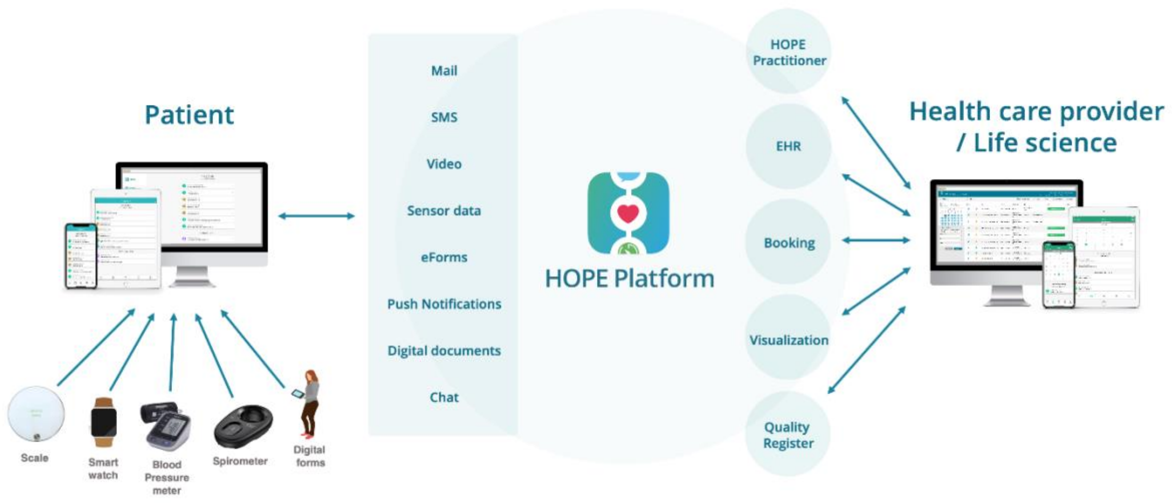

Figure 3. The HOPE Platform serves as a "communication switch" between the healthcare providers systems, and the patient by enabling multiple communication channels.

As mentioned above the purpose of HOPE suite is to provide for *patient involvement.* This includes making individually generated information from sensors, mobile devices, e-forms etc. available to healthcare providers. Equally important is the other way around, making information and notifications available to the patient, e.g. activity plans, video-instructions and electronic documentation and that the communication can be performed via the **channels** and devices preferred by the individual, such as computer, mobile, web, mail, chat and push notifications.

The concept of *channels* should be considered from the users' point of view and should not be confused with the underlaying communication protocol nor with the data type being exchanged.

3 API descriptions

Reading instructions

For readability, the services are not described in the formal wsdl, but in a Java/C#-like style which looks more like the actual proxy-code generated.

3.1 General for the API

The Context-argument

All operations have an argument of type **Context** as first argument, referred to as the context-argument.

attribute	type	code system	card	description
account	string		1	The provider (see *Instance model*)
subject	PatientRef		1	The current patient, if any, for which information is processed
sessionId	string		1	Identifies the user’s session (retrieved at login)
sessionKey	string		1	Authenticates to the user’s session (retrieved at login)
language	Code		1	The language used in service replies (as preferred)
timezone	TimeSpan		1	Obsolete Should be 0 = use provider time zone
transactionId	string		0..1	For tracking events from the UI down to Server-layer

Timezones

When communicating time information between parties, it is of course essential that there is a common agreement on which time zone should be used/expected.

Note that time information isn’t always passed in data types (e.g. DateTime), it is also exist in plain text. E.g to send an e-mail with the text “*Don’t forget your appointment tomorrow at 07:30*) will require the knowledge of the patients time zone (twice – since *tomorrow* only makes sense in the correct time zone).

To make life easier for all parties, all times communicated should be expressed in the **provider time zone**. This is defined on a provider level and should be set to the local time for the patients for that provider.

3.2 Repository

3.2.1 Summary

The *Repository Service* is the core of HOPE Solution. Here is where patient generated data from sensors and e-Forms is stored and retrieved in a structured format in compliance with the FHIR standard.

The service is accessed via a Web Service API which is used from the other applications and services within HOPE Solution as well as from customers applications and services. The general use of the API, as well as the basic data types, is described in [1].

3.2.2 Operations

SaveObservation

Syntax

```
ObservationRef SaveObservation(Context ctx, Observation observation)
```

Remarks

Saves an observation (the object) in the repository. If the object exists (as determined by the guid-attribute) the existing one is replaced, otherwise a new entry is made.

If the guid-attribute is not specified, a new guid is assigned to the object which is also included in the return value.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
observation	Observation	The Observation to store. The guid-attribute is used by the server to determines if the object exists already.
return		
-	ObservationRef	A reference to the stored object.

Preconditions

ctx must be valid (see [1])

observation cannot be null

LookupObservation

Syntax

```
Observation LookupObservation(Context ctx, ObservationRef observation)
```

Remarks

Resolves a reference to an observation and returns that observation (the object), or null if not found.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see [1])
observation	ObservationRef	A reference to the observation
return		
-	Observation	The retrievd observation or null if not found.

Preconditions

ctx must be valid (see [1])

observation cannot be null

RemoveObservation

Syntax

```
void RemoveObservation(Context ctx, ObservationRef observation)
```

Remarks

Note: Consider setting *status = entered_in_error* instead.

This operation allows the observation to be irreversibly removed from the repository. This may be desired during test, if a patient is to be “forgotten” (by GDPR) or if the observation was erroneous in some way, e.g. registered on the wrong patient.

However, in the usual case, observations are rarely removed due since there may be references to the object elsewhere (both within HOPE Solution and in customers systems).

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see [1])
observation	ObservationRef	A reference to the observation that should be removed
return		
-		none (absence of exception acknowledges success)

Preconditions

ctx must be valid (see [1])

observation cannot be null

SearchObservation

Syntax

```
Observation[] SearchObservation( Context ctx,
        DateTime from,
        DateTime until,
        Code code,
        int max_cnt)
```

Remarks

Retrieves all observations matching the condition specified by the arguments.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see [1])
from	DateTime	Compares to *effective* time, inclusive. Use MinValue to include all
until	DateTime	Compares to *effective* time, exclusive. Use MaxValue to include all
code	Code	The *code* (LOINC-code) on top level. Use *null* to include all
max_cnt	int	For technical reasons the number of objects returned is limited by the server (default max = 10.000) but can be limited even more by the caller. Use a negative value to indicate no limits (by the caller).
return		
-	Observation[]	The matching observations, limited by *max_cnt*

Note: *from* may be greater than *until,* resulting in that observations between *from* and *until* are <u>excluded</u> from the result.

Preconditions

- *ctx* must be valid (see [1])

3.2.3 Resources

Observation

The *Observation* type is a subset of the FHIR resource *Observation*. See https://www.hl7.org/fhir/observation.html.

In HOPE Solution it is used for representing information about patient generated data from sensors and e-Forms.

attribute	type	code system	card	description
basedOn	Ref		0..*	The Activity that triggered the measurement, the eForm that was used etc.
status	Code	observation-status	1	Status of the observation.
category	Code	observation-category	0..1	Classifies the general type of observation being made. In HOPE Solution `survey` is used for eForms and `exam` for sensor data.
code	Code	LOINC	1	Type of observation Normally LOINC (see http://loinc.org) should be used. However, there is no technical limitation in HOPE Solution to use other code systems.
subject	PatientRef		0..1	The subject (patient) whose characteristics are described by the observation.
effective	DateTime		1	Clinically relevant time/time-period for observation
performer	ActorRef		0..1	If null, it is assumed in HOPE Solution that this is the same as subject (i.e. the patient).
values	ObservationValue		0..*	The actual observed value (or values)
bodySite	Code	body-site	0..1	Observed body part. Only if applicable.
device	DeviceRef		0..1	The device used to generate the observation data. Only if applicable.

ObservationValue and sub-types

The *ObservationValue* data type is an “abstract type”. This means that there cannot be actual instances of that type. Instead, all instances must be from one of its derived types which hold the actual values of the specific type (string, Code, DateTime etc.).

For convenience, these are all listed in one table.

attribute	type	code system	card	description
	ObservationValue			
code	Code	LOINC	1	Type of value being observed. If there is only one element in *observation.values* it is likely that these have the same value for *code*.
	Types derived from *ObservationValue*			
	ObservationQuantity			
value	Quantity		1	
	ObservationCode			
value	Code		1	
	ObservationString			

value	string		1	
	ObservationDateTime			
value	DateTime		1	
	ObservationQuantitySet			
value	QuantitySet		1	
	ObservationTimeSpan			
value	QuantityTimeSpan		1	

3.3 Authentication

3.3.1 Summary

All operation calls to HOPE Databroker requires the end user to be ***authenticated*** (logged in). The Authentication provides the operations to enable that by supporting the ***PatientId*** (aka *PCC or Pairing codes*), ***BankId***, ***SITHS Enterprise***, ***SITHS Access*** and ***Token*** login methods.

Theoretically, all of the methods can be applied to all user categories, i.e. patients, practitioners and services. In real life it would be very unlikely for a patient to be holder of a SITHS-card, but from a technical perspective the login method and the user category are unrelated. For this reason, we refer in this section to the person (actor) who wants to log in as the ***principal***.

See also the *Security modules* section that describes the relation to other security modules.

The result of an authentication is a session key retrieved from the authentication process. This session key, which has a limited lifetime is passed as part of the *Context*-attribute in every operation.
The only exceptions to "All operation calls" above, are the operations that are involved in the authentication process itself.

3.3.2 Operations

DiscardToken

Syntax

```
void DiscardToken(Context principal_ctx, string token)
```

Remarks

Forgets (discards) either one specific token or all tokens for the principal specified by. A new pairing must then be performed to make token login work again.

Arguments

arguments	type	description
principal_ctx	Context	The context of the principal. *subject* is ignored.
token	string	The token to discard, or null for all tokens.
return		

DiscardAllTokens

Syntax

```
void DiscardAllTokens(Context ctx, ActorRef another_user_ref)
```

Remarks

Discards all tokens for ANOTHER user (i.e. not caller). This is typically initiated by an administrator to 'reset' all tokens for a user.

This requires the caller to be authorized for this operation.

Arguments

arguments	type	Description
Ctx	Context	The context of the call. *subject* is ignored.
another_user_ref	ActorRef	The user for which tokens are discarded.
Return		

LoginByToken

Syntax

```
AuthenticationReply LoginByToken(Context principal_ctx, string token)
```

Remarks

Perform step 5 of the ***PatientId*** login method (see above).

Arguments

Arguments	type	description
principal_ctx	Context	The context of the principal. *subject* is ignored. *sessionKey* is ignored
token	string	The token used for login.
return		
	AuthenticationReply	Logins status and session key.

Logout

Syntax

```
void Logout(Context principal_ctx)
```

Remarks

Log out the principal from the current session.
The token will still be valid.

Arguments

arguments	type	description
principal_ctx	Context	The context of the principal. subject is ignored.
return		

IsLoggedIn

Syntax

```
bool IsLoggedIn(Context principal_ctx)
```

Remarks

Checks if the session is still valid.

Arguments

arguments	type	Description
principal_ctx	Context	The context of the principal. subject is ignored.
return		
	bool	True if valid.

3.4 Code

3.4.1 Summary

The Code module manages codeable information, also knowns as codeable concepts. This is basically a textual description of a code. One code may have different corresponding codeable concepts depending on the language used (as specified in the Context-variable).

Notable is that the Code datatype contains both the code itself (string) and the id of the codesystem (string).

3.4.2 Operations

Lookup

Syntax

```
CodeableConcept Lookup(Context ctx, Code code)
```

Remarks

Looks up a `CodeableConcept for a specified code.`

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
code	Code	The code to lookup. This also contains the code system.
return		
-	`CodeableConcept`	The corresponding codeable concept in the appropriate language or an empty (not null) item if not found.

Save

Syntax

```
void Save(Context ctx, CodeableConcept concept)
```

Remarks

Save a new or updated CodeableConcept.

Not all code systems will accept updates.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
concept	CodeableConcept	The CodeableConcept to save. Approprite code system must be specified in concept.coding.system Approprite language must be specified by concept.language
return		

GetCodeSystem

Syntax

```
CodeSystem GetCodeSystem(Context ctx, string system)
```

Remarks

Get a copy of an entire code system, i.e. all codes and texts for a specific language.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
`system`	string	Id for the code system to get.
return		
-	`CodeSystem`	The code system. If the specified language is not found, the default language for that code system is used.

3.4.3 Resources

Code

The *Code* type corresponds to the FHIR datatype *Coding*. See https://www.hl7.org/fhir/datatypes.html#Coding.

It is a representation of a defined concept using a symbol from a defined "code system".

attribute	type	code system	card	description
code	string	-	1	The code in the code system
System	string	-	1	Identity of the terminology system

CodeSystem

The CodeSystem datatype is used to declare the existence of and describe a code system or code system supplement and its key properties, and optionally define a part or all of its content. Type. This corresponds to the FHIR resource CodeSystem. See https://www.hl7.org/fhir/codesystem.html#codesystem.

attribute	type	code system	card	description
name	string	-	1	The id (name) of the codesystem.
concepts	CodeableConcept		*	The codeable concepts in the code system.

3.5 PatientRegister

3.5.1 Summary

The PatientRegister manages the general information about the demographic attributes for the ***patients***.

Discalimer: *In this context the term patient is used equivalent to individual. So, a patient in HOPE doesn't have to be a patient in the sense that it is ill or subject for care, just that it is the individual for which data is processed in contrast to other actors such as practitioners and family members.*

It goes without saying that the patient object is paramount in all healthcare systems. This is applicable from a technical perspective as well as from a legal perspective since every medical record must, by law, refer to a specific patient.

That is why all operations in HOPE Databroker that refer to a patient require the patient reference to be specified (in *Context.subject*) to be specified. This is also why the *Patient* must exist in the *PatientRegister* before any such operation is called (otherwise, there wouldn't be any patient reference).

Note: As mentioned in the *Abstract identifiers* chapter, the *Patient* object is referred to by an abstract GUID and not by the `identifier`-attribute (e.g. *personnummer*). This is to make sure that the references are never changed (even though the *personnummer* might change).

3.5.2 Operations

Save

Syntax

```
Patient Save(Context ctx, Patient patient)
```

Remarks

Save a new or modified patient object.

The *identifier*-attribute is checked to be valid. See *ValidateIdentifier.*

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
patient	Patient	The new patient object
return		
	Patient	The new patient object, including read-only attributes.

Lookup

Syntax

```
Patient Lookup(Context ctx, PatientRef patient_ref)
```

Remarks

Lookup a patient object by its reference.

Arguments

arguments	type	Description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
patient_ref	PatientRef	The patient reference
return		
	Patient	The patient object, including read-only attributes, or null if not found.

LookupPatients

Syntax

```
Patient[] LookupPatients(Context ctx, PatientRef[] patients)
```

Remarks

Lookup multiple patients at the same time.
This is semantically the same as Lookup, but is beneficial for performance.

Arguments

arguments	Type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
`patients`	`PatientRef[]`	An array of the patient references to lookup
return		
	`Patient[]`	An array of the patient looked up. The size of the array is guaranteed to be same as the patients-argument. Patients not found will be null in the array.

LookupByIdentifier

Syntax

```
Patient LookupByIdentifier(Context ctx, Identifier patient_identifier)
```

Remarks

Lookup a patient by its external identifier, e.g. *personnummer*.

Notable is that the external identifier is transformed into a normalized format before lookup, whenever possible.
So, if *personnummer* is used as identifier format (as defined by settings) , 20121212-1212, 201212121212 and 121212-1212 are all transformed into 201212121212 by the operation.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
`patient_identifier`	`Identifier`	External identifier, e.g. 201212121212 (*personnummer*)
return		
	`Patient`	The patient object, including read-only attributes, or null if not found.

RefreshReferences

Syntax

```
PatientRef[] RefreshReferences(Context ctx, PatientRef[] patient_refs)
```

Remarks

All Ref-objects contain a *title*-attribute which is intended for some short descriptive information about the referenced object. The intention is to be able to use this in a UI without having to lookup the entire referenced object (for performance). A typical example is an Activity-object refers to a patient. In this case *PatientRef.title* would by default hold the patient's name.

But what if the name changes? This will not propagate automatically to the title-attributes in all Activity-objects for that patient.

So, to facilitate for applications that require the title-attribute to be correct, this operation offers just that, to refresh the reference only.

Compared to *LookupPatients,* which looks up the entire patient objects, this operation has better performance.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
`patient_refs`	`PatientRef[]`	An array of patient references to refresh.
return		
	`PatientRef[]`	Refreshed patient references.

Remove

Syntax

```
void Remove(Context ctx, PatientRef patient_ref)
```

Remarks

Remove the specified patient from the database.

WARNING: A removed patient cannot be restored and associated data in other modules will become inaccessible.

Consider setting status to *inactive* instead. (see *SetStatus*)

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
`patient_ref`	`PatientRef`	The patient to remove.
return		

SetStatus

Syntax

```
void SetStatus(Context ctx, PatientRef patient_ref, PatientRegisterStatus status)
```

Remarks

Sets the status of the patient to one of:

active	(default) to indicate the patient is currently being processed.
inactive	to indicate that this patient should not be processed further. However, all data associated with the patient is kept intact.

The status can be changed any number of times in any direction.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
patient_ref	PatientRef	The patient.
status	PatientRegisterStatus	New status value.
return		

Search2

Syntax

```
Patient[] Search2(Context ctx, PatientFilter filter, SortDescriptor[] sort_by,

int max_cnt, int start_idx)
```

Remarks

Search and sort by criteria's specified by a filter.

Filtering

Filtering is done by providing a *PatientFilter*-object. Notable is that the *PatientFilter* class is a base class and that the dynamic type can be a derived class (as long as it is supported).

In the simple case the base class *PatientFilter* is used to search patients by a string that is found in the family name, given name or id. The comparison is case insensitive.

In the more complex case, using *ExPatientFilter,* the search is extended to the attributes (columns) and their corresponding values that are specified in the ColumnDescriptor-attribute. The match between individual columns is aggregated by logical AND.

Matches for the attributes in the *filter* are aggregated by logical AND. I.e. both the condition specified by *name_or_id* and by *status* must be true for the patient to be returned.

Sorting

The *sort_by* argument specifies the attributes (columns) to sort by and if sort order should be Ascending or Descending. A list of columns can be specified representing primary column, secondary column etc. by their index in the list.

Note 2: In the current implementation, all sorting is done by the *string* value.

Pagination

The total number of patients that match the specified filter can be quite large and for performance reasons it is necessary to limit the number actually returned from a single call. For this reason, the operation supports pagination by the use of the arguments *max_cnt* and *start_idx*.

This is also most likely how an application will handle this from a UI-perspective. Typically setting *max_cnt* to the number of patients that fit into one page and allowing the user to go to *previous* or *next* page resulting in calculating the appropriate value for *start_idx.*

Note 1: Pagination is done *after* sorting.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
`filter`	`PatientFilter`	Filter condition (see below)
`sort_by`	`SortDescriptor[]`	Specifies sorting of the returned patients. See above.
`max_cnt`	`int`	Max number of patients to be returned from a search. Note: For performance reasons, there might be a default limitation to the *max_cnt*-argument. Consult the provider for this.
`start_idx`	`int`	Zero-based index of first item to return. So, for the first call for a specific filter this is usually 0. For subsequent calls if during iterating through a number of pages, this is the total number of patients yet retrieved.
return		
	`Patient[]`	Ann array of patients matching the filter conditions.

PatientFilter class

attribute	type	code system	card	description
`name_or_id`	string		1	String that is matched with family name, given name and id. Field matches are aggregated by logical OR.
`status`	`PatientRegisterStatus`		1	Specifies if only *active patients*, only *inactive* or both (*undefined*) should be returned.

ExPatientFilter	Extended PatientFilter			
columns	ColumnDescriptor		*	Specifies attributes (columns) and the corresponding value that should match. Column matches are aggregated by logical AND. Only the *id* and *text*-attributes are used for searching.

ValidateIdentifier

Syntax

```
ValidateIdentifierResult

ValidateIdentifier(Context ctx, Identifier patient_identifier)
```

Remarks

Validates a patient identifier, e.g. a *personnummer*, for the following
a) not null
b) not duplicate in the database
c) has a valid format (if the *PatientIdFormat* is configured to "*Personnummer*").

It also returns transforms the identifier to the normalized computer format (returned in the *identifier*-attribute) and the normalized display-format(returned in the *identifierDisplay*-attribute)

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
patient_identifier	Identifier	The identifier-object.
return		
	ValidateIdentifierResult	Validation result. This is derived from *OperationResult* so *major*, *minor* and *description* holds the status of the validation. For the result of the transformation, see below.

ValidateIdentifierResult : OperationResult

attribute	type	code system	card	description
identifier	Identifier		1	Computer format (e.g. CCYYMMDDNNNN) - no hyphen
identifierDisplay	string		1	Display format (e.g. CCYYMMDD-NNNN)

ResolveUrl

Not applicable

GetExtension

Syntax

```
string GetExtension(Context ctx, Ref obj_ref, string key)
```

Remarks

Retrieves the specified extension for a patient.
Extensions offer a possibility to add “custom attributes” to an object in the FHIR-model.
See `https://www.hl7.org/fhir/extensibility.html#Extension`

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
obj_ref	Ref	The patient reference.
key	string	The name (key) of the extension. Case sensitive.
return		
	string	The value of the extension or null if not found.

GetExtensions

Syntax

```
NameValuePair[] GetExtensions(Context ctx, Ref obj_ref, string key_prefix)
```

Remarks

Get a number of, or all, extensions for a patient.
See remarks for *GetExtension*.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
`obj_ref`	`Ref`	The patient reference.
`key_prefix`	string	The prefix (start of) the extension name. Case sensitive. Use null to get all extensions.
return		
	`NameValuePair[]`	All matching extension in an array of Name-Value pairs.

SetExtension

Syntax

```
void SetExtension(Context ctx, Ref obj_ref, string key, string value)
```

Remarks

Set the value of an extension for a patient.

See remarks for *GetExtension*.

Arguments

arguments	type	description
ctx	Context	The context of the call, including references to the account, the patient and the logged in user. (see above)
obj_ref	Ref	The patient reference.
key	string	The name (key) of the extension. Case sensitive.
value	String	The new value for the extension.
return		