

Persistency Management guide

Persistency management is the term used in Blueriq to address everything that has to do with persisting (or saving, storing) information and later on retrieving, displaying, updating and deleting this information.

In many systems, persistency management is all about databases and SQL-statements. In Blueriq, it is not. We deliberately made the technical way in which data is stored and retrieved not the business engineer's concern, let alone the end user's concern.

Instead, Blueriq provides an intuitive canvas where a business engineer can design what information to persist. Furthermore, easy-to-use services to create, read, update and delete information and a service to display information objects are provided. What happens behind the scenes is, as mentioned before, not of interest to the business engineer.

This guide helps you to understand the big picture and the concepts and gives you best practices. Details can be found in the help files of the individual components. See also this page to understand how Blueriq fits into [Gartner's Pace-Layered Application Layers](#).

In this design guide

- [Terminology](#)
- [Aggregates](#)
- [Aggregate definition](#)
- [User set versus system set](#)
- [Precise aggregate definition](#)
- [Design considerations](#)
- [Working with aggregates](#)
- [Versioning](#)
- [Searching for aggregates](#)
- [References between aggregates](#)
- [Traceability](#)
- [Authorization](#)
- [Migration](#)
- [See also](#)

Experienced Business engineers might use the **Dossier Manager** ([Dossier plugin](#)) for Persistency Management. The functionality described in this design guide is aimed to replace this Dossier Manager.

See [How to migrate dossiers to aggregates](#) on how to migrate dossiers to aggregates.

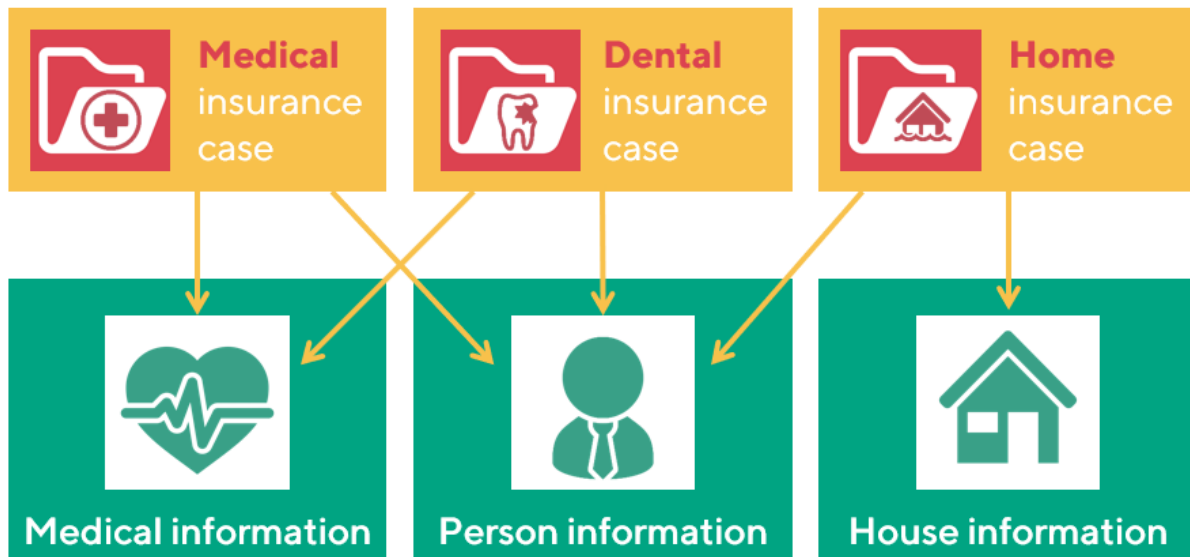
Terminology

The following terms play a role within persistency management in Blueriq and will be discussed in this design guide:

- Aggregate definition
- Entities, attributes and relations
- Create, read, update and delete aggregates
- Lists of aggregates
- Standard metadata
- Custom metadata
- Versioning
- Search for aggregates
- Authorization
- References

Aggregates

Organizations usually offer more than one product or service to their customers. All these products and services have their own cases, so multiple cases can be running concerning the same customer. To get a good overview of a customer there is a need for shared storage of information, which is shared among the several cases. See the example below.



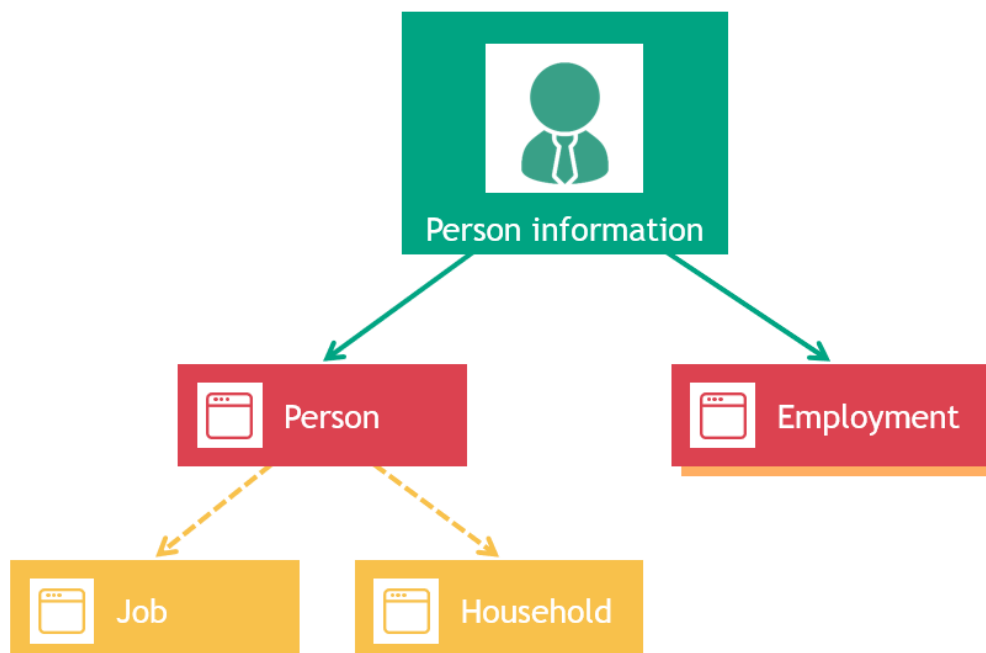
Shown above are three different cases (in grey) that are all part of a single business model of a fictitious insurance company. The first case concerns a medical insurance, the second a dental insurance and the third a home insurance. These cases use and share information (in blue). The medical and dental case both use the medical information and information about persons, whereas the home insurance makes use of the person information and information about houses. These information objects are called **aggregates** in Blueriq and will typically exist of one or more entities with their endogenous relations.

The reason that different aggregates can be created has to do with design, reuse as well as performance. Housing information is usually not needed deciding about a dental insurance. On the other hand, it is not wanted that a medical insurance case and a dental insurance case each use their own medical information, as they both concern the medical information of the same customer. Finally, loading unnecessary information has a negative impact on performance.

Aggregate definition

Aggregates are used to combine information that belongs together, and should be stored and loaded together. In Blueriq, information that belongs together is modeled in an entity, or as multiple entities with relations between them. An aggregate is the selection of parts of your domain that can be stored and retrieved together in a single action, and consists of one or more entities. When an entity is added to an aggregate definition, all entities that are connected to the entity are part of the aggregate as well. It is also possible to create a more [precise aggregate definition](#), excluding certain relations and therefore connected entities.

Shown below is the design of the aggregate Person information, in which the aggregate is blue, and the entities that are defined within the aggregate are pink.



Since it suffices to declare the main entities in a domain model that are to be part of the aggregate, the design of the aggregate definition of Person only consists of the entities Person and Employment. The entity Person has relations with the entities Job and Household, so whenever a person is stored, by default his or her jobs are also stored, as well as the person's household situation. This principle can be extended to multiple levels deep, so if a Job would have a relation to yet another entity, instances of that entity in this relation would also be stored. See also [Aggregate design](#).

Only the To-relations are followed automatically, not the backward (reverse) relations.

When storing such an aggregate, metadata is stored as well. This includes standard metadata, such as when the aggregate has been created and who created it, but also custom metadata that the business engineer can define on the aggregate definition by using expressions. This custom metadata can be seen as identifying attributes for an aggregate, but are not necessarily unique. In the example above, the Person aggregate could have the following custom metadata attributes: `FirstName`, `LastName`, `SocialSecurityNumber`, `CurrentJob`, `HouseholdSize`. As you see, custom metadata for an aggregate can originate from different entities, or even be a complex expressions, for example `NumberOfJobs` is calculated as `SIZE(Person.HasJobs)`.

It is even possible to define custom metadata that does not originate from entities of the aggregate definition at all, since custom metadata consists of an expression. This is not advisable in general, as there is no guarantee that the information needed to fill this attribute is available when reading and updated an aggregate entry at a later point in time.

Only the (custom) metadata is available in lists, as this information is stored in an efficient manner. When a new aggregate entry is created or updated, the (custom) metadata is updated as well.

Please think carefully about the choices for your metadata attributes. If you at a later point in time want to add a different metadata attribute, it will be empty for all existing aggregate entries. Once such an existing entry is updated, the new metadata field is going to be filled.

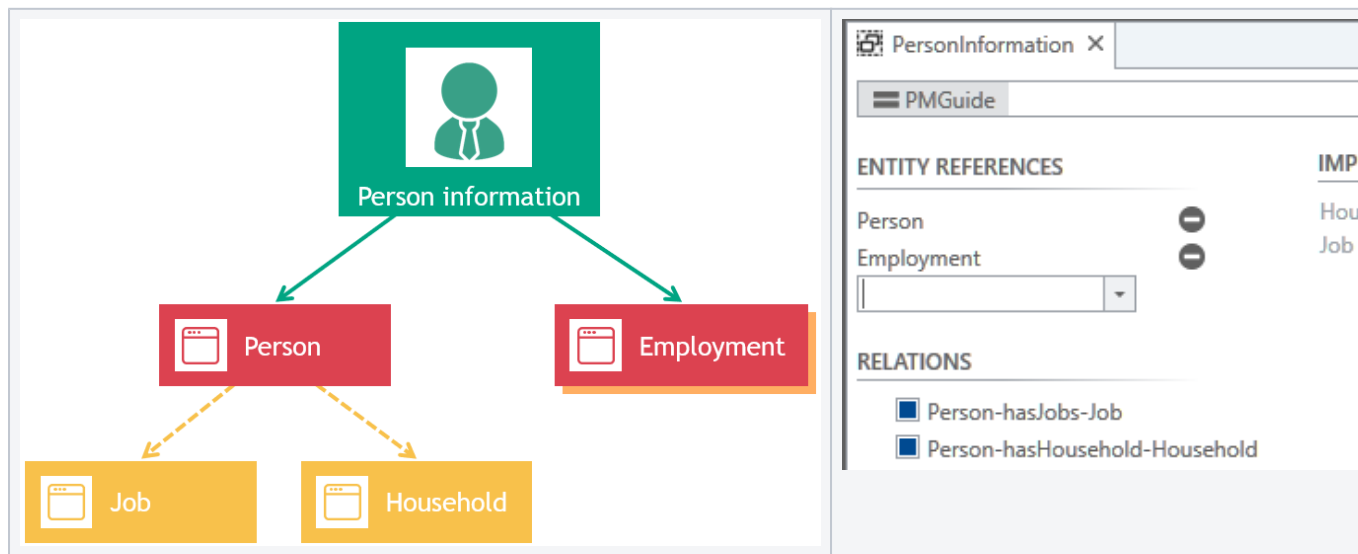
User set versus system set

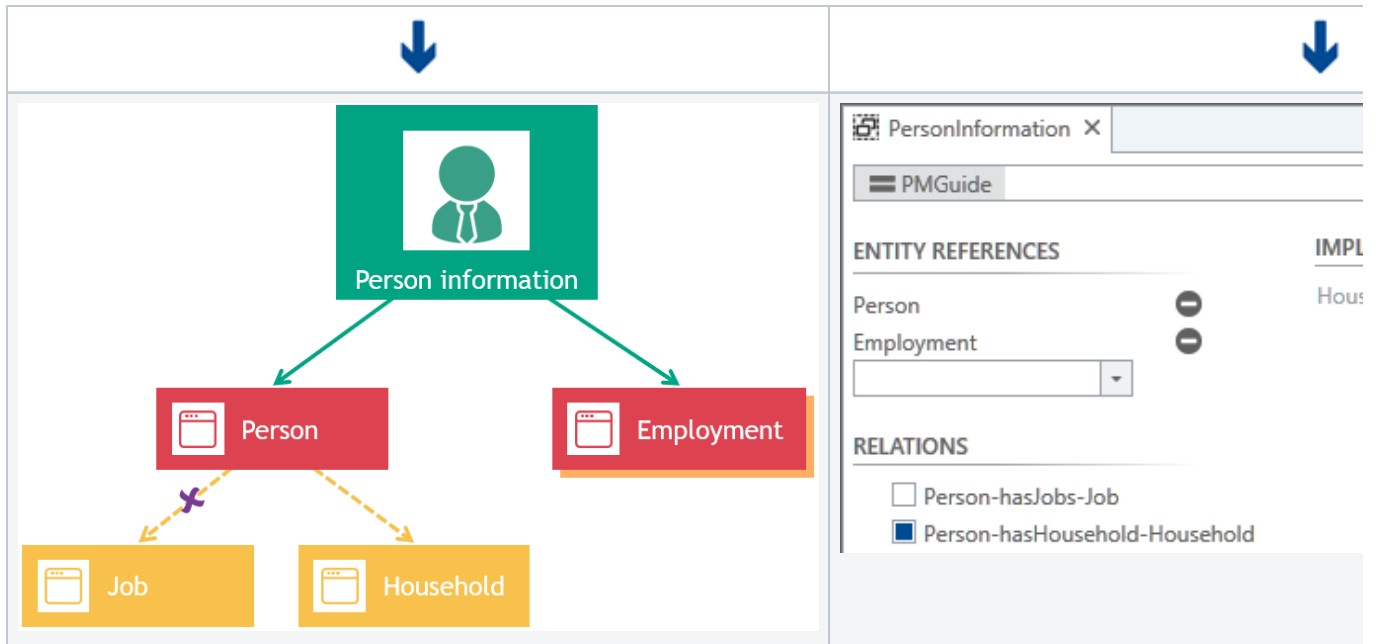
When an aggregate is stored - meaning created or updated - all user set attributes of all entities are stored, as well as all relations between the entities within the aggregate. System set attributes are not stored. When reading an aggregate, the Blueiriq inference engine will re-infer all these system set attributes.

If this not desired - and system set attributes should be stored when creating or updating an aggregate and not re-inferred when reading the aggregate - the business engineer should be sure to make those attributes user set before creating or updating the aggregate.

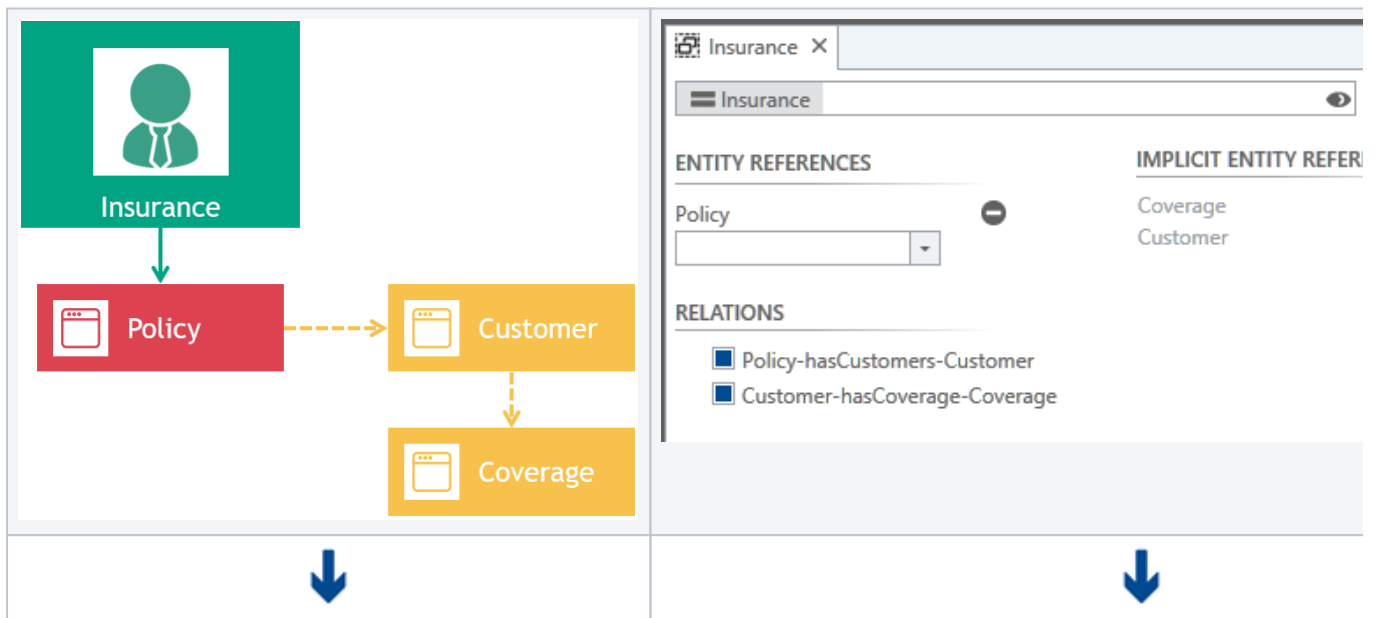
Precise aggregate definition

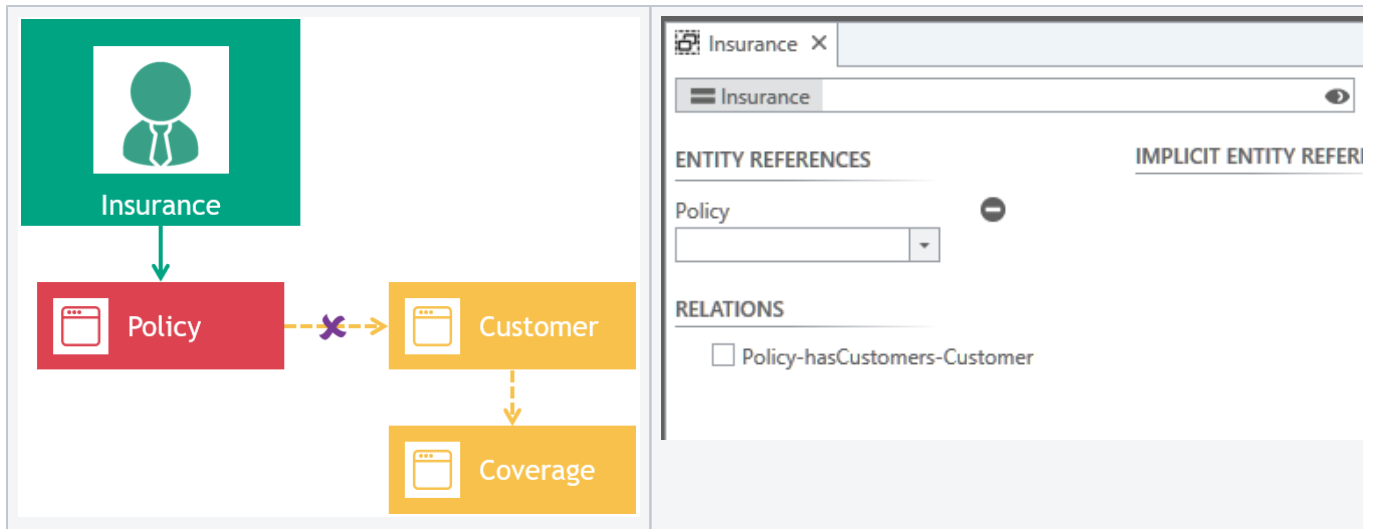
As mentioned before, by default all relations of an entity (and subsequently all connected entities) are part of the aggregate definition for that entity. This is not always wanted, however. Therefore it is possible to exclude relations from the aggregate definition, resulting in the exclusion of the connected entities as well. See the image below, where the relation `hasJobs` from the entity `Person` to the entity `Job` is excluded from the aggregate definition. The entity `Job` is then automatically removed from the aggregate definition.





Removing relations from an aggregate definitions can have cascading impact. If a relation at a "high" level is removed from an aggregate definition, its connected entities are removed from that definition as well. This means that the relations connected to these removed entities are removed from the aggregate definition also, including their connected entities etc. This is shown below, where originally the entities Policy, Customer and Coverage are part of the Insurance aggregate definition. the definition only consists of the entity Policy. After removing the relation Policy.hasCustomers from the aggregate definition, the entity Customer is removed from the aggregate definition as well. Because of the fact that the entity Customer is not part of the aggregate definition anymore, its relation Customer.hasCoverage to the entity Coverage and the entity Coverage itself are removed from the aggregate definition.

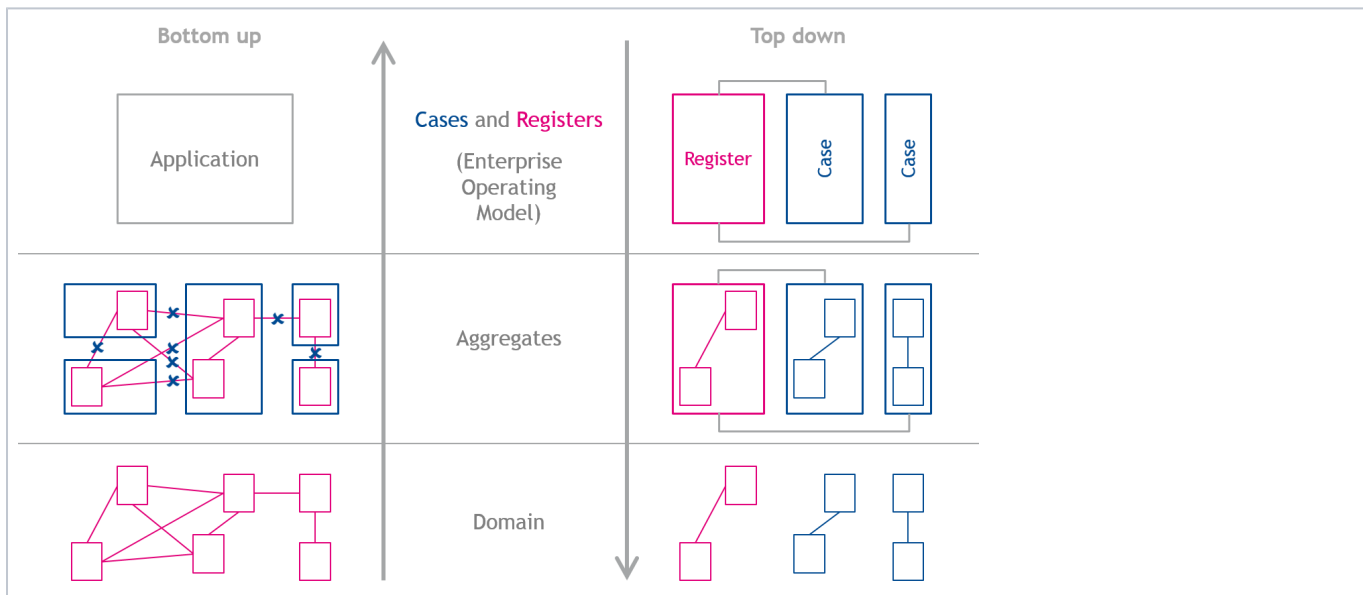




Design considerations

In any field of design - be it art, industry or IT - there's always the question whether a design should be made top down or bottom up. In top down modeling, the starting point is the complete solution, which is broken down in a few iterations into smaller elements, whereas bottom up design starts with creating all building blocks, which are then assembled in iterations and then result in the intended solution.

Historically, Blueriq is a product that is very suitable for bottom up design, but top down is also possible. When modeling in the field of dynamic case management and using aggregates, a top down approach suits better than a bottom up approach. This will be discussed here.



Top down design

Designing Blueriq applications top down means that at first, (amongst others) cases and registrations are identified. Both a registration as well as a case can be seen as an aggregate, so designing aggregates is easy: for each identified case and each identified registration, an aggregate can be declared. In some situations, registrations are not native, so a web service might suit better than an aggregate. The final step, designing the domain is also not that difficult: we just have to fill up the aggregates, which means deciding which entities are part of which aggregate. See the right side of the diagram above to illustrate the top down design.

Bottom up design

Designing an application bottom up starts with meticulously designing the domain, with all its entities and relations. When designing aggregates, some relations need to be excluded from the aggregate design, otherwise the aggregates contain too much data. After designing the aggregates, a separate design has to be made which contains the intended application. See the left side of the diagram above to illustrate the top down design.













Although both design strategies (top down and bottom up) are supported by Blueriq, designing a DCM (dynamic case management) solution is preferably done top down.

Working with aggregates

In order for an end user to work with aggregates in cases, it must be possible that these aggregates are displayed in a list. This could enable the knowledge worker to look for a known person in the system if this person contacts the call center. This list functionality is similar to the list functionality that is already available for instances, tasks and cases and is implemented as a Blueiriq container.

To let the business engineer create meaningful aggregate lists, custom metadata attributes can be added to the aggregate definition and then used as column in the aggregate list. Below is an example of such a list, which illustrates sorting, filtering, custom metadata and paging. For more information on the aggregate list see [Container type: AQ_Aggregate_List](#).

List of person aggregates

ID	Version	Full name	Age	Nr of jobs	
251	1	Tom Barmann	45	1	  
251	2	Tom Barman	45	1	  
251	3	Tom Barman	45	2	  
256	1	Matt Berninger	43	1	  

Displaying entries 1 - 4 of 4

Page 1 of 1

By letting the business engineer create different aggregate lists for different purposes, the end user will work with only the aggregates that are relevant for him. Since only metadata from these aggregates is retrieved from the database while displaying them - and not all data completely - this will boost performance.

Aggregates can be created, read, updated and deleted. For these specific actions, services are available in Blueiriq.

- Aggregate create service: [Service call type: AQ_Aggregate_Create](#)
- Aggregate read service: [Service call type: AQ_Aggregate_Read](#)
- Aggregate update service: [Service call type: AQ_Aggregate_Update](#)
- Aggregate delete service: [Service call type: AQ_Aggregate_Delete](#)

With the (deprecated) dossier manager it is possible to clear the profile with one service call (init). Persistency management functionality does not provide such a service, since such a service is not part of persistency management but instance management. For now, a flow has to be modeled that clears the profile, if needed.

Versioning

As shown in the aggregate list in the previous section, it is possible to specify versioning for aggregates. This means that when creating an aggregate, a version number is stored along with it. When updating an aggregate, it will result in a new aggregate entry with an increased version number. When reading an aggregate, a specific version can be specified. This means that it is possible to load an older version of an aggregate into the profile. One can also decide to just alter the latest version without creating a new version. The same goes for deleting an aggregate, you can choose to delete an aggregate with all its versions, or only a specific version. It is advised to use a control entity to store the aggregate Ids as well as the version Ids.

When using versioned aggregates, a version Id can be specified as input when reading, updating or deleting an aggregate entry. Creating an aggregate entry does not require an aggregate version Id, since this will always be Id 1. A version Id as output can be expected when an aggregate entry is created, read or updated. When an aggregate entry with a specific version is deleted, it is not logical to expect this version Id as output (since it was input), neither is it logical to expect an aggregate version Id as output when deleting all versions of a specific aggregate entry.

When reading a versioned aggregate entry, it is possible to skip the version Id as input, but specify it as output. This means that the aggregate read service will read the latest version of the aggregate entry and tell us which version this actually is. When deleting a versioned aggregate entry, it is possible to skip the version Id. This means that the delete service will delete all versions of this aggregate entry.



There are three versioning strategies when designing aggregates, which can all be modeled in Blueqriq:

	No versioning Updating an aggregate entry means that the original aggregate entry is overwritten. There is always one single version of the aggregate entry present in the database. Reading a non-versioned aggregate entry is trivial: it will read "the" aggregate entry, since there is only one. You should use this strategy if the past state of the aggregate is not important.
	Versioning all Updating an aggregate entry means that a new version of the entry is stored alongside all previous versions. When deleting such a versioned aggregate entry, it is necessary to specify which specific version(s) to delete. Reading an aggregate entry also means that it has to be specified which specific version(s) to read. Beware that the number of versions for an aggregate entry can grow out of control with this strategy. Also, think about the design of your aggregate list when using full versioning. You should use this strategy if it is important what the state of the aggregate in the past was. For example the employment status of a customer at the moment of the start of the application. Any changes during the processing of the application should not be considered.
	Versioning latest This strategy is based on the previous strategy, but with some limitations for the end user. Versioning is still in place, so updating an aggregate entry will create a new version of the entry, but in this strategy reading will always return the latest aggregate entry version. Deleting an aggregate entry with this strategy means deleting all versions. Also be aware of the fact that the number of versions for an aggregate entry can grow out of control. You should use this strategy if you always want to work with the latest version, but you still want to keep a history of the aggregate. A good example is the address of a customer. You always want to use the newest address for correspondence, but keeping the old addresses may be useful.

These strategies are not enforced, and it is up to the Business Engineer to choose one and follow it during modeling. The Business Engineer has all the freedom to choose the correct action for every situation.

Searching for aggregates

	In most situations, opening and displaying a specific aggregate can be achieved with a single service (Service call type: AQ_Aggregate_Read), because the model is aware of the aggregate ID.
	If this is not the case (we sometimes do not know the ID), a list of aggregates can be displayed (Container type: AQ_Aggregate_List). Letting the end user filter the metadata of the aggregate list to find the intended aggregate is then a very efficient method to find the desired aggregate.
	Another strategy may be to prefill the metadata filters on an aggregate list (Container type: AQ_Aggregate_List), so that all aggregates that meet the criteria can be shown at first. The end user can broaden or widen the filters if so desired and after finding the intended aggregate, select it. This must also be done even when there is only one aggregate in the aggregate list.
	There are situations however, where presenting a list to the end user is not wanted. The aggregate must be opened instantaneously, but we do not know the ID. We can use the aggregate search service (Service call type: AQ_Aggregate_Search) to search for the aggregate (based on its metadata) and after finding it, open it. If such a search should result in multiple aggregates, an aggregate list, with all available filter-options, can still be displayed (Container type: AQ_Aggregate_List).

See [How to search for aggregates](#) on how to model different scenarios when searching for aggregates.

References between aggregates

In many situations you want that different aggregate entries refer to another. For example should the case aggregate refer to the person that this case is about.

We assume that there is no relation between the case entity and the Person entity, since Person would be stored together with the case and this is not wanted.

You should model an attribute at the case entity which stores the aggregate Id of the Person aggregate. When loading the case into memory, first the case aggregate is loaded. This retrieves the person aggregate Id, which can be loaded afterwards by a second service. When initially creating the aggregates the order has to be reversed. For updating, the ordering does not matter, as the Id does not change.

When modeling such an attribute that holds a reference to another aggregate, it is advised to check the 'Act as reference' checkbox and indicate that this attribute is a reference to an aggregate. This checkbox is not only useful in [Search in structured and unstructured information](#), but also for a clear domain model.

Traceability

When an aggregate is created, updated or deleted an entry in the trace database is made. See [Aggregate traceability](#).

Authorization

There currently is no authorization on aggregate level. This means that you are responsible for prohibiting the end user to load certain aggregates by choosing adequate authorization on a page or at container level.

Migration

When your application changes over time, the migration of data becomes an important topic. Aggregates help you with this, but cannot solve all your problems.

Lazy migration

For complex scenarios it is advised to create your own database scripts for migration purposes. Database migration scripts generally do everything in a lazy manner, so that not the complete database is updating when the server starts. A conversion hook for lazy migration is available, see [Conversion API](#) and [How to lazily convert aggregates to a new version](#).

A new attribute is added

When a new attribute is added to the domain, all already stored aggregates do not have this attribute. In this situation, it is possible to load existing aggregates, and the value for this new attribute is simply set to unknown.

An attribute is deleted

When loading an existing aggregate that has an attribute that has been deleted from the aggregate definition in the mean time, then this attribute is simply ignored. Upon the next update action, the old aggregate entry is overwritten and the information of the deleted attribute is lost.

An attribute is renamed

Renaming an attribute is regarded as deleting the old attribute and adding a new one. Existing information in the old attribute is lost.

Metadata

For metadata the same principles hold as for domain attributes. A special case is a change in the expression of a custom metadata attribute. Here also the lazy principles applies. When the expression for a custom metadata field changes, the value in the database is updated the next time an update action is performed for the aggregate entry.

Changing data types

It is possible to change the data types of metadata or regular data. It is checked whether the new data type is compatible with the old data type. If so, a simple conversion is done. If not, the old data is ignored and the value is set to unknown.

Examples of possible data type changes

- Integer to Number
- Date to DateTime
- Currency to Percentage

Examples of impossible data type changes

- Number to Integer
- DateTime to Number
- String to Boolean

See also

How to identify business functions, cases and registers: [Identifying business functions, cases and registers](#)

A design guide about dynamic processes can be found here: [Designing dynamic processes](#)

How aggregates can be used to design cases, see [Designing cases using aggregates](#)

The aggregate example project can be found here: [Example project](#)

How to migrate dossiers to aggregates: [How to migrate dossiers to aggregates](#).

A template for dynamic case management can be found here: [DCM Foundation - v4 \[editor\]](#).

For more information on the relation between aggregate concepts, see

