Getting started with Case Modelling

Blueriq Dynamic Case Management has evolved into a new event-driven architecture, native to the Blueriq platform. This topic describes the different components needed to use the new architecture.

The new architecture is developed in the DCM 2.0 program, separately from the current way of modeling processes and Dynamic Case Management.

The DCM architecture is addressed as **Case-Modelling**, as opposed to **Process-Modelling** for the process-engine setup.

Contents

In the Blueriq DCM architecture overview and subpages all components and configuration needed for the new setup for Dynamic Case Management are described. Deployment considerations DCM describes how we currently view topics like scalability, security and backups in the new DCM architecture. A different way of hosting and facilitating Dashboards is explained in Blueriq DCM Dashboard. In the current phase, DCM Dashboarding is still in development, but it will become the main method to model Widgets and Widget flows in a DCM project.

Case-Modelling

The DCM setup is an architecture designed to support modelling Case-Types. There are lots of new features that differ from the currently known Process-Engine setup, tackling multiple aspects of DCM modelling. The following page can be used to help you getting started with the new architecture by explaining setup and the key characteristics of DCM.

Key Features

- Event-driven by default
- Functional and application decomposition
- Event transaction handling
- DCM Dashboard
- Auditability
- Failover

Event-driven by default

> **Gains**
>
> The end-user does not wait on system actions when they do not need to do so, resulting in less users affected by load on the system. This is in contrast to the old architecture, where load on the system will be noticeable by all users and in all actions simultaneously. This means that the new setup can be configured to handle higher loads than previously, since the executions of work on the system is more scheduled. There is also an added possibility to process automatic tasks or case events in parallel. In the current process-engine setup, process evaluation (and automatic task execution) is done inline, so the user has to wait until the process is finished, or waiting upon a next action.

The new DCM architecture is event-driven by default. This means that users will not wait on the results of tasks they are not dependent on, all actions that can be performed asynchronously are done so. This includes for example, a user completing his task or throwing a message event to a case. Other actions demanding direct response are still synchronous, like a user demanding a task to be started.

The new architecture is designed to keep the system working, even when the load on the system increases. When in the old situation users had to wait for longer during peak loads (for example on automatic tasks being performed right after finishing their manual task), the new setup will direct the user to a new page quickly, and finish working in the background. As system load increases, a queue of events can build up in the background leading to delays, but the system will keep working in a stable manner for longer.

The transformation to asynchronous work will also shorten the transactions of the system. In the process setup, the complete result of a task will be calculated in one transaction (for example performing a series of automatic tasks after a manual task), where in the new situations, this will lead to a series of events that can be processed whenever a Runtime is ready to.

The queue of all DCM events can be processed in parallel, leaving more room for scaling and parallel processing configuration. There is one or more Case Engine(s) handling case events, with the possibility to processing more than one event at the same time. Processing automatic tasks is often where the intensive tasks are processed (for example with long waiting times on external services). In the new architecture, there are configuration possibilities to process one or more concurrent automatic task on a Runtime. There is also a possibility to configure a Runtime to not process automatic tasks at all, so it can be focused on user-interaction only.

Task execution

Gains

There is more support throughout the platform for executing tasks. Data will be retrieved and sent back to the Case Engine, keeping the Case Engine responsible for the data management (of Blueriq data). No manual load and save actions are needed to be modelled. The case will only be released when the whole state is consistent again in the database. Every action on a case will be queued until the case is consistent again and only then will it be executed. This is in contrast to the current process-engine setup, where modelling patterns for loading and updating a case is necessary. Some maintenance modelling is needed to keep the data consistent.

Within Case-Modelling, starting a (manual) task is a synchronous action (the user wants to start his task immediately), but completing a task is asynchronous. Performing automatic tasks will always be asynchronous. Some effects of this behavior are explained below.

Order of relevant actions cannot be predetermined

All actions that the Case Engine deems ready to be executed (for example automatic tasks, expired timer, expired task, etc.) are queued as events. All events that eventually are on top of the queue will be executed, but the queue does not guarantee the order in which this this will be done. When constructing a model, the business engineer should consider that certain actions can become relevant at the same time. When this happens, they should never depend on any order.

Automatic tasks

Automatic tasks are not performed inline but pushed to a queue to be executed by any listening runtime. This can be the same runtime where end-users are working from, but also a dedicated runtime for performing automatic tasks (for example in another zone of the infrastructure). Automatic tasks will always be performed by the user “automatic-user”, which does not have any roles or authentication for external use. When system to system authentication is necessary during task execution, configuration can be added for specific service calls during task execution, for example by system-to-system authentication on the specific webservice calls.

Every automatic task must be modelled so that the implementation flow ends in an exit with an OK exit event. This guarantees that both the event and the task are deemed processed if the task is executed correctly in it's entirety. If an implementation flow of an automated task would end in a CANCEL event, the event itself will be deemed processed (since the corresponding flow has resulted in an exit event) while the process task will not (because a cancelled task is deemed like not having been executed at all in the process). This leads to a inconsistent state between event-queue and process, with no available actions to fix this. Error handling should therefor not be implemented through these model patterns within task flows, but should be left to event-processing with it's accompanying maintenance app. A task-flow that is unable to reach it's end is tracked through it's event and the maintenance app is able to perform actions on it. An automatic task should always be started with the intent to finish it correctly and the task implementation flow should reflect this.

Case locks

In essence, every tasks on a case loads the case-state, changes something, and updates the entire case-state as a whole. If tasks would be executed simultaneously, there could be a possibility that they interfere with one another (for example by trying to modify the same data), resulting in potential data loss. To avoid this, case locking is introduced in our event-driven landscape for all actions on a case. When multiple events are queued to be processed for the same case, only one event can access the case at the same time, locking the case for all others.

The execution of an event starts with a synchronous call to the Case Engine that starts a (system) task setting a case-lock, after which the case engine will only process a task completion event corresponding with the same task. All other events (for example an expired timer or an automatic task) will wait until the case is unlocked through this task completion event, after which the next event will be read from the queue. This means all events on a case will be executed in line.

Functional and application decomposition

Gains

By having a Runtime (user/task interaction), Case Engine (case actions), DCM Lists Service (handling lists) and DCM Maintenance App, the DCM toolbox is expanded.

The Maintenance App has more functionality for the maintenance users to keep the system running (doing less manual labor to investigate the current state of a case, since screens are provided to show case states). The components are more scalable than before, since user-interaction is only bound to the Runtime and case processing is done in a different component. Runtimes can be dedicated to manual task execution, or automatic task execution when desired, to better manage load.

The DCM architecture consists of multiple components working together to process cases. The highlights are described below. For a full overview, please check Blueriq DCM architecture overview.

The Case Engine

The Case Engine is responsible for maintaining a case. Everything that has to do with updating case-data, starting or processing tasks, starting or completing a case is done by the Case Engine. The Case Engine is a Blueriq component using a Blueriq model to function. The model is used to gather the process, the dossier and the metadata definitions.

When model changes are performed on process definition or data definition (dossier or metadata), please make sure to reload the model on the Case Engine

The DCM Lists Service

The lists service serves all lists, including the DCM_Caselist, DCM_Worklist and DCM_Tasklist. Data for the lists-service (the actual case information) is stored in a document-database separate from the key-value pairs in the SQL store that the Case Engine uses. All lists services are performed on this document-database, resulting in less load on our SQL store.

The Runtime

The Blueriq Runtime is the client to the Case Engine and in this role it facilitates all user-interaction with the system, for instance performing tasks and displaying case information. This component is the only component exposed to the end-user: lists, and the Case Engine data is always requested from the Runtime. For now, (case) authorization and any front-end is managed by the Runtime as well, marking all other components as internal.

The DCM-Maintenance-App

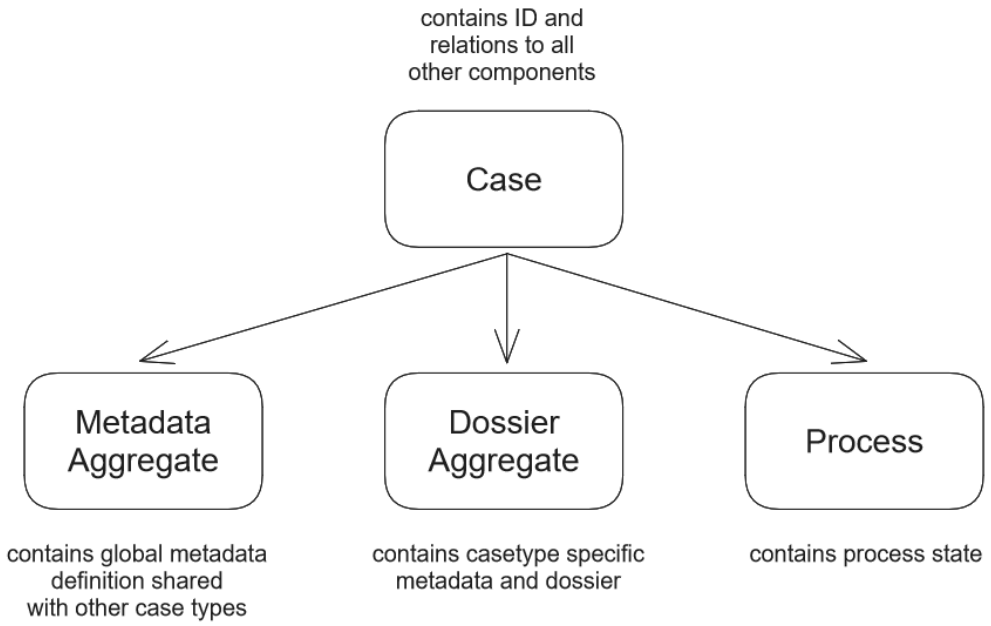
The Maintenance App is a dedicated application for a maintenance user and has the purpose to keep cases valid. Actions on cases are performed through events and issues with these events are maintained through the maintenance app. The maintenance app has some automatic behavior like automatic retries, but it also offers methods to manually do maintenance to the system. This is supported by overviews both on the entire system and for a specific case, to give insight on what is happing within the DCM system.

Case data management

Gains

All case data is stored and managed by the Case Engine. In the process-engine setup, multiple identifiers have a role in working with cases. In this case-modelling setup, a case is stored as a single case, with all references to other data objects stored in the case store. The Case Engine will manage the Blueriq case for you, making it easier to work with case-types in a Blueriq model. We've also improved the lists by storing them in a document database, in which indices can be used to optimize commonly used lists-queries. Optimizing lists was harder with the SQL-database since the structure demanded key-value pair searches on cases.

A case consists of three separate elements in the case-modelling setup, the metadata data (metadata aggregate), the dossier (dossier aggregate) and the process (process-sql-store). These datasets need to stay in sync with each other for these three sets make up a consistent case, meaning that one can not be updated without the other. In the former situation (pre case-modelling), the model was responsible for this, resulting in complex model patterns to do so. In the case-modelling setup however, the Case Engine has assumed this responsibility and it does this by making data storage of all three elements part of event (or task) execution within the Case Engine, instead of behavior derived from the model.



Metadata Aggregate

Metadata is a dataset that is shared between all case types, containing characteristics which exist for every case type (like the people involved in handling a case). This dataset gives the possibility to search through all cases regardless of case type and create (filtered) lists. The metadata, which is stored as an aggregate, is also responsible for maintaining links to any relevant data sources, like other aggregates, the process id and any other external data sources. In the case-modelling setup, all three data sets (metadata/dossier/process) will be created at case creation and the identifiers of the dossier and process will be added to the metadata during creation.

In order to keep the aggregate definition as generic as possible, a number of restrictions are added to the metadata aggregate:

- There is always one singleton, containing the Case_Metadata details
- All other multitons are stored in the aggregate implicitly by their relation to the singleton

The fields of the metadata in the DCM Foundation model are listed to become studio concepts in the future (like involvements, status, case-type, etc.).

Dossier Aggregate

The dossier aggregate contains the case-specific data and will be defined for each case type. The aggregate-type should be unique for each case-type. The metadata of this aggregate-type can be used to add case-type specific labels to the case, that can be used to show or filter lists.

Process profile

The process is always maintained in the case engine and saved in the process sql store. In the case modelling setup, the process is updated using data from the metadata and dossier aggregate, so keep in mind that any data in your task implementation that is not stored in the dossier aggregates, will not be used to update the process state.

A good practice is to keep the process state always completely derivable from rules and attributes in the case dossier. Since the process state can be hard to debug and restore, changing any rules will be easier when there is one source (dossier), which will lead to a process state.

Loading case data using aggregates

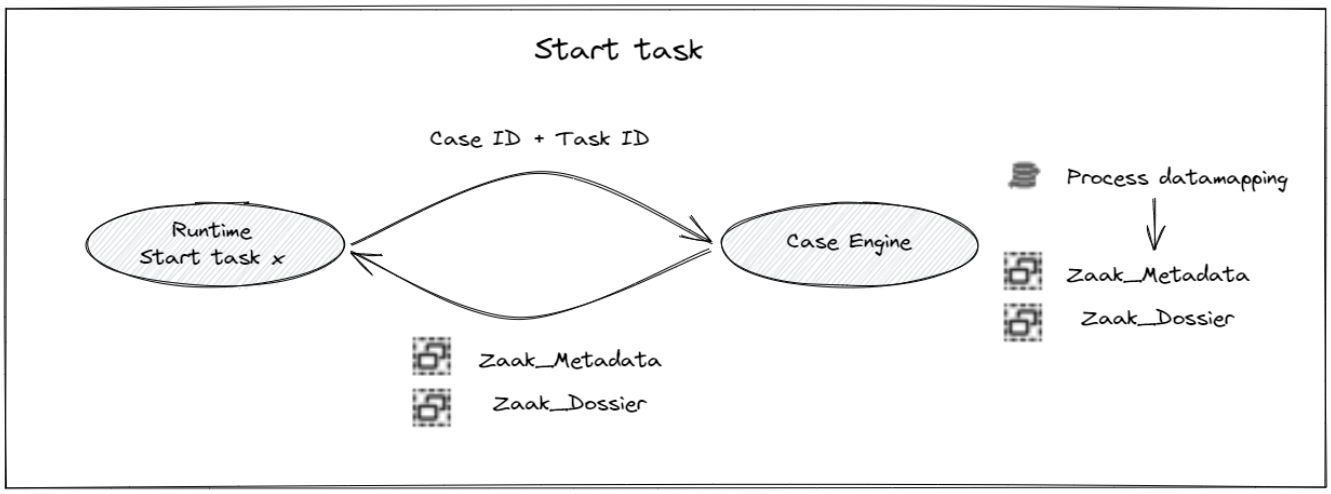
When a task is started, the case engine will gather all relevant information based on the task and other parameters set in the DCM_ExecuteTask service. The complete dataset will be loaded on the start of a task.

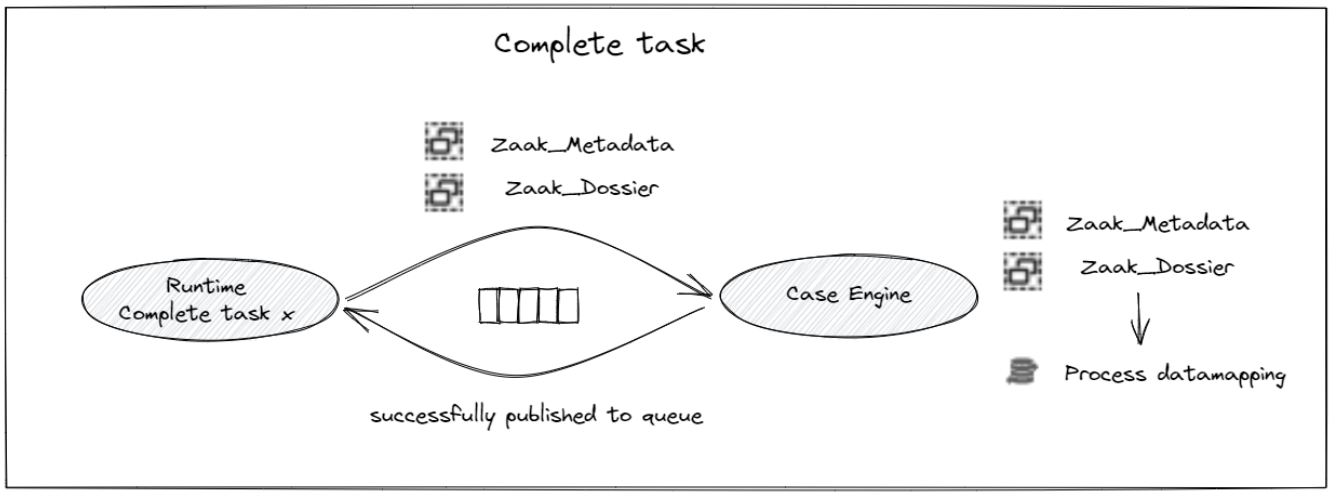
When the case is loaded in different contexts (for example to show the details of the case), the metadata aggregate should always be read first (using the aggregate ID or found by some characteristic known in the metadata). When the metadata is read, only then all references to the other data sources is known in the profile, after which others can be collected.

Loading/Updating case data when executing tasks

The process and aggregates will be updated internally in the case engine as a result of executing a task, so there is no need to model this in the task implementation. The system uses the aggregate definition as contract between a task and the case engine. This means that all data that is needed to map to the process profile should be available in the aggregate definition and the module containing all logic elements.

The mapping performed when loading a case will also be performed in the Case Engine. The aggregates will be loaded, mapping is performed, the aggregate data is then sent back to the Runtime and loaded in the task implementation. All process data that is needed in the task implementation, should be available in the aggregate definition (otherwise it will not be sent to the task implementation).





Case Archiving

When a case is finished (goal has been accomplished), the process serves no more purpose and is finished as well. This leads to the process being removed from the system. The case-dossier aggregates will be removed from the system as well. The comments and timeline records corresponding to the case will not be removed at this time.

Please make sure that one of the last steps in the case model is making sure the data is stored in a way the customer needs. **All data will be removed from the Blueriq system after the process is finished.**

Event transaction handling

Gains

The system is built to keep cases consistent when actions are performed on it. In the process-engine setup, models and maintenance actions were often build in model structures. The new setup offers a better support for keeping cases data-consistent over time. Actions will either succeed or fail. When an event fails to be processed, it can be automatically retried. After a configured number of retries, it will be presented to a maintenance user to fail.

The case modelling setup is designed in such a way to keep cases running correctly. All transactions on the system either succeed, or fail as a whole. The design follows the principle to get cases to a consistent state after the transaction has completed, called "eventual consistent".

Errors, or hiccups could always occur in a working environment. If a user-action fails, the system would tell the user to try the action again later by showing a message on the current page. However, for asynchronous communication, the system contains an automatic retry mechanism, so it can be configured that after a few minutes the message is processed again. This could restore errors that could have been resulted from high system load at the moment. When retrying the message fails for a configured number of times, then the message will be presented to a maintenance user.

As long as the transaction runs, the case will remain under case-lock until the message is processed successfully. Making sure that no other action can interfere before the transaction is finished.

In order to guarantee the system to be eventually consistent, a pattern has been implemented called the inbox/outbox pattern.

Outbox pattern

The outbox pattern is making sure that a set of actions will be committed atomically. which means everything is committed or nothing is. This can be especially challenging when a database and a message broker like RabbitMQ are involved.
If we send a message to the message broker during a database transaction, we cannot guarantee that the transaction will finish. The transaction could be rolled back for example and the message should not have been sent. This could result in the receiver being updated with events that according to the sender never happened.
If we send the messages after committing the database transaction the system could crash before the message has been sent, resulting in not updating the service where the message is should be received.
The outbox pattern saves the message to an outbox table instead of publishing it directly. This happens in the same transaction as the other database actions are performed . This means that either everything succeeds (and the event is committed to the outbox table) or everything fails and nothing is committed to the outbox table. When everything succeeds, the event resides in the outbox table and still needs to be sent. This happens in a different process which reads the outbox table and publishes events found there.
By using the outbox, when a task fails in the case engine, it can be completely repeated as no event will be published to a queue or saved to an outbox table. Only if everything succeeds, an event will be saved in the outbox table and it can be sent.
The outbox poller runs a separate thread that makes sure that messages saved in the outbox table are sent to the message broker. Only after the messages are sent, the records will be removed from the database, making sure that the data is never lost if something fails.
There is one catch with this pattern: The event that is read from the outbox table, published to a queue and then removed from the outbox table. If something goes wrong while updating the outbox table (removing the event) after the event is published, it will be published again the next time the publishing process starts.

Inbox pattern

To mitigate the problem that a message is sent twice, and thus also received twice, we also implemented an inbox pattern, which will make sure no message is executed more than once.
The inbox pattern uses an inbox table in which incoming events are stored. This happens in the same (database) transaction as which the other database related actions are performed. This means that when an action completes successfully, everything is saved in the database including the event in the inbox table. When something goes wrong, nothing is saved in the database. When an event arrives, the inbox table is checked if the event already has been processed. This is the case when the event is already contained in the inbox table. When it is, we know that the event already has been processed so it can be skipped. When it is not contained in the inbox table, we have not handled this event before (or we did try but something went wrong), so we can go ahead and handle the event.
Although the outbox pattern may publish the same message multiple times, by using the inbox pattern we can still guarantee that the same event will only be handled once by the receiver.

The outbox poller

The separate process that reads the outbox table is started in two ways. We listen to the transaction and if it's finished, we start the process for that case only and there is also a scheduled "poller" which reads from the database every 2 minutes by default. From version 16.7 onwards, the interval can be changed, see Outbox poller on how to change the interval. The poller makes sure that messages that failed to be published the first time are sent. Also, messages that do not contain a Case ID are published by the poller. If publishing all messages from the outbox (or multiple outbox tables in case of a multitenant environment) takes longer than the specified interval, a warning message will be added to the log. See the page mentioned above to change the interval if this message appears regularly.

Data storage

The outbox table only contains entries before they are published. The publishing process runs regularly and empties the outbox table, which means that the outbox table will usually only contain a few records or none at all. This is different for the inbox table. As every message* that is handled will be stored here, it can grow fast. Note that instead of the entire message, only the message identifier and creation date is stored in the table, but over time this can still grow large.

There is a point in time in which messages can be removed from the inbox table as they are no longer relevant to check for, but this point in time cannot be decided beforehand, as it completely relies on the business process that is implemented with the DCM system. Therefore the monitoring and emptying of the inbox tables is a project task.

DCM Dashboard

Gains

DCM Dashboard is a new way to model how and when Blueriq models are started in their own context, and how it's flowed from pages to other pages. This also enables sessions to have a single purpose, and staged in their own environment, making fail-over and session management more easy.

The DCM solution can be used with new Dashboard concepts. The DCM Dashboard consists of different new components, like the gateway (managing the user-session and authorisation) and the dashboard-service to interpret the dashboard models and making sure the browser loads the right widgets. More Dashboard about dashboard can be found at Flow Type: DCM Widget.

Auditability

Gains

Blueriq keeps track on who has seen a case in a list, opened the case and performed actions on a case. Especially logging if someone has opened the case, is an addition to Case-Modelling. The information can be viewed by an auditor.

The audit log is new, in addition to the trace and timeline components. Where the timeline is focused on hoe performed actions at the case, and the trace is focused on what has changed on a case, the auditlog is bound to who interacted with cases. The events are published on a queue, and can be stored in a database using Blueriq, or consumed by another application.

Failover

> **Gains**
>
> Failover used to be available only on forms applications (without nested sessions). In DCM it is possible run each widget in its own session, making it easier to scale the application and perform failover.

DCM Dashboarding, as part of Case-Modelling, ensures multiple widgets can be started simultaneously, constructing for example a case details page. Sessions are not nested within one Blueriq sessions anymore, but bound to a Blueriq Dashboard application. This application makes sure session dependencies are processed. It is possible to run each session on a different Runtime, it is even possible to run each Blueriq session update on a different Runtime, making it possible to upscale/downscale Blueriq Runtimes on the fly without losing sessions (and without sticky sessions)