Katholieke
Universiteit
Leuven

**Department of
Computer Science**

DOCUMENT PROCESSING

The complete architecture

Software Architecture (H09B5a and H07Z9a) – Part 2b

Jeroen Reinenbergh (r0460600)
Jonas Schouterden (r0260385)

Academic year 2014–2015

Contents

1 Introduction **3**

2 Overview **3**
2.1 Architectural decisions . . . 3
2.2 Discussion . . . 6

3 Client-server view (UML Component diagram) **6**
3.1 Main architectural decisions . . . 7
3.1.1 Av1a & Av2a: Notifying the appropriate operator within 1 minute . . . 7
3.1.2 Av1b: Storing the status of an individual job . . . 8
3.1.3 Av2b: Temporary storage and user notification upon PDSDB failure . . . 8
3.1.4 P2: Document lookups . . . 10

4 Decomposition view (UML Component diagram) **11**
4.1 DeliveryFunctionality . . . 11
4.2 DocumentDB . . . 12
4.3 DocumentGenerationManager . . . 12
4.4 DocumentStorageFunctionality . . . 13
4.5 JobManager . . . 14
4.6 LinkMappingFunctionality . . . 14
4.7 PDSDB . . . 15
4.8 UserFunctionality . . . 16

5 Deployment view (UML Deployment diagram) **17**

6 Scenarios **18**
6.1 Scenario 1 . . . 18
6.2 A registered recipient logs in . . . 18
6.3 A registered recipient logs out . . . 18
6.4 Updating a document template . . . 18
6.5 Verifying a session . . . 20
6.6 Initiating document processing . . . 21
6.7 Consult document in personal document store . . . 24
6.8 Delivering a document via the personal document store . . . 26

A Element catalog **33**
A.1 AuthenticationHandler . . . 33
A.2 BillingManager . . . 34
A.3 ChannelDispatcher . . . 34
A.4 CustomerOrganizationClient . . . 35
A.5 CustomerOrganizationFacade . . . 35
A.6 Completer . . . 36
A.7 DocumentDB . . . 37
A.8 DocumentDBShard . . . 37
A.9 DocumentDBShardingManager . . . 37
A.10 DocumentGenerationManager . . . 38
A.11 DocumentStorageCache . . . 39
A.12 DocumentStorageFunctionality . . . 39
A.13 DocumentStorageManager . . . 40
A.14 DeliveryFunctionality . . . 40
A.15 EDocsAdminClient . . . 41
A.16 EmailChannel . . . 41
A.17 EmailFacade . . . 42
A.18 Generator . . . 42
A.19 GeneratorManager . . . 43
A.20 JobDBShard . . . 43
A.21 JobDBShardingManager . . . 44

A.22 JobFacade . . . 45
A.23 JobManager . . . 46
A.24 KeyCache . . . 47
A.25 LinkMappingDB . . . 48
A.26 LinkMappingManager . . . 48
A.27 LinkMappingFunctionality . . . 48
A.28 PDSDB . . . 49
A.29 PDSDBReplica . . . 50
A.30 PDSFacade . . . 50
A.31 PDSLongTermDocumentManager . . . 51
A.32 PDSReplicationManager . . . 52
A.33 Print&PostalServiceChannel . . . 53
A.34 Print&PostalServiceFacade . . . 53
A.35 NotificationHandler . . . 54
A.36 OtherDB . . . 54
A.37 RecipientClient . . . 56
A.38 RawDataHandler . . . 57
A.39 RecipientFacade . . . 57
A.40 RegistrationManager . . . 58
A.41 Scheduler . . . 59
A.42 SessionDB . . . 59
A.43 TemplateCache . . . 60
A.44 UserFunctionality . . . 61
A.45 ZoomitChannel . . . 61
A.46 ZoomitFacade . . . 62

B Defined data types **62**

1 Introduction

The goal of this project was to design a document processing system. In this document, we describe the final architecture, which was designed based on the requirements from the domain analysis and the priorities of these requirements given by our stakeholders. The provided initial architecture was used as a starting point to achieve this. Section 2 lists the architectural decisions for all non-functional requirements and discusses the final architecture. Section 3 provides and discusses the main context and decomposition diagrams of our architecture (i.e. the context and primary diagram of the component-and-connector view) along with a discussion of the main architectural decisions involved. Section 4 provides and discusses the more fine-grained decompositions of some of the major components in the main decomposition. Section 5 provides and discusses the deployment of the components of the component-and-connector view on physical nodes. Finally, Section 6 illustrates how our architecture accomplishes the most important functionality and data flows using sequence diagrams. Afterwards, Appendix A lists and describes all components of the component-and-connector view and their interfaces and Appendix B lists and describes the data types used in these interfaces.

2 Overview

This section gives a high-level but complete overview of the system: it lists the design decisions for all non-functional requirements and provides a discussion concerning the strong and weak points of the architecture.

2.1 Architectural decisions

In this section, we give an overview of the architectural decisions made in our architecture in order to achieve the requirements given in the domain analysis and the residual drivers given in the provided initial architecture. However, we will not repeat any decisions that were already documented in this provided initial architecture as they can be found there.

Av1a & Av2a: Notifying the appropriate operator within 1 minute *Av1a* and *Av2a* require the system to notify the eDocs operator within 1 minute in case of failure of the internal infrastructure responsible for generating documents or the internal (sub-)system responsible for storing documents in personal document stores. To achieve this, the `NotificationHandler` handles all incoming notifications and forwards those that are destined for an eDocs operator to the `EDocsAdminFacade`, which in turn delivers these to the external `EDocsAdminClient`. To ensure that notifications get sent to the eDocs operator in a timely fashion, both the `EDocsAdminFacade` and the `NotificationHandler` are deployed on the same node, which is separated from all other nodes to increase communication performance between the two as well as individual performance of both components separately.
For more details, we refer to Section 3.1.1 and Section 5.

Av1b: Storing the status of an individual job *Av1b* requires the system to store the status of an individual job. To achieve this, the `JobManager` accepts all read and write requests regarding the storage of job data, including status information.
For more details, we refer to Section 3.1.2 and Section 4.5.

Av2b: Temporary storage and user notification upon PDSDB failure *Av2b* requires (1) documents that should be delivered via the personal document store to be cached for at least 3 hours in case of unavailability and (2) a clear message to be provided to the recipient in this case. To achieve this, we introduced the `DocumentStorageCache`: this component temporarily stores the `DocumentIds` and corresponding `RecipientIds` of all generated documents that are to be delivered to the `PDSDB` during downtime of the latter component up to a maximum of 3 hours. When the `PDSDB` turns operational again, it notifies the `DocumentStorageManager` with a sign of life, which in turn retrieves all documents and their corresponding meta data from the `DocumentDB` using the previously mentioned ids and subsequently stores them in the `PDSDB` along with the converted `DocumentMetaData`.
Note that the user therefore perceives a maximum total downtime of 3 hours and that the `RecipientFacade` is in charge of presenting the user with a clear error message after having performed a read request in his or her behalf that has failed due to the unavailability of the `PDSDB`. However, during the time needed for the `DocumentStorageManager` to transfer all documents and meta data that the cached ids refer to, it is possible that the user is not able to access (part of) his or her documents via the `PDSDB`, more specifically those that are

still being transferred. Since the requirements do not demand the user to be able to regain full functionality of his or her personal document store at once, finding a better alternative for this gradual revival of the PDSDB is out of scope.
Finally, we would like to stress that the DocumentStorageManager and the PDSDB are necessarily deployed on different nodes for the former to be able to do its caching job while the latter is not operational during downtime. More precisely, the DocumentStorageManager implicitly pings the PDSDB when storing document data in it. The echo message then, in turn, consists of the write confirmation that is subsequently received. If one of those writes should fail, all subsequent writes are internally converted into pairs of DocumentId-RecipientId writes to the aforementioned cache. Once the PDSDB is operational again and all cached ids are processed, subsequent writes to the PDSDB will no longer be redirected through the DocumentStorageCache.
For more details, we refer to Section 3.1.3, Section 4.4 and Section 5.

Av3: Zoomit failure *Av3* requires the system to be able to (1) autonomously detect when an invoice is not accepted by Zoomit, (2) temporarily store at least 2 days of these invoices locally until they are accepted by Zoomit, (3) keep on retrying to deliver a failed invoice to Zoomit in a proper fashion and (4) notify an eDocs operator after 5 failed attempts of at least 10 different invoices. To achieve this, the ZoomitFacade is able to receive a delivery confirmation from the ZoomitChannel (i.e. the invoice was accepted by Zoomit). Furthermore, the ZoomitDeliveryCache caches all Documents and corresponding ZoomitId for which the ZoomitFacade did not (yet) receive a delivery confirmation up to a maximum of 2 days of documents.
The ZoomitFacade then periodically retries to deliver the next document in the cached list of Document-ZoomitId pairs (note that the ZoomitDeliveryCache thus stores its entries in a circular manner) to the ZoomitChannel in a proper fashion by using exponential back-off. Once the ZoomitFacade has received a delivery confirmation regarding one of those cached documents, the corresponding Document-ZoomitId pair is deleted from the ZoomitDeliveryCache and the ZoomitFacade tries to deliver every other document in the list to the ZoomitChannel right after, disregarding the previously employed method of exponential back-off (because the external ZoomitChannelNode on which the ZoomitChannel resides, is now presumed to be operational again).
Finally, the Document-ZoomitId pairs in the ZoomitDeliveryCache need to be stored along with a counter that the ZoomitFacade increments after each failed attempt to send the respective document. Additionally, the ZoomitFacade keeps track of all cached pairs whose counters have reached the value of 5 and sends a notification to an eDocs operator through the NotificationHandler. For more details, we refer to Section 3.1.1, Section 4.1 and Section 5.

P2: Document lookups *P2* requires the system to be able to (1) respond to all incoming document requests in a timely fashion, (2) throttle excessive requests when the arrival rate is larger than a certain value that is specific to the kind of request (determined by the requirements) so that the non-excessive ones continue to be handled in a timely fashion and (3) make sure that the performance of all other functionality of the system remains unaffected in case of a large number of requests. To achieve this, all potential bottleneck components that support the document lookup process are deployed on separate nodes. More precisely, the RecipientFacade and the LinkMappingFunctionality both reside on the same exclusive node as to increase the performance of these components and the communication between them (i.e. when a recipient has presented the former component with a link for which it has to determine the document it maps to). Both the AuthenticationHandler and the SessionDB are also deployed on the same individual node as to not let the authentication subprocess decrease the performance of the document lookup process. The PDSFacade is deployed on the PDSDBManagerNode (discussed in the provided initial architecture), along with the PDSLongTermManager and the two PDSReplicationManagers, as to increase communication performance between this PDSFacade and the PDSDB supercomponent. Notice that *Av1* already demands the PDSDB to be deployed separate from all other functionality according to the provided initial architecture and that *P2* can rely on this design decision to deliver an optimally increased (taking the availability constraints of *Av1* into account) performance of this supercomponent.
Because the DocumentDB, like the PDSDB, stores a large amount of documents and correspondingly handles a large amount of document requests, its documents are partitioned across several DocumentDBShards that each reside on their own DocumentDBShardNode. This sharding technique results in a performance increase that is statistically equivalent to the one that is achieved by active replication, but it is significantly less costly regarding storage costs (which is more of a concern for documents than for other kinds of data). In order to further increase performance of the DocumentDB, the DocumentDBShardingManager, which is in charge of managing all DocumentDBShards and forwarding read and write requests to the appropriate ones, is also deployed on an individual node.

Finally, both the DocumentDB and the PDSDB have the ability to throttle excessive read requests from the RecipientFacade (all other requests remain unaffected) when the rate of these incoming requests exceeds a certain value that is specific to the kind of request (determined by the requirements). Also note that the performance of all other functionality of the system remains unaffected even at this maximum, since all potential bottleneck components that support the document lookup process are deployed on separate nodes, effectively increasing their performance (as discussed above) and thereby providing both an increased level of performance for the document lookup process and a sufficient level of performance for all other functionality. For more details, we refer to Section 3.1.4, Section 3, Section 4.8, Section 4.7, Section 4.2 and Section 5.

P3: Status overview for customer administrators *P3* requires (1) the status overview of all document processing jobs initiated by a certain customer organization to be delivered to the respective customer administrator(s) in a timely fashion (i.e. the provided status of a document processing job should be consistent up to 1 minute ago) and (2) the construction of this status overview not to hinder other parts of the system. To achieve this, all potential bottleneck components that support the status overview process are deployed on separate nodes as to increase their performance. More precisely, both the AuthenticationHandler and the SessionDB are also deployed on the same individual node as to not let the authentication subprocess decrease the performance of the status overview process. Furthermore, the CustomerOrganizationFacade and the JobManager supercomponent are also deployed separate from all other components in the system. More specifically, the former occupies a single node by itself, while the latter is responsible for storing all jobs, which are partitioned across several JobDBShards that each reside on their own JobDBShardNode too. This sharding technique results in a performance increase that is statistically equivalent to the one that is achieved by active replication, but it is less costly regarding storage costs. In order to further increase performance of the JobManager and the corresponding speed at which job statuses can be requested, the JobDBShardingManager, which is in charge of managing all JobDBShards and forwarding read and write requests to the appropriate ones, is also deployed on an individual node.
Note that the performance of all other functionality of the system remains unaffected during the construction of such a status overview, since all potential bottleneck components that support this status overview process are deployed on separate nodes, effectively increasing their performance (as discussed above) and thereby providing both an increased level of performance for the status overview process and a sufficient level of performance for all other functionality.
Also note that the JobDBShardManager and corresponding JobDBShards do not distinguish between jobs of different age during the construction of an overview. Although the requirements mention a different minimal response time for each type of job with regard to its age, our system does not offer this distinction. Instead, the JobDBShardManager and corresponding JobDBShards simply present the customer administrator(s) with a response time that is equal to the lowest of all minimal response times mentioned in the requirements. This strategy was chosen due to the fact that (1) different dynamic storage requirements for jobs along with their periodical synchronization would largely outweigh this simple storage approach with respect to implementation and deployment costs and (2) the same level of performance per type of job can easily be provided by this simple storage approach (i.e. the uniform response time per job should be less than the lowest of all minimal response times) because of the small storage capacity that is needed for job data. For more details, we refer to Section 4.8, Section 4.5 and Section 5.

M1: New type of document: bank statements *M1* requires (1) the interface for initializing document processing jobs to be easily extendible with the functionality to provide the raw data for other types of documents, (2) the interface for registering companies to be easily extendible with the functionality to enable the new types of documents for new customer organizations and configure their template for these types of documents, (3) the functionality for generating documents to be easily extendible with the new generation steps for the new types of documents, (4) the generation of the new types of documents to reuse as much of the existing functionality for generating documents as possible, (5) the storage of generated documents of the new types of documents not to require any changes to the existing system for storing generated documents and (6) the interface for consulting the personal document store and specific documents to be easily extendible with the functionality to show the new types of documents. To achieve this, all relevant interfaces of the components involved in the data flows of these processes, including the specified data types, are generic enough to cope with different document types. Furthermore, all potential changes are encapsulated by the RawDataHandler and the DocumentGenerationManager supercomponent. More specifically, the former will now also need to handle the verification of raw data that is characteristic for these new types of documents. The generic internal structure of the latter also ensures that the generation of new document types will reuse as much of the existing functionality for generating documents as possible. Note that the type and the format of a document are two

different things: the first being the focus of *M1* and the second being the internal structure of a `Document` data type, which remains untouched in the context of *M1*. Since the `RecipientFacade` only has knowledge about the latter, no changes need to be made to the `RecipientFacade` upon introduction of a new document type. For more details, we refer to Section 3, Section 4.3, Section 4.8 and Appendix B.

M2: Multiple print & postal services *M2* requires (1) the system to be able to easily switch print & postal service in the future, (2) the incorporation of this new print & postal service in the system to only affect the final steps of the document processing flow. To achieve this, the `Print&PostalServiceFacade` encapsulates all changes that might need to be made in order to seamlessly incorporate a new print & postal service in the system, e.g. additional methods, external interfaces, ... For more details, we refer to Section 4.1.

M3: Dynamic selection of the cheapest of print & postal services *M3* requires the system to be easily extendible in order to be able to (1) employ multiple print & postal services across the world, (2) receive the daily price and capacity of each print & postal service and (3) dynamically select (i.e. on a daily basis) the most suited print & postal service based on the system's document processing load, the price of the service and the location to which the documents should be sent. To achieve this, the `Print&PostalServiceFacade` encapsulates all changes that might need to be made in order to seamlessly incorporate this new functionality in the system. More precisely, this component will only require (1) an additional external interface to each of the employed print & postal services, (2) an internal interface to the `OtherDB` to consult all SLAs in order to retrieve the approximate location and size of all batches for the day and (3) its internal structure to be adapted to the new functionality. For more details, we refer to Section 4.1.

2.2 Discussion

GEEN JOBFACADE MEER Use this section to discuss your architecture in retrospect. For example, what are the strong points of your architecture? What are the weak points? Is there anything you would have done otherwise with your current experience? Are there any remarks about the architecture that you would give to your customers? Etc.

3 Client-server view (UML Component diagram)

Context diagram The context diagram of the component-and-connector view is given in Figure 1. As shown, six distinct types of external components communicate with the system: (i) the EDocsAdminClient represents a client device of an administrator of eDocs that communicates with the eDocs System, (ii) the CustomerOrganizationClient represents a client device of a Customer Organization (i.e. Customer Administrator and Customer Information System) that communicates with the eDocs System, (iii) the RecipientClient represents a client device of an unregistered or registered recipient of eDocs that communicates with the eDocs System, (iv) the ZoomitChannel represents the servers of Zoomit to which documents can be sent for further delivery, (v) the Print&PostalServiceChannel represents the servers of a print & postal service to which documents can be sent for further delivery, (vi) the EmailChannel represents a mail server of an e-mail provider to which documents can be sent for further delivery. Notice that each external interface on the client side is provided by a separate facade component and that each delivery channel interfaces with the `DeliveryFunctionality` component.

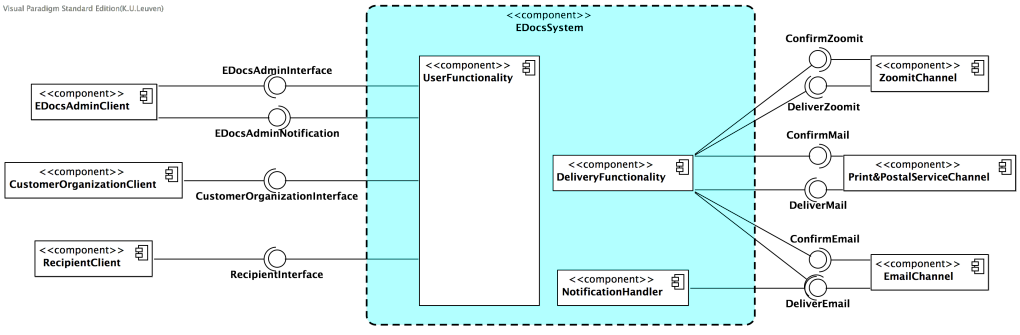

Figure 1: Context diagram for the client-server view.

Primary diagram The main component-and-connector view is shown in Figure 2. All components that were decomposed further are highlighted. The DeliveryFunctionality is further decomposed in Section 4.1, the DocumentDB in Section 4.2, the DocumentGenerationManager in Section 4.3, the DocumentStorageFunctionality in Section 4.4, the JobManager in Section 4.5, the LinkMappingFunctionality in Section 4.6, the PDSDB in Section 4.7 and the UserFunctionality in Section 4.8. The descriptions of all components in Figure 2, their sub-components and their interfaces are given in Appendix A.

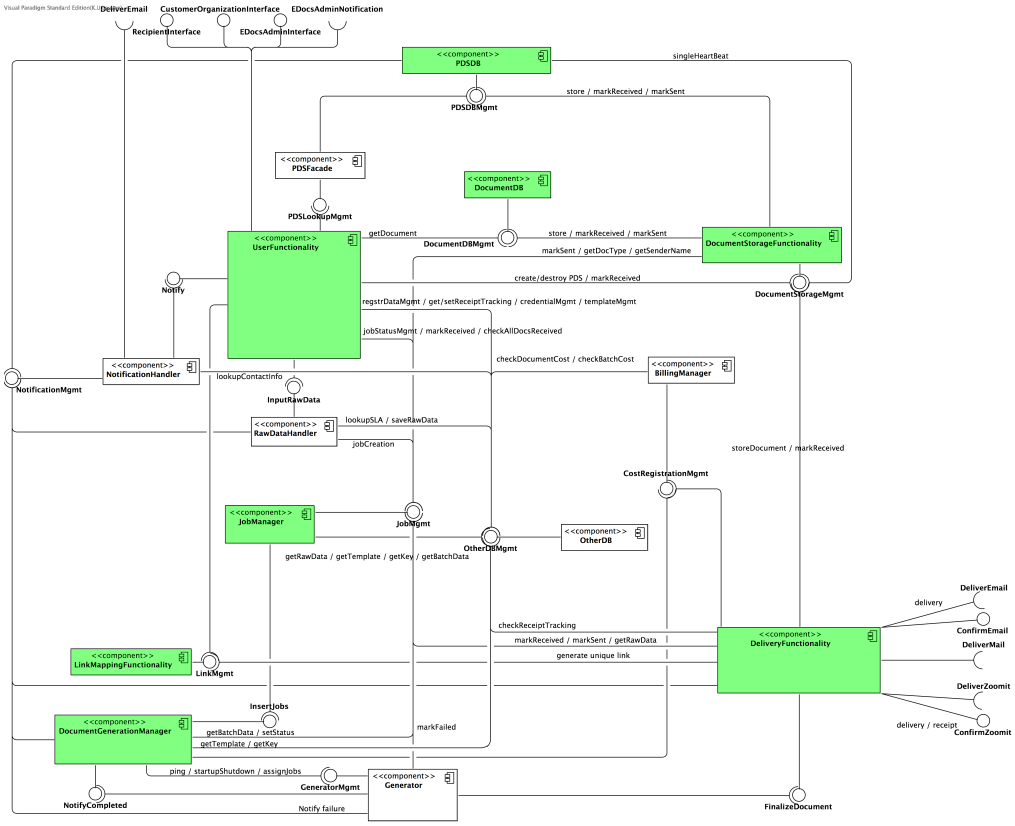

Figure 2: Primary diagram of the client-server view.

3.1 Main architectural decisions

Figure 2 provides the overall description of the architecture and illustrates the most important architectural decisions for achieving the required qualities. Here we document how our architecture fulfils the most important qualities (based on the priorities given by the stakeholders). Although the qualities that were already discussed in the provided initial architecture will not be analysed any further here, their residual drivers will. The resulting qualities thus consist of Av1a & Av2a: Notifying the appropriate operator within 1 minute, Av1b: Storing the status of an individual job and Av2b: Temporary storage and user notification during PDSDB failure. In addition, we also highlight P2: Document lookups, because of the importance of this quality to a significant part of the recipient base.

3.1.1 Av1a & Av2a: Notifying the appropriate operator within 1 minute

Firstly, *Av1a* and *Av2a* require the system to notify the eDocs operator in case of failure of the internal infrastructure responsible for generating documents or the internal (sub-)system responsible for storing documents in personal document stores (cf. the analysis of *Av1* and *Av2* in the provided initial architecture respectively). Therefore, the `NotificationHandler` handles all incoming notifications from the respective components (i.e. the `DocumentGenerationManager` and the `PDSDB`) and forwards those that are destined for an eDocs operator to the `EDocsAdminFacade`, which in turn delivers these to the external `EDocsAdminClient`.

Secondly, to ensure that notifications get sent to the eDocs operator in a timely fashion (i.e. within 1 minute), both the `EDocsAdminFacade` and the `NotificationHandler` are deployed on the same node, which is separated

from all other nodes to increase communication performance between the two as well as individual performance of both components separately (see the deployment view in Section 5). As was already pointed out and discussed in the provided initial architecture, it is the responsibility of the `DocumentGenerationManager` and the `PDSDB` that the `NotificationHandler` itself gets notified in a timely fashion. The `DocumentGenerationManager` and the `PDSDB` are further decomposed in Sections 4.3 and 4.7 respectively.

Alternatives considered

Alternative for NotificationHandler An alternative for the use of the `NotificationHandler` would be for all components in our system to contact the appropriate userfacades (i.e. the `RecipientFacade`, the `CustomerOrganizationFacade` and/or the `EDocsAdminFacade`) directly and to let them handle all subsequent steps of the notification process. This would entail (1) an increase in coupling between those facades and the rest of the system, (2) a decrease in cohesion of all three userfacades and (3) a decrease in adaptability of the system due to the fact that all notification functionality is now spread across several components. For these reasons, we decided to include the `NotificationHandler` as a component in our architecture.

Alternative for joint deployment of NotificationHandler and EDocsAdminFacade In the current architecture, both the `EDocsAdminFacade` and the `NotificationHandler` are deployed on the same node, which is separated from all other nodes to increase overall performance. An alternative for this strategy would be to deploy the `NotificationHandler` on its own node, which would increase the individual performance of this component. Unfortunately, this would also decrease the communication performance between this component and the `EDocsAdminFacade`. Since *Av1a* and *Av2a* only dictate an eDocs operator to receive his or her notifications in a timely fashion and the increase in individual performance of the `NotificationHandler` would not outweigh the decrease in communication performance with the `EDocsAdminFacade` in this case, we chose to go with our original approach and deploy both components on the same node (as discussed above).

3.1.2 Av1b: Storing the status of an individual job

Av1b requires the system to store the status of an individual document processing job in order to be able to show the status of the jobs affected by the failure of the internal infrastructure responsible for generating documents (cf. the analysis of *Av1* in the provided initial architecture). To achieve this, the `JobManager` accepts all read and write requests regarding the storage of job data, including status information. More specifically, the `JobFacade` subcomponent forwards these requests to the `JobDBShardingManager`, which in turn performs the read or write operation on the appropriate `JobDBShard` (see the decomposition of the `JobManager` in Section 4.5).

Alternatives considered

Alternative for JobFacade indirection In our current architecture, the `JobFacade` subcomponent forwards all read and write requests to the `JobDBShardingManager`. An alternative for this approach would be remove this extra level of indirection (i.e. the `JobFacade`) and let the `JobDBShardingManager` handle all requests directly. Although this would greatly simplify the internal workings of the `JobManager` supercomponent, this would also complicate the workings of other components (e.g. the `DeliveryFunctionality`, the `DocumentGenerationManager` and the `UserFunctionality`) that call its interface. The reason for this is that, in our current architecture, all calls to the `OtherDB` based on a `JobId` (i.e. the `JobId` is given as an argument in a method call to the interface) are routed through the `JobFacade`, which in turn fetches the required data from the `OtherDB`. Removing this level of indirection would result in those components (from which the routed calls originated) calling the `OtherDB` directly after having fetched the appropriate job information from the `JobDBShardingManager` themselves. In order to centralize this functionality and alleviate these components from the burden of calling both the `JobDBShardngManager` and the `OtherDB`, we decided to include the `JobFacade` as a fully-fledged component in our architecture.

3.1.3 Av2b: Temporary storage and user notification upon PDSDB failure

Firstly, *Av2b* requires documents that should be delivered via the personal document store to be cached for at least 3 hours in case of unavailability (cf. the analysis of *Av2* in the provided initial architecture). Therefore, we introduced the `DocumentStorageFunctionality`: the `DocumentStorageCache` that is part of this component temporarily stores the `DocumentIds` and corresponding `RecipientIds` of all generated documents that are to be delivered to the `PDSDB` during downtime of the latter component up to a maximum of 3 hours. When

the PDSDB turns operational again, it notifies the DocumentStorageManager (which is also a subcomponent of the DocumentStorageFunctionality) with a sign of life, which in turn retrieves all documents and their corresponding meta data from the DocumentDB using the previously mentioned ids and subsequently stores them in the PDSDB along with the converted DocumentMetaData (i.e. the original DocumentMetaData, including the corresponding RecipientId but omitting the EmailAddress if present). Note that the DocumentStorageManager and the PDSDB are necessarily deployed on different nodes for the former to be able to do its caching job while the latter is not operational during downtime (see the deployment view in Section 5). More precisely, the DocumentStorageManager implicitly pings the PDSDB when storing document data in it. The echo message then, in turn, consists of the write confirmation that is subsequently received. If one of those writes should fail, all subsequent writes are internally converted into pairs of DocumentId-RecipientId writes to the aforementioned cache. Once the PDSDB is operational again and all cached ids are processed, subsequent writes to the PDSDB will no longer be redirected through the DocumentStorageCache. The DocumentStorageFunctionality is further decomposed in Section 4.4.

Secondly, *Av2b* requires a clear message to be provided to the recipient in case of unavailability of the personal document store. This responsibility is delegated to the RecipientFacade in the following way: this component presents the user with a clear error message after having performed a read request in his or her behalf that has failed due to the unavailability of the PDSDB. Note that the recipient therefore perceives a maximum total downtime of 3 hours. However, during the time needed for the DocumentStorageManager to transfer all documents and meta data that the cached ids refer to, it is possible that the user is not able to access (part of) his or her documents via the PDSDB, more specifically those that are still being transferred.

Alternatives considered

Alternative for sign of life In the current architecture, the PDSDB notifies the DocumentStorageManager with a sign of life when the former turns operational again after failure. An alternative for this strategy would be not to rely on this sign of life, but instead to consider all write attempts to the PDSDB as implicit ping messages and their corresponding confirmations as implicit echo messages. An obvious advantage here is that the PDSDB does now not have to actively contact the DocumentStorageManager upon revival. The problem with this approach, however, is that the view of the personal document store for the recipient cannot be controlled by the system. The reason for this can best be made clear with an example scenario: imagine the PDSDB turning operational again after failure and the DocumentStorageCache still containing some document entries due to the fact that no writes have occurred for a while (and the DocumentStorageManager has therefore not started transferring any of the cached documents). The recipient will not be able to request those documents via the PDSDB until another write occurs: a situation for which the time of resolution is rather unpredictable. Since these kind of scenarios would most likely result in damage to the reputation of eDocs and actively sending a sign of life upon revival only requires some minor changes to the two components involved, we decided to go with the latter strategy.

Alternative for gradual revival of the PDSDB In our current architecture, the recipient perceives the PDSDB to be down for a maximum of 3 hours. However, during the time needed for the DocumentStorageManager to transfer all documents and meta data that the cached ids refer to, it is possible that the user is not able to access (part of) his or her documents via the PDSDB, more specifically those that are still being transferred. An alternative for this approach would be to let the PDSDB keep blocking all read requests until the DocumentStorageManager has stopped transferring. This would mean that the maximum waiting time for recipients trying to access their documents in the PDSDB is increased with the total transfer time needed by the DocumentStorageManager. However, since the requirements do not demand the user to be able to regain full functionality of his or her personal document store at once, we decided to go with a gradual revival of the PDSDB in order to be able to offer the recipient a view of the PDSDB (though it could be an incomplete one) as soon as possible to increase performance of the document lookup process (discussed in Section 3.1.4).

Alternative for DocumentStorageCache An alternative for the use of the DocumentStorageCache would be to let the DocumentStorageManager actively look for documents in the DocumentDB that are not yet, but should be, present in the PDSDB upon revival of this last component. A clear advantage of this approach is that the downtime of the PDSDB component is no longer restricted by the aforementioned 3-hour cache. An important disadvantage, however, lies in the fact that the DocumentStorageManager is burdened with a significant amount of extra work and will require more expensive hardware to cope with this. Since the support for a longer downtime of the PDSDB is out of scope, this alternative approach was not chosen.

3.1.4 P2: Document lookups

Firstly, *P2* requires the system to be able to respond to all incoming document requests in a timely fashion. To achieve this, all potential bottleneck components that support the document lookup process are deployed on separate nodes (see the deployment view in Section 5). More precisely, the `RecipientFacade` (which is a subcomponent of the `UserFunctionality` and is responsible for interfacing with the `RecipientClient` as can be reviewed in Section 4.8) and the `LinkMappingFunctionality` both reside on the same exclusive node as to increase the performance of these components and the communication between them (i.e. when the `RecipientClient` has presented the former component with a link for which it has to determine the document it maps to). Furthermore, the `AuthenticationHandler` and the `SessionDB` (both subcomponents of the `UserFunctionality`) are also deployed on the same individual node as to not let the authentication subprocess decrease the performance of the document lookup process. The `PDSFacade` is deployed on the `PDSDBManagerNode` (discussed in the provided initial architecture), along with the `PDSLongTermManager` and the two `PDSReplicationManagers`, as to increase communication performance between this `PDSFacade` and the `PDSDB` supercomponent (see also the decomposition of the `PDSDB` in Section 4.7). Notice that *Av1* already demands the `PDSDB` to be deployed separate from all other functionality according to the provided initial architecture and that *P2* can rely on this design decision to deliver an optimally increased (taking the availability constraints of *Av1* into account) performance of this supercomponent. Furthermore, because the `DocumentDB`, like the `PDSDB`, stores a large amount of documents and correspondingly handles a large amount of document requests, its documents are partitioned across several `DocumentDBShards` that each reside on their own `DocumentDBShardNode` (see the decomposition of the `DocumentDB` in Section 4.2). This sharding technique results in a performance increase that is statistically equivalent to the one that is achieved by active replication, but it is significantly less costly regarding storage costs (which is more of a concern for documents than for other kinds of data). In order to further increase performance of the `DocumentDB`, the `DocumentDBShardingManager`, which is in charge of managing all `DocumentDBShards` and forwarding read and write requests to the appropriate ones, is also deployed on an individual node.

Secondly, *P2* requires the system to be able to throttle excessive incoming document requests when the arrival rate is larger than a certain value that is specific to the kind of request (determined by the requirements), so that the non-excessive ones continue to be handled in a timely fashion. To achieve this, both the `DocumentDB` and the `PDSDB` have the ability to throttle excessive read requests from the `PDSFacade`, through which the recipient's `PDSDB` requests are routed, and the `RecipientFacade` (which is a subcomponent of the `UserFunctionality`) when the rate of these incoming requests exceeds a certain value that is specific to the kind of request (determined by the requirements).

Thirdly, *P2* requires the performance of all other functionality of the system to remain unaffected in case of a large number of incoming document requests. Therefore, all potential bottleneck components that support the document lookup process are deployed on separate nodes, effectively increasing their performance (as discussed above) and thereby providing both an increased level of performance for the document lookup process and a sufficient level of performance for all other functionality. Also note that only requests coming from the `RecipientFacade` and the `PDSFacade`, which are inherently tied to the document lookup process, can be throttled by the `DocumentDB` and the `PDSDB` (as discussed above) and that all other requests thus remain unaffected by this throttling strategy at all times.

Alternatives considered

Alternative for sharding of the DocumentDB An alternative for sharding the `DocumentDB` would be to actively replicate it. However, although the performance increase is statistically equivalent in both cases, the former strategy is significantly less costly regarding storage costs. Since these cost concerns are particularly important for large data files like documents and there is no other availability requirement for document replication in this case, we chose to shard the `DocumentDB` instead of actively replicate it.

Alternative for separate deployment of potential bottleneck components in the document lookup process In our current architecture, all potential bottleneck components that support the document lookup process are deployed on separate nodes. An obvious alternative would be to deploy these components on the same node (assuming this would not contradict with any design decisions that were made for other requirements). Although this approach would greatly increase communication performance between all components involved, it would also decrease overall performance of the document lookup process and, more importantly, of

the other functionality in the system. For these reasons, we decided to deploy these components on separate nodes as discussed above.

Alternative for the DocumentDB and the PDSDB having throttling responsibility In our current design, both the `DocumentDB` and the `PDSDB` have the responsibility to throttle excessive read requests (originating from the `RecipientFacade`) when the rate of these incoming requests exceeds a certain value that is specific to the kind of request (determined by the requirements). Since all document lookup requests pass through the `RecipientFacade`, a valid alternative for this approach would be to delegate all throttling responsibility to this component. A clear advantage of the latter strategy is the centralization of all throttling functionality, effectively increasing the modifiability of the system. An important drawback, however, is the fact that the throttling component can now not take into account the `PDSDB` and `DocumentDB` requests originating from other components in the system. This lack of knowledge results in a fixed suboptimal upper bound for document lookup requests originating from the `RecipientFacade`. The reason for this is that the throttling component now needs take into account the maximum amount of residual requests (i.e. requests originating from all other components in the system) in order to not let the throttling affect any other functionality in the system. Since our current architecture does support an optimal load balancing policy for document lookup requests by throttling at the source components (i.e. the `PDSDB` and the `DocumentDB`), we chose to go with the latter strategy.

4 Decomposition view (UML Component diagram)

An overview of the main component-and-connector view can be found in Section 3. All components that were decomposed any further are highlighted there and will be discussed below in greater detail. This component group consists of the DeliveryFunctionality (Section 4.1), the DocumentDB (Section 4.2), the DocumentGenerationManager (4.3), the DocumentStorageFunctionality (4.4), the JobManager (Section 4.5), the LinkMappingFunctionality (Section 4.6), the PDSDB (Section 4.7) and the UserFunctionality (Section 4.8).

4.1 DeliveryFunctionality

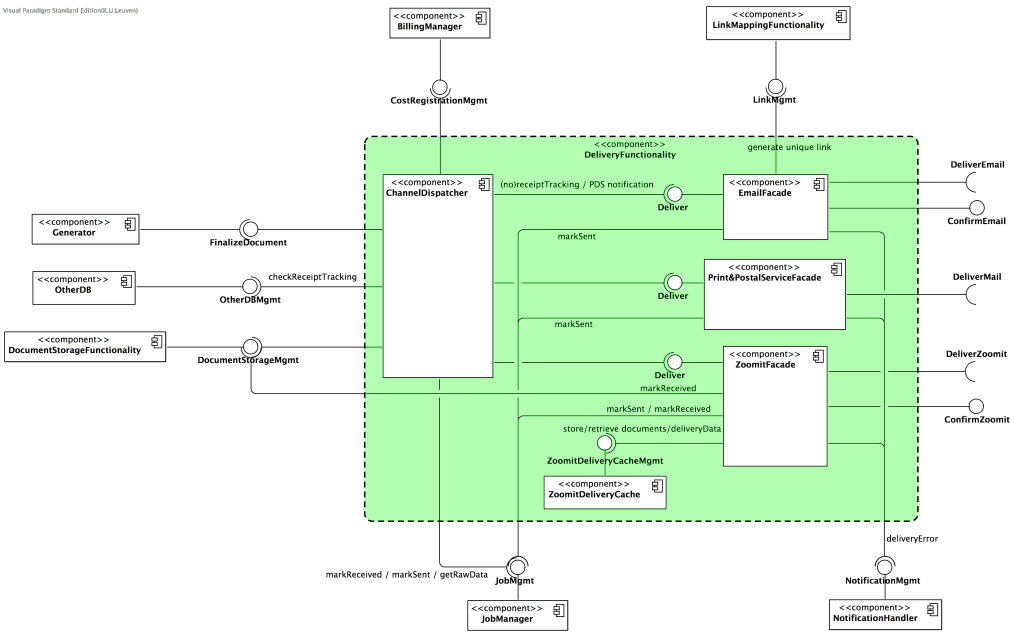

Figure 3: Decomposition of `DeliveryFunctionality`

The decomposition of the `DeliveryFunctionality` is presented in Figure 3. As shown, the `DeliveryFunctionality` is decomposed into five submodules: the `ChannelDispatcher`, the `EmailFacade`, the `Print&PostalServiceFacade`, the `ZoomitFacade` and the `ZoomitDeliveryCache`.

4.2 DocumentDB

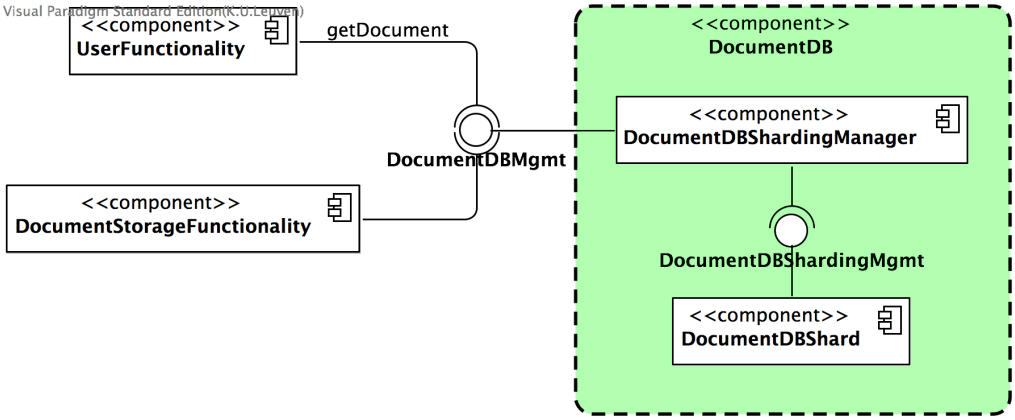

Figure 4: Decomposition of `DocumentDB`

The decomposition of the `DocumentDB` is presented in Figure 4. As shown, the `DocumentDB` is decomposed into only two submodules: the `DocumentDBShardingManager` and the `DocumentDBShard`.

The `DocumentDB` is responsible for the actual storage of all documents, along with their meta data. More specifically, the `DocumentStoragFunctionality` stores documents in it that both this component and the `RecipientFacade` are able to access. Because the `DocumentDB` stores a large amount of documents and correspondingly handles a large amount of document requests, its documents are partitioned across several `DocumentDBShards` that each reside on their own `DocumentDBShardNode` according to *P2*. In order to further increase performance of the `DocumentDB` for *P2*, the `DocumentDBShardingManager` is also deployed on an individual node. It is responsible for managing all `DocumentDBShards` (i.e. maintaining a consistent overview of all shards and their corresponding documents) and forwarding read and write requests to the appropriate ones.

Finally, *P2* requires the `DocumentDB` to be able to throttle excessive read requests from the `RecipientFacade` (see also the decomposition of the `UserFunctionality` in Section 4.8) when the arrival rate is larger than a certain value that is specific to the kind of request (determined by the requirements). This responsibility is delegated to the `DocumentDBShardingManager`, the only component in the document lookup pipeline that has knowledge of all incoming `DocumentDB` requests, to ensure that other `DocumentDB` requests are also taken into account in order to not hinder the other functionality of the system. Note that non-excessive recipient requests to the `DocumentDB` continue to be handled in a timely fashion during this throttling, as is dictated by *P2*.

4.3 DocumentGenerationManager

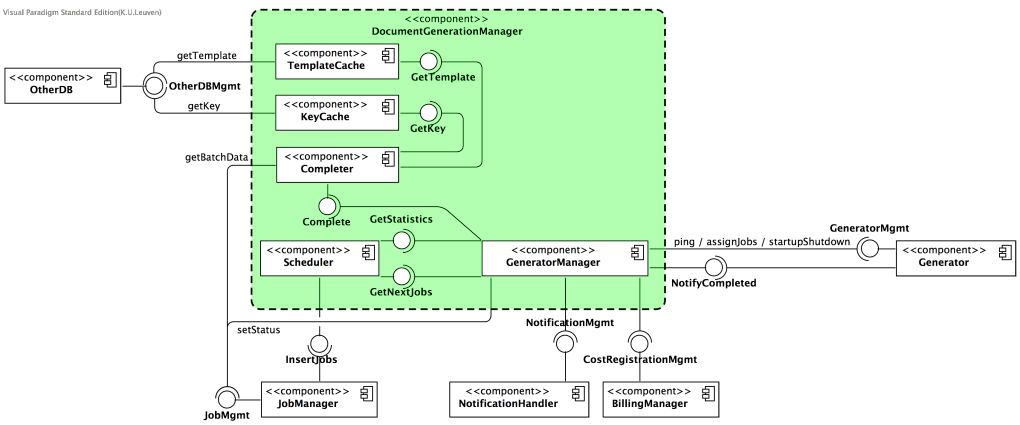

Figure 5: Decomposition of `DocumentGenerationManager`

The decomposition of the DocumentGenerationManager is presented in Figure 5. As shown, the DocumentGenerationManage is decomposed into five submodules: the TemplateCache, the KeyCache, the Completer, the Scheduler and the GeneratorManager.

Since this decomposition was already discussed in the provided initial architecture, we will not repeat those results here. Note, however, that some interfaces have been changed, added or even removed for the DocumentGenerationManag to be able to properly communicate with the rest of the system (see Appendix A).

4.4 DocumentStorageFunctionality

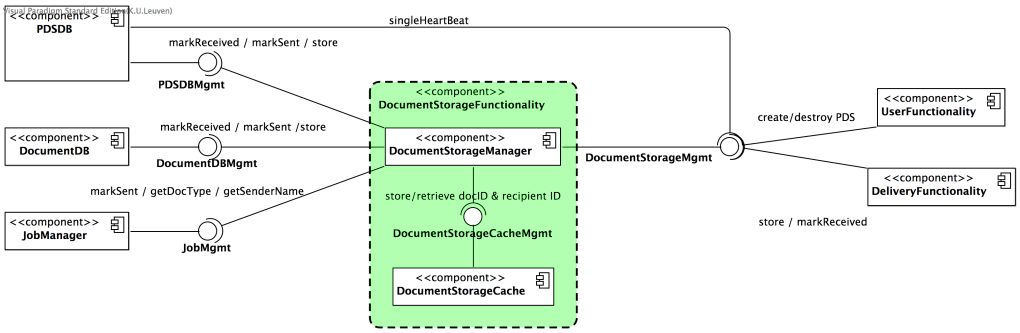

Figure 6: Decomposition of DocumentStorageFunctionality

The decomposition of the DocumentStorageFunctionality is presented in Figure 6. As shown, the DocumentStorageFuncti is decomposed into only two submodules: the DocumentStorageManager and the DocumentStorageCache.

The DocumentStorageManager is responsible for storing generated documents in the correct database(s) (i.e. the PDSDB and the DocumentDB) upon the appropriate method call from the ChannelDispatcher (see also the decomposition of the DeliveryFunctionality in Section 4.1). If the call indicates that the document belongs to a registered recipient, the DocumentStorageManager sends a write request to both the DocumentDB and the PDSDB, accompanied by a call to the JobManager to mark the corresponding job as sent. However, if the call indicates that the document belongs to an unregistered recipient, the DocumentStorageManager only sends a write request to the DocumentDB. Note that in each case, the DocumentStorageManager creates the appropriate DocumentMetaData (see Appendix B) to store along with the document. Also note that all calls to add a receipt timestamp to the meta data of a document pass through the DocumentStorageManager in order to reach both the DocumentDB and the PDSDB in a controlled manner.

Furthermore, the DocumentStorageManager is responsible for transferring an unregistered recipient's documents from the DocumentDB to the PDSDB along with the converted meta data (i.e. the original DocumentMetaData, including the corresponding RecipientId but omitting the EmailAddress if present) when this recipient registers to the eDocs system, and for deleting a registered recipient's documents from the PDSDB when this recipient unregisters. Note that both processes are set in motion by the RegistrationManager (see also the decomposition of the UserFunctionality).

A major responsibility of the DocumentStorageFunctionality is the temporary storage (i.e. for a maximum of 3 hours) of documents that are to be stored in the PDSDB upon failure of the latter component, according to *Av2b*. To achieve this, the DocumentStorageManager stores the corresponding DocumentId-RecipientId pairs in the DocumentStorageCache during PDSDB downtime (which is detected through a failed write request). Note that all those documents are also stored in the DocumentDB (see the decomposition of the DocumentDB in Section 4.2), so that the DocumentStorageManager is able to transfer all cached documents (i.e. documents for which DocumentId-RecipientId pairs have been cached) from the DocumentDB to the PDSDB upon revival of the latter component (which is detected through a sign-of-life call from the PDSDB itself). In order to successfully complete this transfer, the DocumentStorageManager stores those documents along with their converted meta data (i.e. the original DocumentMetaData, including the corresponding RecipientId but omitting the EmailAddress if present) in the PDSDB. Since the requirements do not specify what happens after 3 hours, the behaviour of the system is undefined in this situation.

4.5 JobManager

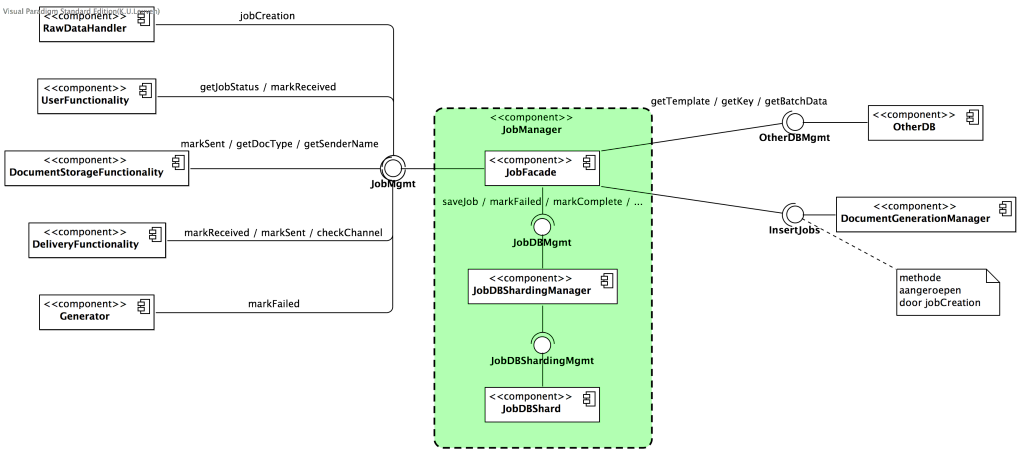

Figure 7: Decomposition of `JobManager`

The decomposition of the `JobManager` is presented in Figure 7. As shown, the `JobManager` is decomposed into three submodules: the `JobFacade`, the `JobDBShardingManager` and the `JobDBShard`.

In order to comply with *Av1b*, the `JobManager` is responsible for storing all document processing jobs. These jobs are (according to *P3*) partitioned across several `JobDBShards` that each reside on their own node. In order to further comply with *P3* and thus increase performance of the `JobManager` and the corresponding speed at which job statuses can be requested, the `JobDBShardingManager`, which is in charge of managing all `JobDBShards` and forwarding read and write requests to the appropriate ones, is also deployed on an individual node (see also the deployment view in Section 5).

Note that the `JobDBShardManager` and corresponding `JobDBShards` do not distinguish between jobs of different age. Although the requirements for *P3* mention a different minimal response time for each type of job with regard to its age during the construction of a job status overview, our system does not offer this distinction. Instead, the `JobDBShardManager` and corresponding `JobDBShards` simply present the customer administrator(s) with a response time that is equal to the lowest of all minimal response times mentioned in the requirements. This strategy was chosen due to the fact that (1) different dynamic storage requirements for jobs along with their periodical synchronization would largely outweigh this simple storage approach with respect to implementation and deployment costs and (2) the same level of performance per type of job can easily be provided by this simple storage approach (i.e. the uniform response time per job should be less than the lowest of all minimal response times) because of the small storage capacity that is needed for job data.

A major responsibility of the `JobFacade` is forwarding all job requests (e.g. marking a job as sent/received/failed/completed, reading the status of a job, ...) to the `JobDBShardingManager`. Additionally, all calls to the `OtherDB` based on a `JobId` (i.e. the `JobId` is given as an argument in a method call to the interface) are also routed through the `JobFacade`. This component then in turn fetches the required data from the `OtherDB` and sends it back to the original requesting component. Removing this level of indirection would result in those components (from which the routed calls originated) calling the `OtherDB` directly after having fetched the appropriate job information from the `JobDBShardingManager` themselves. In order to centralize this functionality and alleviate these components from the burden of calling both the `JobDBShardngManager` and the `OtherDB`, we decided to include the `JobFacade` as a fully-fledged component in our architecture.

Furthermore, the `JobFacade` is also responsible for creating jobs, a process that is initiated by the `RawDataHandler`. This process includes storing the corresponding jobs (via the `JobDBShardingManager`) and inserting them in the `DocumentGenerationManager`.

4.6 LinkMappingFunctionality

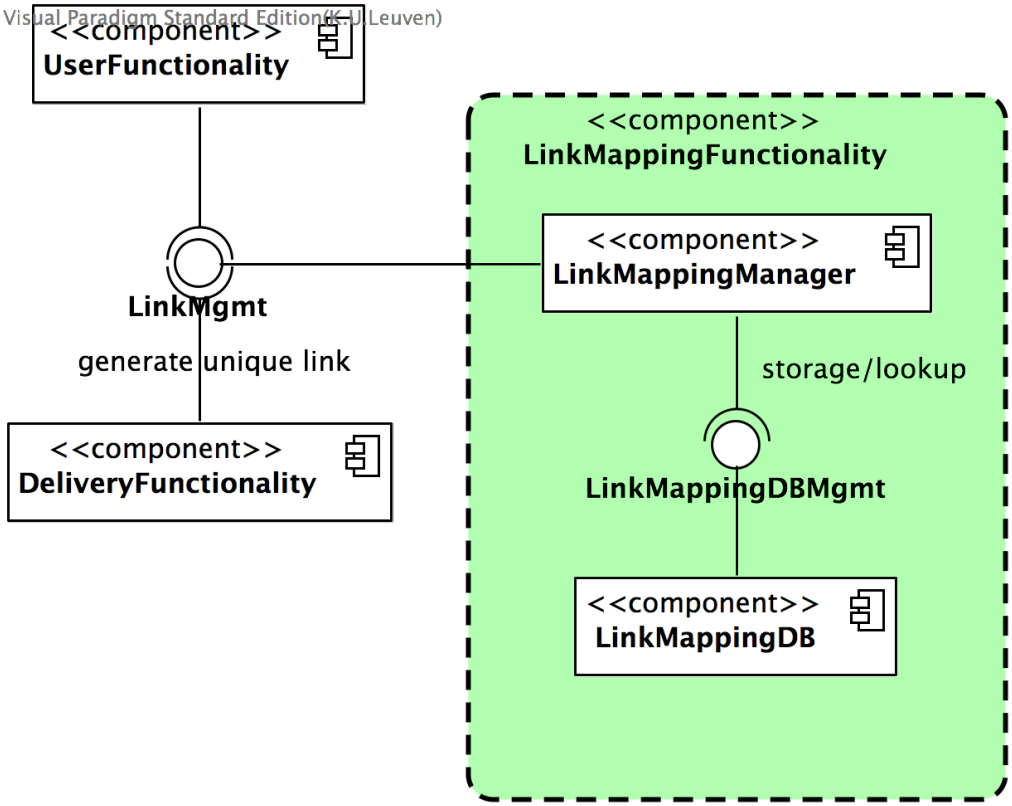

Figure 8: Decomposition of `LinkMappingFunctionality`

The decomposition of the `LinkMappingFunctionality` is presented in Figure 8. As shown, the `LinkMappingFunctionality` is decomposed into only two submodules: the `LinkMappingManager` and the `LinkMappingDB`.

Motivatie voor LinkManager: Checks expiration date ALS DAT NODIG IS–¿ reason: links naar de pdsdb vervallen niet (zolang de gebruik geregistreerd is) LinkManager maps link to (document ID, place where the document is stored)-pairs –¿ REASON: the unique link has two possible sources: an e-mail to an unregistered recipient or an email to a registered recipient. For an unregistered recipient, the RecipientFacade must look with the documentid for the document in the documentDB. For a registered recipient, the RecipientFacade must look with the documentid for the document in the PDSDB. (Mogelijk een boolean ofzo) Does NOT do mapping removal after x years –¿ there has to be a notification when the link has expired

4.7 PDSDB

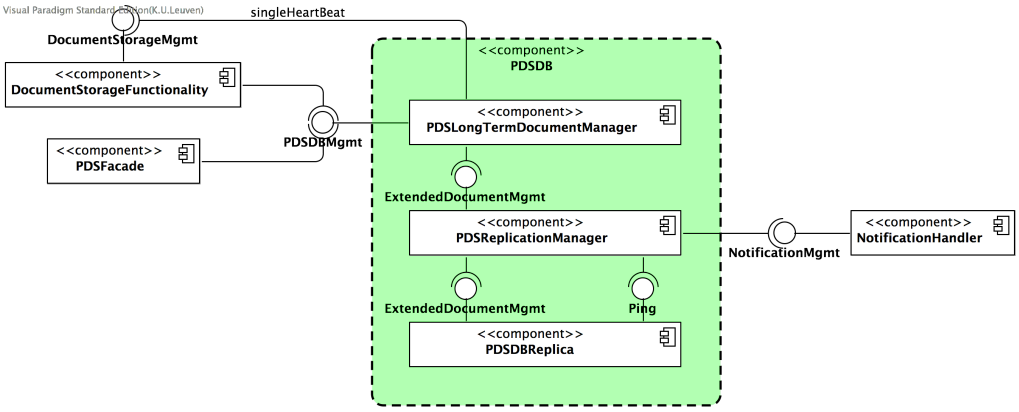

Figure 9: Decomposition of `PDSDB`

The decomposition of the `PDSDB` is presented in Figure 9. As shown, the `PDSDB` is decomposed into three submodules: the `PDSLongTermDocumentManager`, the `PDSReplicationManager` and the `PDSDBReplica`.

Since this decomposition was already discussed in the provided initial architecture, we will not repeat those results here. Note, however, that some interfaces have been changed, added or even removed for the `PDSDB` to be able to properly communicate with the rest of the system (see Appendix A).

Also note that, in order to comply with the design decisions that were made in the light of *Av2b*, the `PDSDB` should notify the `DocumentStorageManager` (see also the decomposition of `DocumentStorageFunctionality` in Section 4.4) upon revival of the former after a period of downtime. This responsibility is delegated to the `PDSLongTermDocumentManager` by having it send a sign of life to the `DocumentStorageManager` upon revival of the `PDSDB`.

Finally, *P2* requires the `PDSDB` to be able to throttle excessive read requests from the `PDSFacade`, through which recipients' `PDSDB` requests are routed, when the arrival rate is larger than a certain value that is specific to the kind of request (determined by the requirements). This responsibility is delegated to the `PDSLongTermDocumentManager`, the only component in the document lookup pipeline that has knowledge of all incoming `PDSDB` requests, to ensure that other `PDSDB` requests are also taken into account in order to not hinder the other functionality of the system. Note that non-excessive recipient requests to the `PDSDB` continue to be handled in a timely fashion during this throttling, as is dictated by *P2*.

4.8 UserFunctionality

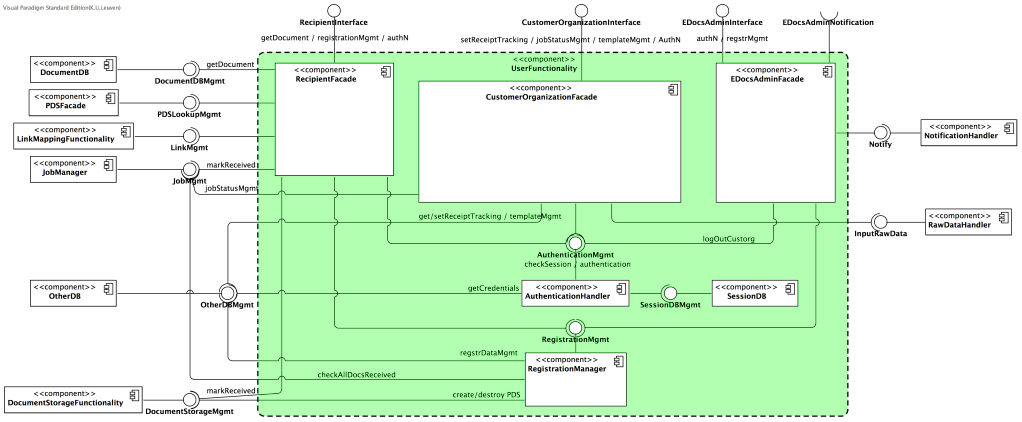

Figure 10: Decomposition of `UserFunctionality`

The decomposition of the `UserFunctionality` is presented in Figure 10. As shown, the `UserFunctionality` is decomposed into six submodules: the `RecipientFacade`, the `CustomerOrganizationFacade`, the `EDocsAdminFacade`,

the `AuthenticationHandler`, the `SessionDB` and the `RegistrationManager`.

5 Deployment view (UML Deployment diagram)

Context diagram Describe the context diagram for the deployment view. For example, which protocols are used for communication with external systems and why?

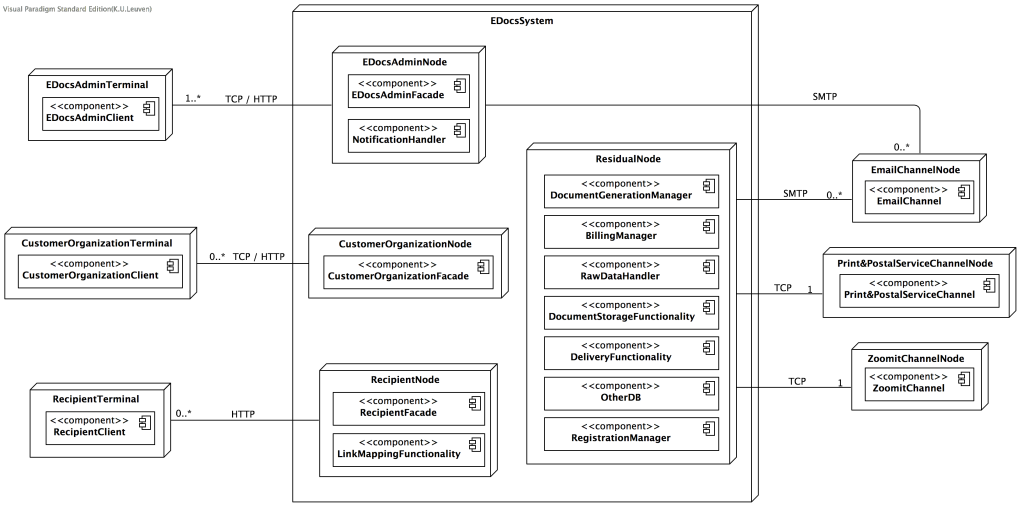

Figure 11: Context diagram for the deployment view.

Primary diagram The primary deployment diagram itself and accompanying explanation. Pay attention to the parts of the deployment diagram which are crucial for achieving certain non-functional requirements. Also discuss any alternative deployments that you considered.

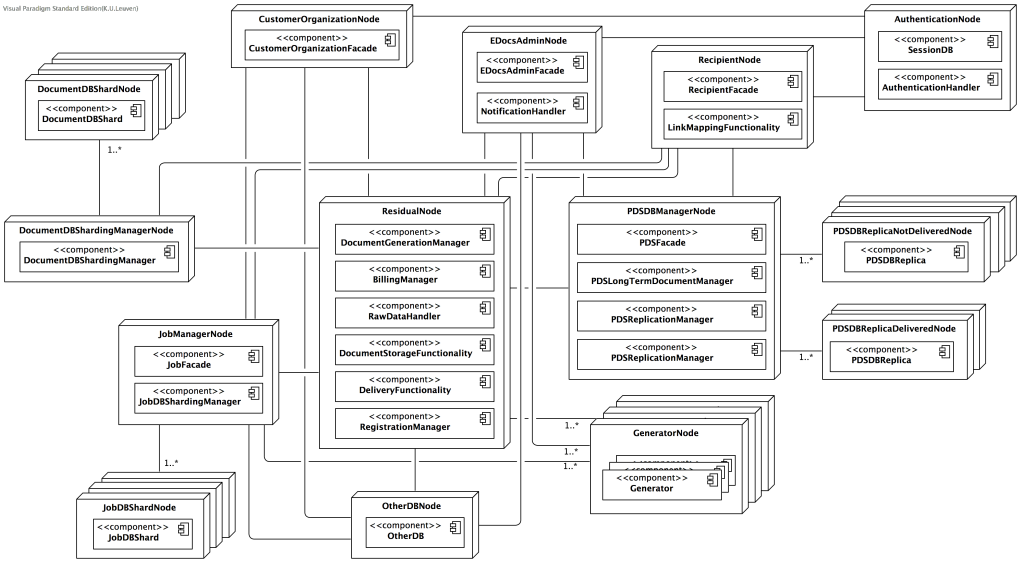

Figure 12: Primary diagram for the deployment view.

6 Scenarios

Illustrate how your architecture fulfills the most important data flows. As a rule of thumb, focus on the scenario of the domain description. Describe the scenario in terms of architectural components using UML Sequence diagrams and further explain the most important interactions in text. Illustrating the scenarios serves as a quick validation of the completeness of your architecture. If you notice at this point that for some reason, certain functionality or qualities are not addressed sufficiently in your architecture, it suffices to document this, together with a rationale of why this is the case according to you. You do not have to further refine you architecture at this point.

6.1 Scenario 1

Shortly describe the scenario shown in this subsection. Show the complete scenario using one or more sequence diagrams.

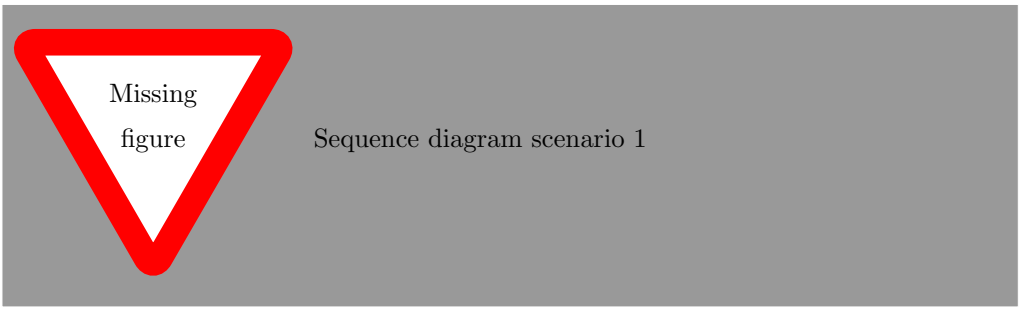

Figure 13: The system behavior for the first scenario.

6.2 A registered recipient logs in

Figure 14 depicts the sequence diagram of a Registered Recipient logging in, according to *UC1: Log in*. The Registered Recipient provides his details to the `RecipientFacade`. The `Authentication` compares these credentials to those stored in the `OtherDB`. if the given credentials match the stored credentials, a new session is opened in the `SessionDB` and the corresponding session identifier is returned to the registered recipient. Otherwise, an InvalidCredentialsException is thrown.
Note that the use case *UC1: Log in* also has the Customer Organization as a primary actor. The login procedure for customer organizations is almost identical to the procedure for registered recipient, which is why we do not provide a separate sequence diagram for this primary actor. Compared to figure 14, the Customer Organization sends it login request to the `CustomerOrganizationFacade` instead of the `RecipientFacade`. Also, another method call to `OtherDB` is used to ask for the credentials of a customer organization.

6.3 A registered recipient logs out

Figure 15 shows the behaviour of a Registered Recipient logging out, according to *UC2: Log out*. The Registered Recipient provides his recipient id and session id to the `RecipienntFacade`. The `AuthenticationHandler` first verifies the session (figure 15). If the session is valid, it logs out the Registered Recipient.
Note that the logout procedure for Customer Organizations is almost the same. The only difference is that the Customer Organization sends his logout request to the `CustomerOrganizationFacade` instead of the `RecipientFacade`. Because of this reason, we do not give a separate sequence diagram for the Customer Organization logging out.

6.4 Updating a document template

Figure 16 shows the flow when a Customer Administrator updates the template used for generating documents of a given type. It corresponds to *UC20: Update document template*. It consists of two main parts. First, the customer administrator asks what the possible document types are that the customer organization is allowed to generate. The system verifies whether the session of the customer administrator is valid (as detailed in figure 17 and then returns a list of the allowed document types using the `OtherDB`. Together with the allowed

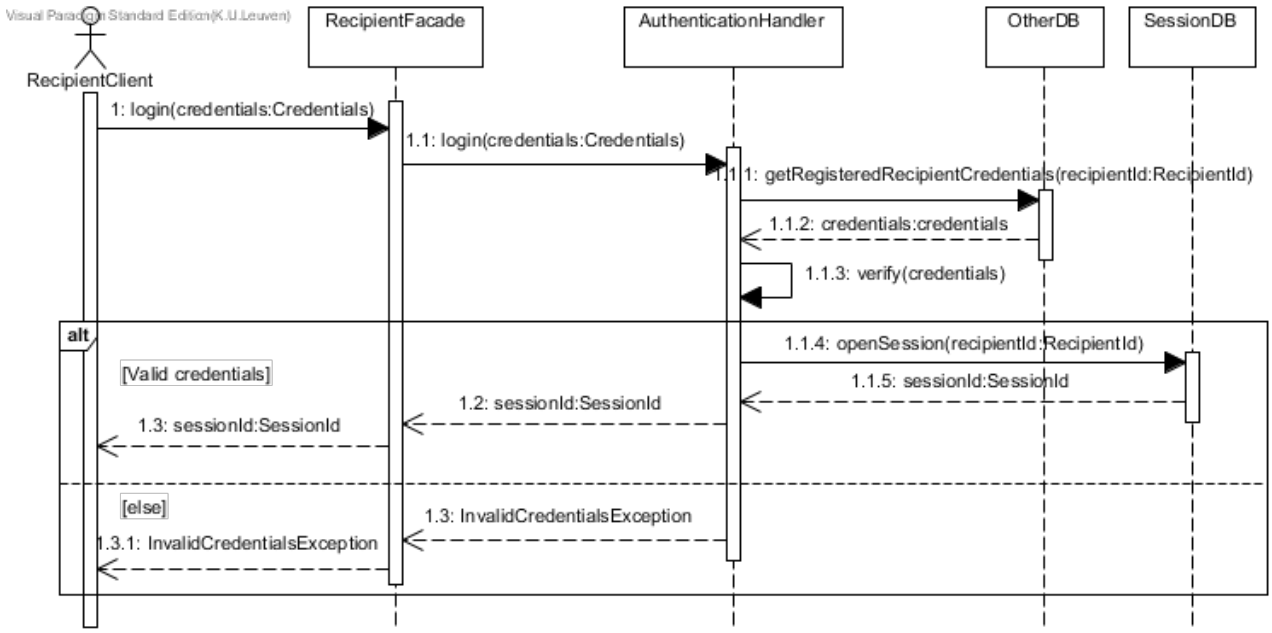

Figure 14: The login behaviour of a Registered Recipient.

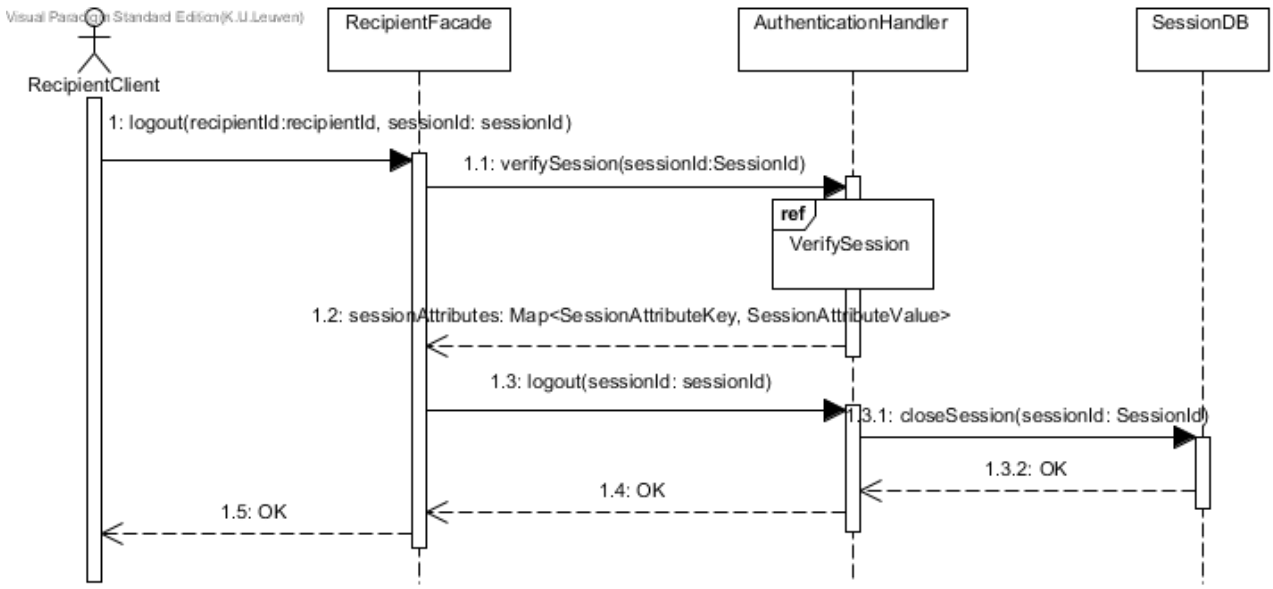

Figure 15: The logout behaviour of the Registered Recipient.

document types, it returns the time when the currently stored templates corresponding to the document types were uploaded.
Secondly, the customer administrator provides the document type it wants to update and the new template to the `CustomerOrganizationFacade`. After verifying the validity of the session again (figure 17), the `CustomerOrganization` asks the `OtherDB` to check whether the document type provided by the customer organization is valid and allowed. If it is not allowed or invalid, an InvalidDocumentTypeException gets thrown. Otherwise, the `CustomerOrganizationFacade` stores the template in the `OtherDB` together with the time when it has received the template, the document type and the customer organization id.

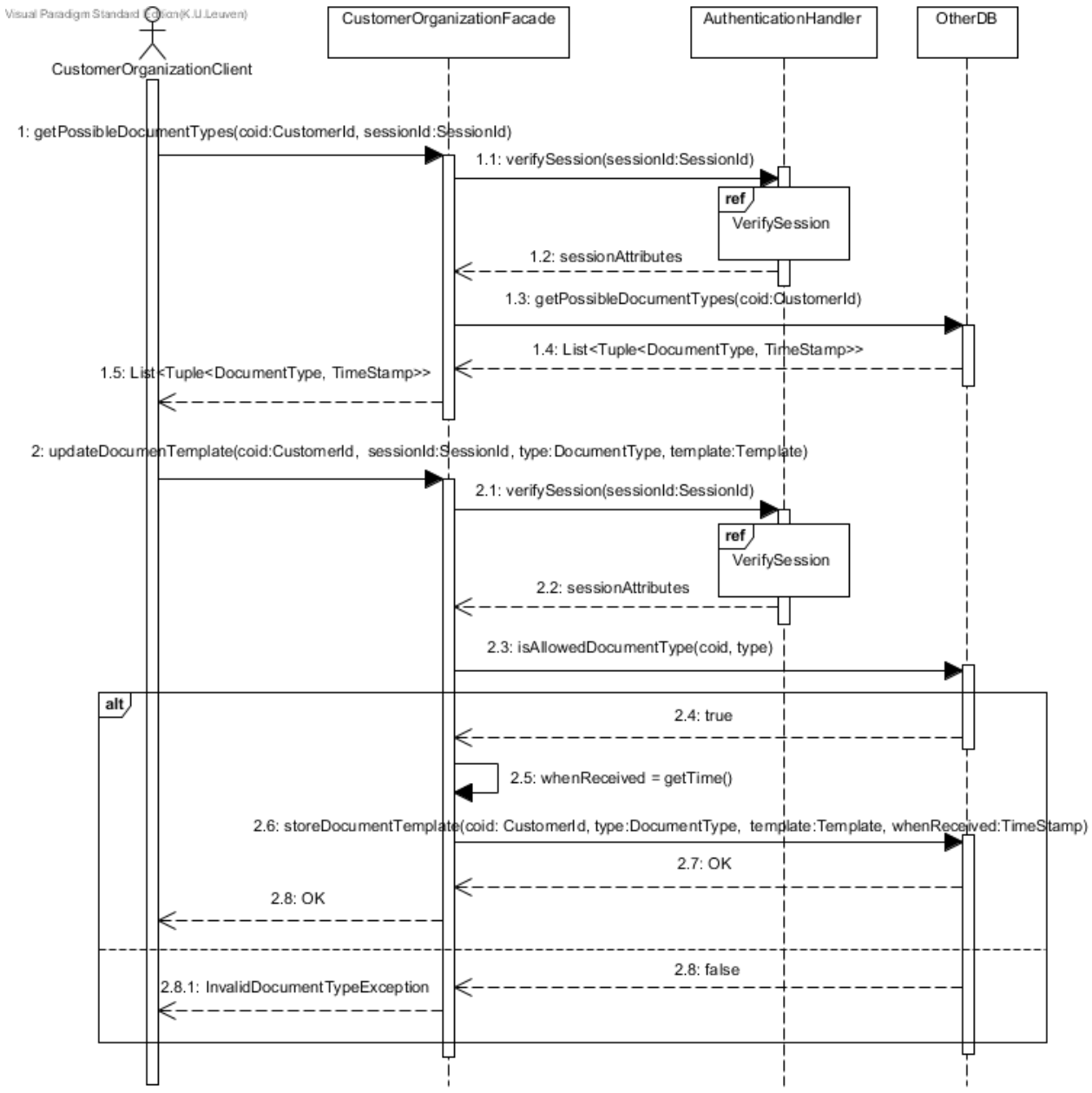

Figure 16: The system behavior for updating a document template.

6.5 Verifying a session

Figure 17 how it is verified whether a session is valid. The `AuthenticationHandler` uses the `SessionDB` to verify whether each incoming session identifier belongs to an existing session. If it belongs to an existing session, the corresponding session attributes are returned to the caller, otherwise an NoSuchSessionException is thrown.

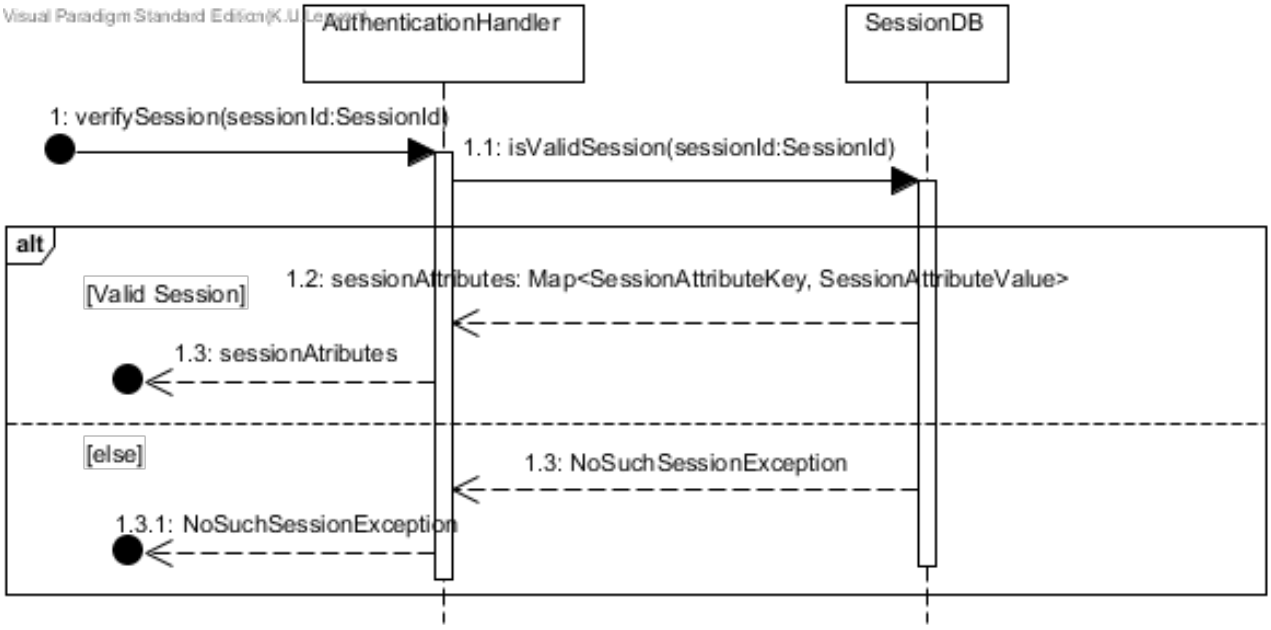

Figure 17: verifySession: sequence diagram depicting the verification whether a session is valid.

6.6 Initiating document processing

Figure 18 shows how a customer organization initiates a batch of document processing jobs. Recurring and non-recurring batches are indicated using the `isRecurring` boolean value. If the batch is non-recurring, the customer organization can indicate another priority than the default priority of that customer organization if the batch should be handled with another priority than the default priority. For recurring batches, it does not matter if a priority is given, as the default priority will be used. The customer organization also delivers the raw data as a `rawDataPackage`. After verification of the session identifier (figure 17), the `CustomerOrganizationFacade` generates a `TimeStamp` of the time when this method is called and forwards this together with all its arguments except for the `sessionid` to `RawDataHandler`, while also indicating if the batch is (non-)recurring.

Figure 19 depicts the first part of the method call in the `RawDataHandler`, which will verify whether the raw data is valid. First, `RawDataHandler` checks whether the customer organization is allowed to generated the given document type. If it is allowed, the received raw data packet will be validated, e.g there will be checked whether the content of the received Excel file can be read or whether a received XML file is correctly formatted. If it is not valid, an exception is thrown. Otherwise, depending on if the batch is recurring, the `RawDataHandler` will query the SLA in the `OtherDB` to check if the number of raw data entries in the raw data package is allowed. If the number of entries exceeds this maximum, an exception is thrown. Next, each individual entry will be checked on validity, so the `RawdataHandler` knows which `RawData` entries are valid and which are not.

After verification, figure 18 shows that the `RawDataHandler` will only use the given priority if the batch is non-recurring. Otherwise, it will use the default priority. The deadline of the document processing jobs is calculated and a `BatchMetaData` object is generated, bundling the information about the batch. The `RawDataHandler` will store the `RawData` entries and `BatchMetaData` in the `OtherDB`. It asks the `JobFacade` to create jobs for the valid raw data entries and to initiate document processing for those jobs. Finally, it will return the invalid raw data entries to the Customer Organization.

Figure 20 shows the `JobFacade` creating and storing jobs for the valid raw data entries, after which it inserts these jobs into the `Scheduler`.

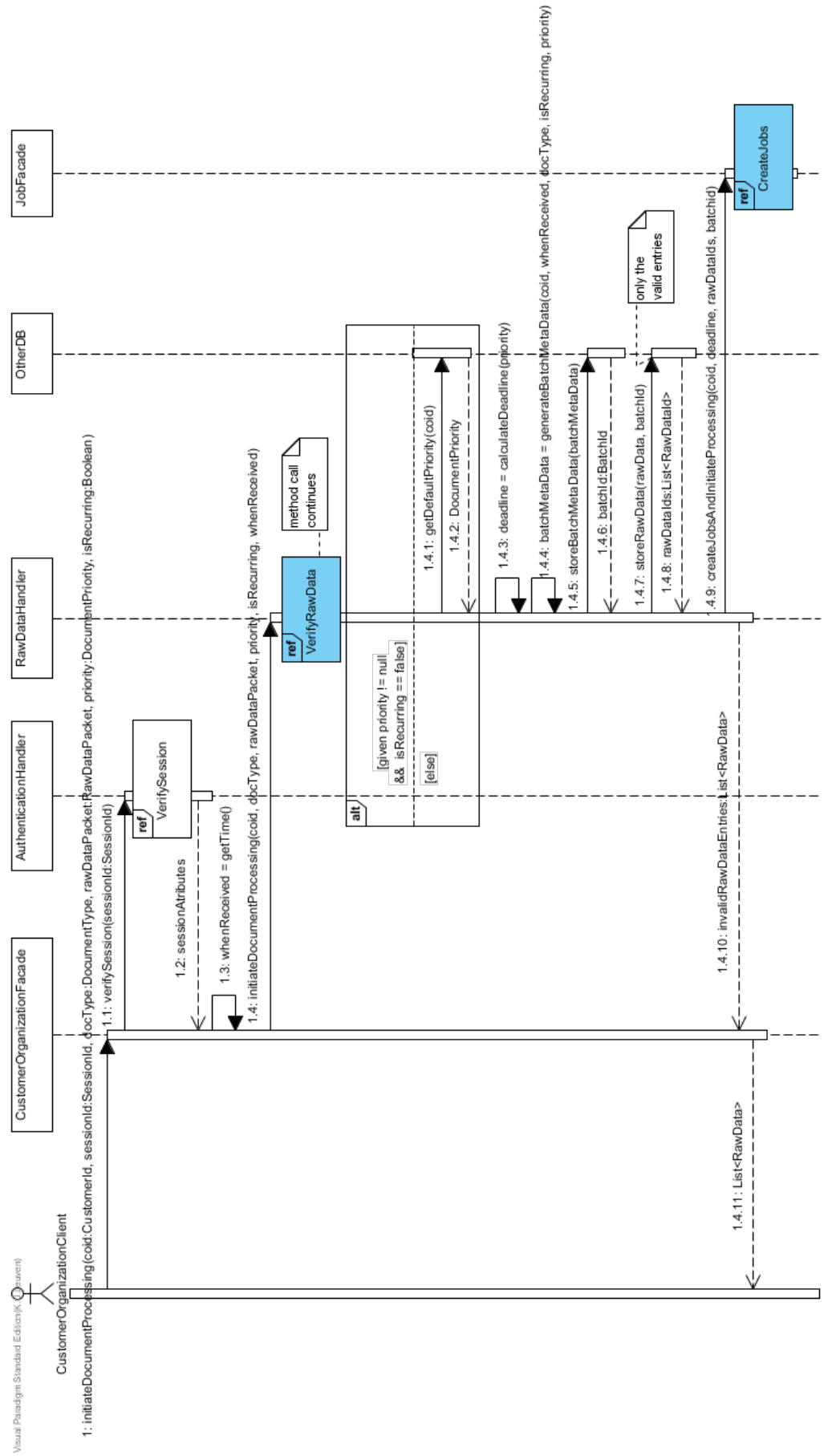

Figure 18: VerifyRawData: verification whether the raw data is valid.

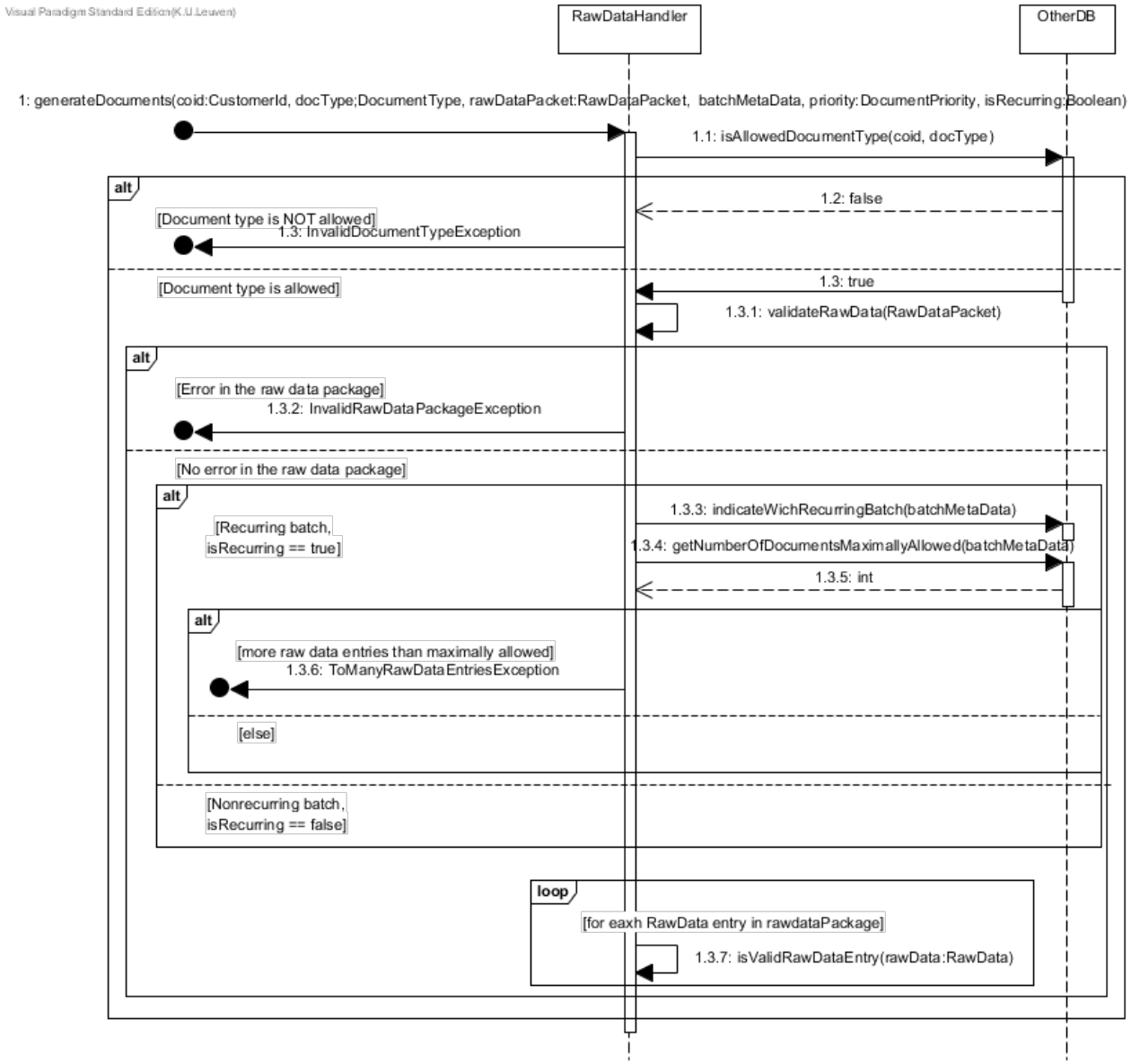

Figure 19: VerifyRawData: verification whether the raw data is valid.

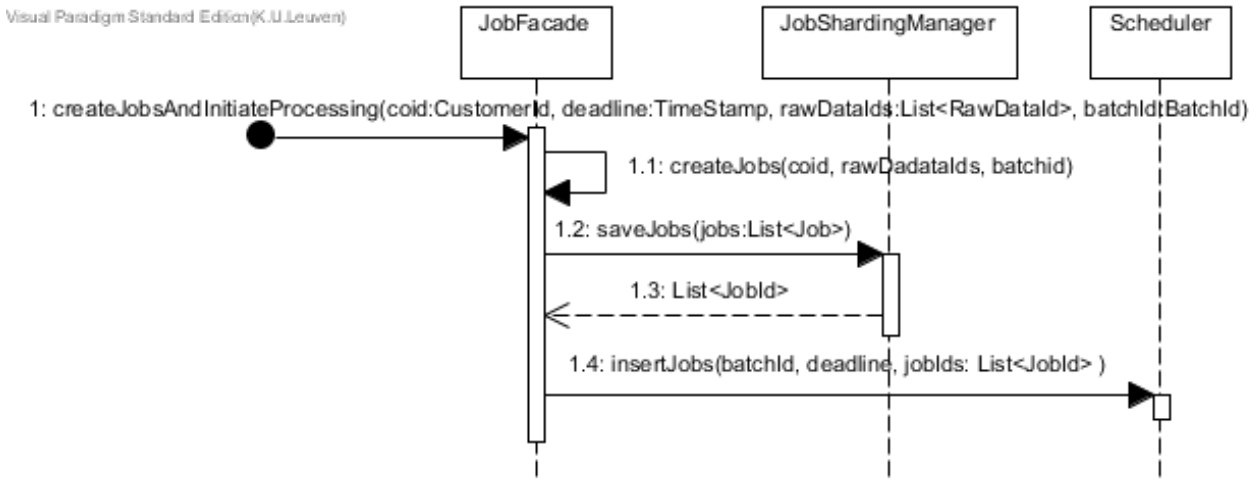

Figure 20: CreateJobs: the `JobManager creates jobs and forwards them to be scheduled by the Scheduler`.

6.7 Consult document in personal document store

Shortly describe the scenario shown in this subsection. Show the complete scenario using one or more sequence diagrams.

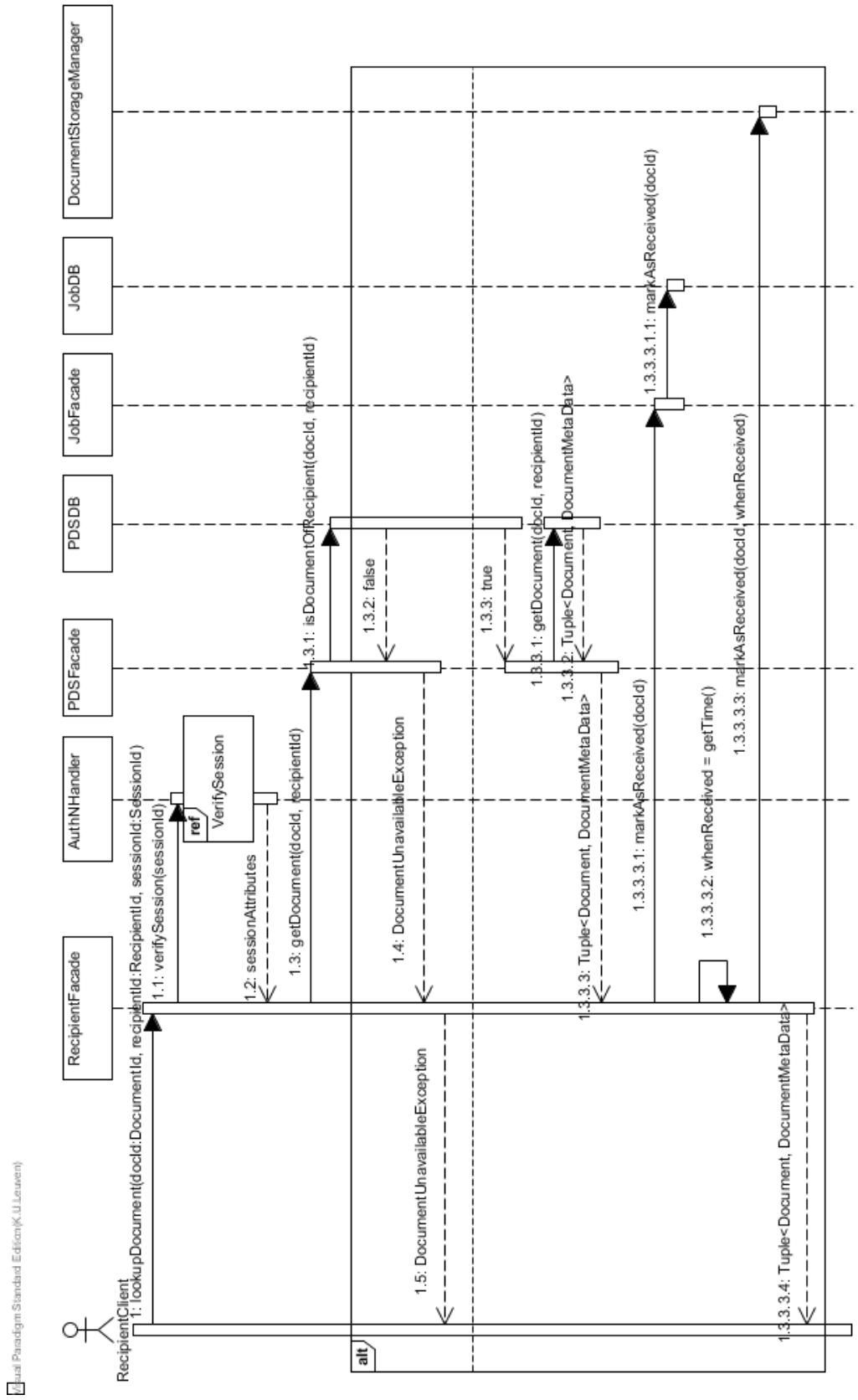

Figure 21: A Registered Recipient consults a document in his or her personal document store.

6.8 Delivering a document via the personal document store

Shortly describe the scenario shown in this subsection. Show the complete scenario using one or more sequence diagrams.

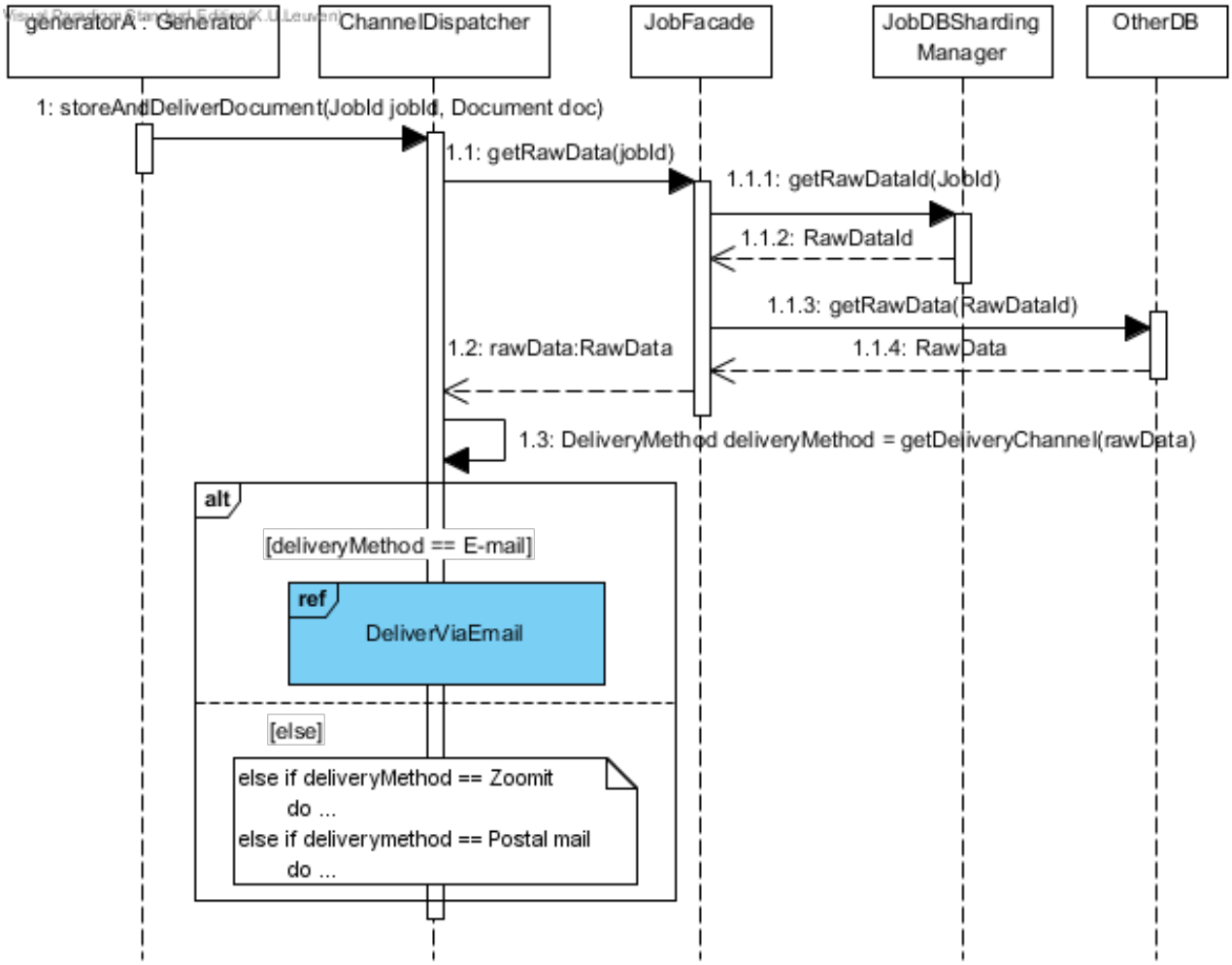

Figure 22: A `Generator` instance asks the `ChannelDispatcher` to store and deliver a document.

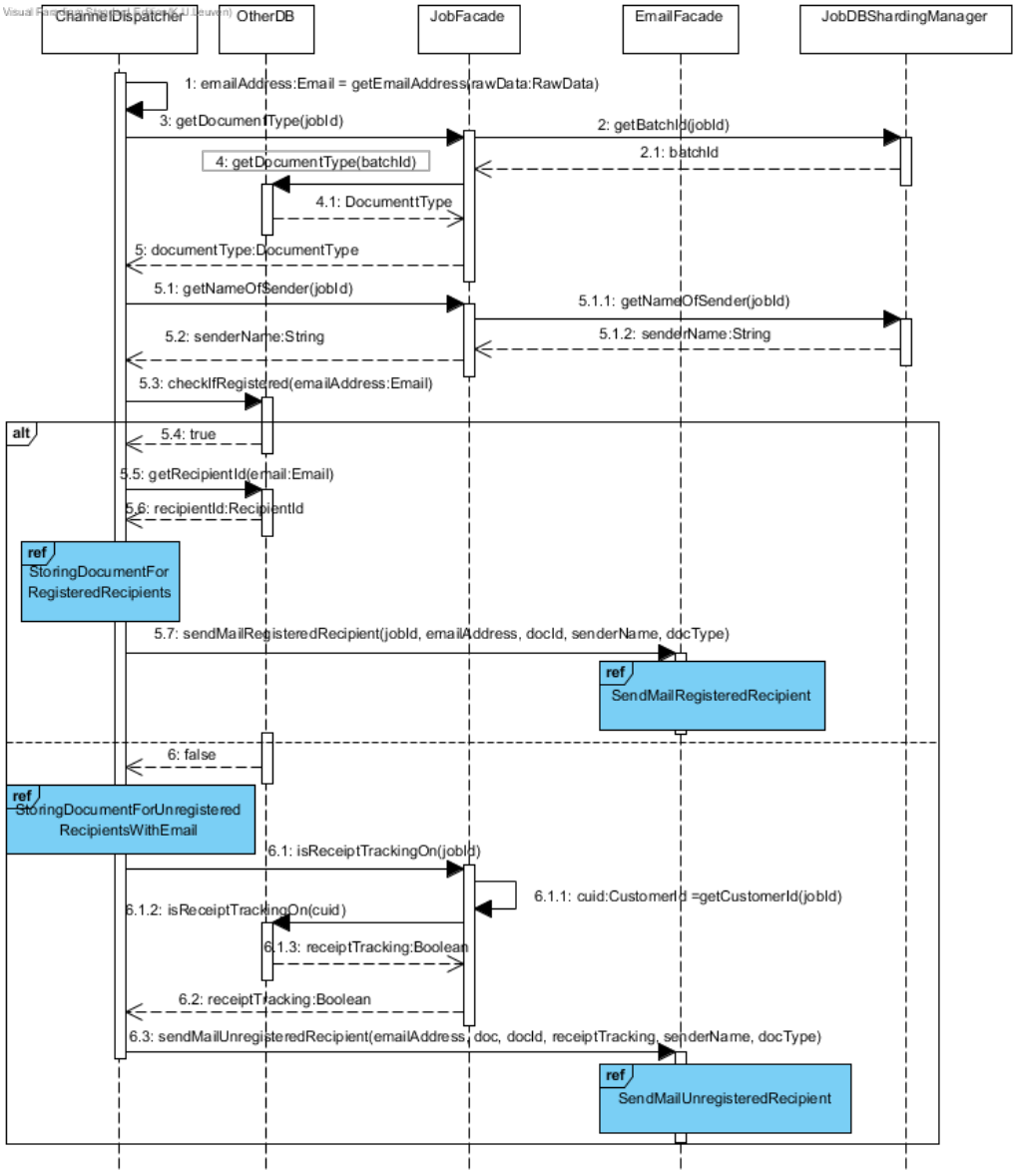

Figure 23: DeliverViaEmail: (Continuation of fig. 22) The `ChannelDispatcher` uses e-mail as a delivery method and will have a different flow when the e-mail should be delivered to a Registered or Unregistered Recipient.

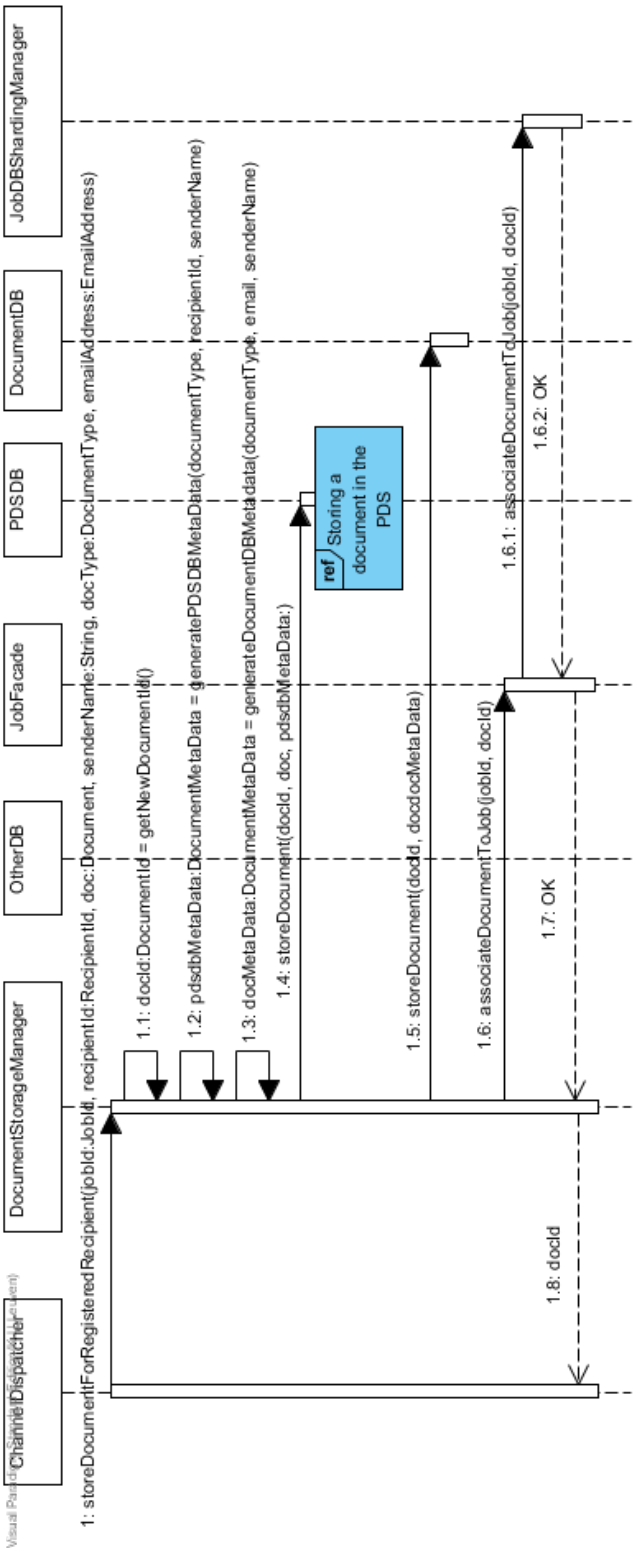

Figure 24: StoringDocumentForRegisteredRecipients: (continuation of fig. 23) The `ChannelDispatcher` requests the `DocumentStorageManager` to store a document for a Registered Recipient.

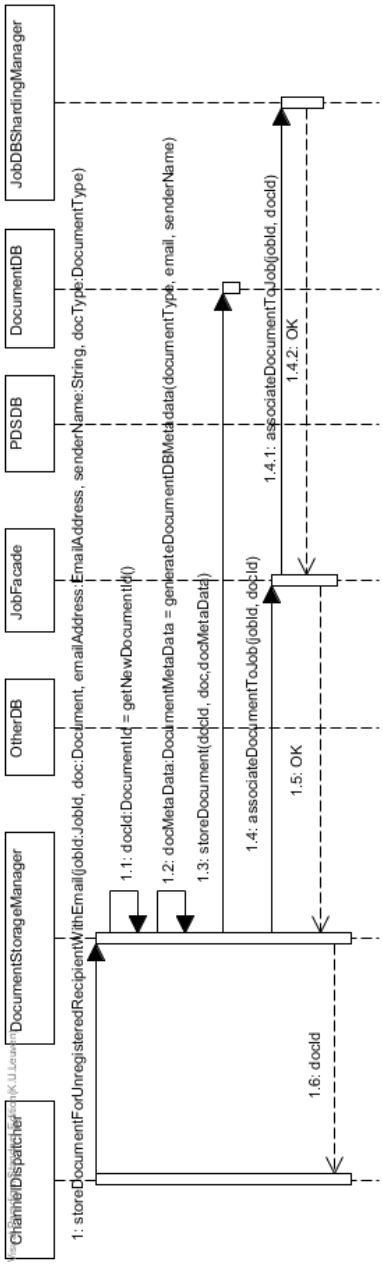

Figure 25: StoringDocumentForUnregisteredRecipientsWithEmail: (continuation of fig. 23) The `ChannelDispatcher` requests the `DocumentStorageManager` to store a document for an Unregistered Recipient.

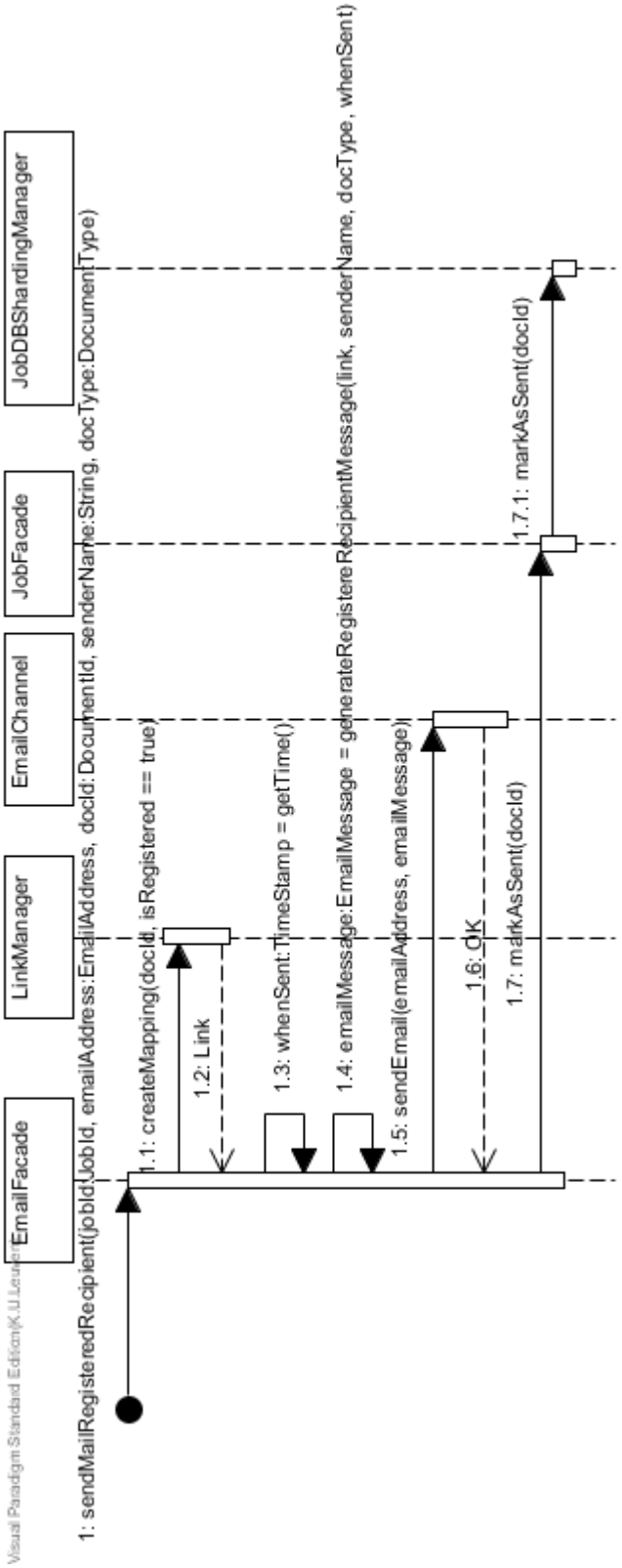

Figure 26: SendMailRegisteredRecipient: (continuation of fig. 23) The `ChannelDispatcher` requests the `EmailFacade` to send an e-mail to a Registered Recipient.

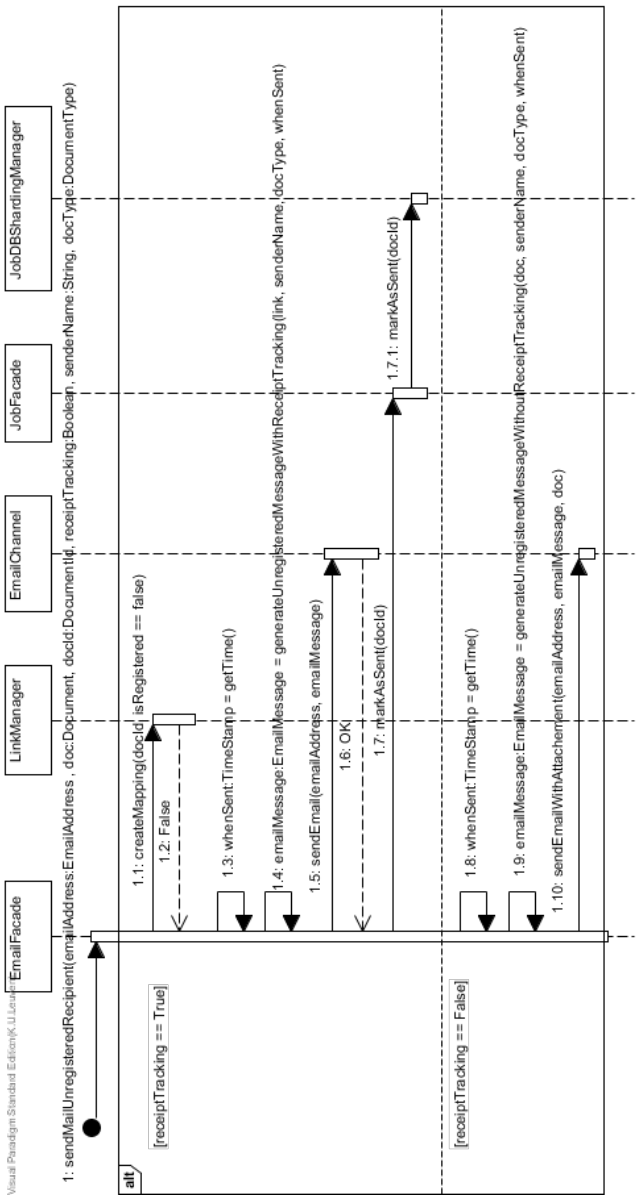

Figure 27: SendMailUnregisteredRecipient: (continuation of fig. 23) The `ChannelDispatcher` requests the `EmailFacade` to send an e-mail to an Unregistered Recipient.

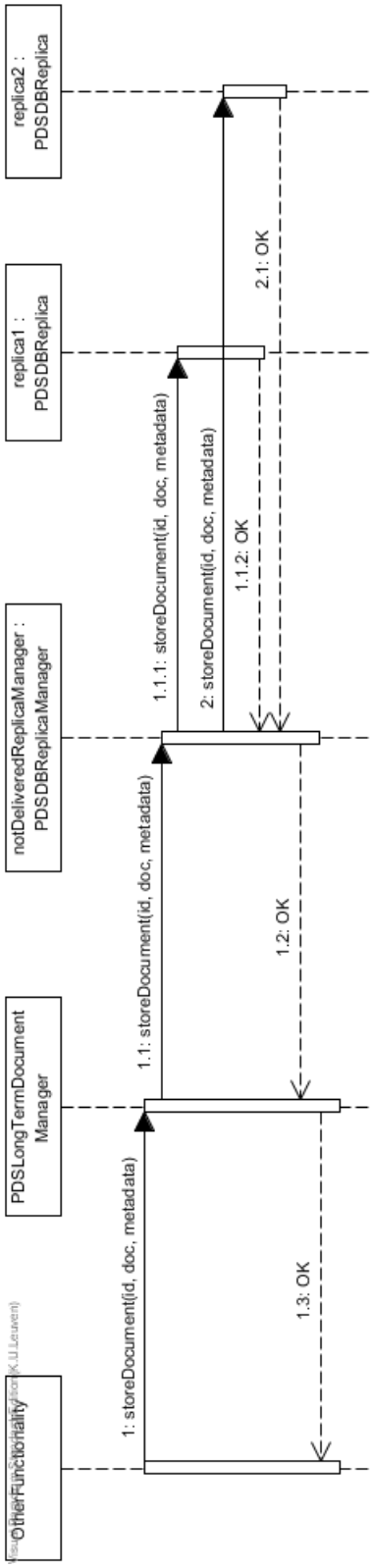

Figure 28: Storing a document in the PDS (continuation of fig. 25).

A Element catalog

In this section, we list all the components and the interfaces they provide. Per component, we describe its responsibilities, declare its super-component (if any) and list its sub-components (if any).

List all components and describe their responsibilities and provided interfaces. Per interface, list all methods using a Java-like syntax and describe their effect and exceptions if any. List all elements and interfaces alphabetically for ease of navigation.

A.1 AuthenticationHandler

- **Description:** The `AuthenticationHandler` is responsible for authenticating Registered recipients and Customer organizations. The architecture does not specify the means of authentication (e.g. the type of credentials). The credentials are stored in the `UserDB`.
- **Super-component:** `UserFunctionality`
- **Sub-components:** None

Provided interfaces

- AuthN
  - `RecipientId getRecipientId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `AuthenticationHandler` fetches and returns the Registered Recipient's identifier corresponding to the `sessionId` from the `SessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifiers, or if the session belongs to a customer organization.
  - `CustomerId getCustomerId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `AuthenticationHandler` fetches and returns the Customer Organization's identifier corresponding to the `sessionId` from the `SessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier, or if the session belongs to a registered recipient.
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `AuthenticationHandler` will remove the session with the given id from the `SessionDB`. If no such session exists, nothing is changed and no exception is thrown.
    - Exceptions: None
  - `SessionId login(Credentials credentials) throws InvalidCredentialsException`
    - Effect: The `AuthenticationHandler` verifies the `credentials` using the `UserDB`. If they are correct, the `AuthenticationHandler` creates a new session using the `SessionDB`, stores the id of the user (i.e. the Registered Recipient id or Customer Organization id) as an attribute in this session and returns the id of the new session. The id of the user is present in the given credentials.
    - Exceptions:
      - InvalidCredentialsException: Thrown if the given credentials are invalid.
- CheckSession
  - `Map<SessionAttributeKey, SessionAttributeValue> verifySession(SessionId sesionId) throws NoSuchsSessionException`
    - Effect: The `AuthenticationHandler` verifies whether a session with the given id exists in the `SessionDB` and if so, returns all its associated attributes.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifiers.

A.2 BillingManager

- **Description:** The `BillingManager` is responsible for all billing tasks. This includes billing the Customer Organization for the generation and delivery of non-recurring document processing jobs.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- CostRegistrationMgmt
  - `void addDocumentGenerationCost(CustomerId cuId, JobId jobId, DocumentPriority priority)`
    - Effect: The `BillingManager` adds the cost for generation documents corresponding to the job identified by `jobId` to the bill of the customer organization identified by `cuId`. The document had a priority `priority`. This method is called by the `GenerationManager`.
    - Exceptions: None
  - `void addDocumentDeliveryCostbyEmail(CustomerId cuId, JobId jobId)`
    - Effect: The `BillingManager` adds the cost for delivering the document by e-mail to the bill of the customer organization identified by `cuId`. The document corresponds to the job identified by `JobId`. This method is called by the `ChannelDispatcher`.
    - Exceptions: None
  - `void addDocumentDeliveryCostbyPrintAndPostalService(CustomerId cuId, JobId jobId)`
    - Effect: The `BillingManager` adds the cost for delivering the document by e-mail to the bill of the customer organization identified by `cuId`. The document corresponds to the job identified by `JobId`. This method is called by the `ChannelDispatcher`.
    - Exceptions: None
  - `void addDocumentDeliveryCostbyZoomit(CustomerId cuId, JobId jobId)`
    - Effect: The `BillingManager` adds the cost for delivering the document by Zoomit to the bill of the customer organization identified by `cuId`. The document corresponds to the job identified by `JobId`. This method is called by the `ChannelDispatcher`.
    - Exceptions: None

A.3 ChannelDispatcher

- **Description:** The `ChannelDispatcher` is responsible for choosing the correct delivery channel for a generated document. It also forwards the document to the `DocumentStorageManager`, which will store the document.
- **Super-component:** `Deliveryfunctionality`
- **Sub-components:** None

Provided interfaces

- FinalizeDocument
  Note that the methods in this interface are made idempotent. The methods of this interface are called by `Generator` instances.
  - `void storeAndDeliverDocument(JobId jobid, Document doc)`
    - Effect: The `ChannelDispatcher` will store the given document `document` and deliver it. This method is made idempotent. To filer duplicate method calls, it has the `JobId` of the document as an argument. This idempotence is to account for the case when case when a `Generator` fails after forwarding the document and before reporting completion to the `DocumentGenerationManager`. In this case, it can be that the `DocumentGenerationManager` restarts jobs for which a document has already been stored or delivered.
    - Exceptions: None

- `void generationError(JobId jobid, Error error)`
  - Effect: Describe the effect of the operation
  - Exceptions: None

A.4 CustomerOrganizationClient

- **Description:** The `CustomerOrganizationClient` is external to the eDocs system and represents a client device of a Customer Organization (i.e. Customer Administrator and Customer Information System) that communicates with the eDocs System.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.5 CustomerOrganizationFacade

- **Description:** The `CustomerOrganizationFacade` provides the main interface of the system to the Customer Organization (i.e. Customer Administrator and Customer Information System).
- **Super-component:** `UserFunctionality`
- **Sub-components:** None

Provided interfaces

- AuthN
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `AuthenticationHandler` will remove the session with the given id from the `SessionDB`. If no such session exists, nothing is changed and no exception is thrown.
    - Exceptions: None
  - `SessionId login(Credentials credentials) throws InvalidCredentialsException`
    - Effect: The `AuthenticationHandler` verifies the `credentials` using the `UserDB`. If they are correct, the `AuthenticationHandler` creates a new session using the `SessionDB`, stores the id of the user (i.e. the Registered Recipient id or Customer Organization id) as an attribute in this session and returns the id of the new session. The id of the user is present in the given credentials.
    - Exceptions:
      - InvalidCredentialsException: Thrown if the given credentials are invalid.
- TemplateMgmt
  - `List<Tuple<DocumentType,TimeStamp>> getPossibleDocumentTypes(SessionId sessionId, CustomerId cuId) throws NotAuthenticatedException`

    * Effect: The `CustomerOrganizationClient` will return a list of the document types that the customer organization is allowed to generate. It also indicates the current template for each document type by returning for each document type the time when the last template was uploaded.
    * Exceptions:
      · NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `Boolean updateDocumentTemplate(SessionId sessionId, CustomerId cuId, DocumentType documentTyp Template template) throws NotAuthenticatedException, InvalidDocumentTypeException`
    * Effect: The `CustomerOrganizationClient` will update the current template for the given `documentType` to the given `template` for the customer organization identified by `cuId`. Returns true when it succeeds.
    * Exceptions:
      · NotAuthenticatedException: Thrown if the given session identifier is invalid.
      · InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.

- InitiateDocumentProcessing
  - `List<RawData> initiateDocumentProcessing(CustomerId cuid, SessionId sessionId, DocumentType docType, RawDataPacket rawDataPacket, DocumentPriority priority, Boolean isRecurring) throws NotAuthenticatedException, InvalidDocumentTypeException, InvalidRawDataPackageException, ToManyRawDataEntriesException`
    * Effect: The `CustomerOrganizationFacade` gets an indication from a customer organization with customer identifier `cuid` to start a batch of document processing jobs. The documents to be generated should be of document type `docType`. The boolean `isRecurring` indicates whether the batch is recurring or non-recurring. The information necessary to generate the documents is given in a `RawDataPacket` containing all the info about the raw data entries of the batch. After verification of the session identifier, the `CustomerOrganizationFacade` generates a `TimeStamp` of the time when this method is called and forwards it together with most of its arguments to the `RawDataHandler`.
    * Exceptions:
      · NotAuthenticatedException: Thrown if the given session identifier is invalid.
      · InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.
      · InvalidRawDataPackageException: Thrown if the raw data package contains an error (e.g. an invalid Excel file or incorrectly formatted XML).
      · ToManyRawDataEntriesException: Thrown if the batch is a recurring batch and contains more raw data entries than maximally allowed for this batch.

A.6 Completer

- **Description:** The `Completer` is responsible for fetching the raw data an applicable meta-data for a group of `JobIds` when a `Generator` instance requires a new group of jobs.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- Complete
  - `CompletePartialBatchData getComplete(BatchId batchId, List<JobId> jobIds )`
    * Effect: The `Completer` fetches data needed by a `Generator` for generation of the documents corrseponding to the`JobIds` belonging to the same batch, which is identified by `BatchId`.
    * Exceptions: None

A.7 DocumentDB

- **Description:** The `DocumentDB` is responsible for actually storing all the documents. It stores documents regardless of the fact that these documents are also stored in the PDSDB. It receives read and write requests from the `DocumentStoragManager`.
- **Super-component:** None
- **Sub-components:** `DocumentDBShardingManager` and `DocumentDBShard`

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.8 DocumentDBShard

- **Description:** A `DocumentDBShard` is responsible for storing a partition of all the documents.
- **Super-component:** `DocumentDB`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.9 DocumentDBShardingManager

- **Description:** The `DocumentDBShardingManager` manages the storage of the documents over multiple `DocumentDBShards`.
- **Super-component:** The `DocumentDB`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.10 DocumentGenerationManager

- **Description:** The `DocumentGenerationManager` monitors the availability of the `Generator` components using the Ping interface. The `DocumentGenerationManager` keeps track of the jobs assigned to and being processed by the `Generators`. To minimize the overhead of the job coordination, the `DocumentGenerationManager` assigns jobs to the `Generators` in groups of more than one job that are part of the same batch. If a `Generator` fails to complete its jobs, the `DocumentGenerationManager` can restart these failed jobs.
  It prioritizes jobs based on thei deadlines ansd schedules them according to *P1*.
- **Super-component:** None
- **Sub-components:** `Completer`, `GenerationManager`, `KeyCache`, `Scheduler`, `TemplateCache`

Provided interfaces

- InsertJobs
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
- NotifyCompleted
  - `void notifyCompletedAndGiveMeMore(GeneratorId id)`
    - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
    - Exceptions: None
  - `void notifyCompletedAndIAmShuttingDown(GeneratorId id)`
    - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
    - Exceptions: None

A.11 DocumentStorageCache

- **Description:** The `DocumentStorageCache` is responsible for storing the `DocumentIds` and `RecipientIds` when the `PDSDB` fails. According to Av2, the system should temporarily store at least 3 hours of documents to be delivered via the personal document store. When the `PDSDB` fails, the documents that are supposed to also be saved in the `PDSDB` are saved in the `DocumentDB`, just as usual. But in this case the `DocumentStorageManager` also stores the `DocumentIds` and `RecipientIds` of those documents in the `DocumentStorageCache` for at least 3 hours. This way, the `DocumentStorageManager` can transfer these documents from the `DocumentDB` to the `PDSDB` using this information if the `PDSDB` comes back online within 3 hours. The requirements do not specify what happens after 3 hours, so in this architecture, the behaviour after those 3 hours is undefined.
- **Super-component:** `DocumentStorageFunctionality`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.12 DocumentStorageFunctionality

- **Description:** The `DocumentStorageManager` is responsible for storing the generated documents in the correct database. If the document belongs to a registered recipient, it sends a write request to both the `DocumentDB` and the `PDSDB`. Otherwise, it only sends a write request to the `DocumentDB`, which stores all the documents.
  It is also responsible for copying documents from the `DocumentDB` to the `PDSDB` when an Unregistered Recipient registers to the eDocs system.
  Another responsibility of the `DeliveryFunctionality` is storing at least 3 hours of documents when the `PDSDB` fails.
- **Super-component:** None
- **Sub-components:** `DocumentStorageCache` and `DocumentStorageManager`

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation

        * Exceptions: None

- InterfaceB
    - `returntType2 operation3()`
        * Effect: Describe the effect of the operation
        * Exceptions: None

A.13 DocumentStorageManager

- **Description:** The `DocumentStorageManager` is responsible for storing the generated documents in the correct database. If the document belongs to a registered recipient, it sends a write request to both the `DocumentDB` and the `PDSDB`. Otherwise, it only sends a write request to the `DocumentDB`, which stores all the documents.
  It is also responsible for copying documents from the `DocumentDB` to the `PDSDB` when an Unregistered Recipient registers to the eDocs system.
  Another responsibility of the `DocumentStorageManager` is storing at least 3 hours of documents when the `PDSDB` fails.
- **Super-component:** `DocumentStoragefunctionlity`
- **Sub-components:** None

Provided interfaces

- DocumentStorageMgmt
    - `void markAsReceived(DocumentId documentId, TimeStamp whenReceived)`
        * Effect: The `DocumentStorageManager` marks the document identified by the given document identifier as received, by storing the given time in the document meta-data of the document. The given time is the time when the recipient should have received the document. Note that this method only has effect the first time it is called for a valid document identifier and time stamp.
        * Exceptions: None

A.14 DeliveryFunctionality

- **Description:** The `DeliveryFunctionality` is responsible for delivering the documents generated by eDocs to the recipients.
- **Super-component:** `ChannelDispatcher`, `EmailFacade`, `Print&PostalServiceFacade`, `ZoomitFacade` and `ZoomitDeliveryCache`
- **Sub-components:** the direct sub-components, if any.

Provided interfaces

- InterfaceA
    - `returntType1 operation1(ParamType param) throws SomeException`
        * Effect: Describe the effect of the operation
        * Exceptions:
            · SomeException: Describe when the exception is thrown.
- InterfaceB
    - `returntType2 operation3()`
        * Effect: Describe the effect of the operation
        * Exceptions: None

A.15 EDocsAdminClient

- **Description:** The `EDocsAdminClient` is external to the eDocs system and represents a client device of an administrator of eDocs that communicates with the eDocs System.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.16 EmailChannel

- **Description:** The `EmailChannel` is responsible for the delivery emails. It is external to the eDocs system and represents a mail server of an e-mail provider.
- **Super-component:** None
- **Sub-components:** None.

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.17 EmailFacade

- **Description:** The `EmailFacade` is responsible for creating and sending emails used in the delivery of documents. It will send documents to Unregistered recipients by e-mail when receipt tracking is turned off. When receipt tracking is turned on for an Unregistered Recipient, it will send an e-mail containing a short description of the received document and a unique link, which can be followed to get document. For Registered Recipients, it will send an e-mail containing a short description of the document and a link to the document. It also marks jobs as sent using the `JobManager`.
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

Provided interfaces

- Deliver
  - `void sendMailRegisteredRecipient(Jobid jobId, EmailAddress emailAddress, DocumentId docId, String senderName, DocumentType docType)`
    - Effect: The `EmailFacade` will send an e-mail to the Registered Recipient with a notification that a document corresponding to the job identified by `jobId` has been added to his or her personal document store. The e-mail will contain a short description of the received document (i.e., the sender of the document (`senderName`), the `documentType` of the document and the date when the document was sent). The e-mail will also contain a unique link pointing to the document in the personal document store. After sending the e-mail to the given `emailAddress`, the `EmailFacade` will mark the document as sent.
    - Exceptions: None
  - `void sendMailUnregisteredRecipient(EmailAddress emailAddress, Document document, DocumentId documentId, Boolean receiptTracking)`
    - Effect: If `receiptTracking` is true, receipt tracking is turned on. The `EmailFacade` will send an e-mail to the Unregistered Recipient, which will contain a short description of the received document (i.e., the sender of the document (`senderName`), the `documentType` of the document and the date when the document was sent). The e-mail will also contain a unique link that can be used to download the document. After sending the e-mail to the given `emailAddress`, the `EmailFacade` will mark the document as sent.
      If receipt tracking is turned of, the `EmailFacade` will send the document as an attachment with an e-mail.
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.18 Generator

- **Description:** A `Generator` generates the documents and forwards them to `DeliveryFunctionality` to store and deliver them. Its availability is monitored by the `DocumenGenerationManager` with the Ping interface. A `Generator` is also responsible of notifying the `NotificationHandler` that it does not have all of the data required to fill in the template.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- AssignJobs
  - `void assignJobs(CompletePartialBatchData batchData)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
- Startup/ShutDown
  - `void startUp(GeneratorId generatorId)`
    - Effect: Starts up the `Generator` instance an gives it the given `GeneratorId`.
    - Exceptions: None
  - `void shutDown()`
    - Effect: The `Generator` completes its assigned group of document generation jobs and report back completion to the `DocumentGenerationManager`, after which is shuts down.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `Generator` will respond to the ping request by sending an echo response. This is used by the `GeneratorManager` to check whether the `Generator` is available.
    - Exceptions: None

A.19 GeneratorManager

- **Description:** The `GenerationManager` is responsible for monitoring the `Generator` instances. It starts up or shuts down these instances based on the number of required instances indicated by the `Scheduler`.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- NotifyCompleted
  - `void notifyCompletedAndGiveMeMore(GeneratorId id)`
    - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
    - Exceptions: None

A.20 JobDBShard

- **Description:** The `JobDBShard` is responsible for storing partition of all the jobs.
- **Super-component:** `JobManager`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.21 JobDBShardingManager

- **Description:** The `JobDBShardingManager` manages the storage of the documents over multiple `JobDBShards`.
- **Super-component:** `JobManager`
- **Sub-components:** None

Provided interfaces

- JobDBMgmt
  - `RawData getRawData(JobId jobId)`
    - Effect: The `JobDBShardingManager` returns the raw data entry corresponding to the job identified by the given job identifier.
    - Exceptions: None
  - `void markAsReceived(DocumentId documentId) throws NoSuchJobExceptionException`
    - Effect: The `JobDBShardingManager` marks the job corresponding to the document identified by the given document identifier as received by the recipient.
    - Exceptions:
      - NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to a document identified by the given document identifier.
  - `BatchId getBatchId(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobDBShardingManager` returns the batch identifier of the batch corresponding to the job identified by the given job identifier.
    - Exceptions:
      - NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job associated to the given job identifier.
  - `CustomerID getCustomerId(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobDBShardingManager` returns the customer organization identifier of the customer organization corresponding to the job identified by the given job identifier.
    - Exceptions:
      - NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job associated to the given job identifier.

A.22 JobFacade

- **Description:** The `JobFacade` is responsible creating jobs and storing them using the `JobDBShadingManager` over different `JobDBShards`. It is responsible for retrieving data connected to a specific job. It can retrieve the raw data or customer organization info for specific jobs. The `JobFacade` is also used for marking jobs as sent and received.
- **Super-component:** `JobManager`
- **Sub-components:** None

Provided interfaces

- SetStatus
  - `void setJobStatusAsTemporarilyFailed(List<JobId> statusesOfJobs)`
    - Effect: The `JobManager` marks the job as "temporarily failed" for each of the jobs identified by the given `JobIds`. Used by the `DocumentGenerationManager` for jobs that where assigned to a failed `Generator` instance.
    - Exceptions: None
- GetBatchData
  - `RawData getRawData(JobId jobId)`
    - Effect: The `JobFacade` returns the raw data entry corresponding to the job identified by the given job identifier.
    - Exceptions: None
  - `Tuple<JobId, RawData> getRawData(List<JobId> jobIds)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
  - `BatchMetaData getBatchMetaData(JobId jobId)`
    - Effect: The `JobFacade` returns the meta-data of the batch of the document processing job identified by the given job identifier.
    - Exceptions: None
  - `DocumentType getDocumentType(JobId jobId)`
    - Effect: The `JobFacade` returns the document type of the document corresponding to the job identified by the given job identifier.
    - Exceptions: None
  - `String getNameOfSender(JobId jobId)`
    - Effect: The `JobFacade` returns the name of the sender of the document corresponding to the job identified by the given job identifier.
    - Exceptions:
      - NoSuchJobException: Thrown if the `JobFacade` cannot find a job identified by the given job identifier.
- JobMgmt
  - `void createJobsAndInitiateProcessing(Customerid cuid, TimeStamp deadline, List<rawdataId> rawDataIds, batchId batchId)`
    - Effect: The `JobFacade` creates a job for each of the raw data identifiers it gets as an argument. The job contains the time of deadline for the document processing job and the batch identifier. It stores the jobs and inserts the jobs in the `Scheduler`, together with their deadline and batch identifier.
    - Exceptions: None
  - `void markAsReceived(DocumentId documentId) throws NoSuchJobExceptionException`

* Effect: The `JobFacade` marks the job corresponding to the document identified by the given document identifier as received by the recipient.
* Exceptions:
  · NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to a document identified by the given document identifier.

– `Boolean isReceiptTrackingOn(JobId jobId) throws NoSuchJobExceptionException`
* Effect: The `JobFacade` queries the `JobDBShardingManager` for the customer organization identifier belonging to the job identified by `jobId` and uses it to ask the `OtherDB` whether that customer organization has reciept tracking enabled. The `JobFacade` returns true when the customer organization has receipt tracking enabled and returns false otherwise.
* Exceptions:
  · NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to the given job identifier.

A.23 JobManager

- **Description:** The `JobManager` is responsible creating jobs and storing them. It is responsible for retrieving data connected to a specific job. It can retrieve the raw data or customer organization info for specific jobs. The `JobFacade` is also used for marking jobs as sent and received.
- **Super-component:** None
- **Sub-components:** `JobFacade`, `JobDBShardingManager` and `JobDBShard`

Provided interfaces

- SetStatus
  – `void setJobStatusAsTemporarilyFailed(List<JobId> statusesOfJobs)`
    * Effect: The `JobManager` marks the job as "temporarily failed" for each of the jobs identified by the given `JobIds`. Used by the `DocumentGenerationManager` for jobs that where assigned to a failed `Generator` instance.
    * Exceptions: None
- GetBatchData
  – `RawData getRawData(JobId jobId)`
    * Effect: The `JobManager` returns the raw data entry corresponding to the job identified by the given job identifier.
    * Exceptions:
      · NoSuchJobException: Thrown if the `JobManager` cannot find a job identified by the given job identifier.
  – `Tuple<JobId, RawData> getRawData(List<JobId> jobIds)`
    * Effect: Describe the effect of the operation
    * Exceptions: None
  – `BatchMetaData getBatchMetaData(BatchId batchId)`
    * Effect: The `JobManager` returns the meta-data of the batch of the document processing job identified by the given job identifier.
    * Exceptions: None
  – `DocumentType getDocumentType(JobId jobId)`
    * Effect: The `JobManager` returns the document type of the document corresponding to the job identified by the given job identifier.
    * Exceptions:
      · NoSuchJobException: Thrown if the `JobManager` cannot find a job identified by the given job identifier.

- `String getNameOfSender(JobId jobId)`
  - Effect: The `JobManager` returns the name of the sender of the document corresponding to the job identified by the given job identifier.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobManager` cannot find a job identified by the given job identifier.

- JobMgmt
  - `void createJobsAndInitiateProcessing(Customerid cuid, TimeStamp deadline, List<rawdataId> rawDataIds, batchId batchId)`
    - Effect: The `JobManager` creates a job for each of the raw data identifiers it gets as an argument. The job contains the time of deadline for the document processing job and the batch identifier. It stores the jobs and inserts the jobs in the `Scheduler`, together with their deadline and batch identifier.
    - Exceptions: None
  - `void markAsReceived(DocumentId documentId) throws NoSuchJobExceptionException`
    - Effect: The `JobManager` marks the job corresponding to the document identified by the given document identifier as received by the recipient.
    - Exceptions:
      - NoSuchJobException: Thrown if the `JobManager` cannot find a job associated to a document identified by the given document identifier.
  - `Boolean isReceiptTrackingOn(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobManager` looks up the customer organization identifier belonging to the job identified by `jobId` and uses it to ask the `OtherDB` whether that customer organization has reciept tracking enabled. The `JobManager` returns true when the customer organization has receipt tracking enabled and returns false otherwise.
    - Exceptions:
      - NoSuchJobException: Thrown if the `JonManager` cannot find a job associated to the given job identifier.

A.24 KeyCache

- **Description:** The `KeyCache` caches the keys which are most recently used for document generation. The `Completer` has to fetch a key every time a `Generator` instance requests new jobs, while the key will be the same for all jobs belonging to the same batch. The `KeyCache` avoids that the key storage system becomes a bottleneck for document generations. The keys are cached based on the `CustomerId` of a Customer Organization.
- **Super-component:** `DocumentGenerationManager`.
- **Sub-components:** None

Provided interfaces

- GetKey
  - `Key getKey(CustomerId customerId) throws NoSuchKeyException`
    - Effect: The `KeyCache` looks into its cache for the `Key` belonging to the customer organisation with id `customerId`. If the `Key` is in its cache, it returns it. If the `Key` is not in its cache, it asks `OtherDB` for the `Key` and stores it in its cache, after which it returns that `Key`.
    - Exceptions:
      - NoSuchKeyException: Thrown if there is no key for the given `customerId`.

A.25 LinkMappingDB

- **Description:** The `LinkMappingDB` is responsible for actually storing a mapping between unique links and `DocumentIds`. With this information, it also stores information about where a document can be found, i.e. only in the `DocumentDB` or both in the `PDSDB` and the `DocumentDB`.
- **Super-component:** `LinkMappingFunctionality`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.26 LinkMappingManager

- **Description:** The `LinkMappingManager` is responsible for creating unique links which points to a document. It also sends read and write requests to the `LinkMappingDB` to get and store the mappings between the unique links and the documents.
- **Super-component:** `LinkMappingFunctionality`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.27 LinkMappingFunctionality

- **Description:** The `LinkMappingFunctionality` is responsible for creating unique links which point to documents. It is also responsible for storing these mappings and ultimately mapping a link to a document.
- **Super-component:** None
- **Sub-components:** `LinkMappingManager` and `LinkMappingFunctionality`

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.28 PDSDB

- **Description:** The `PDSDB` component is responsible for storing the database of documents in the personal document stores. That database is separated from all other persistent data so that its failure *"does not affect the availability of other types of persistent data"*, as required by *Av2*.
- **Super-component:** None
- **Sub-components:** `PDSDBReplica`, `PDSLongTermDocumentManager`, `PDSReplicationManager`

Provided interfaces

- DocumentMgmt
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: The `PDSDB` will fetch and return the document corresponding to `DocumentId` id.
    - Exceptions: None
  - `List<Document> getAllDocumentMetaDataOf(RecipientId recipientId)`
    - Effect: The `PDSDB` will fetch and return all the meta-data of the documents belonging to the Registered Recipient identified by `recipientId`.
    - Exceptions: None
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `PDSDB` will store the given document`doc` together with the provided meta-data `md`.
    - Exceptions: None
  - `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
  - `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
    - Effect: The `PDSDB` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
    - Exceptions:
      - PDSUnavailableException: Thrown if the personal document store is unavailable.
  - `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
    - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
    - Exceptions: None

A.29 PDSDBReplica

- **Description:** The `PDSDBReplica` is responsible for actually storing the documents.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- ExtendedDocumentMgmt
  - `List<Document> getAllDocumentsOf(TimeStamp whenFailed)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `List<Document> getDocumentsSince(TimeStamp whenFailed)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
    - Effect: The `PDSDBReplica` will store the documents and their meta-data.
    - Exceptions: None
  - `void storeDocument(DocumentId documentId, Document document, DocumentMetaData md)`
    - Effect: The `PDSDBReplica` stores the given document with its `DocumentId` and meta-data.
    - Exceptions: None
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: The `PDSDB` will fetch and return the document corresponding to `DocumentId` id.
    - Exceptions: None
  - `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
    - Effect: The `PDSDBReplica` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
    - Exceptions: None
  - `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
    - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `PDSDBReplica` will respond to the ping request by sending an echo response. This is used by the `PDSReplicationManager` to check whether the `PDSDBReplica` is available.
    - Exceptions: None

A.30 PDSFacade

- **Description:** The `PDSFacade` is responsible for handling all read requests from Registered recipients to the `PDSDB`. It checks whether the documents requested by a Registered Recipient actually belong to him or her.
  It is also responsible for handling queries when Registered Recipients search for documents in their personal document store. It fetches the meta-data of the documents of the searching Registered Recipients, filters through that meta-data and returns an overview of the document meta-data and document identifiers matching the search criteria.

- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- PDSDBDocMgmt
  - `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
    - Effect: The `PDSFacade` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
    - Exceptions:
      - PDSUnavailableException: Thrown if the personal document store is unavailable.
  - `Tuple<Document, DocumentMetaData> getDocument(DocumentId documentId, RecipientId recipientId) throws PDSUnavailableException, DocumentUnavailableException`
    - Effect: The `DocumentFacade` first checks whether the document identified by the given document identifier actually belongs to the registered recipient. If it does, it fetches the document from the `PDSDB` and returns it. Otherwise, it throws an exception.
    - Exceptions:
      - PDSUnavailableException: Thrown if the personal document store is unavailable.
      - DocumentUnavailableException: Thrown if the given document is not available for the given user. An example is when the document identified by the `documentId` exists, but the document does not belong to the Registered Recipient. Another example is when the document identifier does not exist.
  - `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
    - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
    - Exceptions: None

A.31 PDSLongTermDocumentManager

- **Description:** The `PDSLongTermDocumentManager` is responsible for managing the different storage clusters. Each cluster consists of a `PDSReplicationManager` and one or more `PDSDBReplica` instances. In the architecture, two clusters are defined. The `PDSLongTermDocumentManager` reads to and write from clusters, and periodically transfers documents from the one cluster to the other.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- DocumentMgmt
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: The `PDSDB` will fetch and return the document corresponding to `DocumentId` id.
    - Exceptions: None
  - `List<Document> getAllDocumentMetaDataOf(RecipientId recipientId)`
    - Effect: The `PDSLongTermDocumentManager` will fetch and return all the meta-data of the documents belonging to the Registered Recipient identified by `recipientId`.
    - Exceptions: None
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `PDSDB` will store the given document`doc` together with the provided meta-data `md`.
    - Exceptions: None

- `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
  - Effect: The `PDSLongTermDocumentManager` will store the documents and their meta-data.
  - Exceptions: None
- `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
  - Effect: The `PDSLongTermDocumentManager` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
  - Exceptions:
    - PDSUnavailableException: Thrown if the personal document store is unavailable.
- `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
  - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
  - Exceptions: None

A.32 PDSReplicationManager

- **Description:** The `PDSReplicationManager` is responsible for managing the `PDSDBReplicas`. The `PDSReplicationManager` passes read requests to one `PDSDBReplica` and writes to all `PDSDBReplicas`. It monitors their availabilty using the ping/echo.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- ExtendedDocumentMgmt
  - `Tuple<Document, MetaData> getDocument(DocumentId documentId)`
    - Effect: The `PDSReplicationManager` will fetch and return the document corresponding to `DocumentId` id.
    - Exceptions: None
  - `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
    - Effect: The `PDSReplicationManager` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
    - Exceptions:
      - PDSUnavailableException: Thrown if the personal document store is unavailable.
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `PDSReplicationManager` will store the given document `doc` together with the provided meta-data `md`.
    - Exceptions: None
  - `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
    - Effect: The `PDSReplicationManager` will store the given list of documents and their meta-data.
    - Exceptions: None
  - `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
    - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
    - Exceptions: None

A.33 Print&PostalServiceChannel

- **Description:** The `Print&PostalServiceChannel` is responsible for the printing the document and sending it by mail. It is external to the eDocs system and represents the servers of a print &postal service.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.34 Print&PostalServiceFacade

- **Description:** The `Print&PostalServiceFacade` is responsible for delivering a document to the `Print&PostalChanne` so it can be printed and sent by mail. It also marks jobs as sent using the `JobManager`.
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.35 NotificationHandler

- **Description:** The `NotificationHandler` is responsible for sending notifications to the appropriate parties, e.g. the eDocs operators and the customer administrators.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- NotifyOperator
  - `void notifyOperatorOfPDSDBReplicaFailure(PDSDBReplicaId replicaId, TimeStamp dateTime)`
    - Effect: The `NotificationHandler` will send the given `PDSDBReplicaId` of the failed `PDSDBReplica` with the given time of failure `dateTime` to the eDocs operators. This method is called by a `PDSReplicationManager`.
    - Exceptions: None
  - `void notifyOperatorOfDocumentGenerationFailure(NotificationMessage msg, TimeStamp whenFailed)`
    - Effect: The `NotificationHandler` will send a textual message `msg` to the eDocs operators, which contains further information about the specific failure. This method is called by the `DocumentGenerationManager`. More specifically, it is called by the `GeneratorManager`.
    - Exceptions: None

A.36 OtherDB

- **Description:** The `OtherDB` is responsible for storing all information that is not required to be stored separately by non-functional requirements. For example, it stores the raw data and data about customer organizations and registered recipients. It also stores the templates for documents and the keys of customer organizations to sign the documents during generation.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- GetKey
  - `Key getKey(CustomerId customerId)`
    - Effect: The `OtherDB` returns the key belonging to the Customer Organization identified by `customerId`.
    - Exceptions:
      - NoSuchKeyException: Thrown if there is no key for the given `customerId`.
- TemplateMgmt
  - `Template getTemplate(CustomerId customerId, DocumentType documentType, TimeStamp whenReceived)`
    - Effect: The `OtherDB` returns the `Template` belonging to the customer organisation with id `customerId` corresponding to a document of type `documentType` and received at time `whenReceived`.
    - Exceptions:
      - NoSuchTemplateException: Thrown if there is no template for the given arguments.
  - `Boolean storeDocumentTemplate(CustomerId cuId, DocumentType documentType, Template template, TimeStamp whenReceived) throws InvalidDocumentTypeException`
    - Effect: The `OtherDB` stores the given template with the given time stamp for the customer organization identified by the given `CustomerId`.
    - Exceptions:

· InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.

- UserDataMgmt
  - `Credentials getRegisteredRecipientCredentials(RecipientId recipientId) throws NoSuchRecipientException`
    * Effect: The `OtherDB` returns the credentials belonging to the Registered Recipient identified by `recipientId`.
    * Exceptions:
      · NoSuchRecipientException: Thrown if no Registered Recipient with the given credentials exists.
  - `Credentials getCustomerOrganizationCredentials(CustomerId customerId) throws NoSuchCustomerOrganizationException`
    * Effect: The `OtherDB` returns the credentials belonging to the Customer Organization identified by `customerId`.
    * Exceptions:
      · NoSuchCustomerOrganizationException: Thrown if no Customer Organization with the given credentials exists.
  - `List<Tuple<DocumentType, TimeStamp>> getPossibleDocumentTypes(CustomerId customerId)`
    * Effect: The `OtherDB` returns the a list of document types that the Customer Organization identified by `customerId` can generate. For each document type, it also returns the time when the last `Template` for that document type was uploaded.
    * Exceptions: None
  - `Boolean isAllowedDocumentType(CustomerId cuid, DocumentType docType)`
    * Effect: The `OtherDB` checks in the SLA of the customer organization with customer identifier `cuid` if the document type `docType` is allowed. It returns true if it is allowed, otherwise it returns false.
    * Exceptions: None
  - `void indicateWhichRecurringBatch(CustomerId cuid, BatchMetaData batchMetaData)`
    * Effect: The `OtherDB` stores to which of the recurring batches of the customer organisation identified by `cuid` this batch corresponds. The `batchMetaData` contains a description of which recurring batch this batch is.
    * Exceptions: None
  - `int getNumberOfDocumentsMaximallyAllowed(CustomerId cuid, BatchMetaData batchMetaData)`
    * Effect: The `OtherDB` returns the number of documents that is maximally allowed for the customer organization identified by `cuid` for this recurring batch. The `batchMetaData` contains a description of which recurring batch this batch is.
    * Exceptions: None
  - `Boolean checkIfRegistered(Email emailAddress)`
    * Effect: The `OtherDB` returns true if the given e-mail address belongs to a Registered Recipient. Otherwise, it returns false.
    * Exceptions: None
  - `RecipientId getRecipientId(Email emailAddress)`
    * Effect: If the system has a Registered Recipient with the given e-mail address, the `OtherDB` returns its recipient identifier.
    * Exceptions:
      · NoSuchRecipientException: Thrown if the `OtherDB` does not contain the information of a Registered Recipient with the given e-mail address.
  - `Boolean isReceiptTrackingOn(CustomerId cuid)`
    * Effect: The `OtherDB` checks if the customer organization with customer identifier `cuid` has receipt tracking turned on. It returns true if it has receipt tracking turned on, otherwise it returns false.

        * Exceptions: None
    - `DocumentType getDocumentType(BatchId batchId)`
        * Effect: The `OtherDB` returns the document type of the batch identified by the given batch identifier.
        * Exceptions: None

- BatchDataMgmt
    - `BatchId storeBatchMetaData(BatchMetaData batchMetaData)`
        * Effect: The `OtherDB` stores the given `batchMetaData` and returns the batch identifier with which this meta-data can be queried.
        * Exceptions: None
    - `BatchMetaData getBatchMetaData(BatchId batchId)`
        * Effect: The `OtherDB` returns the meta-data if the batch identified by the given `batchId`.
        * Exceptions: None

- RawDataMgmt
    - `List<RawDataId> storeRawData(List<RawData> rawDataEntries)`
        * Effect: The `OtherDB` stores the given raw data entries. It returns the identifiers of the stored raw data entries.
        * Exceptions: None
    - `List<RawData> getRawData(List<RawDataId> rawDataIds)`
        * Effect: The `OtherDB` returns the raw data entries corresponding to the given raw data identifiers.
        * Exceptions: None
    - `RawData getRawData(RawDataId rawDataId)`
        * Effect: The `OtherDB` returns the raw data entry corresponding to the given raw data identifier.
        * Exceptions: None

A.37 RecipientClient

- **Description:** The `RecipientClient` is external to the eDocs system and represents a client device of an unregistered or registered recipient of eDocs that communicates with the eDocs System.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- InterfaceA
    - `returntType1 operation1(ParamType param) throws SomeException`
        * Effect: Describe the effect of the operation
        * Exceptions:
            · SomeException: Describe when the exception is thrown.
    - `void operation2(ParamType2 param)`
        * Effect: Describe the effect of the operation
        * Exceptions: None

- InterfaceB
    - `returntType2 operation3()`
        * Effect: Describe the effect of the operation
        * Exceptions: None

A.38 RawDataHandler

- **Description:** The `RawDataHandler` is responsible for verifying the raw data and its entries. It forwards the validated raw data to the `JobManager` to create jobs.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- InputRawDara
  - `List<RawData> initiateDocumentProcessing(CustomerId coid, DocumentType docType, RawDataPacket rawDataPacket, DocumentPriority priority, Boolean isRecurring, TimeStamp isReceived) throws InvalidDocumentTypeException, InvalidRawDataPackageException, ToManyRawDataEntriesException`
    - Effect: The `RawDataHandler` verifies the received raw data. If it is correct, it stores the raw data and batch meta-data in the `OtherDB`. Next, it calculates the deadline of the document processing. It generates a `BatchMetaData` object containing information about the batch and stores it in the `OtherDB`. It also stores the valid raw data entries in the `OtherDB`. The `RawDataHandler` then informs the `JobFacade` to create jobs and to initiate document processing for the valid raw data entries. It returns a list of the invalid raw data entries.
    - Exceptions:
      - InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.
      - InvalidRawDataPackageException: Thrown if the raw data package contains an error (e.g. an invalid Excel file or incorrectly formatted XML).
      - ToManyRawDataEntriesException: Thrown if the batch is a recurring batch and contains more raw data entries than maximally allowed for this batch.

A.39 RecipientFacade

- **Description:** The `RecipientFacade` is responsible for the interaction of Registered and Unregistered Recipients with the eDocs system. It provides methods for authentication, for consulting the personal document store, for downloading documents, ... Another responsibility of the `UserFacade` is marking documents as received, since this is the last internal component through which the document is passed before the requesting recipient actually receives it and failure of another component in the document lookup process can no longer prevent this from happening.
- **Super-component:** `Userfunctionality`
- **Sub-components:** None

Provided interfaces

- AuthN
  - `SessionId login(Credentials credentials ) throws InvalidCredentialsException`
    - Effect: The `RecipientFacade` forwards the given `credentials` to the `AuthenticationHandler`, which verifies them and returns a new session identifier if correct. This session identifier can be used in future requests to the `RecipientFacade`.
    - Exceptions:
      - InvalidCredentialsException: Thrown if the `AuthenticationHandler` indicated that the given credentials where incorrect.
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `RecipientFacade` removes the session corresponding to the `sessionId` using the `AuthenticationHandler`. As a result, this session cannot be used anymore to access the system without logging in again. If no session corresponds to the `sessionId`, it does not exist, nothing is changed but no exception is thrown.

    - Exceptions: None

- DocumentMgmt
  - `PDSOverview getPDSOverview(SessionId, RecipientId recipientId) throws NotAuthenticatedException, PDSUnavailableException`
    - Effect: The `RecipientFacade` first verifies the given session identifier `sessionId` using the `AuthenticationHandler`. The `RecipientFacade` then requests all the document meta-data of the documents of the recipient identified by `recipientId` from the `PDSFacade`. It generates a document overview , which is returned to the caller.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
      - PDSUnavailableException: Thrown if the personal document store is unavailable.
  - `Tuple<Document, DocumentMetaData> lookupDocument(RecipientId recipientId, SessionId sessionId, DocumentId documentId) throws NotAuthenticatedException`
    - Effect: The `RecipientFacade` first verifies the given session identifier `sessionId` using the `AuthenticationHandler`. The `RecipientFacade` then gets the document identified by `documentId` and its meta-data using the `PDSFacade`, which checks whether the document belongs to the Registered Recipient, and returns them.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
      - DocumentUnavailableException: Thrown if the given document is not available for the given user. An example is when the document identified by the `documentId` exists, but the document does not belong to the Registered Recipient. Another example is when the document identifier does not exist.

A.40 RegistrationManager

- **Description:** The `RegistrationManager` is responsible for the registration or unregistered recipients and customer organizations. For the registration of unregistered recipients, the `RegistrationManager` gets called by the `RecipientFacade`, as recipients can register themselves. Customer organizations get registered by an eDocs operator, so the `EDocsAdminClient` calls those methods.
- **Super-component:** `UserFunctionality`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.41 Scheduler

- **Description:** The `Scheduler` receives the new jobs initiated by a Customer Organization and adds them to a queue of all jobs that have not been processed yet. To lower the size of this queue, the Scheduler is only given the information it needs, i.e., the id of the batch, its deadline and the ids of the individual jobs. The raw data of each job and the meta-data of the batch is stored in `OtherDB` and fetched by the `Completer` when needed.
  The `Scheduler` also indicates to the `GenerationManager` the number of required `Generator` instances through its GetStatistics interface.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetNextJobs
  - `Tuple<BatchId, List<JobId>> getNextJobs()`
    - Effect: The `Scheduler` returns the `JobIds` of the group of jobs that belong to the batch identified by `BatchId` that should be generated next. This method is called by the `GeneratorManager` when a `Generator` instance requires a new group of jobs.
    - Exceptions: None
    - `Tuple<BatchId, List<JobId>> jobsCompletedAndGiveMeMore(List<JobId>)`
      - Effect: The `Scheduler` gets notified that the document processing jobs belonging to the list of `JobIds` are completed. It returns the a list of `JobIds` belonging to a batch identified by `BatchId`. The returned list of `JobIds` identify document processing jobs which are not yet started.
      - Exceptions: None
- InsertJobs
  - `void insertJobs(BatchId batchId, TimeStamp deadline, List<JobId> jobIds )`
    - Effect: The `Scheduler` adds the jobs identified by their `JobId` to its queue of all jobs that have not been processed yet. To lower the size of this queue, the Scheduler is only given the information it needs, i.e., the id of the batch, its deadline and the ids of the individual jobs. This method provides new jobs synchronously to the `Scheduler`, which it schedules synchronously. This means that when the method call returns, the given jobs are scheduled.
    - Exceptions: None
- GetStatistics
  - `int getNumberOfFutureJobs()`
    - Effect: The `Scheduler` returns the amount of documents that should be generated in the near future. The `GeneratorManager` queries this method at regular intervals and adjusts the number of `Generator` instances accordingly.
    - Exceptions: None

A.42 SessionDB

- **Description:** The `SessionDB` stores the session identifiers for currently active sessions.
- **Super-component:** `UserFunctionality`
- **Sub-components:** None.

Provided interfaces

- SessionMgmt
  - `RecipientId getRecipientId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `SessionDB` fetches and returns the Registered Recipient's identifier corresponding to the `sessionId` from the `sessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifiers, or if the session belongs to a customer organization.
  - `CustomerId getCustomerId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `SessionDB` fetches and returns the Customer Organization's identifier corresponding to the `sessionId` from the `sessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier, or if the session belongs to a registered recipient.
  - `SessionId openSession(RecipientId recipientId)`
    - Effect: The `SessionDB` generates a new session identifier for the given `recipientId` and stores this as an active session.
    - Exceptions: None
  - `void closeSession(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `SessionDB` closes the active session associated with the given `sessionId`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier.
  - `Map<SessionAttributeKey, SessionAttributeValue> isvalidSession(SessionId sesionId) throws NoSuchsSessionException`
    - Effect: The `SessionDB` verifies whether a session with the given id exists in the `SessionDB` and if so, returns all its associated attributes.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifiers.

A.43 TemplateCache

- **Description:** The `TemplateCache` caches the templates which are most recently used for document generation. The `Completer` has to fetch a templateevery time a `Generator` instance requests new jobs, while the template will be the same for all jobs belonging to the same batch. The `TemplateCache` avoids that the template storage system becomes a bottleneck for document generations. The templates are cached based on the `CustomerId` of a Customer Organization, the type of the document and the date and time at which the batch was provided by the Customer Organization (in order to account for template updates).
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetTemplate
  - `Template getTemplate(CustomerId customerId, DocumentType documentType, TimeStamp whenReceived`
    - Effect: The `TemplateCache` looks into its cache for the `Template` belonging to the customer organisation with id `customerId` corresponding to a document of type `documentType` and received at time `whenReceived`. If the `Template` is in its cache, it returns it. If the `Template` is not in its cache, it asks `OtherDB` for the `Template` and stores it in its cache, after which it returns that `Template`.
    - Exceptions:
      - NoSuchTemplateException: Thrown if there is no template for the given arguments.

A.44 UserFunctionality

- **Description:** The `UserFunctionality` is responsible for the interaction of registered recipients, unregistered recipients, customer organizations and eDocs operators with the eDocs system. It provides methods to register, to login and to logout, to consult the personal document store and download documents, to consult the status of document processing jobs, ...
- **Super-component:** None
- **Sub-components:** `RecipientFacade`, `CustomerOrganizationClient`, `EDocsadminfacade`,`RegistrationManager` `AuthenticationHandler` and `SessionDB`

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.45 ZoomitChannel

- **Description:** The `ZoomitChannel` is responsible for delivering documents via Zoomit. It is external to the system and represents the servers of Zoomit to which a document can be sent.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void operation2(ParamType2 param)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- InterfaceB
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.46 ZoomitFacade

- **Description:** The `ZoomitFacade` is responsible for sending documents to Unregistered Recipients through Zoomit. It is also responsible for receiving messages from Zoomit when a document has been received by Zoomit or when a Zoomit user as received his or her document. The `ZoomitFacade` can use the `JobManager` to mark jobs as sent or received.
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

Provided interfaces

- InterfaceA
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.

B Defined data types

List and describe all data types defined in your interface specifications. List them alphabetically for ease of navigation.

- `BatchId`: A piece of data uniquely identifying a batch of document processing jobs in the system. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `BatchMetaData`: A data structure listing the metadata belonging to a batch of jobs. This includes the `CustomerID` of a Customer Organization, the `DocumentType` of the documents to be generated, the `TimeStamp` of when the batch was received, ...
- `CompletePartialBatchData`: A complex data structure listing all data a `Generator` needs to complete document generation jobs that are part of the same batch. It contains an array of `Tuple<JobId, RawData>`. The `JobIds` identify jobs that are all part of the same batch. The `RawData` belongs to these document processing jobs. Also listed in the `BatchMetaData` are the values of the `BatchMetaData`, `Key` and `Template` data types belonging to the batch.

  `CompletePartialBatchData` also contains a `BatchMetaData` entry, a `Key` and a `Template`. *Important to note:* a value of CompletePartialBatchData contains all information necessary to generate **some** jobs of belonging to same batch. It does not have to contain the information of all jobs belonging to same batch.
- `Credentials`: The authentication credentials of a Registered Recipient or Customer Organization. The credentials always contain an identifier of the recipient or customer organization and a proof of his or her identity. The architecture does not specify the specific credentials used, but a possibility is using a username and password.
- `CustomerId`: A piece of data uniquely identifying a Customer Organization in the system. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `DeliveryMethod`:
- `Document`: A data file corresponding to a document. The architecture specifies the format of this data type as a PDF-file.
- `DocumentId`: A piece of data uniquely identifying a document in the system.
- `DocumentMetaData`:

- `DocumentPriority`: A data type representing the priority of document generation jobs. They have values representing the Critical, Diamond, Gold and Silver priorities. The exact format of this data type is not specified by the architecture.
- `DocumentType`: A piece of data describing the type of a document. This architecture does not specify the exact format of this data type, but possibilities are a long integer, a string, a URL etc.
- `Echo`: The response to a ping message. This data element does not contain any meaningful data.
- `EmailAddress`:
- `EmailMessage`:
- `Error`: Description of data type.
- `GeneratordId`: A piece of data uniquely identifying a `Generator` in the system. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `JobBatch`: Description of data type.
- `JobId`: A piece of data uniquely identifying a document processing job in the system.
- `Key`: A data structure containing the key of the Customer Organization which is used to sign its documents during the generation process. This architecture does not specify the exact format of this data type, but possibilites are a long integer, a string, a URL etc.
- `NotificationMessage`: A textual message which can be used to include extra information about the event of the notification.
- `PDSOverview` The overview of a personal document store that can be shown to a Registered Recipient. Through this overview, the Registered Recipient can consult documents. The architecture does not specify the exact format of such an overview, but a likely possibility is an HTML page.
- `PDSDBReplicaId`: A piece of data uniquely identifying a `PDSDBReplica` in the system. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `PostalAddress`:
- `RawData`: A data structure listing a raw data entry used in a document processing job.
- `RawDataPacket`: A data structure containing all the raw data entries in a batch, given by the customer organization to the `CustomerOrganizationFacade`. The architecture does not specify this data structure, but possibilities are an XML file or Excell file.
- `RawDataId`: A piece of data uniquely identifying a raw data entry in the system. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `RecipientId`: A piece of data uniquely identifying a Registered Recipient in the system. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `SessionId`: A piece of data uniquely identifying a session of a registered recipient of customer organization in the eDocs system. This contains at least the user identifier (i.e. the `CustomerId` for a customer organization or the `RecipientId` for a registered recipient) and the time the session was initiated.
- `SessionAttributeKey`: The key of an attribute attached to a session. This architecture does not specify the exact format of this key. a possible value is a long integer or a flat string.
- `SessionAttributeValue`: The value of an attribute attached to a session. This value van be of any primitive type.
- `TimeStamp`: The representation of a time (i.e. date and time of day) in the system.
- `Template`: A document used as a template for the generation of documents.

- `TemplateId`: A data structure uniquely identifying a template in the system. It lists three values. It contains `CustomerId` which identifies the Customer Organization who the template belongs to. It also contains a `DocumentType`, specifying for which kind of document it is a template for. The last piece of information it contains is a `TimeStamp` specifying when the system received the template.
- `ZoomitId`: