



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

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Overview</b>	<b>3</b>
2.1	Architectural decisions . . . . .	3
2.2	Discussion . . . . .	3
<b>3</b>	<b>Attribute-driven design documentation</b>	<b>3</b>
3.1	Decomposition 1: eDocs (X1, Y3, UCa, UCb, UCc) . . . . .	3
3.1.1	Module to decompose . . . . .	3
3.1.2	Selected architectural drivers . . . . .	3
3.1.3	Architectural design . . . . .	3
3.1.4	Instantiation and allocation of functionality . . . . .	3
3.1.5	Interfaces for child modules . . . . .	4
3.1.6	Data type definitions . . . . .	5
3.1.7	Verify and refine . . . . .	5
3.2	Decomposition 2: OtherFunctionality(P2, Av2b, UC12, UC13, UC14, UC15) . . . . .	5
3.2.1	Module to decompose . . . . .	5
3.2.2	Selected architectural drivers . . . . .	5
3.2.3	Architectural design . . . . .	6
3.2.4	Instantiation and allocation of functionality . . . . .	7
3.2.5	Verify and refine . . . . .	8
<b>4</b>	<b>Client-server view (UML Component diagram)</b>	<b>8</b>
4.1	Main architectural decisions . . . . .	8
4.1.1	ReqX: requirement name . . . . .	10
<b>5</b>	<b>Decomposition view (UML Component diagram)</b>	<b>10</b>
5.1	ComponentX . . . . .	10
<b>6</b>	<b>Deployment view (UML Deployment diagram)</b>	<b>10</b>
<b>7</b>	<b>Scenarios</b>	<b>10</b>
7.1	Scenario 1 . . . . .	11
<b>A</b>	<b>Element catalog</b>	<b>11</b>
A.1	Component 1 . . . . .	11
A.2	AuthenticationHandler . . . . .	12
A.3	BillingManager . . . . .	13
A.4	CustomerOrganizationFacade . . . . .	13
A.5	Completer . . . . .	13
A.6	DB . . . . .	13
A.7	DocumentDB . . . . .	14
A.8	CustOrgRegistrationManager . . . . .	14
A.9	DocumentGenerationManager . . . . .	15
A.10	DocumentStorageManager . . . . .	15
A.11	DeliveryFunctionality . . . . .	16
A.12	EDocsFacade . . . . .	16
A.13	EmailFacade . . . . .	16
A.14	Generator . . . . .	17
A.15	GeneratorManager . . . . .	17
A.16	JobFacade . . . . .	18
A.17	KeyCache . . . . .	18
A.18	LinkManager . . . . .	18
A.19	PDSDB . . . . .	19
A.20	PDSDBReplica . . . . .	19
A.21	PDSFacade . . . . .	20
A.22	PDSLLongTermDocumentManager . . . . .	21

A.23 PDSReplicationManager . . . . .	21
A.24 Print&PostalServiceFacade . . . . .	22
A.25 OtherFunctionality . . . . .	22
A.26 RawDataDB . . . . .	23
A.27 RawDataHandler . . . . .	24
A.28 RecipientFacade . . . . .	24
A.29 RecipientRegistrationManager . . . . .	25
A.30 Scheduler . . . . .	25
A.31 SessionDB . . . . .	26
A.32 TemplateCache . . . . .	27
A.33 TemplateDB . . . . .	27
A.34 TriggerManager . . . . .	28
A.35 UserDB . . . . .	28
A.36 ZoomitFacade . . . . .	28
<b>B Defined data types</b>	<b>29</b>

# 1 Introduction

The goal of this project was to develop an architecture for a system for document processing. This part of the project consisted of

## 2 Overview

### 2.1 Architectural decisions

Briefly discuss your architectural decisions for each non-functional requirement. Pay attention to the solutions that you employed (in your own terms or using tactics and/or patterns).

**ReqX: requirement name** Provide a brief discussion of the decisions related to *ReqX*.  
*Employed tactics and patterns: List all patterns and tactics used to achieve ReqX, if any.*

### 2.2 Discussion

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 Attribute-driven design documentation

### 3.1 Decomposition 1: eDocs (X1, Y3, UC<sub>a</sub>, UC<sub>b</sub>, UC<sub>c</sub>)

#### 3.1.1 Module to decompose

In the first run, the eDocs System is decomposed as a whole

#### 3.1.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *X1*: name
- *Y3*: name

The related functional drivers are:

- *UC<sub>a</sub>*: name
- *UC<sub>b</sub>*: name
- *UC<sub>c</sub>*: name

**Rationale** A short discussion of why these drivers were selected for this decomposition.

#### 3.1.3 Architectural design

**Topic** Discussion of the solution selected for (a part of) one of the architectural drivers.

**Alternatives considered**

**Alternatives for solution** A discussion of the alternative solutions and why that were not selected.

#### 3.1.4 Instantiation and allocation of functionality

**Decomposition** Main aspects of the resulting decomposition.

**ModuleB** Per introduced component a paragraph describing its responsibilities.

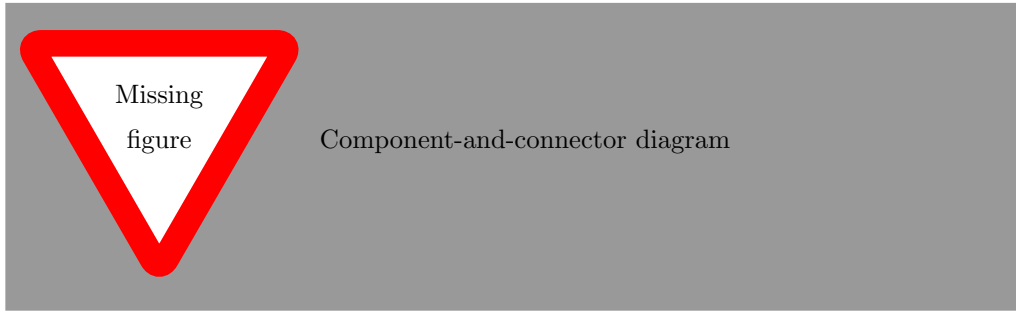


Figure 1: Component-and-connector diagram of this decomposition.

**ModuleC** Per introduced component a paragraph describing its responsibilities.

**Behaviour** If needed and explanation of the behaviour of certain aspects of the design so far.

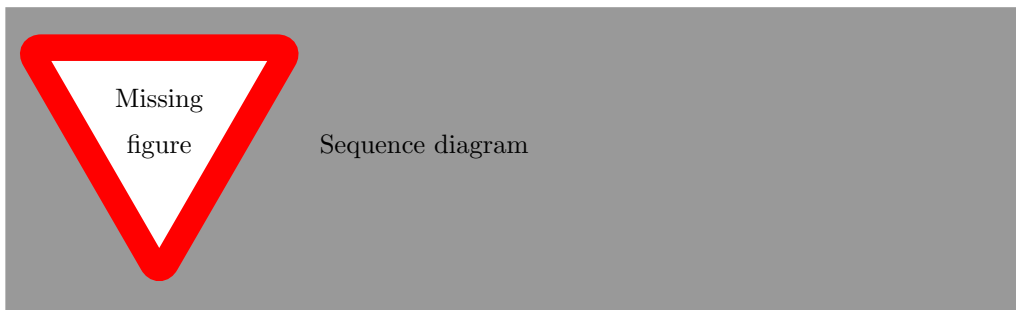


Figure 2: Sequence diagram illustrating a key behavioural aspect.

**Deployment** Rationale of the allocation of components to physical nodes.

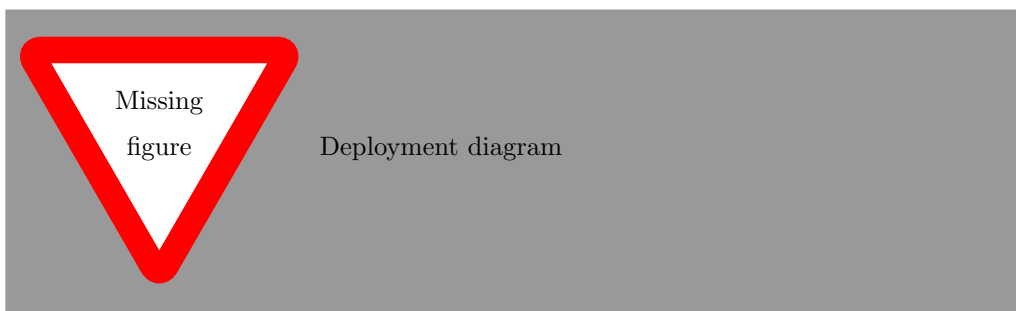


Figure 3: Deployment diagram of this decomposition.

### 3.1.5 Interfaces for child modules

#### PDSDB

- DocumentMgmt
  - `void storeDocument(DocumentId id, Document doc, Metadata md)`
    - \* Effect: The PDSDB will store the given documentdoc together with the provided metadata md.
    - \* Exceptions: None
  - `Tuple<Document, Metadata> getDocument(DocumentId id)`
    - \* Effect: Describe the effect of calling this operation.

- \* Exceptions: None
- `void markReceived(DocumentId id)`
  - \* Effect: Describe the effect of calling this operation.
  - \* Exceptions: None

### 3.1.6 Data type definitions

Describe per complex data type used in the interfaces what it represents.

**returnType** This data element represents X.

**ParamType** This data element represents Y.

### 3.1.7 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for.

#### ModuleB

- *Z1*: name
- *UCd*: name

#### ModuleC

- *UCba*: name  
Description which part of the original use case is the responsibility of this component.

## 3.2 Decomposition 2: OtherFunctionality(P2, Av2b, UC12, UC13, UC14, UC15)

### 3.2.1 Module to decompose

In this run we decompose `Otherfunctionality`.

### 3.2.2 Selected architectural drivers

The non-functional driver for this decomposition is:

- *P2*: Document lookups
- *Av2b*: Temporary storage for PDSDB

The related functional drivers are:

- *UC12*: Consult personal document store
- *UC13*: Search documents in personal document store
- *UC14*: Consult document in personal document store
- *UC15*: Download document via unique link

**Rationale** P2 and Av2b were chosen because their priorities are amongst the highest ones of all remaining non-functional drivers and the domains on which their focus lies complement the previous decomposition perfectly.

### 3.2.3 Architectural design

**Link mapping** In order to support document lookups via links, which are either unique (unregistered recipient) or part of a notification e-mail (registered recipient), the **UserFacade** is concerned with the mapping of each incoming link request to the appropriate document, which is either located in the **PDSDB** (for registered recipients) or in the **DocumentDB** (for unregistered recipients). The mappings themselves are stored in the **MappingDB** component and are fetched by the **UserFacade** when needed.

**Dedicated DocumentDB and LinkMappingDB for P2** Since a large number of document lookups and downloads should not affect the performance of other functionality of the system, both link mappings and documents are stored in dedicated databases, each deployed on a different node. This decision ensures that documents can be looked up via the personal document store or a notification in a timely fashion, because it prohibits either of those two components to be a bottleneck in the document lookup process. Since, according to the previous decomposition, the **PDSDB** is deployed on a separate node too, this component's performance is already satisfactory and no changes need to be made to improve it.

**Sharding for DocumentDB for P2** In order to improve performance of the **DocumentDB**, we chose to partition this database into multiple shards. This approach was driven by the need for fast response time while keeping in mind the high storage cost for documents. More precisely, a **ShardingManager** is responsible for reading and writing all documents in one of the **DocumentDBShards**, making sure that every document is stored only once. This sharding technique provides roughly the same advantages in response time as active replication, but is significantly less costly when it comes to storage capacity. Note that there must also be a (sub)component monitoring all requests to the shards, in order to throttle excessive requests when necessary. Since this is a rather simple task and the (sub)component must be aware of the details concerning the sharding architecture to efficiently fulfil its purpose, this functionality is delegated to the **ShardingManager** itself. Finally, this **ShardingManager** is also responsible for implicitly pinging all **DocumentDBShards** upon reading and writing in them.

**DocumentStorageCache for Av2b** This component temporarily stores the id's of all generated documents that are to be delivered through the personal document store during downtime of the **PDSDB** component up to a maximum of 3 hours. When the latter component turns operational again, the **DocumentStorageManager** retrieves all documents and their corresponding meta data from the **DocumentDB** using the previously mentioned IDs and subsequently stores them in the **PDSDB**. Note that the recipient therefore perceives a maximum total downtime of 3 hours plus the time needed for the **DocumentStorageManager** to transfer all documents and meta data that the cached IDs refer to.

Note that the **DocumentDB** and the **PDSDB** store documents in exactly the same way, both including the document itself, its meta data and its ID, although there is no need for the meta data in the former database. This storage tactic ensures that the **DocumentStorageManager** does not have to fetch or convert any information when transferring documents between both databases, making the transfer as efficient as possible. This does not introduce any overhead to the storage system, since the meta data is but a fraction of the document data. Finally, we would like to stress that 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 converted into document ID writes to the aforementioned cache. Upon revival of the **PDSDB**, this database will send a single heartbeat to the **DocumentStorageManager**, causing this component to begin transferring the missing document data. Once all cached document IDs are processed, subsequent writes to the **PDSDB** will no longer be redirected through the **DocumentStorageCache**.

**UserFacade** We introduce this component to be able to distinguish between registered recipients and unregistered recipients in the first steps of the document lookup process. The **UserFacade** also maps incoming links to documents that are located either in the **PDSDB** or in the **DocumentDB** by first fetching the correct mapping from the **LinkMappingDB** mentioned above. 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 pipeline can no longer prevent this from happening.

**PDSFacade** This component handles all read requests that are intended for the personal document store. This extra level of indirection calculates and aggregates all intermediate query results as to relieve the **PDSDB**

of this burden, which is also the reason why both components are best deployed on different nodes. Upon querying the personal document store, this component first collects the recipient's documents' meta data and subsequently performs queries on this collection. This approach introduces no significant overhead, because only those documents that are needed, will be fetched in their entirety. It is also the responsibility of the **PDSFacade** to check the recipient's ID against the recipient ID that can be retrieved from the document meta data and to only pass on those documents for which they are a match.

**DocumentStorageManager** The storage of documents is handled by the **DocumentStorageManager**, which receives generated documents from the intermediary **OtherFunctionality2** component and stores them in either the **DocumentDB** (for unregistered recipients) or in both the **DocumentDB** and the **PDSDB** (for registered recipients) according to the method that is invoked on it. The necessity for this component follows from the fact that synchronisation is needed between the two storage components mentioned above. Note that the **PDSDocMgmt** interface in the **PDSDB** component now requires an extra method to save documents.

### Alternatives considered

**Alternative for DocumentStorageCache** We could just as well do without the previously introduced **DocumentStorageCache** by letting the **DocumentStorageManager** actively look for documents in the **DocumentDB** that are not yet, but should be, present in the **PDSDB** upon revival of the **PDSDB**. 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 approach was not chosen.

### 3.2.4 Instantiation and allocation of functionality

**Decomposition** Figure ?? shows the components resulting from the decomposition in this run. Extra attention is required concerning the **DocumentDBShard**. This component actually represents multiple instances, each containing a different partition of the document data to accommodate for parallel lookups in different partitions while minimizing storage costs. The responsibilities of the resulting components are as follows:

**DocumentDB** Responsible for storing all generated documents.

**ShardingManager** Responsible for managing all reads and writes to the **DocumentDBShards**.

**DocumentDBShard** Responsible for storing a partition of the documents in the **DocumentDB**

**LinkMappingDB** Responsible for storing the mappings between links and document locations.

**UserFacade** We introduce this component to be able to distinguish between registered recipients and unregistered recipients in the first steps of the document lookup process. The **UserFacade** also maps incoming links to documents that are located either in the **PDSDB** or in the **DocumentDB** by first fetching the correct mapping from the **LinkMappingDB** mentioned above. 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 pipeline can no longer prevent this from happening.

**PDSFacade** This component handles all read requests that are intended for the personal document store. This extra level of indirection calculates and aggregates all intermediate query results as to relieve the **PDSDB** of this burden, which is also the reason why both components are best deployed on different nodes. Upon querying the personal document store, this component first collects the recipient's documents' meta data and subsequently performs queries on this collection. This approach introduces no significant overhead, because only those documents that are needed, will be fetched in their entirety. It is also the responsibility of the **PDSFacade** to check the recipient's ID against the recipient ID that can be retrieved from the document meta data and to only pass on those documents for which they are a match.



**DocumentStorageManager** The storage of documents is handled by the **DocumentStorageManager**, which receives generated documents from the intermediary **OtherFunctionality2** component and stores them in either the **DocumentDB** (for unregistered recipients) or in both the **DocumentDB** and the **PDSDB** (for registered recipients) according to the method that is invoked on it. The necessity for this component follows from the fact that synchronisation is needed between the two storage components mentioned above. Note that the **PDSDocMgmt** interface in the **PDSDB** component now requires an extra method to save documents.

**DocumentStorageCache** Responsible for temporarily storing the IDs of all generated documents that are to be delivered through the personal document store during downtime of the **PDSDB** component.

**OtherFunctionality2** Encapsulates requirements that are not tackled in this run and cannot be assigned to other introduced components.

## UserFacade

The **UserFacade** provides three interfaces to the recipient: *AuthN* for authentication, *UniqueLinkMgmt* for document lookups via unique links and *PDSDBMgmt* for access to the personal document store. Links to documents in the **PDSDB** that are part of a notification e-mail are therefore handled by the latter, in which case the **UserFacade** also needs to perform a check to see if the requesting recipient is registered in the eDocs system.

### 3.2.5 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for.

Residual Av2b2 : TIJDIGE notification AAN DE USER

USE CASE 12: residual drivers: het opslaan van de recipient id's in de pdsdb –*z* hoort bij deliveryfunctionality. –*z* voorlopig verondersteld dat recipientID in meta data zit die samen met doc opgeslagen wordt in zowel docDB als PDS Ook het inloggen (authenticatie). USE CASE 13: residual drivers: we veronderstellen dat de DocumentStorageManager metadata bij elk document opslaat, like the name of the sender, the data range in which the document should be received and the document type –*z* hoort bij deliveryfunctionality. The metadata is saved in both the pdsdb en database components, because when a user registers, this metadata has to be copied from the one to the other. Dit is om gemakkelijk de documenten in de pdsdb te kunnen zoeken. USE CASE 14: residual driver: mark as received, puntje 3. –*z* beter te doen bij de deliveryfunctionality USE CASE 15: residual driver: tracking.

## LinkManager VOORLOPIG NIET NODIG IN DEZE DECOMPOSITIE

Motivatie voor LinkManager: Checks expiration date ALS DAT NODIG IS –*z* 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 –*z* 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 –*z* there has to be a notification when the link has expired

## 4 Client-server view (UML Component diagram)

The context diagram of the client-server view. Discuss which components communicate with external components and what these external components represent.

The primary diagram and accompanying explanation.

### 4.1 Main architectural decisions

Discuss your architectural decisions for the most important requirements in more detail using the components of the client-server view. Pay attention to the solutions that you employed and the alternatives that you considered. The explanation here must be self-contained and complete. Imagine you had to describe how the architecture supports the core functionality to someone that is looking at the client-server view only. Hide unnecessary details (these should be shown in the decomposition view).



#### 4.1.1 ReqX: requirement name

Describe the design choices related to *ReqX* together with the rationale of why these choices were made.

#### Alternatives considered

**Alternative(s) for choice 1** Explain what alternative(s) you considered for this design choice and why they were not selected.

## 5 Decomposition view (UML Component diagram)

Discuss the decompositions of the components of the client-server view which you have further decomposed.

### 5.1 ComponentX

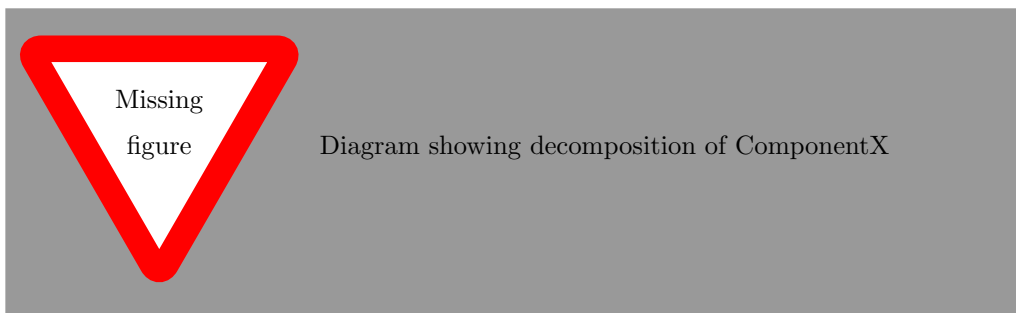


Figure 6: Decomposition of ComponentX

Describe the decomposition of ComponentX and how this relates to the requirements.

## 6 Deployment view (UML Deployment diagram)

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

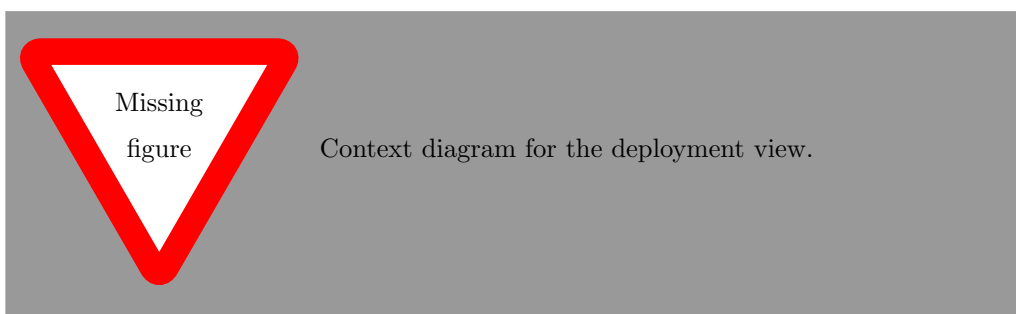


Figure 7: Context diagram for the deployment view.

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.

## 7 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

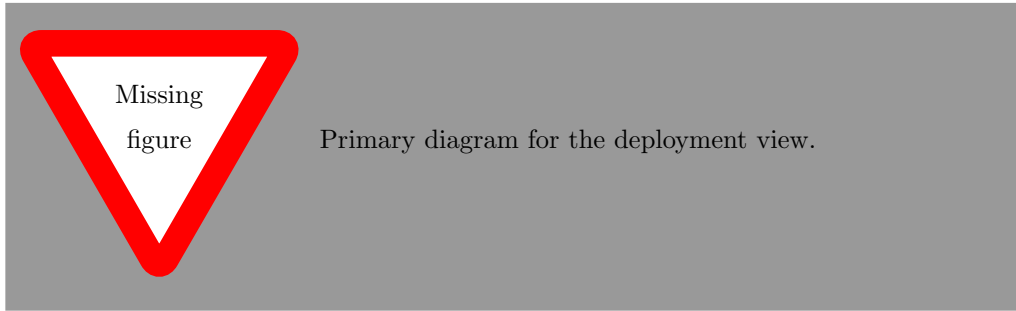


Figure 8: Primary diagram for the deployment view.

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 your architecture at this point.

## 7.1 Scenario 1

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

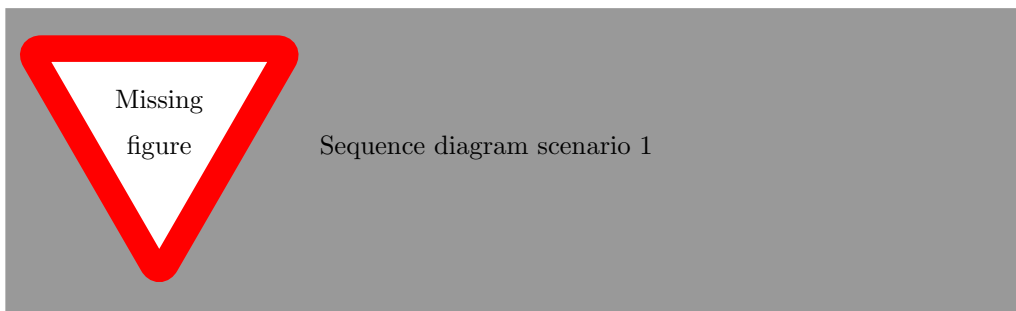


Figure 9: The system behavior for the first scenario.

## A Element catalog

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 Component 1

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

#### Provided interfaces

- InterfaceA
  - `returnType1 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`
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.2 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:** `UserFacade`
- **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 sessionId) throws NoSuchSessionException`
    - \* 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.3 BillingManager

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

#### Provided interfaces

- InterfaceA
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - `SomeException`: Describe when the exception is thrown.

### A.4 CustomerOrganizationFacade

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

#### Provided interfaces

- InterfaceA
  - `returnType1 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 Completer

- **Description:** The `Completer` is responsible for fetching the raw data and 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 corresponding to the `JobIds` belonging to the same batch, which is identified by `BatchId`.
    - \* Exceptions: None

### A.6 DB

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

## Provided interfaces

- InterfaceA
  - `returnType1 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.7 DocumentDB

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

## Provided interfaces

- InterfaceA
  - `returnType1 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
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.8 CustOrgRegistrationManager

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

## Provided interfaces

- InterfaceA
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - SomeException: Describe when the exception is thrown.
- InterfaceB
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.9 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 their deadlines and schedules them according to *P1*.
- **Super-component:** None
- **Sub-components:** `Completer`, `GenerationManager`, `KeyCache`, `Scheduler`, `TemplateCache`

### Provided interfaces

- `InsertJobs`
  - `returnType1 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.10 DocumentStorageManager

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

### Provided interfaces

- `InterfaceA`
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - `SomeException`: Describe when the exception is thrown.
- `InterfaceB`
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None



## A.11 DeliveryFunctionality

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

### Provided interfaces

- InterfaceA
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - SomeException: Describe when the exception is thrown.
- InterfaceB
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.12 EDocsFacade

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

### Provided interfaces

- InterfaceA
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - SomeException: Describe when the exception is thrown.
- InterfaceB
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.13 EmailFacade

- **Description:** The `EmailFacade` is responsible for sending documents to Unregistered by e-mail when receipt tracking is turned off. When receipt tracking is turned on for an Unregistered Recipient, it sends an e-mail containing a short description of the received document
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

## Provided interfaces

- InterfaceA
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - SomeException: Describe when the exception is thrown.
- InterfaceB
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.14 Generator

- **Description:** A **Generator** generates the documents and forwards them to **OtherFunctionality** to store and deliver them. Its availability is monitored by the **DocumentGenerationManager** with the **Ping** interface. A **Generator** is also responsible of notifying the **OtherFunctionality** 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 and 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 it 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.15 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.16 JobFacade

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

## Provided interfaces

- `InterfaceA`
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - `SomeException`: Describe when the exception is thrown.
- `InterfaceB`
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.17 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 `OtherFunctionality` 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.18 LinkManager

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

## Provided interfaces

- InterfaceA
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - `SomeException`: Describe when the exception is thrown.
- InterfaceB
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.19 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.

## A.20 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
- 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.21 PDSFacade

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

## 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.
  - `void operation2(ParamType2 param)`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.22 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.

## A.23 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 availability 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

## A.24 Print&PostalServiceFacade

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

### Provided interfaces

- InterfaceA
  - returnType1 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
  - returnType2 operation3()
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.25 OtherFunctionality

- **Description:** Responsibilities of the component.
- **Super-component:** None
- **Sub-components:** the direct sub-components, if any.

### 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 **OtherFunctionality** will store the given document **document** and deliver it. This method is made idempotent. To filter 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
- GetBatchData
  - Tuple<JobId, RawData> getRawData(List<JobId> jobIds)
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None
  - BatchMetaData getMetaData(BatchId batchId)
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None
- GetKey
  - Key getKey(CustomerId customerId)
    - \* Effect: The OtherFunctionality returns the key belonging to the Customer Organization identified by customerId.
    - \* Exceptions:
      - NoSuchKeyException: Thrown if there is no key for the given customerId.
- GetTemplate
  - Template getTemplate(CustomerId customerId, DocumentType documentType, TimeStamp whenReceived)
    - \* Effect: The OtherFunctionality 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.
- SetStatus
  - void setJobStatusAsTemporarilyFailed(List<JobId> statusesOfJobs)
    - \* Effect: The OtherFunctionality stores the JobStatus 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
- NotifyOperator
  - void notifyOperatorOfPDSDBReplicaFailure(PDSDBReplicaId replicaId, TimeStamp dateTime)
    - \* Effect: The OtherFunctionality 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 OtherFunctionality 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.26 RawDataDB

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.



## Provided interfaces

- InterfaceA
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - `SomeException`: Describe when the exception is thrown.
- InterfaceB
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.27 RawDataHandler

- **Description:** The `RawDataHandler` is responsible for verifying the raw data and its entries.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

## Provided interfaces

- InterfaceA
  - `void validateRawData(List<RawData> rawData) throws InvalidRawDataException`
    - \* Effect: The `RawDataHandler` verifies the received raw data. If it is correct, nothing happens.
    - \* Exceptions:
      - `InvalidRawDataException`: Thrown if the raw data is invalidated for some reason. This exception can contain a message about why the raw data was invalidated.
- InterfaceB
  - `returnType2 operation3()`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.28 RecipientFacade

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

## 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 were incorrect.
  - `Boolean logout(SessionId sessionId)`

- Effect: The **UserFacade** 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.

## A.29 RecipientRegistrationManager

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

### Provided interfaces

- InterfaceA
  - **returnType1 operation1(ParamType param) throws SomeException**
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - **SomeException**: Describe when the exception is thrown.
- InterfaceB
  - **returnType2 operation3()**
    - \* Effect: Describe the effect of the operation
    - \* Exceptions: None

## A.30 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 **OtherFunctionality** 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, 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.31 SessionDB

- **Description:** The **SessionDB** stores the session identifiers for currently active sessions.

- **Super-component:** **UserFacade**

- **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 sessionId) throws NoSuchSessionException`
  - \* 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.32 TemplateCache

- **Description:** The `TemplateCache` caches the templates which are most recently used for document generation. The `Completer` has to fetch a template every 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 `OtherFunctionality` 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.33 TemplateDB

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

### Provided interfaces

- `InterfaceA`
  - `returnType1 operation1(ParamType param) throws SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - `SomeException`: Describe when the exception is thrown.

### A.34 TriggerManager

- **Description:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

#### Provided interfaces

- InterfaceA
  - Credentials `getRegisteredRecipientCredentials()` throws `SomeException`
    - \* Effect: Describe the effect of the operation
    - \* Exceptions:
      - `SomeException`: Describe when the exception is thrown.

### A.35 UserDB

- **Description:** The `UserDB` stores all information about registered users of the eDocs system, i.e. Customer Organizations and Registered Recipients.
- **Super-component:** DB
- **Sub-components:** None

#### Provided interfaces

- `UserDataMgmt`
  - Credentials `getRegisteredRecipientCredentials(RecipientId recipientId)` throws `NoSuchRecipientException`
    - \* Effect: The `UserDB` 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 `UserDB` returns the credentials belonging to the Customer Organization identified by `customerId`.
    - \* Exceptions:
      - `NoSuchCustomerOrganizationException`: Thrown if no Customer Organization with the given credentials exists.

### A.36 ZoomitFacade

- **Description:** The `ZoomitFacade` is responsible for sending documents to Unregistered Recipients through Zoomit.
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

#### Provided interfaces

- InterfaceA
  - `returnType1 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 possibilities 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 possibilities are a long integer, a string, a URL etc.
- **Document**: A data file corresponding to a document. The architecture specifies the format of this data type as a PDF-file. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- **DocumentId**: A piece of data uniquely identifying a document in the system.
- **DocumentMetaData**: The meta-data stored with a document.
- **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.
- **Error**: Description of data type.
- **GeneratorId**: A piece of data uniquely identifying a **Generator** in the system. This architecture does not specify the exact format of this identifier, but possibilities 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 possibilities 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 possibilities are a long integer, a string, a URL etc.
- **RawData**: A data structure listing the raw data used in a document processing job.
- **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 possibilities 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 can 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.