

SUBMISSION OF WRITTEN WORK

Class code: BNDN
 Name of course: Second Year Project: Software Development in Large Teams [...]
 Course manager: Thomas Hildebrandt
 Course e-portfolio: <https://learnit.itu.dk/>
 Thesis or project title: Second Year Project
 Supervisor: Thomas Hildebrandt

Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1. Adam William Engsig	01/05-1993	adae@itu.dk
2. Anders Fischer-Nielsen	06/05-1993	afin@itu.dk
3. Anders Wind Steffensen	10/02-1993	awis@itu.dk
4. Cecilie Strunge Jensen	28/09-1992	csje@itu.dk
5. Mikael Lindemann Jepsen	17/02-1992	mlin@itu.dk
6. Morten Albertsen	01/08-1988	moalb@itu.dk
7. _____	_____	_____@itu.dk

Second Year Project:
Software Development in Large Teams
with International Collaboration

IT University of Copenhagen
Spring term 2015

Authors:

Adam William Engsig (adae)
Anders Fischer-Nielsen (afin)
Anders Wind Steffensen (awis)
Cecilie Strunge Jensen (csje)
Mikael Lindemann Jepsen (mlin)
Morten Albertsen (moalb)

Contents

1	Introduction	3
1.1	Conventions	3
2	Requirements	3
2.1	Scope	3
2.2	Non scope	4
2.3	Hospital Workflow	4
2.4	Initial Plan of Division of Work	5
3	Workflows, DCR graphs, and REST	5
3.1	Workflows	5
3.2	DCR graphs	6
3.2.1	Events	6
3.2.2	Relations	7
3.3	REST	8
3.3.1	HTTP Methods for CRUD Operations	8
3.3.2	Be Stateless	9
3.3.3	Resource Directory Structure	9
3.3.4	Transfer XML or JSON	9
4	Workflow elaboration	9
4.1	Initial Workflow Design	9
4.2	Feedback from External Partner	10
4.3	Final Workflow	11
4.4	Workflow Given by Lecturers	13
5	Usermanual	13
5.1	Compiling the Project	14
5.2	Running the Client	14
5.2.1	The Login Window	14
5.2.2	The Main Window	14
5.2.3	History	16
5.2.4	The Settings File	17
5.3	The DCR Graph Parser	17
5.4	Running Locally	17
6	System Implementation	18
6.1	Overview	18
6.2	Overall Architecture	19
6.2.1	Client	19
6.2.2	Server	19
6.2.3	EventAPI	19
6.3	Interface-based Programming	20
6.4	Dependency Injection	21
6.5	Design of Client	21
6.5.1	MVVM-principles in Client	21
6.6	Design of Server and EventAPI	22
6.6.1	Multi-layered Design	22
6.6.2	Execution of an Event	23
6.7	Distribution	24
6.7.1	Location Transparency	24
6.7.2	Global Identification of Events	24

6.7.3	System Deployment	25
6.8	RESTful APIs	25
6.8.1	Restfulness of the System	25
6.8.2	UnRESTful Parts of the System	25
6.9	Minimal Logic Handling in Controllers	26
6.9.1	Exception and Response Handling	26
6.10	Persistence	29
6.10.1	The Relational Database	29
6.10.2	Data Distribution	31
6.11	Role-Based Access Control	32
6.11.1	Motivation	32
6.11.2	Implemented Solution	32
6.11.3	Discussion of the Implemented Solution	33
6.12	Concurrency Control	33
6.12.1	Motivation	33
6.12.2	Implemented Solution	34
6.12.3	Discussion of Implemented Solution	34
7	Testing	35
7.1	Unit Testing	35
7.2	Integration Testing	36
7.3	System Testing	36
7.4	Acceptance Testing	36
7.5	Discussion of Testing Approach	36
8	Reflection on Project	37
8.1	Reflections on the System	37
8.1.1	What the Team Dislikes	37
8.1.2	What the Team Likes	37
8.2	Thoughts on Group Processes and Tools	38
9	Conclusion	38
A	List of Work Distribution	40
B	Healthcare Process From Brazil	42
C	Usermanual for DCR graph parser	45
D	Possible solutions for creating the database	45

1 Introduction

The following report describes the system developed in the course “*Second Year Project: Software Development in Large Teams with International Collaboration*” taught by Thomas Hildebrandt and Rasmus Nielsen at the fourth semester of the Bachelor of Science in Software Development at the IT University of Copenhagen.

The report gives an introduction to Dynamic Condition Response Graphs (*DCR graphs*), an overview of the software architecture of the delivered system, how the system has been tested, and why the team believe it is functioning. Both the construction of the system and the report was completed in the Spring term 2015.

The purpose of the system is to support the functionality of DCR graphs, and to test the system on a workflow description of the processes in a Brazilian healthcare system. The processes are described by an external partner in Brazil, Eduardo A. P. Santos.

All DCR graphs presented throughout this report is generated through the online tool available at DCRGraphs.net.

1.1 Conventions

The following conventions is used throughout the report:

- `monospaced` text for code and classes
- “The system” refers to the actual product.
- “The solution” refers to the implemented solution to a problem.
- “The project” refers to the process of developing the system.
- If project or solution is prefixed with VS it refers to Visual Studio projects and solutions.
- Specific roles in workflows will be written with an *italic* font.
- Specific events in workflows will be written with a **bold** font

2 Requirements

This section states the given requirements to the system and how they were interpreted.

From the project description:

“The goal is to develop a functioning and correct web-service-based distributed workflow system that can support distributed coordination of workflows provided by the external and possibly international customers, and reconfigured if the workflow changes.”

2.1 Scope

The system should fulfill the following requirements:

- The system must be able to handle generic DCR graphs and the logic they follow. This is to be tested on a DCR graph produced from a workflow of a Brazilian hospital and a DCR graph given by our lecturers.
- The system must be able to create, reconfigure, and delete workflows.

- The architecture of the system must resemble a peer-to-peer distribution.
- The implementation should be based on REST services.
- The system should provide a graphical user interface (UI) to the user.
- The UI should be able to present an event log or history, that describes which operations have been executed, aborted or changed in a given workflow. Because “log” and “lock” sounds the same the team chose to call the “logging” feature “History”.
- The system must persist data such that it can be restarted and handle isolated occurrences of crashes.
- The system must implement concurrency control that ensures that an illegal state cannot be reached.
- The system must implement role based access control (RBAC) that ensures that a user is only able to make changes that the given user has permission to make.
- The system must allow reconfiguration of already existing events and their relations. This can be achieved by deleting and then recreating the individual events.

2.2 Non Scope

The following was intentionally left out during the design and implementation of the system:

- The team realized that some workflows might require storing of data such as a patient record. The team has chosen not to support this feature.
- Multiple instances of the same workflow. The system only supports a single instance of each workflow, as the team did not see this as being the cornerstone of the given assignment - however the system supports creation of identical workflows.
- The system does not provide a UI for creation of DCR graphs. The team recommends the use of <http://www.DCRGraphs.net>
- Furthermore it is not in the scope of the project to support nested events on a workflow as is possible in the full specification of DCR graphs.
- Being able to handle crashes of services is not part of the scope of this project, no handling for this has been implemented.

2.3 Hospital Workflow

This section introduces the workflow, that was used as a sample workflow throughout the project. The workflow was based on a textual description, provided by our external partner. The workflow concerns medical care and treatment at a hospital and is seen from the point of view of the hospital, i.e. all assignments in the workflow are handled by the employees of the hospital.

The hospital workflow is designed in multiple iterations: the initial draft and the final workflow. The initial draft was the interpretation of the textual description see Appendix B Healthcare Process From Brazil. The initial draft was then reviewed by the external partner. The feedback was taken into consideration in the development of the final workflow.

The team believes that the final workflow supports the tasks the hospital needs.

2.4 Initial Plan of Division of Work

At the beginning of the project the group had an introductory meeting, discussing the members' expectations of the project, and outlining a group constitution.

To cope with different expectations of the members, everyone had a chance to express what part of the project they wanted to contribute to the most. The team had an agreement that no member should be assigned to work on a single part of the system. The decision was made to make sure that everyone felt ownership of the entire project and thereby making everyone able to test and work on any part of the project when necessary.

Having everyone involved equally in the project was declared as a goal for the team, however it was decided not to have everyone contribute equally in team management and implementation of the system. Four members would mainly be in charge of implementation and the remaining two would mainly handle team management. To make sure that everyone were focused on the entire project, a minimum of 30% of a group member's time should be spent on either code or management.

The implementation of the system was focused on first documenting the proposed system architecture. Subsequently the assigned team members decided on the implementation details, and the entire team discussed the possible solutions in unison.

The team wanted tests to be a high priority during development.

To ensure that the report would reflect the project as well as securing against adding new bugs with new implementation, both a feature freeze and a code freeze were planned. These were planned two and one weeks respectively prior to handin.

The plan has to a large degree been followed and the team believes that overall the contribution has been equally divided between the members. For a list of work contribution see Appendix A List of Work Distribution

3 Workflows, DCR Graphs, and REST

This section will give readers a theoretical introduction to workflows, DCR graphs, and REST-based web services. These concepts will become relevant as they are used in the delivered solution and will be discussed later in this report.

3.1 Workflows

A workflow is a standardized work description of a process that is executed repeatedly.

An instance of a workflow describes an ongoing process with several stages and describes in which order these states must be reached. A workflow contains several tasks that must be executed for the workflow to either finish in a successful or unsuccessful state. The workflow must also guarantee that the process finishes.

Certain types of workflow system notations exist. Today, most workflow systems are based on the flow-oriented process notation¹. These, unlike those described with DCR graphs, do not support that the design of the workflow can change dynamically. Furthermore the user cannot deviate from the plan even if it made sense to do so business wise. One of the strengths of DCR graph-based workflows is that any route can be taken to complete the workflow as long as the rules of the workflow are followed.

¹Concurrency and Asynchrony, *Søren Debois, Thomas Hildebrandt, and Tijs Slaats*, IT University of Copenhagen

3.2 DCR Graphs

This section will give a brief introduction to DCR graphs. DCR graphs present an alternative notation to the notation of standard flow-oriented processes.

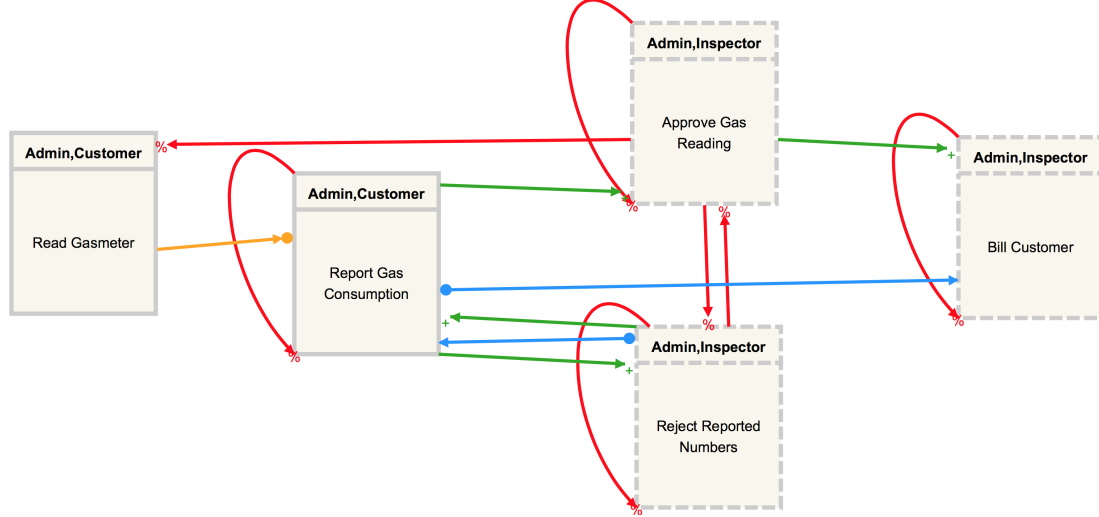


Figure 1: Sample Pay Yearly Gas Bill

A DCR graph represents a workflow. A workflow could, for instance, be the yearly reporting of gas consumption in a household, as seen in Figure 1. A DCR graph is made up of a number of the following two components: *events* and *relations*. The term DCR *graph* is due to events representing nodes, and relations representing edges between nodes.

3.2.1 Events

Events represent activities within a workflow. In the example, Figure 1, there are five events, among others **Read Gasmeter** and **Bill Customer**. An event may have a role associated with it e.g. the event **Read Gasmeter** has a *Customer*, and the event **Bill Customer** an *Inspector*. The role determines who are allowed to execute the event.

A state for each event is needed for the functionality of a DCR graph to be implemented. The state of an event is comprised of the following information, see Table 1

	Options	Default value
Pending	false true	false
Included	false true	true
Executed	false true	false

Table 1: Default values for event state

Executed describes if the event has been executed at least once. Depending on the value of *Included* the event is either included: true, or excluded: false. *Pending* describes if the event is expected to be executed eventually. For an event without relations to be executable the event must be included. After the execution of any event, their *Pending* value must be set to false, and the *Executed* value to true.

3.2.2 Relations

Relations describe constraints among events. There are 4 basic relations, explained in the following section.

Condition - yellow arrow: This relation states that to execute the event **B**, event **A** must be executed or excluded first. See Figure 2.

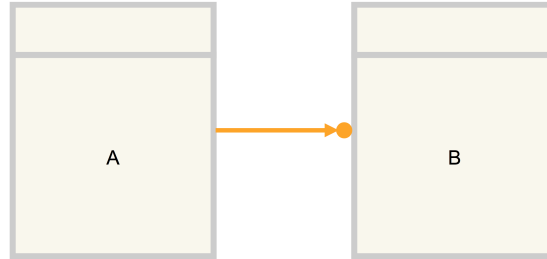


Figure 2: Illustration of Condition relation between event **A** and event **B**

Exclude - red arrow: An exclusion relation states that once event **A** has been executed, the *Included* boolean of event **B** must be set to false, such that event **B** is excluded. Note that an event may exclude itself, meaning once the event has executed, the *Included* value of the event must be set to false such that it is excluded. See figure 3.

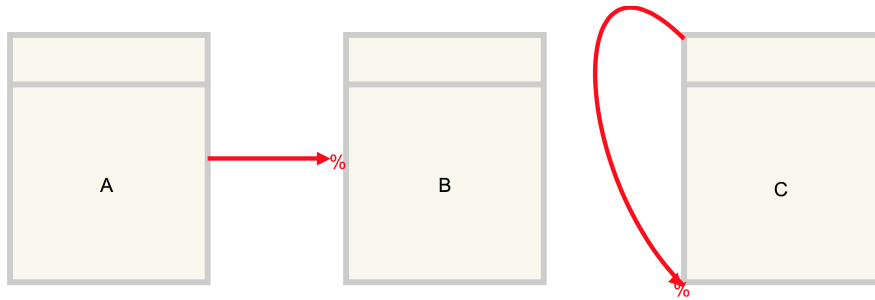


Figure 3: Illustration of exclusion relation between event **A** and event **B**. Event **C** is self excluding.

Include - green arrow: This relation states that event **A** includes event **B**. This means that after the execution of event **A**, the *Included* value of event **B** must be set to true, such that event **B** is included. See Figure 4

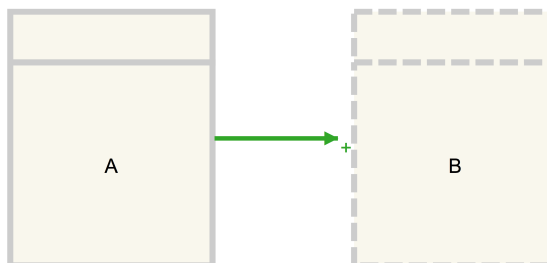


Figure 4: Illustration of inclusion relation between event **A** and event **B**

Response - blue arrow: The response relation states that once event **A** is executed, event **B**

is expected to be executed eventually. Once event **A** has been executed, the *Pending* value of event **B** must be set to true. See Figure 5

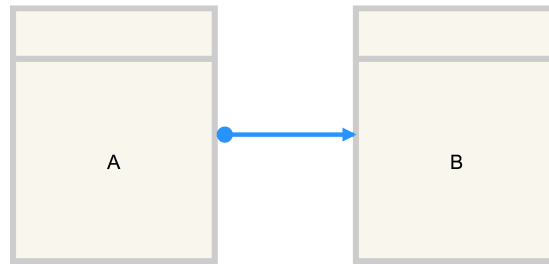


Figure 5: Illustration of response relation between event **A** and event **B**

To sum up: for an event to be executable, the *Included* value must be true, and each of its condition relations must be either excluded or executed.

Furthermore an execution of an event leads to its *Executed* and *Pending* values to be set to true and false respectively. Each of the executed events' response relations must have their *Pending* value set to true, each of its inclusion relations must have their *Included* value set to true, and its exclusion relations must have their *Included* value set to false.

The three concepts: state, events, and relations, will be mirrored in the system.

3.3 REST

A web service is one or more servers that allows for clients to retrieve and manipulate resources at the server(s).

Representational State Transfer (*REST*²) is an architectural style used in web services. No knowledge about the internal logic of a REST service should be required to use the REST service. The user of a REST service should not need to know whether communication is happening directly with the service or with an intermediary layer. This is known as RESTful layering.

A REST implementation should fulfill these four principles³

- Use HTTP methods⁴ as intended for CRUD⁵ operations.
- Be stateless - no context is stored of clients between requests.
- Resources should be accessed by URLs that expose a directory-like structure.
- Transfer resources as XML or JSON.

These are explained in the following sections.

3.3.1 HTTP Methods for CRUD Operations

A GET request should return a resource. POST requests should be used for creating new resources at the server, while DELETE and PUT requests are used for respectively deleting and updating resources. GET, PUT, and DELETE requests should be idempotent, which means that multiple identical requests result in the same response. POST on the other hand is not supposed to be idempotent, such that multiple identical requests create multiple resources at the server.

²Distributed Systems - Concepts and Design p. 386.

³<http://www.ibm.com/developerworks/library/ws-restful/ws-restful-pdf.pdf>

⁴ <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>

⁵http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

3.3.2 Be Stateless

For a REST service to be stateless, a server must not store the state of the clients. The canonical example is a client reading pages in a document held at a server. The client must request one page at a time. Instead of the server knowing what page the client is currently at, this responsibility is given to the client so that the client must provide this information when requesting the next page.

This has the following consequence: Previously, clients could simply issue a GET request for the next page and the server would be able to determine the next page based on the last requests of the clients. With REST the client must issue a GET request for a specific page to receive a given page. The server does not have to store an internal representation of the state of the client.

3.3.3 Resource Directory Structure

A RESTful service must provide access to its resources in a directory-like structure. An example of this could be that a server stores a collection of books, which consists of chapters that are then split into pages. To access a given page the resource should be located at:

`http://myRestService.com/Books/ATaleOfTwoCities/ChapterOne/12`

A GET request issued at this address would find page 12 of chapter one in the book “A Tale of Two Cities”.

3.3.4 Transfer XML or JSON

Typically a REST service will serve its resources as either XML, JSON, or both. It should also be able to receive content in both formats.

4 Workflow Elaboration

In this section, the solution for the workflow description given by the external partner is described. The feedback from the external partner is discussed and the revised workflow is explained. Finally, a workflow given by the lecturers is explained.

4.1 Initial Workflow Design

In this section, the solution for the workflow description given by the external partner is described. The feedback from the external partner is discussed and the revised workflow is explained. Finally, a workflow given by the lecturers is explained.

Through analysis of the textual description given by the external partner the initial workflow design was developed. Events were created based on the actions described in the text, and roles were found by looking for actors doing those actions. The relations were based on rules and the order of which the patient went through the events.

The team intentionally decided to make a rather simple design of the initial workflow to make sure that all the actions of the healthcare system were supported and all relations defined in DCR graph notation were in use.

The team focused on representing the activities directly handled by the employees of the hospital. The patient's activities such as entering the hospital is left out, as it is not the responsibility of an employee. The different types of employees were assigned to their respective events. The following roles were found necessary: CheckInReceptionist, QueueReceptionist, Examiner, and Specialist.

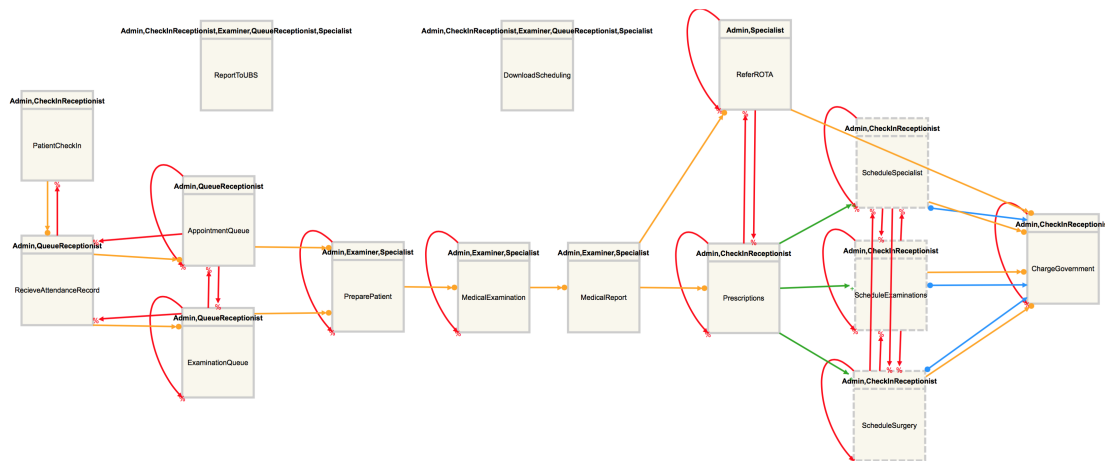


Figure 6: The initially drafted workflow

The initial healthcare workflow has three events which is initially executable: **DownloadScheduling** and **ReportToUBS** that both can be executed at any time and hold no constraints to other activities. The third initially executable event is **CheckInPatient** which is the starting point when a patient enters the hospital and addresses the *CheckInReceptionist*. When checked in, the patient is handed an attendance record. Then the *QueueReceptionist* handles the **ReceiveAttendanceRecord** event. The *QueueReceptionist* then assigns the patient to a queue: **AppointmentQueue** or **ExaminationQueue**. When a patient has been through either an examination or an appointment with a specialist a medical report is made. Depending on which queue the patient was in, the medical report is conducted by the *Specialist* for appointments or the *Examiner* for examinations. Based on the medical report the *CheckInReceptionist* gives prescriptions if needed. Afterwards the *CheckInReceptionist* schedules the patient to either a meeting with another *Specialist*, further examinations, or surgery which are all represented as events in the graph. In any case the government is charged. In the medical report, it is also possible for a patient to be referred to ROTA, which is a fast track queue for patients needing treatment urgently. In this case the government is also eventually charged.

4.2 Feedback from External Partner

The initial workflow was presented to our external partner. He reviewed the team’s initial healthcare workflow, and clarified some elements in the workflow that were misinterpreted. The following section will describe these elements.

First of all, response relations should be used to force the user to end in an accepting state where there are no pending activities.

It should not be possible for the same patient to be checked in multiple times without having seen a *Specialist*.

Preparation of a patient is not handled by either an *Examiner* or a *Specialist*, but a new role, *Nurse*, is introduced to handle this.

The external partner also explained that a *Specialist* and an *Examiner* are not to share any activities, as they are not doing the same job. An *Examiner* is not able to recommend any further treatment - after a patient has left the *Examiner*'s office, a medical examination report is conducted and the government is charged. A *Specialist* has multiple options to what a patient's further treatment should involve, which is not specified thoroughly in the initial workflow.

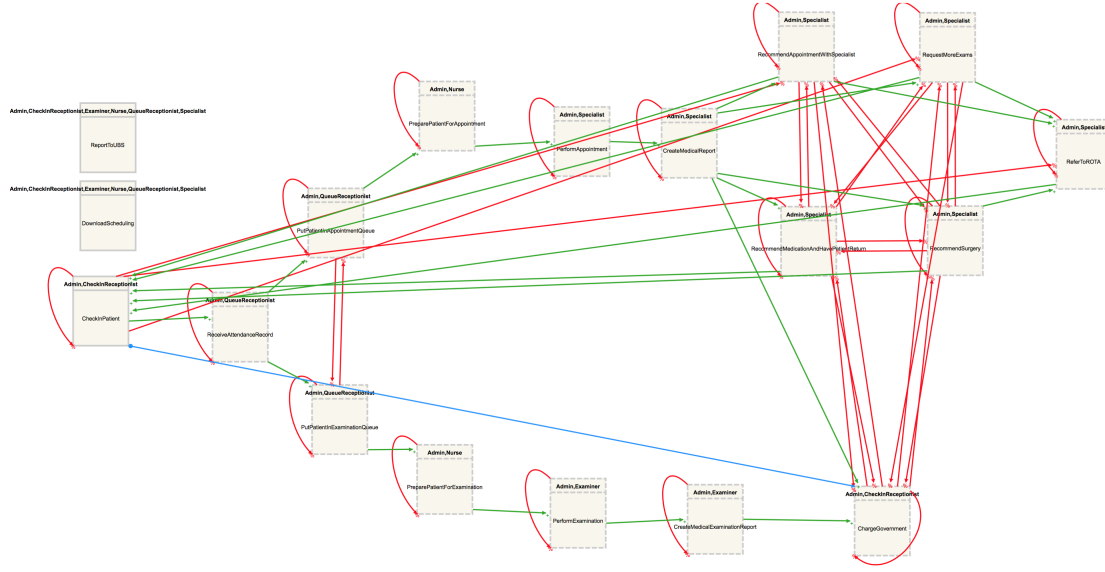


Figure 7: Final workflow

The initial workflow only allows the patient to be handled by the hospital once. This does not allow for a patient to return e.g. for further examination. When a patient has been taken care of by an *Examiner* or a *specialist*, further treatment is planned if needed and the government is charged. This was not the intention. Instead information about all treatments given to a patient should be stored until all treatments are completed. This means that a patient should be able to check in multiple times without the government being charged, but it is required that a patient sees either an *Examiner* or a *Specialist* between each check in.

ROTA is a fast track queue for patients needing urgent treatment. Only Specialists can refer patients to ROTA if the patient needs to either:

- see another *Specialist*
- have further exams
- both of the above
- needs surgery

The urgency of each case is decided by an *Auditor*, who is working for the Brazilian government.

4.3 Final Workflow

The final workflow turned out more complex than the initial interpretation. After feedback from the external partner, the team has added the role *Nurse*, which was not described in the textual description. Furthermore we divided the examinations and the appointments to different events. Although similar, they are not identical.

- All events, except **DownloadScheduling**, **ReportToUBS**, and **CheckInPatient** are excluded to begin with.
- All events, except **DownloadScheduling** and **ReportToUBS**, are self excluding, meaning once they are carried out, they cannot be executed again, unless they are subsequently included by some other activity.

- The only requirement is, once a patient has been checked in, the government will *eventually* be charged, this is clarified by the response relation between the events **CheckInPatient** and **ChargeGovernment**. This may happen either after the patient has gone through the workflow a number of times or a single time. Notice that we are able to leave out a lot of response relations, because the single response relation drives the workflow towards **ChargeGovernment**.

In the following section, the final workflow and its different branches are described. Before branching out, every patient entering the hospital is handled the same way by the hospital employees.

1. **CheckInPatient**: Requires that the government will eventually be charged, hence the response relation to **ChargeGovernment**. It excludes **ReferToROTA**; why will be explained later.
2. **ReceiveAttendanceRecord**: **ReceiveAttendanceRecord** has inclusion relations to both **PutPatientInAppointmentQueue** and **PutPatientInExaminationQueue**.
3. **PutPatientInExaminationQueue** and **PutPatientInAppointmentQueue** exclude each other, since the given hospital description allows only one of them to occur. As soon as one of these event are executed, one of the two branches are chosen.

From here on, the workflow splits into two separate branches. We first describe the branch related to a patient that is put in an appointment queue:

1. **PutPatientInAppointmentQueue**: This event includes the **PreparePatientForAppointment** event and excludes itself.
2. **PreparePatientForAppointment**: This event includes **PerformAppointment** and excludes itself.
3. **PerformAppointment**: This event includes **CreateMedicalReport** and excludes itself.
4. **CreateMedicalReport**: From this point forth, there are five different paths a patient could take, and hence **CreateMedicalReport** has five inclusion relations to the events explained in point 5 to 8 as well as the event **ChargeGovernment**.
5. **RecommendAppointmentWithSpecialist**: Because a *Specialist* may either recommend an appointment with another specialist, request more exams or both, **RecommendAppointmentWithSpecialist** does not exclude **RequestMoreExams**. However, it excludes **RecommendMedicationAndHavePatientReturn**, **RecommendSurgery**, and **ChargeGovernment**, as these are not legal options from here on. **ChargeGovernment** is not an option, since this will not take place at this point. **RecommendAppointmentWithSpecialist** includes **ReferToROTA**, since a specialist may refer the patient to ROTA.
6. **RequestMoreExams**: Has similar relations as those described in point 5, but starting from **RequestMoreExams** instead.
7. **RecommendMedicationAndHavePatientReturn** and **RecommendSurgery** both exclude the events mentioned in point 5 and 6, **ChargeGovernment** as well as each other. They have include relations to both **ReferToROTA** and **CheckInPatient**. If the patient is checked in again, we do not want to **ReferToROTA** to be available for execution anymore, and this explains the exclusion relation from **CheckInPatient** to **ReferToROTA** mentioned earlier in the description of the final workflow.

8. **ReferToROTA** includes only **CheckInPatient**, as this is what is allowed to happen from here on. Notice, that we do not model whether the referral is approved or not; but we assume that in either case, the patient will return to the hospital. We do not include the rejection or approval of the ROTA referral as this takes place outside of the hospital.

Now, the second branch will be described. To clarify, this is the branch in which a patient was put into an examination queue:

1. **PutPatientInExaminationQueue**: This event includes the **PreparePatientForExamination** event and excludes itself.
2. **PreparePatientForExamination**: This event includes **PerformExamination** and excludes itself.
3. **PerformExamination**: This event includes **CreateMedicalExaminationReport**, and excludes itself.
4. **CreateMedicalExaminationReport**: This event includes **ChargeGovernment**, and excludes itself. **ChargeGovernment** has its pending value set to true, because of the response relation from **CheckInPatient**.

ChargeGovernment is the event which puts the workflow in a successful state, since the *Pending* value of **ChargeGovernment** is set to true, when a patient is checked in. When executing this event the government is finally charged. The government is not charged with each visit to the hospital, but is eventually charged for the full treatment of a patient.

It is possible for both branches to end in this state.

4.4 Workflow Given by Lecturers

Another workflow presented in the system is given by the lecturers and is designed by *Søren Debois, Thomas Hildebrandt, and Tijs Slaats*. They created a mortgage credit application workflow based on the process of applying for a mortgage in the Danish mortgage institution BRFKredit. The workflow is reimplemented in DCRGraphs.net using the same notation as previously presented workflows in this report, and also to be able to simulate the workflow in the system developed in this project.

Besides the roles found during the analyses, the team chose to add a role called *Admin* to all events on all workflows, such that an admin user can execute all events. This will be explained in Section 5 Usermanual.

5 Usermanual

The VS solution contains nine different VS projects:

- A Client project, containing a Windows client which is used for interacting with the workflows and events.
- A Server project, containing a Server which stores information about workflows.
- An Event project, containing an EventAPI which stores, handles, and simulates a number of events.
- Furthermore the VS solution has a DCR graph parser which is only meant to be used for debugging and administration. The graph parser is therefore not tested or documented.
- Besides these projects, a Common project is used for shared classes.

- Four test projects that test the Client, Server, EventAPI and Common subsystems respectively.

The team have chosen to use Microsoft Azure as the web hosting solution for the Server and EventAPI projects. Read more about this in Section 6.7.3 System Deployment.

5.1 Compiling the Project

This section describes how the user can run the supplied software on a Windows PC.

The team has strived for making it as easy as possible for the user to be able to run the programs and start the two web service projects EventAPI and Server.

To be able to compile the project the user is required to have Microsoft Visual Studio 2013 installed. Open the BNDN solution file - found in <Handin-folder>/Code/BNDN.sln - choose “Build Solution” from the “BUILD” menu.

When compiled, the client can be run by right clicking the Client project, choosing “Debug”, and then “Start new instance”.

When compiled in Debug mode, the EventAPI will contact a localhost Server when needed. When compiling in Release mode, it will contact the Azure hosted Server. This is described in Section 5.4 Running Locally.

5.2 Running the Client

When first compiled, the Client will automatically be set to contact the Server running in Azure. Just double click the Client.exe - found in <Handin folder>/CompiledClient/ - with the Flow icon to start the program.

5.2.1 The Login Window

The first thing shown to the user is a login window, see Figure 8. When the application is run for the first time the Username and Password fields will be empty with a watermark that states what the fields are used for.

The top field is for the username. The next field is for the password. In the system, passwords are case sensitive.

To login with the entered username and password, click the “Login” button in the bottom of the window.

Between the password field and the “Login” button there is an empty area. This area is used to give the user feedback on what is happening behind the scenes.

The workflows described in Section 4.3 Final Workflow and 4.4 Workflow Given by Lecturers have all been added to the system to get the user started. The usernames are the roles listed in Section 4 Workflow elaboration, and the password is set to “Password” without quotation marks. Remember that each workflow has the user *Admin*, which can execute all events.

5.2.2 The Main Window

When the user is logged in a new window appears. It has two main components: The list of workflows, and the list of events for the selected workflows. These components can easily be found in Figure 9. The list of workflows will show all of the workflows in which the user has been assigned a role. This means that a *GasStationOwner* will not see workflows related to hospitals. The first workflow in the list to the left is automatically selected for the user.

When a workflow is selected, only the events the user has access to is shown. It might take a short while for the data to be retrieved. Each event has a name, three boxes which informs the user of the state, and an “Execute” button which is enabled if the event is executable.

The “Refresh” button in the bottom left corner will reload all workflows from the Server, and then load the events of the first workflow in the workflow list.

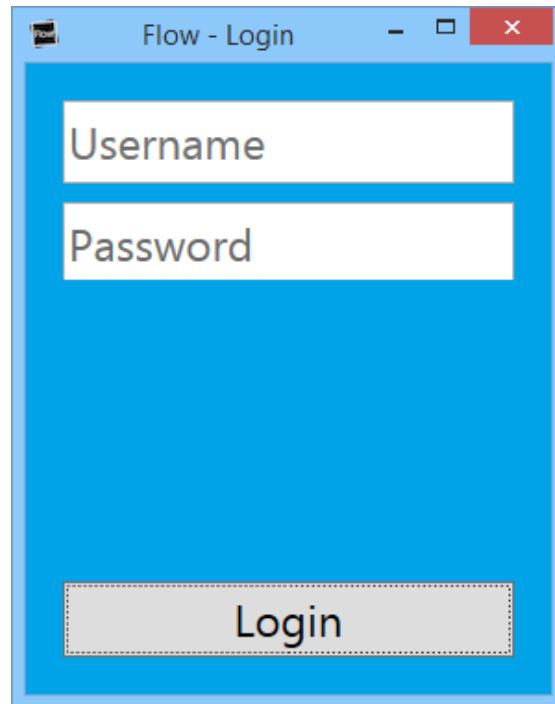


Figure 8: Screenshot of Client's Login Window

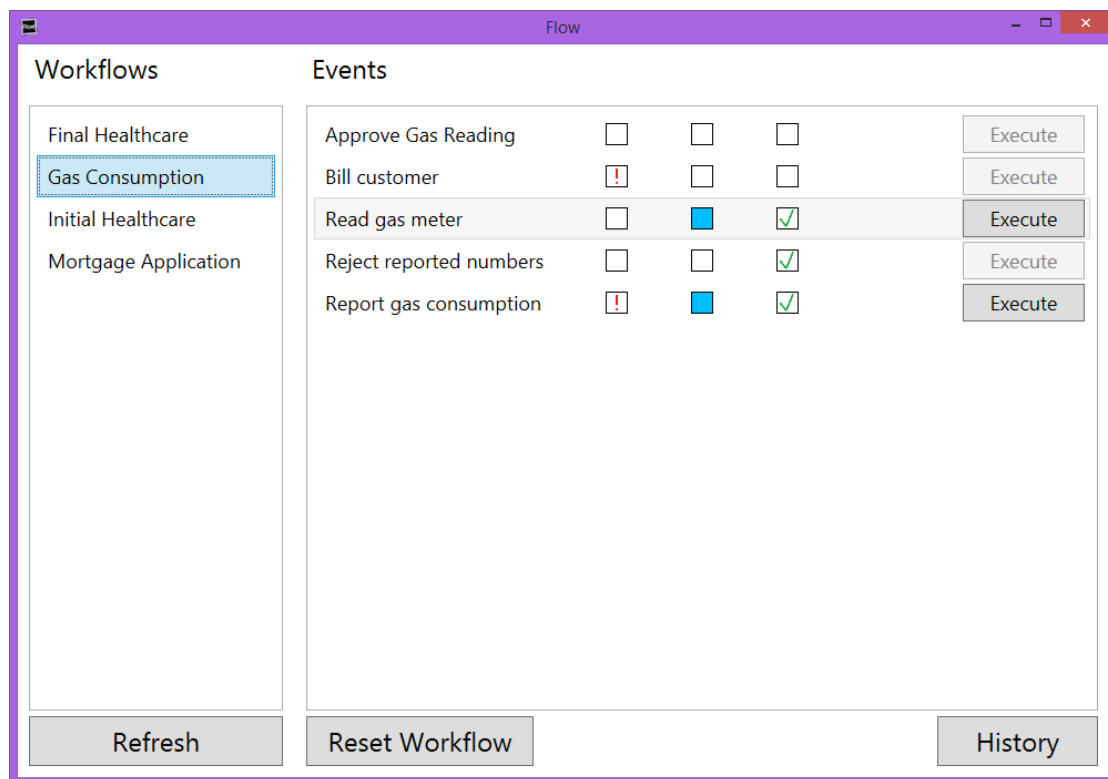


Figure 9: Screenshot of Client's Main window

5.2.2.1 State of Events Each event shown in the client has three boxes showing its state, see Figure 10

- The leftmost box states whether the event is pending or not. There will be a red exclamation mark when the event is pending.
- The center box tells whether the event is included or not. If the box is blue the event included. Otherwise it is excluded.
- The right hand box tells whether an event has been executed at least once, or not. When the event is executed a green checkmark will appear in the box.

The “Execute” button will only be enabled when the event is executable. When enabled and clicked, all “Execute” buttons will be disabled and then try to execute the event. If execution succeeds the events will refresh their state to show the new state of the workflow, otherwise if something goes wrong the status bar will be updated.

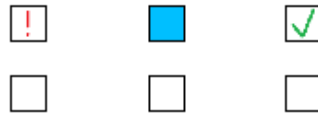


Figure 10: Screenshot from Client’s Mainwindow, demonstrating the possible look for the three fields for the state of an event

5.2.2.2 The Status Bar In the main window a status bar will appear in the top right corner if an error occurs, see Figure 11. An error will be presented in a red color in the status bar. If the error occurred during execution of an event, the “Execute” buttons will not be enabled again until the “Refresh” button is clicked.



Figure 11: Screenshot from Client’s Mainwindow, demonstrating the status bar displaying an error

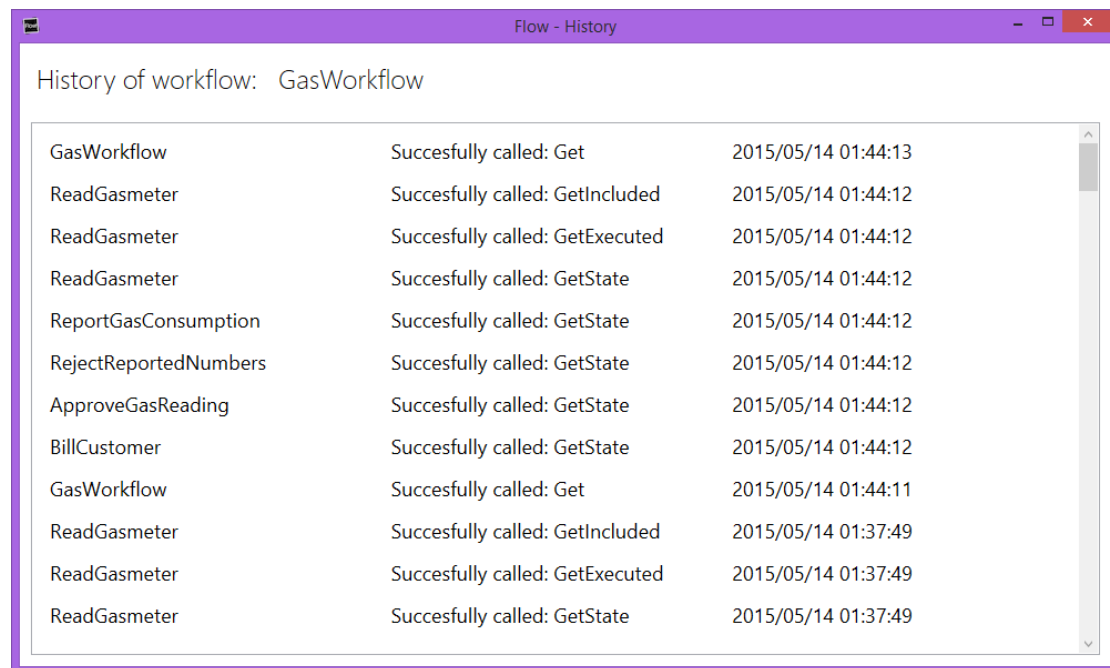
5.2.2.3 Reset Workflow The team chose to implement a “Reset Workflow” button, which reset all events within the current workflow to their initial values. When the reset is done, the client will refresh all workflows and events to reflect the new state of the workflow.

5.2.3 History

In the bottom right corner of the main window a “History” button is located. When clicked, it will open a new window, see Figure 12.

This window will show the workflow specific history of the Server, and the history for each of the events in the workflow with the newest history entries first.

The History window has a status bar in the top right corner. If anything should go wrong when interacting with the Server or EventAPIs, a status text will appear.



Flow - History

History of workflow: GasWorkflow

GasWorkflow	Successfully called: Get	2015/05/14 01:44:13
ReadGasmeter	Successfully called: GetIncluded	2015/05/14 01:44:12
ReadGasmeter	Successfully called: GetExecuted	2015/05/14 01:44:12
ReadGasmeter	Successfully called: GetState	2015/05/14 01:44:12
ReportGasConsumption	Successfully called: GetState	2015/05/14 01:44:12
RejectReportedNumbers	Successfully called: GetState	2015/05/14 01:44:12
ApproveGasReading	Successfully called: GetState	2015/05/14 01:44:12
BillCustomer	Successfully called: GetState	2015/05/14 01:44:12
GasWorkflow	Successfully called: Get	2015/05/14 01:44:11
ReadGasmeter	Successfully called: GetIncluded	2015/05/14 01:37:49
ReadGasmeter	Successfully called: GetExecuted	2015/05/14 01:37:49
ReadGasmeter	Successfully called: GetState	2015/05/14 01:37:49

Figure 12: Screenshot from Client's History Window

5.2.4 The Settings File

When a user is successfully logged in through the Client, a small settings file will be saved on the disk. This file will include the username of the last successful login, and the address of the Server on which the login was completed. If the settings file does not exist, the Client will point to the hosted Azure Server. This can be changed through the settings file.

The settings file is saved as settings.json in the same directory as the Client.exe file. It is a JSON object with up to two properties, ServerAddress and Username. If the ServerAddress is set to "http://localhost:13768/" the Client will access the local Server.

5.3 The DCR Graph Parser

Because the team did not want to issue every HTTP request manually every time a workflow should be created in the system, a simple DCR graph parser was developed, see Figure 13. The parser can upload workflows and events to the Server and EventAPIs respectively. The program was developed as an internal tool, and as such not a tool designed for hand-in. The team thought it might be useful if the readers wanted to try out another workflow than the ones provided.

The parser will not handle failures. It does not clean up after itself if an error should occur. A short user manual for the DCR graph parser can be found in the Appendix, Section C Usermanual for DCR graph parser.

5.4 Running Locally

From Visual Studio it is possible to host the team's EventAPI and Server locally. In order to make the EventAPI contact the local Server, the Solution Configuration must be set to Debug, see Figure 14. If the configuration is set to Release, the EventAPI will contact the hosted Server instead.

Visual Studio is set up to start an instance of the Server and EventAPI, if they are not running already. The server is now running, and the DCR graph parser or client can be started

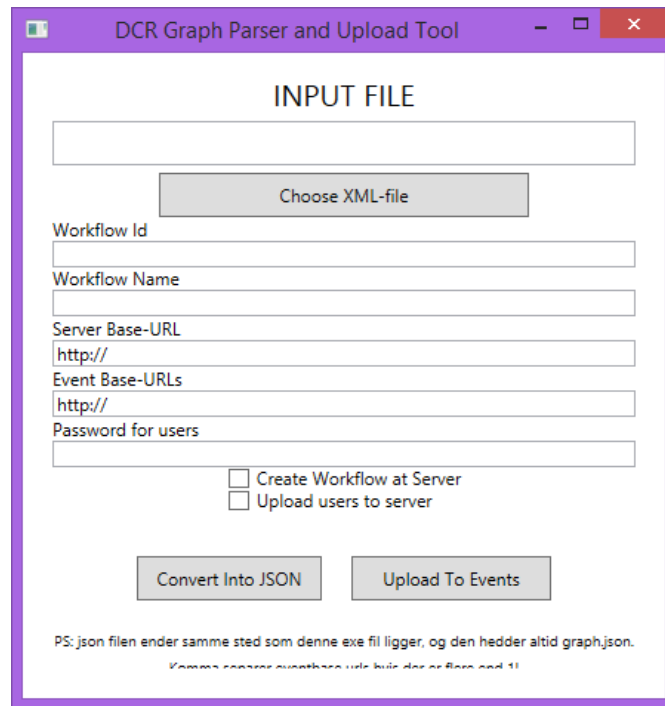


Figure 13: Screenshot of the simple DCR graph parser, that was developed

In some cases our database system cannot automatically create the database. We have provided possible solutions for these problems in Appendix, Section D Possible solutions for creating the database

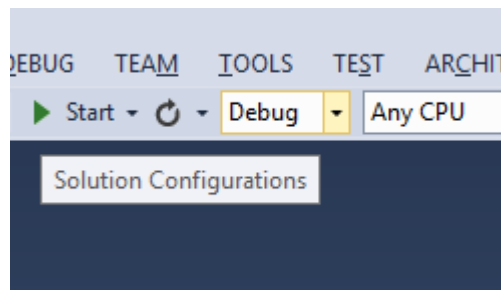


Figure 14: Screenshot describing where to set Solution Configuration in Visual Studio

6 System Implementation

This section presents how the system was implemented. The system's main parts are presented, after which individual parts will be explained in greater detail. This section also covers how the team implemented persistence, concurrency and role-based access control. The system was written and tested using the programming language C# on the .NET platform

6.1 Overview

The following has been implemented in the final system:

- A peer-to-peer architecture which can be distributed across an arbitrary number of web services.
- The workflow system is generic in the sense that it supports not only a single, built in workflow. Furthermore a workflow is able to dynamically change while it is running.
- Workflow events have *location transparency*; events can be held at different servers or at the same server. The internal logic does not differ across locations.
- A *graphical user interface* that provides a more appealing user experience for the end user to interact with.
- *History*, enabling the user to see a log of what happened at both the Server and at a single event.
- *Persistence*, if the machines running the Server or an EventAPI are shut down, once restarted they will be able to restore from where they left off.
- Role-based access control, which ensures that a user is only able to execute events that the given user has permission to.
- *Locking* prevents simultaneous access to events. Simultaneous access could otherwise cause a corrupted state in a workflow.

6.2 Overall Architecture

This section aims at providing a panoramic view of the architecture of the delivered system. Now, a brief explanation of the three main parts, that constitute the delivered software:

6.2.1 Client

The client is the software an end user will typically interact with when using the system. The Client was described in the Section 5 Usermanual. The user uses the Client to execute events in a workflow, reset a workflow, or see a history of what happened at the events and at the Server. The Client interacts with the Server and possibly several EventAPIs.

6.2.2 Server

The Server is a centralized instance that holds information about all workflows. For each workflow it provides the addresses of all events in the given workflow. The Server is contacted by the Client, when the Client wants to know what workflows exist at the Server and know what events are related to a specific workflow.

Furthermore, the EventAPI also interacts with the Server, when an event wants to add itself to an already existing workflow at the Server. The Server is intended to be a REST-based service, see Section 3.3 REST

In the current setup, the Client has no way of discovering the Server automatically, and hence the Client is hardcoded to the address of the Server, stored in a configuration file. It is possible to have multiple Servers, however a workflow must be located at a single Server. Furthermore a Client can only contact one server per instance.

6.2.3 EventAPI

The EventAPI holds events and is responsible for the execution of events. The Client will contact the EventAPI when asking for the state of events. The EventAPI contacts the Server when events are created or deleted. EventAPI is a REST-service.

The three main subsystems and their interactions are sketched in the Figure 15.

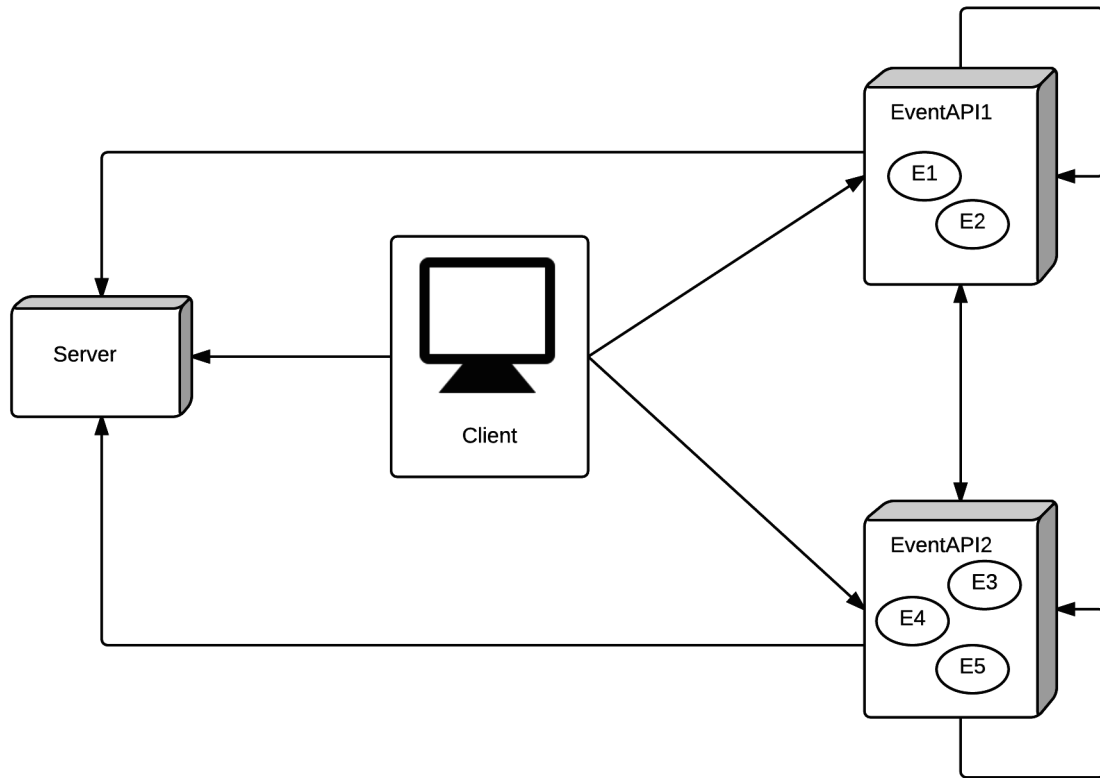


Figure 15: Illustration of the system's three main components and their interaction.

6.3 Interface-based Programming

This section presents how the team decided to let the system architecture be structured around interface-based programming.

At the drafting of the initial system architecture, strict boundaries of the responsibility of modules were a clear goal. Using interfaces to specify functionality of a component and only exposing the interface to other components of the system helps in achieving the desired separation. Necessary methods were drafted for every interface before focusing on implementation details. When implementation of the system began, using interfaces enabled quicker prototyping, e.g. when implementing storage and retrieval retrieval of data in the system.

The team had specified a storage interface for storing objects, which could quickly be implemented using in-memory lists. This meant that work could move on to other components, without spending too much time perfecting the storage component.

Swapping modules was also made easier using interface-based programming. For instance after implementing the necessary functionality of other components, the team was able to switch out the concrete implementation of the storage component with a persistent storage module. This was possible without having to rewrite or modify other components of the system.

Using interfaces for adding functionality to existing implementations was also of great use in the system architecture. After having implemented most functionality, new requirements were added to the project description. This meant that some extra functionality - namely logging of requests and errors - had to be added.

Specifying this functionality in an interface and then making sure that existing modules implemented this interface enabled an easy implementation without major issues.

6.4 Dependency Injection

Dependency injection⁶ is a technique in software development that allows the developer to inject an implementation of an interface into a given module that depends on that interface. We use constructor injection meaning the dependency is given in the constructor call as seen in the example code-snippet in Figure 16. Dependency injection hence allows for injecting different implementations of the same interface, e.g. into `EventStorage`.

```
public EventStorage(IEventContext context)
{
    if (context == null)
    {
        throw new ArgumentNullException("context");
    }
    _context = context;
}
```

Figure 16: Here, in `EventStorage`, a context, implementing `IEventContext`, is passed to the constructor, that the `EventStorage` is to use.

A major motivation for using dependency injection was not to inject different implementations during runtime, but instead for the team to mock dependencies when unit testing classes. The use of mocking is described in Section 7.1 Unit Testing.

Dependency injection is very useful for testing since it is possible to test a module in isolation by injecting mocked interfaces into the module being tested. An example of isolation is to not run storage tests against the actual database but instead against a mocked implementation of the database.

This enabled testing in a controlled environment and allowed for more verifiable tests without side effects.

6.5 Design of Client

This section presents the design of the Client. When the team designed the system, the intent was to make the Client "as stupid as possible".

The intention was that the Client should provide a visual representation of raw data with a thin shell of error handling. To some extent this design was achieved.

The only data logic the Client has implemented is the filtering of workflows and events. The rest of the Client is just lists of data, which is presented in a design which the team believes presents a good overview.

6.5.1 MVVM-principles in Client

The Client was implemented with the Model-View-ViewModel (MVVM) design principle in mind. Using Windows Presentation Foundation (WPF) this is made possible by "binding" views to the properties of view models.

The MVVM principle makes it possible to separate the views from both the model and the business logic. Additionally, it converts data into a form suitable for presentation in the view, by adding a layer - the view model - in between the view and the model. The view model also maps user actions to logic functionality.

In the system, all view models inherits from a base class, which enables invoking a method whenever a property changes. WPF uses this to update the views in the GUI, whenever data changes.

⁶http://en.wikipedia.org/wiki/Dependency_injection

Using MVVM also enables easier testing, since all user interface actions are methods of the view models.

6.6 Design of Server and EventAPI

This section will give an insight into the design characteristics of the Server and the EventAPI. The team has chosen to use ASP.NET WebAPI⁷ to implement the web services Server and EventAPI. ASP.NET WebAPI uses Controllers, which are objects that handles incoming HTTP requests. In the following, Controller should hence not be confused with a Controller as found in the Model-View-Controller⁸ design pattern.

6.6.1 Multi-layered Design

We have based our architectural design of Server and EventAPI on a multi-layered approach. With a multi-layered approach the team achieved separation between classes - low coupling - and independence among classes - high cohesion. Each layer has a distinct responsibility, and provides its service to the layer above it. How the Server and EventAPI are implementing layered architecture, can be seen in Figures 17 and 18. It can be seen that both the Server and EventAPI has the layers Controller, Logic, and Storage. The Controller layer is responsible for handling HTTP requests. The Logic layer implements the business logic, concurrency, and access control. The Storage layer facilitates storage and retrieval of data needed by the Logic layer. Additionally the EventAPI has a Communication layer which provides communication to the Server or other EventAPIs.

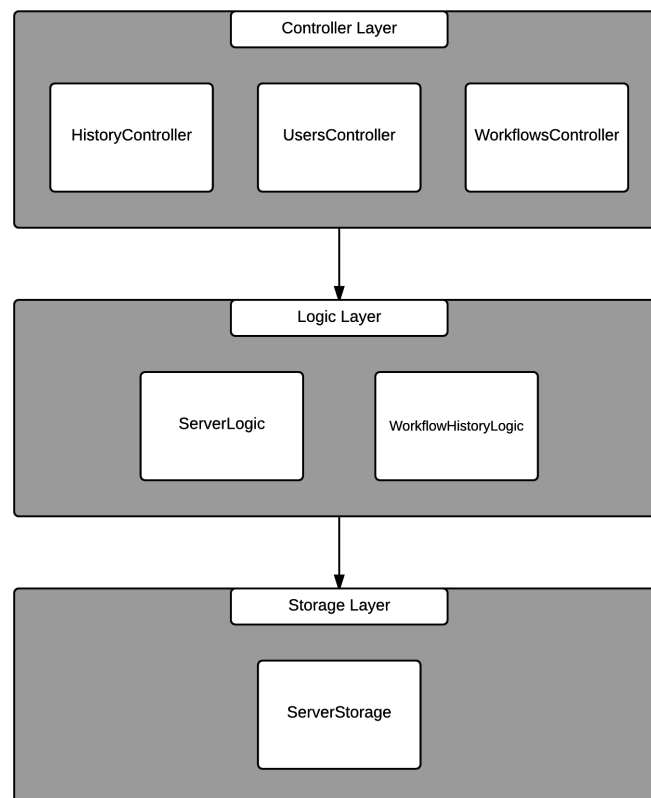


Figure 17: Representation of the layers in the Server

⁷<http://www.asp.net/web-api>

⁸http://en.wikipedia.org/wiki/Model_View_Controller

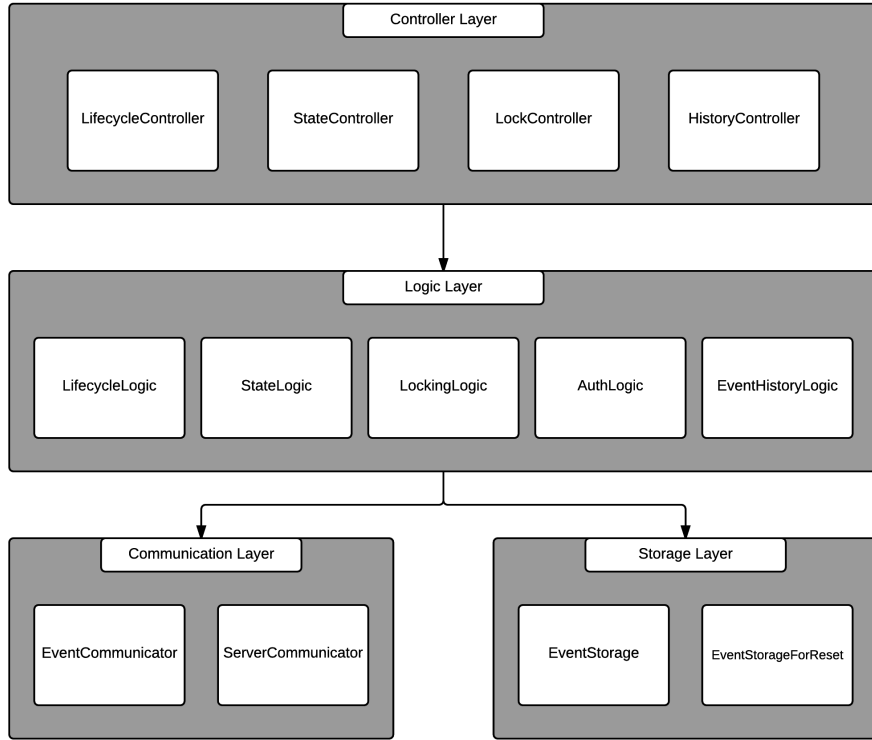


Figure 18: Representation of the layers in the EventAPI

6.6.2 Execution of an Event

This section intends on giving the reader a clearer understanding of what takes place and in what order, when an event is requested to execute.

The algorithm for execution of an event **A** is:

1. **A** receives HTTP request for execution.
2. **A** checks which event needs to be executed.
3. **A** checks whether the sender of the HTTP request has the necessary role to execute the event.
4. **A** checks whether it is currently locked. If it is, the request is put into a lock queue and wait until **A** is unlocked.
5. Check whether **A** is executable. This includes
 - (a) **A** checks if its Include value is true.
 - (b) **A** checks for each of its conditions that they are either executed or excluded.
6. **A** attempts to lock all response, inclusion, and exclusion events ordered alphabetically by Id. The lock should use the eventId of the **A** as the lockowner; this includes locking itself the same way. Read more in the Section 6.12 Concurrency Control.
7. **A** requests all response events to update their pending to true.
8. **A** requests all inclusion events to update their included to true.

9. **A** requests all exclusion events to update their included to false.
10. **A** sets its own Executed to true.
11. **A** sets its own Pending to false.
12. **A** attempts to unlock all of the events it previously locked.
13. **A** responds to the caller that everything went well.

If a check fails, or failure occurs, event **A** will unlock all locked events and respond to the caller with an error message.

The implementation of this algorithm is initiated by **StateController**'s **Execute** method and delegated to **StateLogic**'s **Execute** method.

6.7 Distribution

One of the requirements to the system was that it had to be distributed.

The intention was that events should be distributed across several EventAPIs. The fact that the Server is centralized could potentially create a performance bottleneck and the Server has therefore been developed to have as little responsibility as possible to minimize this issue. Reducing this bottleneck comes down to two things: Handling as few requests as possible and not doing heavy processing. The sooner a Client can be routed away from the Server to EventAPIs, the better. The Server supplies the Client with a list of events. From then on, the Client now communicates directly with EventAPIs bypassing the Server as a middle man.

Since events can be deployed on an infinite number of EventAPI instances, the system is distributed. Processing requests across several EventAPI instances - instead of just a single EventAPI - helps balancing the workload to avoid performance bottlenecks. Making the system distributed presented some problems which are explained in Section 6.12 Concurrency Control

6.7.1 Location Transparency

The EventAPI has been designed in such a way that it does not distinguish between events located at the same EventAPI and events located at other EventAPI instances. The same logic is used for either case.

This implies that an EventAPI will issue a HTTP request to itself, even though the target event is located at the same EventAPI instance.

Therefore, the team believes events have location transparency.

6.7.2 Global Identification of Events

An event is uniquely determined by the combination of its own ID (`eventId`) and the ID of the workflow the event belongs to (`workflowId`). Two events on the same workflow cannot share the same `eventId`. Therefore two events cannot share the same global ID. This property allows for an event to be uniquely determined in the system and it allows exposing a directory-like structure in the URLs as prescribed by REST - see Section 3.3 REST. EventAPIs locate their resources as such:

`<baseurl>/events/<workflowId>/<eventId>`

As an example, the event **Read Gasmeter** belonging to the workflow Pay Gas would be located on the following URL:

`http://www.myRestService.com/events/PayGas/ReadGasmeter`

6.7.3 System Deployment

The nature of a distributed system means that it is running across several computers. Hosting the system on a cloud platform like Amazon EC2 or Microsoft Azure was an obvious choice. With the team's chosen system architecture, the system requires the Server to be present. Hosting the Server in a fixed, known location makes communication between the Client, the EventAPI, and the Server easier.

Choosing Microsoft Azure for hosting the system instead of other hosting platforms was simply a result of the convenience of using Azure. The platform has Visual Studio integration and Microsoft supplies students at the IT University of Copenhagen with a free hosting service that offers enough scalability for this project. The entire backend of the system is therefore hosted on Azure across several virtual machines. One virtual machine hosts the Server that Clients access when they first enter the system. Several EventAPIs are hosted across other virtual machines.

6.8 RESTful APIs

This section covers to what extent the team believes the Server and the EventAPI are RESTful. API is here used to refer to the Server and the EventAPI.

6.8.1 Restfulness of the System

The APIs exposes their resources through a RESTful interface. Creating objects in the system is done through HTTP POST requests. Updating objects and information is done using PUT requests. Retrieving information is done through GET requests, where the request has no information apart from URL request parameters.

No knowledge of the implementation of the APIs is accessible from outside the system. Only incoming and outgoing information is accessible unless stated below.

Users of the APIs do not need to know about the internal state and representation of data in the APIs. Users of the APIs send and receive information in Data Transfer Objects - DTOs - as JSON.

The APIs act as resource providers that provide information when requested to do so, without needing to know anything about the users of the APIs and vice versa.

RESTful layering is also present in the implementation in the sense that a user of the APIs at no given time knows whether it is communicating directly with an API or whether the request is handed to an intermediary.

6.8.2 UnRESTful Parts of the System

Some behaviour at the APIs cannot be described as RESTful. The **Execute** method on the EventAPI's **StateController** is one example. The intent of this method is to execute an event. **Execute** is called with a PUT request. A PUT request should replace the targeted resource with the resource provided in the request. However, **Execute** does not. The resource that is given in the PUT request is instead used as input for executing the method.

The central problem is that the execution of an event hardly corresponds to a resource, instead it is an operation that the caller wants the EventAPI to carry out.

The second problem is that this deviation propagates to the caller of the EventAPI as well. Now the caller also needs to know that the method should be invoked through a non-compliant HTTP PUT request. Now there is coupling between the implementation on EventAPI and how clients should invoke **Execute**. Architecturally this resembles SOAP⁹ more than REST, since it is function based.

⁹<http://en.wikipedia.org/wiki/SOAP>

A possible fix for this would be to PUT some sort of Execute resource to a given event, which would then be stored on the event after execution. It could be argued that a resource is then simply stored at the event even though the state of the event changes.

6.9 Minimal Logic Handling in Controllers

The following section describes the implemented structure of the controllers in the Server and the EventAPI. Controllers across the EventAPI and the Server share architectural similarities since both are implemented as REST web services.

During the implementation of the Server and the EventAPI it was a design goal for the Controllers to do as little work as possible besides receiving the incoming HTTP requests and checking for invalid input.

It was therefore the intention of the design that controllers should only have the following four responsibilities:

- Checking that input can be converted into an instance of the given argument type
- Delegate the call to a logic layer
- Catch and form relevant exceptions which is mapped to HTTP responses
- Request a history entry to be recorded in all three cases

As far as possible the controllers should not handle any logic, but instead simply pass on the incoming information to another layer that will then process and return the necessary information. This ensures encapsulation of responsibility and enables the use of several smaller classes for handling domain-specific logic. The team aimed for several controllers, since one big controller with a lot of different functionality is hard to test and reuse.

An implementation of the aforementioned design intention is presented below, see Figure 19.

6.9.1 Exception and Response Handling

First of all, there are a number of exceptions that may arise through the execution of an HTTP request. The team's exception handling approach comes down to distinguishing between two types of exceptions. Those that can be handled locally and those where the exception are propagated all the way up to the controller level. In a given scenario the responsibilities of the involved components determine the scenario type.

In the following subsections the handling of either of these two types of exceptions will be elaborated.

6.9.1.1 Exception is Handled Immediately If an exception can be dealt with locally and the upper layers do not need to know about what caused the exception, an action is taken accordingly. An example of such an exception scenario is found in the logic layer. Assuming an event during execution has locked four out of six related events, but when attempting to lock the fifth an exception is thrown. The logic must unlock the four locked events before returning to the caller. In this scenario the logic layer can - and should - deal with the exception. The requested operation cannot be completed and therefore a "clean up" by unlocking the four events.

The implementation of this is presented below, see Figure 20.

```

[Route("events/{workflowId}/{eventId}/lock")]
[HttpPost]
public async Task Lock(string workflowId, string eventId, [FromBody] LockDto lockDto)
{
    if (!ModelState.IsValid)
    {
        var toThrow = new HttpResponseException(Request.CreateErrorResponse(HttpStatusCode.BadRequest,
            "Provided input could not be mapped onto an instance of LockDto"));
        await _historyLogic.SaveException(toThrow, "POST", "Lock", eventId, workflowId);
        throw toThrow;
    }

    try
    {
        await _lockLogic.LockSelf(workflowId, eventId, lockDto);
        await _historyLogic.SaveSuccessfulCall("POST", "Lock", eventId, workflowId);
        return;
    }
    catch (ArgumentNullException e)
    {
        _historyLogic.SaveException(e, "POST", "Lock");

        throw new HttpResponseException(Request.CreateErrorResponse(HttpStatusCode.BadRequest,
            "Lock: Seems input was not satisfactory"));
    }
}

```

Figure 19: A code snippet from `LockController` in `EventAPI`. `Lock` checks for input validity, and if it is valid, it delegates the work to the `LockingLogic` layer. The try-catch design will be discussed in the following section.

6.9.1.2 Exception is Propagated Upwards If it is not possible for a layer to take proper action when an exception is thrown, it is propagated upwards to a layer which has interest in and can handle the exception. When an exception occurs and the operation has to abort, the user of the Server or EventAPI must be notified of the error.

In these scenarios there really is no obvious action to take in the lower layers. In some cases it is even possible for a layer to wrap the existing exception in another exception type which provides more information to the calling layer. If an exception is propagated to the controller layer, an appropriate HTTP response, based on the exception, is sent to the caller.

For instance, if a request is made to create an event with an ID identical to the ID of an already existing event, no countermeasure besides returning a bad request response to the caller exists. The lower layer should not determine what to do here, and therefore it propagates the exception upwards to the upper layer. `HttpResponseExceptions` thrown by the controller layer results in an HTTP error code being returned to the caller to give an idea of what went wrong.

In the code-snippet seen in Figure 21, it is realized that the event that is to be created have already been created. An `EventExistsException` is therefore thrown. This exception is then allowed to propagate up to `LifecycleController`, seen in Figure 22, where it is caught. The catching of the exception ultimately leads to `LifecycleController` issuing a bad request response.

This also explains the need for the try-catch blocks pointed out in the previous section. Note that catching different exceptions lead to slightly different HTTP response exceptions being issued to the caller each with a more descriptive error message than a default error message.

It is important to note that with this approach the decision on what type of response to issue back to caller is made at the controller layer.

One could imagine an alternative approach where the throwing layer - in this case `EventStorage` - would decide on what response to return. This would break the encapsulation of the class since

```

public async Task<bool> LockList(SortedDictionary<string, RelationToOtherEventModel> list, string eventId)
{
    if (eventId == null || list == null)
    {
        throw new ArgumentNullException();
    }
    var lockedEvents = new List<RelationToOtherEventModel>();
    // For every related, dependent Event, attempt to lock it
    foreach (var tuple in list)
    {
        var relation = tuple.Value;
        var toLock = new LockDto { LockOwner = eventId, WorkflowId = relation.WorkflowId, EventId = relation.EventId };

        try
        {
            await _eventCommunicator.Lock(relation.Uri, toLock, relation.WorkflowId, relation.EventId);
            lockedEvents.Add(relation);
        }
        catch (Exception)
        {
            break;
        }
    }

    // if something has gone wrong, the lockedEvents list only includes some of the relations in the list.
    // Therefore is the sizes are not equal - the ones who are locked must be uncoked.
    if (list.Count != lockedEvents.Count)
    {
        await UnlockSome(eventId, lockedEvents);

        return false;
    }
    return true;
}

```

Figure 20: Code-snippet from LockingLogic LockList method. Example of an exception that are thrown in modules below the LockingLogic layer, but are handled locally. Local exception handling is performed because the layer can actually do something about the exception.

```

public async Task InitializeNewEvent(EventModel eventModel)
{
    if (eventModel == null)
    {
        throw new ArgumentNullException("eventModel", "eventModel was null");
    }

    if (await Exists(eventModel.WorkflowId, eventModel.Id))
    {
        throw new EventExistsException();
    }

    _context.Events.Add(eventModel);

    await _context.SaveChangesAsync();
}

```

Figure 21: Code-snippet from the method InitializeNewEvent in EventStorage

a lower layer would interfere with the responsibilities of a higher layer.
By throwing an exception stating what the issue was at the lower level and then let top layers

```

try
{
    await _logic.CreateEvent(eventDto, ownUri);
    await _historyLogic.SaveSuccessfulCall("POST", "CreateEvent", eventDto.EventId, eventDto.WorkflowId);
    return; // Important that we return here.
}
catch (EventExistsException e)
{
    _historyLogic.SaveException(e, "POST", "CreateEvent");

    throw new HttpResponseException(Request.CreateErrorResponse(HttpStatusCode.BadRequest,
        "CreateEvent: Event already exists"));
}
catch (ArgumentNullException e)
{
    _historyLogic.SaveException(e, "POST", "CreateEvent");

    throw new HttpResponseException(Request.CreateErrorResponse(HttpStatusCode.BadRequest,
        "CreateEvent: Seems input was not satisfactory"));
}
catch (FailedToPostEventAtServerException e)
{
    _historyLogic.SaveException(e, "POST", "CreateEvent");

    throw new HttpResponseException(Request.CreateErrorResponse(HttpStatusCode.InternalServerError,
        "CreateEvent: Failed to Post Event at Server"));
}
catch (FailedToDeleteEventFromServerException e)
{
    _historyLogic.SaveException(e, "POST", "CreateEvent");

    // Is thrown if we somehow fail to PostEventToServer
    throw new HttpResponseException(Request.CreateErrorResponse(HttpStatusCode.InternalServerError,
        "CreateEvent: Failed to delete Event from Server. " +
        "The deletion was attempted because, posting the Event to Server failed. "));
}
}

```

Figure 22: Code-snippet from the method `CreateEvent` in `LifecycleController`

handle the exception, we encapsulate the responsibilities of the layers.

6.10 Persistence

This section describes how the finished system achieves persistence on the Server and the EventAPI. In this section an entity refers to the in-memory representation of a row in a relational database.

Data persistence is needed in case of system crashes or restarts. Therefore both the Server and the EventAPI implement data persistency in the form of an SQL database.

To map a relational data model to in-memory objects Microsoft's Entity Framework was used. By having persistent data, the system became more robust. This became more apparent when the deployment on Microsoft's Azure hosting platform was taken into consideration. Azure will shut down the deployed Server and EventAPIs when not in use. Resuming from a stored state is therefore a necessity if data should not be lost.

6.10.1 The Relational Database

To persist data of events and workflows on both the Server and the EventAPI two relational data models were created. When mapping objects to relational data the concepts of redundancy and normalization were used.

Data models are only used in the storage layer of the subsystems. POST and PUT requests with DTOs are converted to entities by a logic layer and are finally persisted. Similarly GET requests requires the logic to retrieve entities from the database and convert them to the wanted DTOs which can then be sent with a HTTP response. Data independence was achieved by having different kinds of data - data for transferring, DTOs, and data for saving, entities. This enabled implementing new DTOs or easily adding data to an existing entity without changing anything in the DTOs.

The data model of the EventAPI contains seven models, and can be seen on Figure 23. The History entity has no relations to other entities which is intentional, because History data should not be deleted - even if an event is deleted. Furthermore, four models are created - one for each type of graph relation. This design choice was made to be able to extend the relations individually in the future. These models derive from the same base class and it is therefore easy to extend all of them simultaneously.

The three fields *InitialExecuted*, *InitialPending*, and *InitialIncluded* on the Event model are used to reset an event to its initial state. Multiple roles on an Event are allowed which is seen on the one-to-many relation from Event to Role.

Notice that an Event has a composite key of the WorkflowId and EventId - two events can have the same Id as long as they do not exist on the same workflow - see Section 6.7.2 Global Identification of Events.

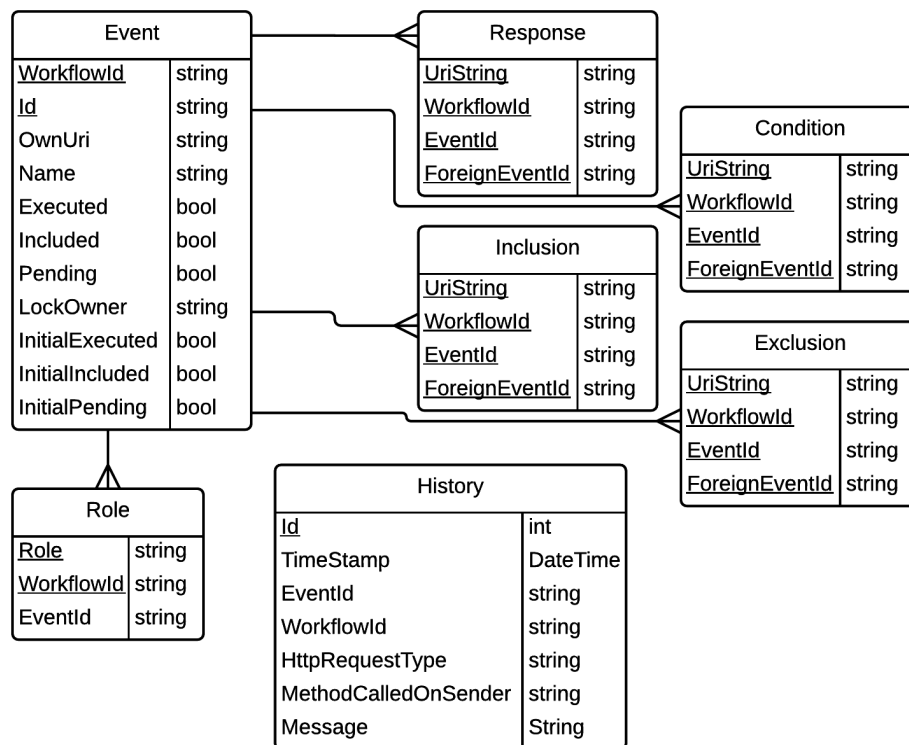


Figure 23: The Data Model of an EventAPI

The data model of the Server can be seen in Figure 24. The data model contains seven entities.

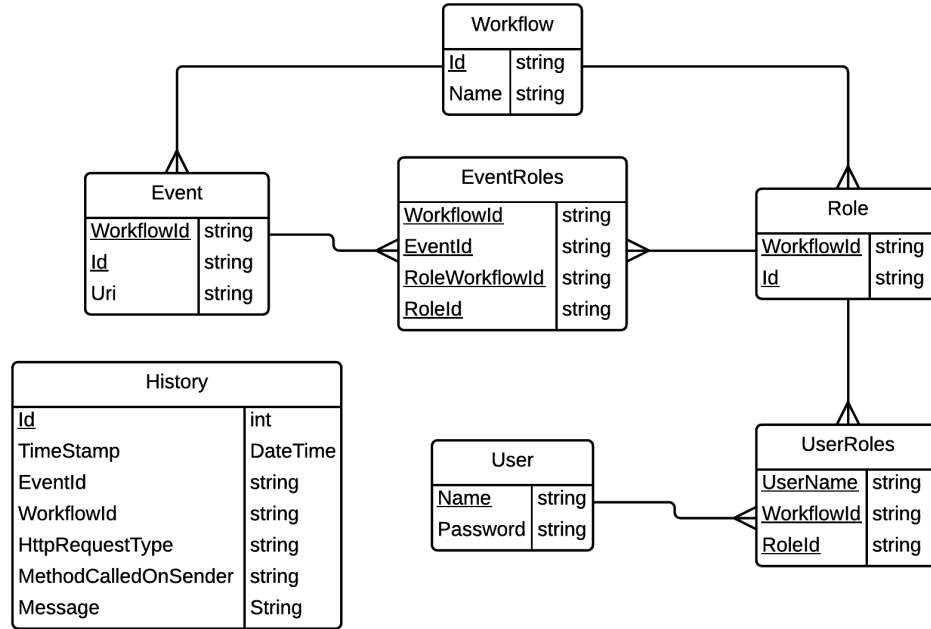


Figure 24: The Data Model of the Server

Notice the EventRoles table. This table is created to normalize the many-to-many relation between events and roles. The EventRoles entity is never used outside of the database. This entity helps Entity Framework minimize the amount of redundant data. The model allows for an event to have many roles and roles to be on many events, but a role is unique for a workflow. Since two events should be able to have the same ID on two different workflows, the Event model has a composite key of the WorkflowId and EventId. To support role-based access control a hashed and salted password for each user is stored, see Section 6.11 Role-Based Access Control.

6.10.2 Data Distribution

The aforementioned data models allows storage of workflow and event information on the Server and the EventAPI, respectively. By storing data in multiple locations a bottleneck is removed since all data does not have to be stored and retrieved in one place.

On the Server, by only storing where to find events, but not any information about the state of each event, the Server and events are encapsulated since the Server has no need to know the state of individual events. An event keeps track of its own state, the Server simply serves event addresses.

Furthermore, fewer errors are encountered by not saving copies of states of related events on each event. Instead events are forced to request data each time they need it. This, once again, encapsulates events and their responsibilities. It should be noted that this benefit comes at the cost of a performance impact.

By distributing data it is not possible to get access to all of the data of events at once and get an overview of the stored information of every event. This can function as a defense mechanism if an attacker would want to access all the data in the system. A problem with distributed data is that the roles of the system - when using role-based access control - have to be agreed on systemwide, which complicates matters.

6.11 Role-Based Access Control

This section describes how RBAC was implemented in the delivered system. RBAC is implemented across the system. Server, Client, and EventAPI all handle roles and in conjunction make RBAC possible.

6.11.1 Motivation

Without RBAC any user would be allowed to execute any event within a workflow. This would allow for troublesome and unintended effects. Examples of this could be drug addicts prescribing themselves morphine or taxpayers approving their own annual statement.

RBAC will prevent users who cannot prove that they have one of the roles required from executing the event. For instance, in the workflow "Pay Annual Gas Bill", only users who can prove themselves to be a *Customer* would be allowed to execute the **Read Gas Meter** event.

In short, because events within a workflow are assigned only to some roles, the system needs a way of ensuring that only people assigned with these roles are allowed to execute the given events. This was the motivation for implementing role-based access control.

6.11.2 Implemented Solution

In the finished system a user can be assigned multiple roles and an event supports execution by multiple roles. This was implemented to make it easy to parse exported graphs from `DCRGraphs.net` into system events.

In the relational database of the Server, user passwords are hashed with the SHA-512 algorithm to ensure that no sensitive information is stored in clear text.

Furthermore, to secure against dictionary attacks¹⁰ the passwords are salted by adding a few randomly generated characters in front of all the passwords before and after they are hashed.

The way the system implements role-based access control is such that a user issues a login call to the Server with a username and password. The Server will then retrieve the user by username, hash the provided password, and try to match the provided hashed password against a salted hash retrieved in the database of the Server.

If a match is found the server will send a list of the roles assigned to the user to the Client. Roles are defined in the workflow a given event belongs to, which means that multiple workflows can share role IDs without giving unwanted access to the wrong users.

The Client receives a dictionary of roles, where the key is the ID of the workflow and the value is a list of roles for the given workflow.

When the Client retrieves the addresses of events from the Server, the Client is able to discard the events on which the user has no execution rights, due to the fact that the Server will map an event to the roles the event supports.

During implementation it was also determined that users should not be able to be given roles that are not used by events in a given workflow.

This limits the possibility of users getting access to events not assigned to them because an administrator of the given workflow did not check the current database for unused roles. Roles

¹⁰http://en.wikipedia.org/wiki/Dictionary_attack

are not removed from the server when it is no longer in use. In this scenario it is possible for a user to be assigned an unused role.

6.11.3 Discussion of the Implemented Solution

The implemented solution is not ideally secure. All communications are sent via the HTTP protocol which transfers data in clear text. HTTPS could have been enforced but was not done due to HTTPS warnings caused by using self signed, and therefore untrusted, certificates.

Roles are also sent as JSON, which means that everyone with some insight into the implementation of the system could forge a list of roles and try to execute an event using that list, or simply just copy the roles from an intercepted request. One way to overcome the latter problem is to encrypt the list of roles, and send the encrypted access rights instead. This solution requires a shared secret which is only known to the Server and the EventAPI. The use of asymmetric encryption could also be used such that the access rights would include a value which would uniquely identify the Server. This value would then be encrypted so that only EventAPIs could read it and verify that the access rights were from the Server. The latter solution can pose problems because the server should deliver a unique value for every EventAPI, as they should not share encryption keys. The lifetimes of these access rights and unique values should also be limited because the longer a secret is used the greater the risk of its misuse will be.

"Distributed Systems - Concepts and Design"¹¹ mentions a way to login where the password is never transmitted over the network. This method makes the client send a request containing a username to login at a server. The server then responds with the access rights which are encrypted with a key that can be derived from the password.

6.12 Concurrency Control

This section will describe the motivation for having concurrency control, which solutions were considered, and the solution that was ultimately implemented.

Two major solutions to concurrency control are in use in software today: pessimistic concurrency control¹² (PCC) and optimistic concurrency control¹³ (OCC).

PCC uses the concept of locking which prevents multiple transactions from accessing shared data simultaneously. OCC on the other hand uses a working copy of shared data to carry out a transaction and the changes are validated before possibly committing. If a transaction discovers a conflict between itself and a concurrent transaction, the implementation will decide which transaction aborts.

6.12.1 Motivation

Since the system is distributed, multiple users will be able to work on the same workflow at the same time. It is important that when multiple concurrent transactions happen, the workflows in the system will not enter any illegal states. Furthermore, it is important that two concurrent transactions will complete and do so in a serially equivalent manner.

Concurrency control must be implemented to cope with conflicts between operations in different transactions sent to the same receiver. These conflicts include lost updates and dirty reads, thereby possibly reaching an illegal state.

¹¹Distributed Systems - Concepts and Design p. 475

¹²Distributed Systems - Concepts and Design p. 692

¹³Distributed Systems - Concepts and Design p. 707

The team considered both OCC and PCC to work around the stated problems since both solutions would solve these.

6.12.2 Implemented Solution

The chosen concurrency control method was PCC. The implementation uses strict two-phase locking. At first a growing phase is carried out, in which the event will try to lock every other event it needs to complete the transaction. The event will then execute, update the states of the relation, and finally release all locks.

If the event fails to acquire the needed locks in the growing phase it will abort the transaction by unlocking events that were locked in the growing phase.

Strict two-phase locking ensures that the order in which the transactions are completed will be serially equivalent as well as preventing certain situations where deadlocks could otherwise occur.

If an event is locked, both reads and writes are delayed until the event is unlocked. Reading of an event is not allowed before the event is unlocked, because of the assumption that if an event is locked then it will almost certainly change its state and therefore affect the read values.

To prevent transactions from aborting, e.g. if they have to acquire a lock on an event which has already been locked, a "First In First Out" queue of lock requests has been implemented. When an event is unlocked, the next request in the queue, if any, will lock the event if it requires to update the state of the event. Reads, as mentioned, do not lock the event. This is due to the fact that read operations do not conflict each other in PCC.

If it takes more than ten seconds to acquire read or write access, the request will abort and the system will return a timeout exception to the caller.

To prevent deadlocks a global order in which the events will acquire locks have been implemented. The order is alphabetical and is based on the *eventId*. As the event itself is in the ordered list, it is locked in the global order as well. The order of unlocking is not important as long as the executing event unlocks itself last. If an event unlocked itself before others, it might be prevented from unlocking the other events.

With this approach deadlocks will never be encountered as every event has to lock in the same order. Argumentation for why this works is given in the next section.

6.12.3 Discussion of Implemented Solution

OCC is presented as a solution which allows for more concurrency than PCC because of the absence of locks. Supporters of OCC argue that this solution is superior when few conflicts arise.

The team discussed the use of OCC but came to the conclusion that it would require a substantially higher amount of work to implement than PCC. This is due to the fact that OCC needs to operate on top of a transaction abstraction in order to be able to abort and restart transactions. Furthermore OCC has to create a working copy as well as using protocols for validation and committing. Additionally a logic deciding what transaction to abort in case of conflicts needs to be implemented. Finally, using OCC, a transaction can do a considerable amount of work before it would be aborted, thus wasting resources.

The team estimated that PCC would require a smaller amount of work to implement while still providing serial equivalence and ensuring the completion of started transactions. Unfortunately PCC does create processing overhead in the form of locks, but the team argued that PCC would still be the most feasible solution, since performance was not a priority in this project.

One of the flaws in the final solution is the fact that reads are made first and then locks are acquired on the write set. This creates a window of opportunity where changes to the read

values could occur. Since the process does not check up on the read set again, we could reach a state where non-executable events execute.

Two possible solutions to this problem are, to either acquire locks on both the read and write set before executing, or to check the read set again after acquiring locks on the write set. This bug could produce an illegal state in the final system and should be fixed in a future release. The bug was found after code freeze and was not deemed sufficiently critical to be fixed before hand-in.

The argumentation for why globally ordered locking is safe from deadlocks can be explained using set theory.

Assume two events, **A** and **B**, exist. **A** and **B** have lock sets, whose elements are events. If the intersection of the lock sets of **A** and **B** is not the empty set, then the intersection is the set of events which can create deadlocks between **A** and **B**. Because events have unique and comparable IDs the intersection can be in a total order. Since the order will be the same for both **A** and **B**, then the event **C** must be the first element in the ordered set for both **A** and **B**. **A** and **B** will send lock request in the total order applied to their respective lock sets. **A** and **B** will therefore request the lock on **C** before any other event in the intersection set. If it is assumed that the lock request from **A** will arrive at **C** first, the request from **B** is put in the request queue. **A** will finish acquiring all the locks on the rest of the events in its lock set. When all locks are acquired and the changes to the elements are done. **A** will unlock all events in its lock set and allow **B** to acquire the locks in its lock set and commit its changes. No deadlocks can occur and serially equivalence has been achieved.

Even in the case that another event **D** has an intersection set with **B** and acquires locks while **B** is waiting for **B** to finish, serial equivalence is still achieved, since it is not known whether event **B** or **D** was executed first in an asynchronous system since a happens-before relation between the two is not established.

Overall it is believed that the implementation of concurrency control is correct and efficient enough since the current performance bottleneck of the system is the latency of sending HTTP requests.

If further requirements would require OCC to be implemented, some performance increase could potentially be gained, though this would require a sizable amount of work on both logic and architecture.

7 Testing

This section describes how testing the final system has been carried out and includes a discussion of what extent the system has been tested.

A variety of testing approaches have been used to test the system during development. These include unit, integration, system, and acceptance testing in varying degrees. Acceptance testing has been applied after some initial tests were developed.

A testing evaluation was performed nearing the end of the project to decide on which modules should be tested, and in which order. Some components play a larger role than others in the system and have therefore been put through more scrutinising tests.

The coverage analysis tool JetBrains DotCover assisted in this by providing coverage analysis and an overview of the test coverage.

7.1 Unit Testing

The major components handling data have been unit tested to ensure that their functionality was correct. Test projects uses the naming convention *TargetProject.tests*.

Automated unit testing has been written using the NUnit testing framework¹⁴. Only code writ-

¹⁴<http://www.nunit.org/>

ten by the team has been tested. Frameworks and libraries have been assumed tested.

Dependency injection and mocking, using the Moq library¹⁵, has been used extensively in many unit tests to ensure that the implementations of interfaces were tested in an isolated and predictable environment.

More specifically mocking a storage module and saving objects in a list for validation was often used. By validating the objects coming in and out of methods it is possible to assert with greater certainty that the implementation being tested is working as expected. Components have been unit tested in their order of importance.

Assertions of exceptions being thrown in boundary cases were also performed in methods where these were expected to be thrown.

Black and white box testing have both been used. Most unit tests were developed by testing the expected functionality of a method, not the implementation of the method.

In certain cases white box testing with branch coverage has been done. Throwing of special exception types is a great example of this.

7.2 Integration Testing

Integration testing was performed by hand during implementation. Every time components were developed they were tested informally against the other components. Unfortunately no documented integration tests exist.

7.3 System Testing

System testing was, like integration testing, performed during implementation. Unit testing was of higher importance to the team and validating the functionality manually by performing tasks supporting the requirements were acceptable for the team.

7.4 Acceptance Testing

Acceptance testing would preferably be done by the receiver and the user of the system using test cases specified by the client in cooperation with the developers. When reaching the halfway point of the development of the system, the software was sent to the product owner of the system, our external partner. Unfortunately he did not have the time to review the software, and proper acceptance testing by the product owner has not been completed.

The team has, using a person acting as the customer, tried to complete acceptance testing to ensure that all requirements have been met.

7.5 Discussion of Testing Approach

The JetBrains DotCover tool has been used to check the amount of code covered by tests. DotCover does not guarantee that sufficient testing has been performed, it can only tell the developer using it which statements have and have not been tested and evaluate which methods are the most risky and should have a higher priority in testing. This helped the team identify the most critical components to test in the system, and also helped to give an overview of remaining tests. After finishing the system, DotCover gave an assessment of test coverage. The result was 79% test coverage with 947 test cases.

The testing approach has mainly been centered around unit testing with focus on testing internal logic. Integration testing has also been used to test the communication between components, but not as in-depth as stated above.

¹⁵<https://www.nuget.org/packages/Moq/>

Since the system was developed with less focus devoted to development of a powerful front-end client, testing the back-end logic was deemed to be more important.

The weakest point in the testing suite is arguably the lack of integration testing. Since mocking is used in almost all the tests, we do not have enough assurance that the components interact properly. Therefore either top-down or bottom-up automated integration tests should be the top priority of an extension of the test suite. This would create much more certainty that the system functions as it should as well as allow for more security when changing module implementation.

8 Reflection on Project

This section intends on summarizing how the group has tackled and organized the workload. Reflections on the process will be presented with focus on what went well and what did not.

8.1 Reflections on the System

The team is mostly satisfied and proud of the system, even though the team is aware of the fact that some issues remain. By using well known and tested frameworks, best practices in object oriented programming, and following the architectural style of REST, the team feels that the system is robust with an architecture that is easy to extend.

8.1.1 What the Team Dislikes

The team have not prioritised a comprehensive solution to RBAC as it was not deemed as the feature with the most relevance to the system. However the team is not completely satisfied with the solution and believe it could be much more intelligent and secure.

The GUI of the Client is designed for the sake of basic usability. There was only a requirement for a UI which did not necessarily need to be graphical. The team does not think the Client is the most user friendly GUI nor the most well designed. Had the team focused more on the Client, it could have been possible for instance to implement a feature which would allow the user to choose between showing and hiding events that are not executable at a given time.

Even though testing was a priority for the team, it was not carried out thoroughly enough. Unfortunately a bug was found after code freeze which could produce faulty states, as described in the section Concurrency Control , and the team feels that this should be fixed in a future release.

8.1.2 What the Team Likes

Initial discussions about the system architecture were great for specifying an architecture that everyone in the team understood and agreed on. Many of the key decisions were taken before starting implementing the system, and these discussions really helped reach a satisfying implementation.

The team feels that the interface-based programming was used in a satisfying manner, with only a small amount of coupling between classes. Having a layered architecture throughout the system was a goal from the beginning, and implementing these layers as initially discussed succeeded. By doing so, the principle of having small controllers with well defined purposes made it possible to have low coupling between classes. Designating responsibilities to layers and following these when handling exceptions was also done in a pleasing manner according to the team.

Transferring data in an independent manner was beneficial in that it allowed the team to change and add DTOs during implementation. Adding additional functionality during the project was

made a lot easier due to this. The team felt that the RESTfulness of the system was developed in accordance with their expectations. Omitting some specified and required functionality could have made the system even more RESTful, but overall the team is satisfied with implementation. The use of dependency injection was widespread, which helped the team in testing components.

Even though the team focused a lot on testing, it was not done thoroughly enough as previously mentioned. Most methods in the major classes of the system have been tested with acceptable test coverage. Mocking was used to a great extent in testing, which made testing faster and easier for the team.

8.2 Thoughts on Group Processes and Tools

The team strived to run the project as a SCRUM project. This meant that the team held sprint planning meetings, did task division, used a SCRUM board for each sprint, had daily standup meetings, and made sprint retrospectives.

The team has primarily worked like this. The benefits of using SCRUM were that it was clear to see what tasks were currently due, which tasks were in progress, and which members were responsible of finishing what.

In the last part of the project the team realized that it would be beneficial to have a common objective to pursue during a sprint. Not just in terms of individual completed tasks, but also in terms of what should be presentable to a fictional product owner.

The lesson was that the team overlooked the extensive task of joining finished work and making sure that the individual modules cooperated as intended.

If the team had a product owner - with whom weekly meetings could have been held - the team might have been forced to focus more on deliverables instead of simply completing individual modules.

GitHub was used for version control, and branches were used to implement new features. The branch would then be merged into the master branch when it was deemed finished.

This enabled easier reversion and made merging bigger changes easier. Isolating a feature on a single branch also helped the members get an overview of the state of the system during implementation.

9 Conclusion

The finished system meets the stated requirements. The system is dynamic and can be used for almost any DCR graph exported from `DCRGraphs.net` unless the graph contains nested events which was not in the scope of the project.

The system supports creation, reconfiguration, and deletion of workflows, however reconfiguration must be done by deleting and creating events. The system is based on the principles of REST, and provides a UI for using these services. The system provides a log of operations that have been executed or aborted on these REST services and what changes have occurred in workflows and events.

The system fulfills the stated requirement for peer-to-peer distribution among events. Persistence of data is achieved by saving data in a distributed manner on database servers.

The role-based access control allows the system to verify users, allow different users different access rights, and different users the same rights, which are the features required by the requirement elicitation. The concurrency control is pessimistic and allows for multiple events to concurrently execute. The system is built to not creating deadlocks and therefore execution should never abort, if crashes or internal errors do not occur.

The system has been tested using by both automated unit-tests as well as varying amounts of manual integration, system, and acceptance testing.

Overall the team is satisfied with the project.

A List of Work Distribution

The following list shows the estimated contribution of each member in a given part of the project. The ordering of names specifies which members have had the biggest responsibility of the component.

- Architecture
Split equally between all team members.
- Client
 - Views
AWIS, MLIN, AFIN
 - View Models
AWIS, MLIN, ADAE
 - Connections
AWIS, MLIN, ADAE
- Client Tests
 - View Models Tests
MLIN, CSJE
 - Connections Tests
MLIN
- Common
 - DTO
AFIN, rest of team
 - HTTPClient toolbox
MLIN, AWIS
- Common Tests
 - DTO
AFIN, rest of team
 - HTTPClient toolbox
MLIN
- Server
 - Storage
AWIS, AFIN, ADAE, CSJE
 - Logic
MOALB, AFIN, ADAE, MLIN
 - Access Control
MLIN, AFIN, AWIS
 - Exception Handling
MOALB, AFIN
 - History
AFIN, AWIS
- Server Tests
 - Storage Tests
MOALB, AFIN

- Logic Tests
AWIS, AFIN, MOALB
 - History Tests
AFIN
- Event
 - Concurrency
ADAE, CSJE, AWIS
 - Access Control
AFIN, MLIN
 - Storage
AFIN, ADAE, CSJE, MLIN, AWIS
 - Logic
MLIN, MOALB, AFIN
 - Communicators
MLIN, MOALB
 - Exception Handling
MOALB
 - History
AFIN, AWIS
- Event Tests
 - Concurrency Tests
AWIS
 - Access Control Tests
MLIN
 - Storage Tests
MLIN, MOALB, AFIN
 - Logic Tests
MLIN, MOALB, AWIS, ADAE
 - Communicators Tests
AWIS, AFIN
 - History Tests
AFIN
- DCR graph
ADAE, CSJE
- Workflows
 - Initial draft of the Brazilian Healthcare Workflow
CSJE, ADAE, MOALB
 - Revised version of the Brazilian Healthcare Workflow
CSJE, MOALB
- SCRUM Process
CSJE, MOALB
- Report
Equally divided between all team members.

The team believes that overall the contribution has been equally divided between the members.

B Healthcare Process from Brazil

Project: Build the AS-IS model of the medical care of a hospital located in Curitiba, Brazil. This private hospital provides services (appointments, exams, surgeries) to the population. Brazilian government pays every provided service by the hospital.

General description:

1) Patient goes first to a medical care unit called UBS* (Healthcare Basic Unit). Each region in Curitiba has at least an UBS. The objective of an UBS is to provide the first care to the population. In UBS, a non-specialist physician initially treats the patient. The physician may recommend the patient an appointment with a specialist or request some exams. In both cases, UBS is responsible to schedule appointments and exams. Thus, at the end of the first appointment, UBS provides where and when the patient must go (to an appointment with a specialist or to do exams). Our interest here relies on appointments and exams scheduled in a specific hospital;

* UBS is a unit managed by the government of Parana State (Curitiba is the capital of Parana). So, its process is out of the scope of the present project.

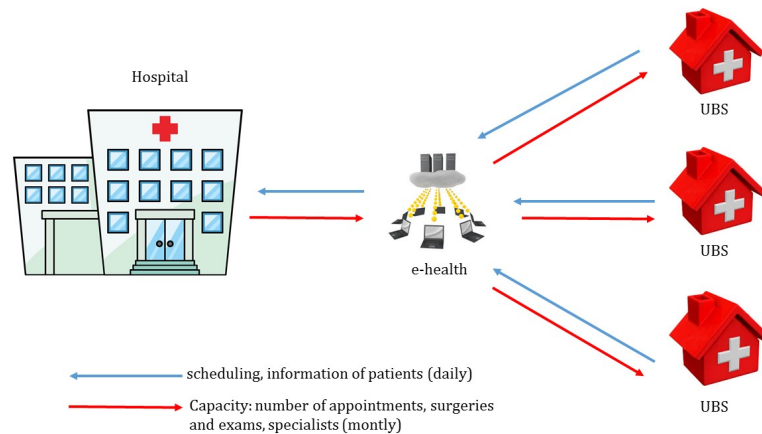
2) The hospital inform monthly to all UBS the amount of appointments and exams available. Thus, each UBS schedules an appointment or exams according to such availability;

3) There is an information system (called e-health) that manages the scheduling of patients. The e-health supports the healthcare public system of Paraná State. Private hospitals cannot modify or make any changes in e-health. The only thing that they can do (using e-health) is to inform the availability of appointments and exams (as described in 2);

4) In a daily basis, the hospital received (from e-health) the scheduling of patients. Thus, the first task that employers have to do is to download the file that contains a list of patients for that day (appointments and exams). This list includes private information of each patient, the time of appointment or exams, and the specialist for that appointment;

5) When the patient arrives in some of these clinics, the first reception has to check his/her scheduled time (according to information from e-health) and register some information about him/her. Then the patient is referred to a second reception where he/she has to provide an attendance record issued by the first reception. In such record is informed which specialist the patient should go or which exams are requested. The second

reception is responsible to organize the service queue of both appointments and exams;



6) In the case of an appointment, the specialist calls each patient according to the queue organized by the second reception. The medical examination is executed and the specialist creates a medical report. The specialist may request some exams, recommend an appointment with another specialist, or both, or may recommend some medications and ask the patient to return, or may recommend some surgery;

7) In the case of exams, the patient is first prepared. Then the exam is performed. A medical examination report must be created;

8) Depending of the specialist recommendation, after appointment or exams the patient has to go to the first reception in order to schedule new appointment or exams or surgery or return. The first reception has to provide information about date, time, addresses of appointments and exams. This step is called 'checkout';

9) It is possible that a specialist recommend for a specific patient a maximal priority over other patients concerning exams, appointments (with other specialist) or surgery. In this case, the patient is referred to other sector called ROTA. In this sector an auditor (a specialist working for Brazilian Government) will confirm the urgency of each case;

10)The hospital will charge the Brazilian government the services provide to all patients. Thus, is necessary that the all reports store the whole set of relevant information.

What the operations managers of the hospital want:

- i) To build a pervasive healthcare process model for the hospital;
- ii) To represent the exception alternatives in the healthcare process (cancellations and re-scheduling of appointments and exams);
- iii) To reduce the total time of patients in clinic. A certain level of automation has been suggested to reach this objective. For example, the managers want to give permissions to the specialists to schedule another appointment or exams. Currently this activity is under responsibility of first reception;
- iv) The current process does not consider the emergency cases. It is necessary to create a process to treat this situation;
- v) To restrain the permissions of the specialists. The aim is each specialist can only schedule exams related to your specialty;
- vi) To propose performance indicators to monitor the quality of healthcare process.

C Usermanual for DCR Graph Parser

The parser has only one window in which a great deal of information can be entered.

In the first box, called INPUT FILE, the user can write the path of a **DCRGraphs.net** XML-file. The user can also browse for such a file using the Choose XML-file button just below it. This will open a default Windows Open Dialog. Below the button is a box labeled Workflow Id. This is used by FlowIT to uniquely define a workflow.

The Workflow Id should not contain spaces or symbols, but numbers, uppercase, and lowercase letters are OK.

The box labeled Workflow Name will be the name of the workflow, which the client presents. Anything can be written in this box, but if the string is too long, the Client will not be able to show everything without scrolling horizontally in the Client user interface. In the Server Base-URL box needs to know the base URL of the server.

This will be `http://flowit.azurewebsites.net/` when publishing workflows onto the hosted Azure FlowIT server. It will be `http://localhost:13768/` when posting to the local IIS instance when running FlowIT locally. Remember to change the Client Settings file, when running locally, see Section 5.2.4 The Settings File).

The Event Base-URLs will be the base address(es) of the Event-Machines meant to host the workflow. For localhost usage it will be `http://localhost:13752/` (the two localhost-ports can be changed through Visual Studio or IIS-settings if desired, these are the ports the projects are configured for).

For the hosted Azure Event Machines the addresses will be one or more of `http://flowites1.azurewebsites.net/`, `http://flowites2.azurewebsites.net/`, `http://flowites3.azurewebsites.net/`, or `http://flowites4.azurewebsites.net/`. If multiple Event Machines are desired put a comma between the base-addresses. The input is not checked before usage, so be careful what is typed into these fields.

The last textbox is the password chosen for all of the default users, which are the actors or roles from the **DCRGraphs.net** XML-file.

Before we get to the buttons there are two checkboxes. These are almost self-explanatory. The first one will try to create the workflow on the chosen server. This will fail if the workflow exists already.

The second one will create the users with the typed in password, if the users does not already exist. In the case where the users are existing, the parser will not change the passwords of the existing users, it will just add the roles to the users.

The leftmost button in the bottom of the window, will parse the selected XML-file and turn it into JSON-objects which are then saved as text in `graph.json` which will be written to disk in the same folder as the parser-executable.

The other button will upload the parsed data directly into the Flow IT system, based on the parameters given.

D Possible Solutions for Creating the Database

In some installations it is not possible to automatically create the databases used by the Server and EventAPI. The team has identified two causes of these problems:

- The database instance is not created on the computer
- The database instance has a database with the same name.

The first problem can be fixed by creating a Microsoft LocalDb instance called `MSSQLLocalDb` through a command line tool installed with Visual Studio. From the Windows Command Prompt

it is possible to view a list of all database instances by executing the command:

```
sqllocaldb info
```

If the list contains an entry called “MSSQLLocalDb”, this is not the cause of the problem. If “MSSQLLocalDb” is not in the list, it can be created by executing the command:

```
sqllocaldb create MSSQLLocalDb
```

Then try to restart the Server and EventAPI to see if the databases are created.

The second cause should not occur unless the Server or EventAPI has been run before, and the database files has been deleted. Typically the problem is that Microsoft SQL LocalDb keeps track of all of its databases and therefore it has to be removed, in order to allow the same database to be created again. The following command will tell Microsoft SQL LocalDb to stop the MSSQLLocalDb database instance:

```
sqllocaldb stop MSSQLLocalDb
```

When the instance is stopped, it can be deleted by the following command:

```
sqllocaldb delete MSSQLLocalDb
```

When this command has been executed, remember to check that the Server and EventAPI database files does not exist. These files can be found in the following directories:

```
{Solution Directory}\Server\App_Data
```

and

```
{Solution Directory}\Event\App_Data
```

There will be two files in each of these directories, which can safely be deleted, when the database is no longer needed.

If both the database files and the database instance is deleted, it should be possible to run the Server and EventAPI without issues. If EntityFramework cannot attach the database file, try the solution to the first cause.