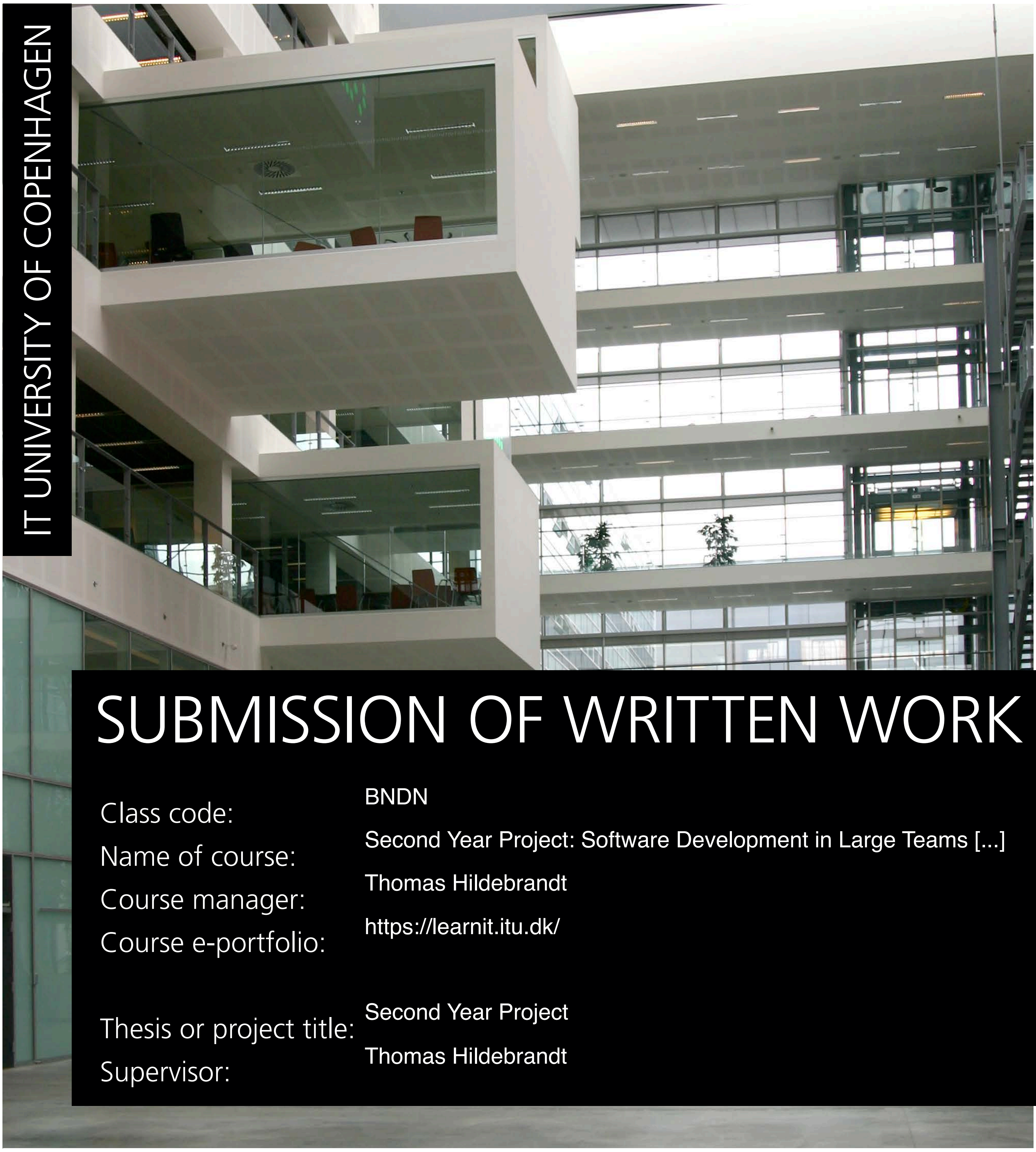
IT UNIVERSITY OF COPENHAGEN



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 Wind Steffensen (awia)
Anders Fischer-Nielsen (afin)
Cecilie Strunge Jensen (csje)
Mikael Lindemann Jepsen (mlin)

Morten Albertsen (moalb)

Contents

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. 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.

Figure 1: reflection

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
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.

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.

Figure 2: Link to table

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 notation[1]. 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.

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.
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 3. 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 3, 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.

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

Figure 3: Sample Pay Yearly Gas Bill

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

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 XYZ:

3.2.2 Relations

3.3 REST

3.3.1 HTTP Methods for CRUD Operations

3.3.2 Be Stateless

3.3.3 Resource Directory Structure

3.3.4 Transfer XML or JSON

4 Workflow elaboration

4.1 Initial Workflow Design

4.2 Feedback from External Partner

4.3 Final Workflow

4.4 Workflow Given by Lecturers

5 Usermanual

5.1 Compiling the Project

5.2 Running the Client

5.2.1 The Main Window

5.2.1.1 State of Events

5.2.1.2 The Status Bar

5.2.1.3 Reset Workflow

5.2.2 History

5.2.3 The Settings File

5.3 The DCR Graph Parser

5.4 Running Locally

6 System Implementation

6.1 Overview

6.2 Overall Architecture

6.2.1 Client

6.2.2 Server

6.2.3 EventAPI

d

6.3 Interface-based Programming

6.4 Dependency Injection

6.5 Design of Client

6.5.1 MVVM-principles in Client

6.6 Design of Server and EventAPI

6.6.1 Multi-layered Design

6.6.2 Execution of an Event

6.7 Distribution

6.7.1 Location Transparency

6.7.2 Global Identification of Events

6.7.3 System Deployment

6.8 RESTful APIs

6.8.1 Restfulness of the System

6.8.2 UnRESTful Parts of the System

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 4.

Figure 4: A code snippet from `LockController` in EventAPI. `Lock` checks for input validity, and if it is valid, it delegates the work to the `LockLogic` layer. The try-catch design will be discussed in the following section.

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 5.

Figure 5: 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.

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.

Figure 6: Code-snippet from the method `InitializeNewEvent` in `EventStorage`

In the code-snippet seen in Figure 6, 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 7, 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.

Figure 7: Code-snippet from the method `CreateEvent` in `LifecycleController`

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 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 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 8. 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.

The data model of the Server can be seen in Figure 9. The data model contains seven entities. 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.

Figure 8: The Data Model of an EventAPI

Figure 9: The Data Model of the Server

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 attacks [2] 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 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

[2] `http://en.wikipedia.org/wiki/Dictionary_attack`

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"[3] 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 control[4] (PCC) and optimistic concurrency control[5] (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.

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.

[3]Distrubuted Systems - Concepts and Design p. 475
[4]Distributed Systems - Concepts and Design p. 692
[5]Distributed Systems - Concepts and Design p. 707

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 framework[6]. Only code written by the team has been tested. Frameworks and libraries have been assumed tested.

Dependency injection and mocking, using the Moq library[7], 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.

[6]`http://www.nunit.org/`
[7]`https://www.nuget.org/packages/Moq/`

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.
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 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.**

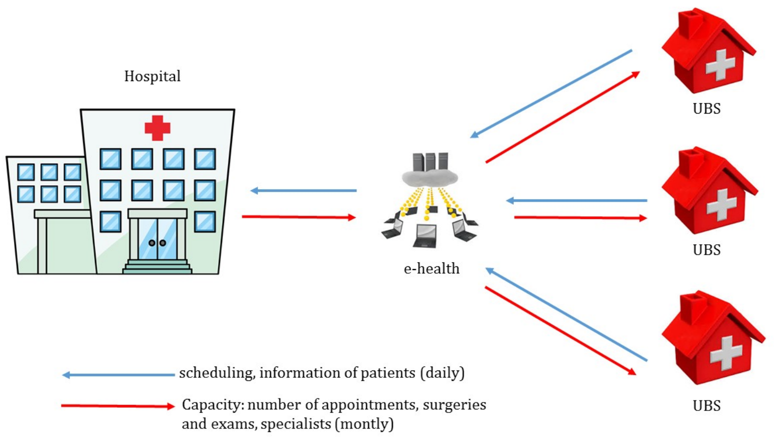
2) The hospital inform monthly to all UBS the amount of appointments and exams available. Thus, each UBS schedules an appointment or exams according to such availability;

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;


Hospital
e-health
UBS
UBS
UBS
scheduling, information of patients (daily)
Capacity: number of appointments, surgeries and exams, specialists (montly)

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.