API Prototype for medical product documentation Systems

Approved: _____ Date: 10/27/2015

Prof. Dr. Christoph Wentzel, Committee Chair

Approved: _____ Date: 10/27/2015

Prof. Dr. Ronald Moore, Committee Member

Approved: Yan Shi Date: 10/28/2015

Dr. Yan Shi, Committee Member

Suggested content descriptor keywords:

_____

_____

_____

API Prototype for medical product documentation Systems

A Thesis

Presented to

The Graduate Faculty

University of Wisconsin-Platteville

In Partial Fulfillment

Of the Requirement for the Degree

Master of Science

Computer Science

By

Emmanuel Guenther

2015

Declaration of authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged.

Dreieich, October 27, 2015

Signature

Declaration of confidentiality

This thesis may not be published or accessible for third parties neither complete nor in part, without the written permission of the bayoonet AG. The submitted Master Thesis are held by the main speakers under wraps. I am aware that confidentiality does not affect the creation of a master poster as well as performing the defense. The confidentiality obligation shall automatically expire after five years.

Dreieich, October 27, 2015

Signature

Abstract

This thesis deals with the topic of web API software in the ASP.NET environment. The task is to design a general API, which fulfills the necessary requirements of being used in the environment of medical devices. The API is implemented on a prototypical basis to operate with Qware Risk Manager software, which is used to conduct risk analysis of medical devices and developed by bayoonet AG.

The main idea of the API is to enable the Risk Manager software for being connected with third party software and software systems to import and export conducted data. Additionally the design of a generalized API enables the possibility to use the API not only with the Risk Manager software but also with other software. This thesis analyzes the general steps of placing a medical device onto the market in Europe and the approval process for a medical device for the United States market. This includes the data privacy regulations of both processes for these two countries.

In order to choose the best design for the API this thesis analyzes the current state of the art API techniques in ASP.NET. This current API analysis is extended through the investigation of the generalization types, which can be used with .NET technology. The conducted general goals which have to be achieved by the API include the aspect of fulfilling the mandatory requirements of the FDA CFR 21 Part 11 to be able to sign a digital document.

The outcome of this thesis is a generalized prototypical API implementation, which can process the operations of importing an analysis tree and create a new project, including a new version of the project, into the Qware Risk Manager software and is able to work with other software in the same way. This API is also analyzed for its economic value for business purposes.

Abstract

Diese Arbeit beschäftigt sich mit dem Thema Web API Software in der ASP.NET Umgebung. Die Aufgabe besteht darin, eine allgemeine API zu entwerfen, welche die notwendigen Anforderungen im Umfeld von medizinischen Geräten erfüllt. Die API basiert auf einer prototypischen Umsetzung, um mit Qware Risk Manager Software, mit welcher die Risikoanalyse von Medizinprodukten durchgeführt wird und durch die bayoonet AG entwickelt wurde, zu arbeiten.

Die Hauptidee der API ist es, die Risiko Management Software mit Software von Drittanbietern und anderen Softwaresysteme zu verbinden, um den Import und Export von Daten zu ermöglichen. Zusätzlich soll durch ein allgemeines Design der API ermöglicht werden, dass es die Möglichkeit gibt, die API nicht nur mit der Risk Manager Software, sondern auch mit einer anderen Software zu benutzen.

Diese Arbeit untersucht die allgemeinen Schritte, ein Medizinprodukt auf den Markt in Europa zu platzieren und den Genehmigungsprozess für ein medizinisches Gerät für den Markt der Vereinigten Staaten. Dazu gehören die Datenschutzbestimmungen für diese beiden Länder in diesen Prozessen.

Um das beste Design für die API zu wählen, analysiert diese Arbeit den aktuellen Stand der Technik API Techniken in ASP.NET. Diese API Analyse wird durch die Untersuchung der Generalisierungstypen, die mit der .NET Technologie verwendet werden kann, erweitert. Zu den allgemeinen Zielen, welche von der API erreicht werden sollen, gehört auch der Aspekt der Erfüllung der verbindlichen Anforderungen des FDA CFR 21 Part 11, welches ermöglicht, ein Dokument digital zu unterzeichnen.

Das Ergebnis dieser Arbeit ist eine allgemeine prototypische Implementierung der API, welche Prozesse zum Importieren eines Analysebaumes und zum Anlegen neuer Projekte, inklusive Versionen eines Projekts, in der Qware Risk Manager Software verarbeiten kann und in der Lage ist, auch mit anderer Software auf die gleiche Weise zu arbeiten. Die API wird auch auf

ihren wirtschaftlichen Wert für Geschäftszwecke analysiert.

Contents

Abbreviations

FD&C Act	Federal Food, Drug, and Cosmetic Act
FDA	Food And Drug Association
HTTP	Hypertext Transfer Protocol
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
SOAP	Simple Object Access Protocol
WCF	Windows Communication Foundation
XML	Extensible Markup Language

Figures

Tables

1 Introduction

The European Union and the United States of America have standards which define the general requirements for a medical device to be placed on the market. In both countries, the highest and most important requirement for a medical device is to ensure that a device cannot harm users, patients, or other people. This safety requirement can be fulfilled by the manufacturer by conducting a risk assessment according to the EN ISO 14971 standard. Without the risk assessment and the fulfillment of the risk assessment requirements, a medical device manufacturer in the US or Europe is not able to get the approval to bring their device on the market.
In the last twenty years the development of software products for enhancing business processes has been established. This was the reason due to a higher usage of computer systems in business. The recent past has shown that computer systems can be used to improve business processes using a specialized software. In the medical device area the development of specialized software started roughly in the year 2000. This specialized software helps to improve the general quality of medical products and devices. For other medical devices specialized software enables the possibility to be able to create them.
A specialized software for risk assessment is the Qware Risk Manager by the bayoonet AG. The software is able to conduct a risk assessment according to EN ISO 14971 standard including usability engineering and referenced standards.
The Federal Food and Drug Administration (FDA) of the United States of America provides a regulation for delivering digital records with digital signatures. The requirements for this regulation are defined in the CFR 21 Part 11. The Qware Risk Manager fulfills these requirements and it can digitally sign documents of the risk assessment.
This thesis is facing the design and prototypical implementation of an API for the Qware Risk Manager to provide access for third party software products. For this API it is important to accomplish all the security functions the software is already offering. These security functions include that the data which shall be stored runs through the authorization context. This means

that the authorization features and settings cannot be disabled or infiltrated. A user shall not be able to store data without checking the user's permissions to prevent corrupted data saved into the database or that the database gets corrupted through uncontrolled database saving actions. The security functions also include the CFR 21 Part 11 requirements which contain a control audit. This is necessary to be able to trace changes to the data and who of the users did the changes.

The motivation for this API is that the current market does not offer an API which can be used with a software product for medical devices and fulfills the requirements of the CFR 21 Part 11. This leads to a generalized design to enable the API to be also used with other software products.

The resulting approach of implementing the prototype of this thesis shall not be only usable with the Qware Risk Manager. It shall be a new approach of designing a general API which can work with software designed for the medical device process and can be used with several other software programs. This opens up a wide range of applications for this API.

The development of such an API does not only opens up a wide range of applications for the API but the API also opens up several new usage capabilities for the Qware Risk Manager. One of these capabilities can be a mobile or tablet application of the software which can access the data saved in the database using the API. Another capability is that the software can be connected to several other software systems, for example an SAP system, to import and export data. Therefor it is necessary to ensure that the API is not vulnerable from the outside.

There are actually many reports which are uncovering vulnerable medical devices. Today's medical devices are using more and more embedded software which are communicating. This leads to that the security risk of medical devices is increasing because they are using new technologies like wireless transmissions or the internet. Data security and integrity is very important for data records of medical devices.

For the consideration of all of these motivational aspects, the design and implementation of such an API will be very useful and inventive.

2 Current Concept of Approval for Medical Devices

This chapter will analyze the concept to receive an approval of a medical device for the US market and the concept of placing a medical device on the market in the European Union. The analysis includes all the necessary regulations but concentrates on the general concept.

2.1 European Union

Since the European Union opened the borders for free movement of goods between their member states, it was necessary to create technical harmonized standards. These technical harmonized standards are a product of the new approach to technical harmonization and standards from 1985. This new approach includes the development of standards wherefore the European Union can issue a standardization assignment to private standardization bodies. These bodies develop a draft version of a standardization. If the European Union agrees with the draft version, it will be published in the Official Journal of the European Union and the developed standard covers the general requirements.

A directive of the European Union has to be transposed by every member state into their national laws. European directives cover the general important and mandatory requirements for a wide range of industrial products. These requirements are to be met in order to receive the CE mark and to place a product on the European market. On the other hand, the European Union also has regulations for which it is not necessary to transpose them into national law. These regulations are self-executing.

Harmonized standards are European standards for products. They are worked out by the organizations CEN, CENELEC and ETSI on behalf of the European Commission and EFTA, which means it exists a standardization mandate (mandate) at these European standardization

organizations. If a manufacturer applies a harmonized standard, it is automatically assumed that all the essential and mandatory requirements are fulfilled.
The European directives contain three directives for medical devices. These three directives are:

- the Active Implantable Medical Device (AIMD) Directive - 90/385/EEC
- the Medical Device Directive (MDD) - 93/42/EEC
- the In Vitro Diagnostic Device Directive (IVD) - 98/79/EC

We focus on the Directive 90/42/EEC which defines the general requirements for medical devices. The Directive 90/385/EEC and 98/79/EC exclusively to active implantable medical devices and devices for in vitro diagnostic.
The general steps in the certification process are shown in Figure 2.1.

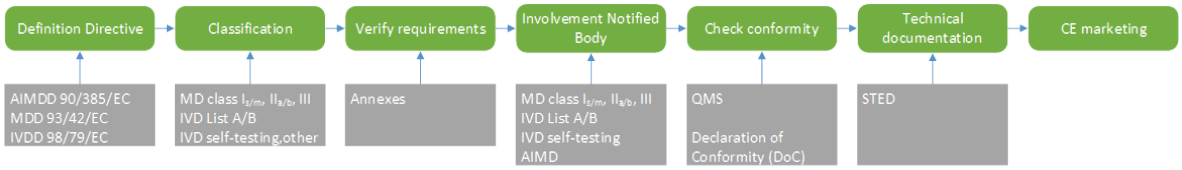

Figure 2.1: General Process for Medical Products in the EU [tto]

These steps are the same for all medical devices which are covered by the three medical device directives. There are some exceptions for classes of medical devices, which do not need the involvement of a notified body. These are mentioned in the associated directive description.
A notified body is a third party company or institution which offers their assessment services. These assessment services are product and quality system management related conformity assessment procedures. They require being third parties and being independent of their clients and other interested parties.
Additionally they are under surveillance by their national notifying authorities, which have the possibility to withdraw and modify the notification if the conditions are no longer fulfilled. Notified bodies are advised to any manufacturer which have their office inside the European Union or in third countries. They may carry out their activities in other countries with their

own personnel or with subcontractors. If the notified body carries out their activities to subcontractors, the notified body needs to ensure that the subcontractor meets the provisions of the Medical Device Directive.
In particular, a notified body has to fulfill the following [DIRb]:

- "sound vocational training covering all the assessment and verification operations for which the body has been designated,
- satisfactory knowledge of the rules on the inspections which they carry out and adequate experience of such inspections,
- the ability required to draw up the certificates, records and reports to demonstrate that the inspections have been carried out."

The European Union accredits notified bodies.

2.1.1 Medical Device Directive 93/42/EC

The Medical Device Directive 93/42/EC applies to medical devices and accessories. Accessories are treated as medical devices within the directive. A medical device manufacturer defines the medical purpose of the device. This medical purpose has to be covered by the definition of a medical device of the MDD.
The definition of a medical device is "any instrument, apparatus, appliance, material or other article, which is intended to be used for human beings for the purpose of:

- diagnosis, prevention, monitoring, treatment or alleviation of disease,
- diagnosis, monitoring, treatment, alleviation of or compensation for an injury or handicap,
- investigation, replacement or modification of the anatomy or of a physiological process,
- control of conception," [DIRb]

and which is not proposed to be used in or on the human body by a pharmacological, immunological or metabolic means. This definition also includes software, which is needed for an accurate usage and explicitly for diagnostic or therapeutic purposes.

An accessory is an article which is not considered as a device but intended to be used together with a device. It needs the device to be enabled and to be able to use it according to the intended usage by the manufacturer.

The term manufacturer is also defined by the directive. A manufacturer is the natural or legal person who produces a device. This includes the responsibility of the design, packaging and labeling of a device before placing it on the market.

All devices specified by the given definition have to fulfill the essential requirements for safety, performance and labeling. The MDD excludes medical devices like in vitro diagnostic devices, active implantable devices, cosmetic products, human blood or products out of parts of human blood, transplants or cells out of human origin or animal origin. The necessary safety requirements are not only applicable to patients but also include users and other persons. These safety requirements include protection against radiation, devices connected or equipped with an energy source, construction and environmental properties, infection and microbial contamination, and chemical, physical and biological properties. For the labeling it is required to use the national language of a Member State.

The manufacturer has to demonstrate the fulfillment of the essential requirements for all his devices. This also includes revised devices.

The medical devices are classified into one of the four classes I, IIa, IIb, and III. The classification determines which conformity assessment procedure has to be followed. For example, a class I medical device would be a reusable surgical instrument which is not connected to an active device. An active medical device is defined as a device which is powered by a source of electrical energy, whereas the energy is not generated by the human body. A class III device would be all implantable or long-term invasive devices. The classification indicates the level of risk of the medical device.

Irrespective of the class the manufacturer is liable to maintain a technical file for the device or the device family. This liability is limited to the lifetime of the medical device. Additionally he is responsible to issue and keep the declarations of conformity file for the CE marked devices.

Custom-made devices, which are devices specifically made according to a written prescription by a qualified medical practitioner, need to produce a statement concerning devices for special purposes. The written prescription defines specific design characteristics and the intended use for a particular patient.

The general rule for requirements concerning characteristics and performances of the usage of the device and the evaluation of undesirable side-effects is, that they have to be based on clinical data. The evaluation of this data has to follow a defined procedure.
The directive defines that clinical data has to be based on:

- clinical investigation(s) of the device concerned; or
- clinical investigation(s) or other studies reported in the scientific literature, of a similar device for which equivalence to the device in question can be demonstrated; or
- published and/or unpublished reports on other clinical experience of either the device in question or a similar device for which equivalence to the device in question can be demonstrated.

The rules for clinical investigations are defined in Article 15 and Annex X of the MDD. It is very important that the obtained data has to remain confidential. The requirement of confidentiality is described in Article 20 of the MDD.
All medical devices which are covered by the MDD have to carry a CE mark to be able to be placed on the market. To place a device on the market means making a device available to be purchased by a customer, but this not include devices which are intended for clinical investigations. Devices intended for clinical investigations or are custom made not require the CE mark. In this case the manufacturer is responsible to keep the documentation according to the MDD.

Quality Systems

The MDD requires that a medical device manufacturer keeps a product-related, suitable and effective quality assurance system. The quality assurance system has the purpose to ensure that the products quality follows the requirements of the MDD. All elements, requirements and provisions adopted for the quality assurance system have to be documented in a regular and methodical manner. They must be in the form of written policies and procedures. These policies and procedures are, for example, quality programs, quality plans, quality manuals and quality records.

The requirements for the quality assurance system defined in the MDD can be achieved with the application of the harmonized standard EN ISO 13485. Some of the requirements of the MDD for a quality assurance system include:

- the technical documentation
- reference to the essential requirements according to Annex I of the MDD
- information about harmonized standards and medical device regulations
- risk analysis
- labeling and instructions for use
- different languages
- post-marketing surveillance
- reporting under the vigilance system
- retention of certain documents

Technical Files/Design Dossier

The MDD defines that all devices for clinical investigation, custom-made, class I, IIa, IIb or III, require a technical documentation. Whereas for class I, IIa, and IIb the term technical files and for class III the term design dossier is used. This documentation includes a device documentation, the technical documentation, and a design dossier.
The required essential content of a technical file is as follows [DIRb]:

- "a general description of the product, including any variants planned,
- design drawings, methods of manufacture envisaged and diagrams of components, sub-assemblies, circuits, etc.,
- the descriptions and explanations necessary to understand the above mentioned drawings and diagrams and the operations of the product,

- the results of the risk analysis and a list of the standards referred to in Article 5, applied in full or in part, and descriptions of the solutions adopted to meet the essential requirements of the Directive if the standards referred to in Article 5 have not been applied in full,
- in the case of products placed on the market in a sterile condition, description of the methods used,
- the results of the design calculations and of the inspections carried out, etc.; if the device is to be connected to other device(s) in order to operate as intended, proof must be provided that it conforms to the essential requirements when connected to any such device(s) having the characteristics specified by the manufacturer,
- the test reports and, where appropriate, clinical data in accordance with Annex X,
- the label and instructions for use."

A raw material manufacturer or subcontractor can submit master files to the notified body. These master files can be referenced in a technical file of the device manufacturers.
The MDD strictly requires a formal risk analysis for each device or device family. This increases the responsibility of medical device manufacturers beyond previous regulations. The preferred standard for this risk analysis is the harmonized standard EN ISO 14971 which defines risk management of medical devices.

European Medical Device Vigilance System

The MDD requires Member States of the European Union to embrace needed steps for recording and evaluating incident information which are brought to their knowledge. The device manufacturer has to report [DIRb]

- "any malfunction or deterioration in the characteristics and/or performance of a device, as well as any inadequacy in the labeling or the instructions for use which might lead or have led to the death of a patient or user or to a serious deterioration in his state of health,
- any technical or medical reason in relation to the characteristics or performance of a device for the reasons referred to above, leading to a systematic recall of devices of the same type by the manufacturer".

A manufacturer has to collect all the feedback about the device and to evaluate it. The feedback reports are classified after a special process. If the medical device security adviser and its council come to the result that a corrective action is necessary and the corrective action is an adverse event, the manufacturer has to report the incident.
The European Medical Device Vigilance System is a European reporting system for incidents. All incidents happened in Member States of the European Union have to be recorded in the system. The European Union supports a platform to submit the incident reports and makes them available to all other member states.
The medical device manufacturer has the obligation to keep an up to date systematic procedure for reviewing gained experience from devices in the post-production phase. The manufacturer is also responsible to apply suitable resources which implement necessary corrective actions on the device.
Corrective actions on a device can be:

- Recall a device
- Modifications of the actually in use device
- Add more surveillance to an actually in use device
- Modifications to a new device design or components for the future
- Modifications to the process of manufacturing
- Modifications to the labeling
- Modifications to descriptions of the instructions for use

The appropriate establishment of the post-marketing surveillance system (PMS) is focus in the inspections by the notified body.
If a Member State determines because of the reporting of the device manufacturer that a medical device, which is properly installed, maintained and used, may compromise the health or safety of patients, users or other persons, it shall take appropriate temporary measures. This is described in the Safeguard clause Article 8 of the MDD. Appropriate measures can be to remove such devices from the market or forbid or limit the possibility of the device being placed on the market or put into service. This may also be the case if a device is not complying with

the directive and has the CE mark. The Commission shall be immediately informed about such measures including the reasons for the decision.

Certification Procedure

The MDD defines that all of the medical devices which are placed on the market require the CE marking. The certification process for the CE mark requires a technical documentation, a risk analysis, the proof of compliance with the essential requirements of the MDD and a product-related declaration of conformity delivered by the manufacturer.
The only medical devices which are not required to have an official CE mark including the involvement of a notified body are non-sterile class I devices. These devices are marked under the responsibility of the manufacturer with the CE mark without a notified body number.
All other devices need the certification of a Notified Body. The manufacturer is then allowed to put the CE mark in combination with the number of the notified body on the device.

2.1.2 Active Implantable Medical Device Directive 90/385/EC

The Active Implantable Medical Device Directive 90/385/EC specifies the regulations and requirements for an active implantable medical device (AIMD). An active implantable medical device is any active medical device which is purposed to be completely or partially put into the human body. The purpose for an AIMD can be diagnostics or therapeutic reasons. In the perspective of the law AIMDs are treated as a subset of medical devices. Some examples for an AIMD are pacemakers, defibrillators, infusion pumps, and ventricular assist systems.
The directive does not apply to the following devices:

- medicinal products for human use;
- human blood, blood products, plasma or blood cells of human origin or to devices which contain them at the time of placing on the market;
- transplants, tissues or cells of human origin or to products incorporating them or derived from them;
- transplants, tissues or cells of animal origin, unless a device is manufactured utilizing animal tissue which is rendered non-viable or non-viable products derived from animal

tissue.

For the reason that a human body stays a long time in direct contact with AIMDs, the standards and requirements are rigorous to protect the health and safety of patients.

Classification

An active implantable medical device as defined in 93/385/EC are not classified as in the MDD 93/43/EC or IVDD 98/79/EC. All devices require a certification of a notified body. There is no self-certification possible. The manufacturer has to decide which of the conformity assessment routes he want to use. They are not depending on a classification.
The following conformity assessment routes are described in the corresponding Annexes:

- Annex II, Full Quality Assurance
- Annex III, EC Type Examination
- Annex IV, EC Verification
- Annex V, Product Quality Assurance

Essential Requirements

The essential requirements of the AIMD are described in Annex I. They are divided into two parts. The first part are the general requirements and the second part specific requirements. The specific requirements are related to the design and construction of the AIMD. This separation is the same as for the MDD 93/42/EC.
For AIMDs it is very important to the design, manufacturing and packaging that the device is packed in a non-reusable pack. This ensures that the device is sterile when it is placed on the market and remains sterile after storage and transport conditions.
It is also needed to conduct a risk assessment with conformity to EN ISO 14971, build up a full quality system conforming to EN ISO 13485, and compile a technical file of the product. These requirements are essential requirements. The requirements for the technical file are the same as for the MDD 93/42/EC.

Vigilance System

The AIMDD 93/385/EC requires to report incidents to the European Vigilance System. The requirements for the reviewed experiences gained from the post-production phase and corrective actions are equally to the MDD.

2.1.3 In Vitro Diagnostic Device Directive 98/79/EC

The In Vitro Diagnostic Device Directive 98/79/EC defines the requirements for in vitro diagnostic medical devices. These devices include reagents, calibration and control materials, instruments, equipment, and systems that are used to examine specimens from the human body in order to diagnose diseases and monitor the health state or therapeutic procedures.
Some examples of IVD medical devices are:

- HIV tests
- Clinical chemical tests
- Pregnancy tests
- Blood sugar monitoring systems

Classification

The IVDD 98/79/EC defines four different classification groups which are based on the associated risk on the usage of the product.
The first classification group is List A. This group contains devices with the highest potential risk, which includes reagents, calibrators, and controls who can determine blood groups or infections.
The second group is List B which contains high-risk devices. High-risk devices are reagents, calibrators and controls which have the following functionality [Süda]:

- Determination of blood groups (Duffy and Kidd system)
- Determination of irregular anti-erythrocytic antibodies

- Detection of rubella, toxoplasmosis, phenylketonuria, and infections with cytomegalovirus or chlamydia
- Detection of tumor marker PSA
- Evaluation of the risk of trisomy 21
- Devices for self-testing of blood sugar-levels

The third group are self-testing devices which are subject to special requirements. The last group are other IVD devices which are not termed in Annex II or intended for self-testing. A typical device is a clinical chemistry test.

Essential Requirements

The essential requirements of the IVDD 98/79/EC are basically the same as for MDD 93/42/EC and AIMDD 90/385/EC. This includes the quality assurance to EN ISO 13485 and the technical documentation including risk assessment according to EN ISO 14971. Devices for self-testing have the additionally requirement that they have to be easy to use and have a low risk of incorrect interpretation of results.
An additional requirement which is not mentioned in the other medical device directives is that the regulatory data in accordance to the IVDD shall be stored in a European database.
These database is targeted to contain the following data:

- Related data to registration of manufacturers and devices
- Related data to issued, modified, supplemented, suspended, withdrawn or refused certificates
- Related data obtained with the vigilance procedure

Vigilance System

The requirements and procedure for the European Vigilance System are the same as described in chapter 2.1.1.

Certification Process

All classification groups except the other IVD devices need the involvement of a notified body for the CE mark certification process.

The following figures 2.2 to 2.4 show example conformity assessment procedures for list A, list B, and self-testing classification groups.

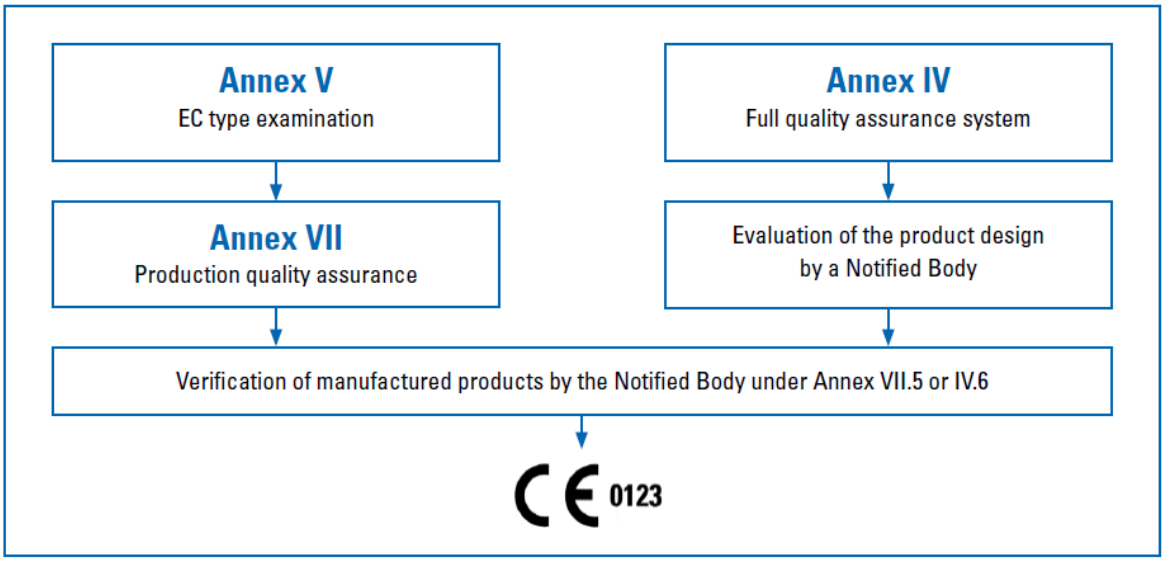

Figure 2.2: IVD conformity assessment procedure for list A HIV test kit [Südb]

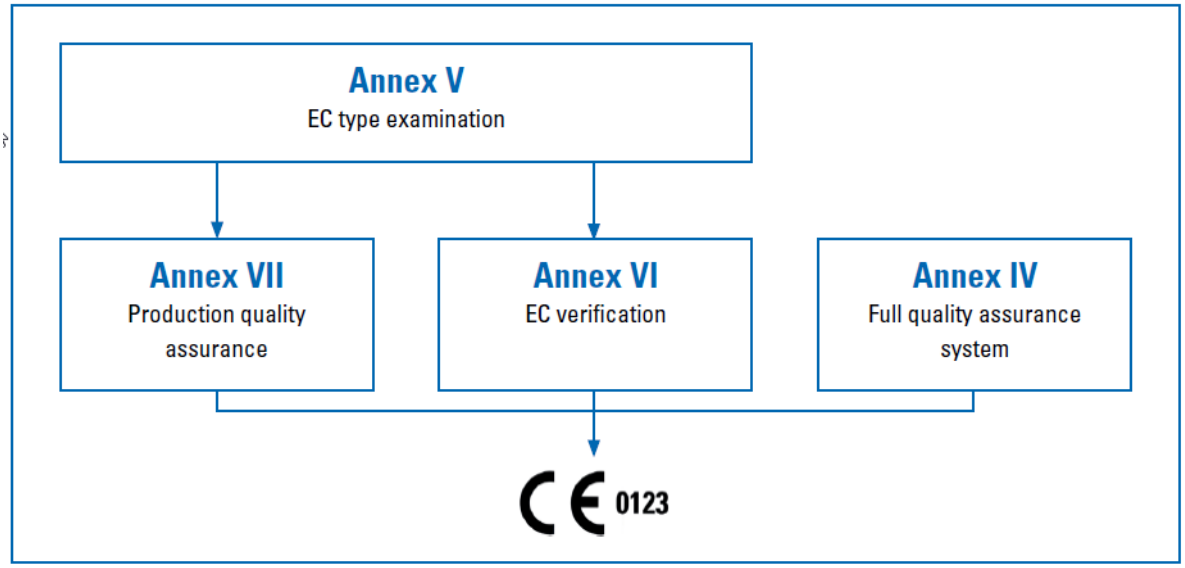

Figure 2.3: IVD conformity assessment procedure for list B Tumor maker PSA [Südb]

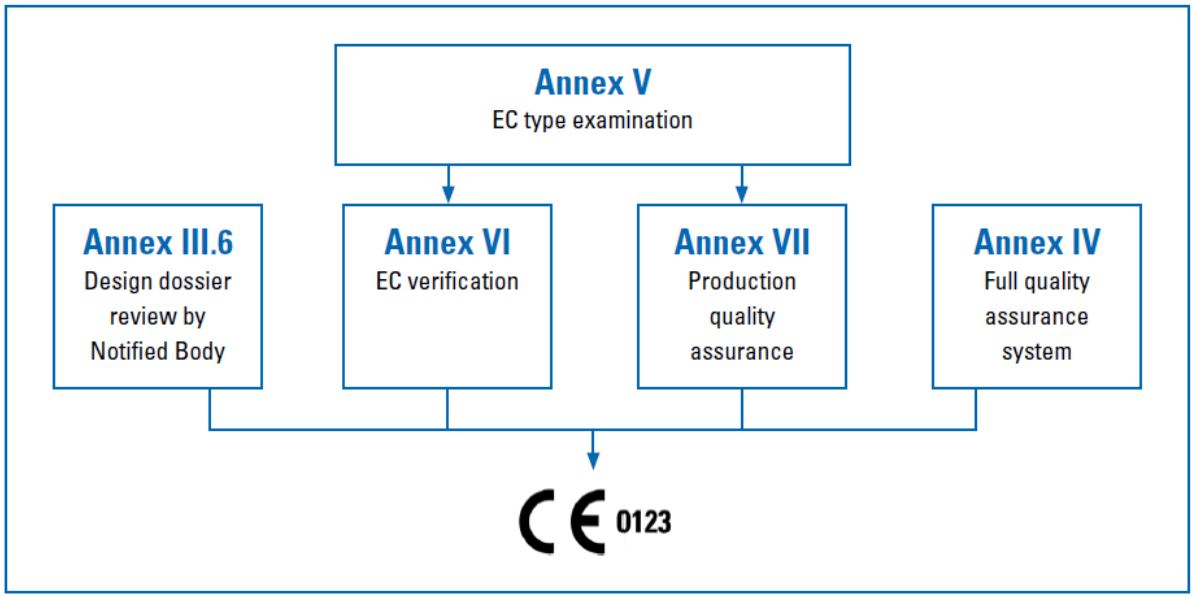

Figure 2.4: IVD conformity assessment procedure for self-testing pregnancy test [Südb]

2.2 United States of America

The process of market approval for the United States of America involves the Food and Drug Administration (FDA). The FDA specified regulations and processes for the approval of medical products.

The first step in the FDA system is to classify a medical device. This classification is based on the risk, intended use and indications for use. These three factors are the necessity of controls to provide an assurance of safety and effectiveness. The three classes and requirements which apply to medical devices are:

- Class I General Controls
  - With exemptions
  - Without exemptions
- Class II General Controls and Special Controls
  - With Exemptions
  - Without Exemptions

- Class III General Controls and Premarket Approval

Class I devices are devices which are not designed to be used for supporting, sustaining or preventing damage of a human life or health. These devices have no presence of a possible inappropriate risk for illness or injury. Class I devices are divided into two parts. One are devices with exemptions and the other without exemptions. If a device has no exemptions it is required to perform a premarket notification (510k). For a premarket notification it is necessary to show an essential equivalence with an approved device which is already on the market. All other devices which classification includes exemptions are referenced to the limitations on these exemptions.

This is different for Class II devices. Class II devices are devices which provide a higher risk than Class I devices. They have also the definition for exemptions and devices with an exemption are referenced to the limitations on these exemptions. For a device that has no exemptions, it is necessary to build up a performance standard. This performance standard shall provide the necessary and acceptable assurance of the safety and effectiveness of the device. This is done by the pre-market approval process.

The third class of medical devices are Class III devices. Class III devices are defined as devices which are proposed to be used with a major importance for supporting, sustaining or preventing damage of human life or health or have a high presence of possible inappropriate risk for illness or injury. These devices require a premarket approval application.

The FDA has established a classification database with nearly 1,700 different common types of devices and assembled them into 16 medical specialties. The medical specialties are referred to as so called panels. Each of these common types of devices have assigned their regulation class which is based on the needed level of control including a general description of the device. This general description includes the intended use and information about marketing requirements.

A manufacturer can search through this database to find the correct classification and necessary requirements for his device. If the manufacturer is not able to determine a classification for his device or it is a new type of device and there is no essential equal device, the device is classified as a Class III device. For a new device, which has a low or medium risk, the manufacturer can submit a de novo request. In that instance the FDA determines the classification based on the risk of the device.

The FDA is also able to reclassify a medical device. This can happen if the FDA receives

new information about a device. On one hand this could be a reclassification in a higher class because the new information show that the regularity controls for the actual class are not sufficient and there are more strict measures required for guaranteeing safety and effectiveness. On the other hand it is also possible that a device is reclassified into a lower class if the new information show that less strict measures are required. All of the new information have to be "valid scientific evidence". The FDA does not only have to do that on its own initiative. It can also be as a response to a petition.

Figure 2.5 shows the general overview about the market requirements in the USA.

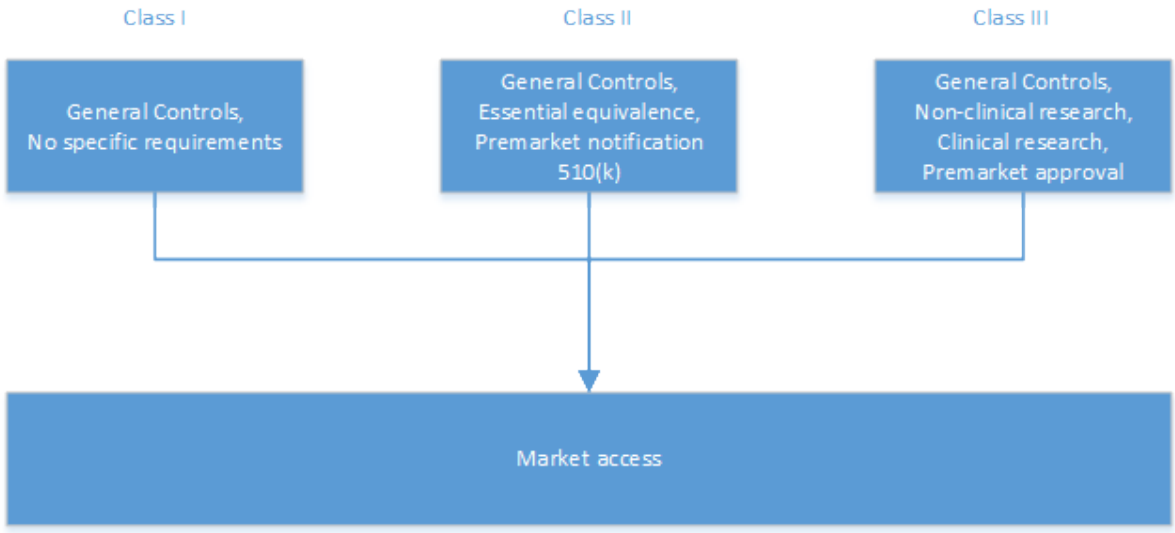

Figure 2.5: Market requirements of medical devices in the USA

2.2.1 General Controls

General controls are the elementary regulations for managing devices to guarantee safety and effectiveness defined in the Food, Drug and Cosmetic Act (FD&C Act). These regulations are necessary for all medical devices independent from their classification.

The general controls include the following regulations [FDAg]:

- Adulteration
- Misbranding
- Device registration and listing

- Premarket notification
- Banned devices
- Notification, including repair, replacement, and refund
- Restricted devices
- Good Manufacturing Practices (GMP)

Adulteration

A medical device is considered as adulterated if it contains dirty, stale, or moldered substances, or is prepared, packed, or held under unhygienic conditions. The adulteration regulations are listed under Section 501 of the Federal Food, Drug, and Cosmetic Act (FD&C Act). A device is also considered as adulterated if:

- "Its container is compounded, in whole or part, of any poisonous or health-damaging substances;
- It contains an unsafe color additive for the purpose of coloring;
- Its strength is different from, including that its pureness or quality falls below, that which it postulates to characterize;
- It is focus to a performance standard and does not fulfill with all the requirements of the standard;
- It is a Class III device and is not conform to the requirements of an approved premarket approval application or a notice of completion of a product development protocol;
- It is a banned device;
- It is in violation of good manufacturing practice requirements; or
- It is not conform to an Investigational Device Exemption (IDE)." [FDAg]

Misbranding

The regulations for misbranding defined in Section 502 of the FD&C Act cover drug and device labeling. This section contains not only regulations which apply to drugs and devices but there are also specific regulations for drugs or devices. The following cases define a device to be misbranded [FDAg]:

- The labeling is wrong or misleading
- The packaging does not have a label containing:
  - Name of the place of business of the manufacturer, packer, or distributor
  - A statement of the quantity of the content, concerning weight, measure, or a numerical count

  A reasonable variation and exemption for small packages may be permitted.
- A label containing required information is not placed prominently or is not clearly stated to be read and understood
- Has no label containing adequate directions for use
- The label containing directions for use is dangerous to health
- The label does not comply with the color additive regulations
- The device name is not prominently printed
- A restricted device which
  - uses false or misleading advertising
  - is sold, distributed, or used not conforming the restricted device regulations
- All advertisements or descriptive materials for a restricted device are not including
  - a true statement of the name of the device which is prominently printed, and
  - a brief statement for intended uses, relevant warnings, precautions, side effects, and contradictions.
- A device, subjected to a performance standard, does not have the correct labeling of the standard

Device registration and listing

Manufacturers and specified processors require to register their establishments with the Food And Drug Association (FDA). This registration includes a list of all devices which they are operate. This is required for manufacturers along with repackers, relabelers, and importers.
Manufacturer are also required to submit a list of all devices they produce or process to the FDA.

Banned devices

The FDA is authorized to ban devices from the market. This process is described in Section 516 of the FD&C Act. A ban of a device can happen if a device presents substantial deception or unreasonable risk of illness or injury. The determination is made out of all the available data and information, including the appropriate classification panel. If the risk of illness or injury cannot be corrected by a change of the labeling, it is necessary to publish a regulation in the Federal Register by the FDA. Otherwise if the risk of illness or injury can be corrected by a change of the labeling, it is necessary to contact the responsible person for the device for correcting it. Therefore it is necessary to fulfill the correction in a specific time. If the time is exceeded the FDA can still publish the regulation. Also all interested persons get the opportunity for an informal hearing on the proposal. After that the FDA will affirm, modify, or revoke the proposed regulation. In case of an affirmation or modification of the proposal the FDA will publish a final regulation, in case of a revocation a notice to this effect will be published.

Notification

Section 518 of the FD&C Act describes the general requirements for the FDA to order a notification and the regulations for repair, replacement, or refund of a medical device.
A notification is used to notify all health professionals who use the device and other person including manufacturers, importers, distributors, retailers, and device users that a device is resulting health risks and that these risks can be reduced or eliminated.
The requirements for the FDA to order a notification for a device are [FDAg]:

- Presence of unreasonable risk of substantial harm to public health

- Necessity to notify to eliminate risk
- No practicable means available in the FD&C Act Act to eliminate risk

The FDA is also authorized to order a repair, replace, or refund of a device which shows unreasonable health risks. This is done after an informal hearing.
The criteria which determine a repair, replacement or refund are the following [FDAg]:

- Device shows unreasonable risk of substantial harm to public health
- Device not fulfilling the requirements for the prevailing state of the art
- The risk is not caused by a wrong installation, maintenance, repair, or use by a user
- A notification is not sufficient and repair, replacement, or refund is needed

Restricted devices

The FDA is authorized with the regulations described in Section 520 to restrict the selling, distribution, or use of a device. The reason for restricting a device is if the device has no reasonable assurance of its safety and effectiveness. Such a restricted device is only able to be sold with an oral or written authorization or under specified conditions of the regulation. The oral and written authorization is done by a licensed practitioner.

Good Manufacturing Practices/Quality System Regulation

The FDA is able through Section 520(f) to release regulations which define the requirements for used methods and controls. These methods and controls contain manufacturing, packing, storing, and installing a device. The requirements define a conformity to the actual good manufacturing practices. The quality system regulations can be achieved with conformity to the ISO 13485 standard.
The risk management is also a part of the quality system regulation. The definition of the risk management in the quality system regulations include:

- Identification and description of hazards
- Occurrence of hazards

- Expected consequences of hazards
- Assessment of the likelihood of the hazards

The risk management is focused on controlling of risks which is a central requirement of the quality system regulation. The international standard ISO 14971, which describes the procedure for risk management of medical devices, is included in the standards database of the FDA and applies for the quality system regulation without restrictions.

2.2.2 Special Controls

Special controls are regulation requirements which apply for class II devices. For these devices general controls are not sufficient to provide a good assurance for safety and effectiveness. It is necessary to establish special controls to provide these assurance. A requirement to establish such special controls are sufficient information about to device. Special controls are in general device specific and include the following:

- Performance standards
- Postmarket surveillance
- Patient registries
- Special labeling requirements
- Premarket data requirements
- Guidelines

The 21 CFR defines performance standards for electronic medical devices and special electronic medical devices. The general regulations for electronic medical devices include certification and identification requirements and variances and exemptions regulations. Additionally there are special regulation definitions for ionizing radiation emitting, microwave and radiofrequency emitting, light-emitting, and sonic, infrasonic, and ultrasonic radiation-emitting medical devices. This regulations include special labeling and cover the limits of emissions and exposure. For other devices the 21 CFR Part 861 defines procedures for performance standards.

For the post-market surveillance manufacturers are required to report every incident and event to the FDA which needs an immediate action to prevent a risk of damage to public health. The FDA requires the manufacturer to get appropriate information about the event. If a device on the market caused death or serious injury through a malfunction or has malfunctioned, manufacturer are required to report that fact immediately to the FDA. It can happen, that a related device on the market would be likely to have the same malfunction and can cause death or serious injury.
It is not only required by the manufacturer to report those incidents, facilities who use these devices, for example hospitals, ambulatory surgical facilities, or nursing homes, are also required to report incidents. These facilities require to report the following information:

- Devices related to death to the FDA and the manufacturer
- Devices related to serious injuries to the manufacturer. If the manufacturer is unknown the report has to be to the FDA.
- A summary to the FDA on an yearly basis which contains all reports send during that period

2.2.3 Premarket Notification (510k)

The premarket notification is a notification to the FDA where the manufacturer shows that the device they want to place on the market is as safe and effective as a similar device which is already sold legally on the US market.
A device is defined as substantially equivalent with an device on the market if

- it has the identical intended use and technological characteristics, or
- it has the identical intended use and different technological characteristics, but it does not bring up new questions on safety and effectiveness and shows that it has at the minimum the safety and effectiveness as the legally device on the market.

Devices which apply to the 510(k) procedure have to submit an application if:

- The device is commercially sold for the first time.

- The proposed intended use changed for an already in commercial distribution device.
- A legally marketed device is changed or modified for which the change can affect the safety or effectiveness.

The required components of a 510(k) include:

- A detailed description of the device
- An identification of the to be compared with device on the market
- A description of the intended use of the device. In case of a difference from the compared device it is necessary to describe why these differences will not have any effect on the safety and effectiveness.
- A summary of the technological characteristics where the technological characteristics of the new device are compared with the one on the market. In case of different technological characteristics it is needed to provide a summery and describe how equal the characteristic is to a characteristic of the marketed device.

2.2.4 Premarket Approval (PMA)

The PMA is an assessment process of the FDA for safety and effectiveness of medical devices. This process applies to most of the Class III devices, for which the special controls alone including the 510(k) are not sufficient to assure safety and effectiveness.
The main part of the PMA documentation is the scientific evidence section. The rest of the PMA documentation are administrative elements. The scientific section is separated into two research parts. One is the non-clinical research and the other one clinical research.
The non-clinical research part contains information about the microbiology, toxicology, immunology, biocompatibility, wear and shelf life of the medical device and additional information. The regulations and requirements for conducting a non-clinical research are defined in the Good Laboratory Practice For Nonclinical Laboratory Studies (21 CFR Part 58).
The clinical data required for the clinical research includes study protocols, safety and effectiveness data, adverse reactions and complications, device failures and replacements, patient

information, patient complaints, tabulations of data from all individual subjects, results of statistical analyses, and any other information gained from clinical research.
Clinical studies have to be done according to the Good Clinical Practices regulations which include Protection of Human Subjects, Institutional Review Board (IRB) and Investigation Device Exemption. The investigational device studies are parted into three different classes. The classes are significant risk (SR) device studies, non-significant risk device studies and exempt studies. An investigated device which is in the class for significant risk devices requires the approval of the FDA and the IRB. A non-significant risk device needs only the approval of the IRB. For exempt studies which can be a comparison study between two already approved devices on their indications for use that does not have an additional risk for patients does not require any approval from the IRB or FDA.
The manufacturer has to show that the benefit-risk ratio of his device is in an acceptable range. This includes investigations using laboratory animals, involving human subjects and non-clinical investigations including in vitro studies. A benefit-risk ratio in an acceptable range is accomplished if the investigations show that the benefit to health from using the device as its intended use overbalance any probable risk.
The effectiveness is determined through the delivery of clinically significant results which show the intended use and conditions of use of the device for a substantial helping of the aimed population.

2.2.5 De novo classification

The de novo classification process is a process for new medical devices which are automatically classified as class III devices. The classification is unrelatedly to the level of risk because there are no legally marketed devices to compare with. The de novo process then reclassifies the device. This is possible if the devices are able to show that general controls or general and special controls provide the necessary assurance of safety and effectiveness.
A manufacturer has also the possibility to submit a de novo request if a 510(k) review was already subjected to the device and the device determined to be not substantially equivalent (NSE). This can have the reasons that there was a lack of an identifiable comparable device, a new intended use, or a different technological characteristic which brings up different questions of safety and effectiveness. That is only possible if there is an existing class III classification

regulation or at least one approved PMA. If not, it will not be reclassified into class I or II and will, in general, not be eligible for the de novo process.
In general the following criteria shall be fulfilled for a device for which a de novo classification is submittedc [FDAb]:

- The device should be at a low or moderate risk
- The device should meet the legal standards for class I or class II (general controls or general and special controls provide safety and effectiveness assurance)
- The manufacturer should be able to describe all of the known risks and benefits including how the known risks can be successfully alleviated and effectiveness can be guaranteed through general controls or general and special controls

The required contents of a de novo application are the following [FDAb]:

- Administrative information (e.g. applicant name, address, phone)
- Regulatory History (prior submissions)
- Device information and summary
- Change summary (all detailed changes made to prior 510(k) or pre-submission)
- Classification summary
- Classification recommendation
- Proposed special controls (for class II devices)
- Supporting protocols and data
- Summary of benefits
- Summary of known potential risks to health
- Risk and mitigation information
- Benefit-risk meditations
- Device labeling

The de novo applications are processed by the FDA.

2.2.6 FDA 21 CFR Part 11

The FDA offers the possibility to submit applications for medical devices in a digital form. To be able to submit applications in a digital form it is necessary to have a digital signature on it. This digital signature replaces the handwritten signature and it is not necessary to print the complete application and send it to the FDA. The Code of Federal Regulations (CFR) Title 21 Volume 1 Part 11 by the Food and Drug Administration for electronic records and electronic signatures describes the necessary regulations and criteria for using electronic records, electronic signatures, and handwritten signatures on electronic records instead of paper records and to be considered as trustworthy, reliable, and equal to handwritten signatures on paper records. These regulations apply to created, modified, maintained, archived, retrieved, or transmitted records, which are in an electronic form.

A general description is provided as the first subpart of the CFR Part 11. The first subpart also includes a definition for the requirements of using electronic records and electronic signatures on electronic. For records, which are only required to be maintained and not submitted to the Food and Drug Administration, a person can use electronic records instead of paper records or electronic signatures instead of traditional signatures for all records or only partly. Records, which need to be submitted to the Food and Drug Administration, need to be included in the public docket No. 92S-0251 to be accepted in electronic form and fulfill the requirements of the FDA 21 CFR Part 11. When the type of submission is not included in the public docket, the submission will be considered as not official and it is necessary to send the records in paper form.

An electronic record is defined as any combination of information types in a digital form which is processed by a computer system. An electronic signature for an electronic record is compilation of symbols by a computer which is used by an individual as a legal binding equivalent of the handwritten signature.

Additionally to the first subpart there are two further subparts which describe the requirements and regulations for electronic records and electronic signatures.

Electronic Records

The subpart "Electronic Records" contains a definition of procedures and controls for closed systems and open systems. This subpart also contains requirements on how signatures need to be shown on an electronic record and that electronic signatures shall be linked to the electronic record. A signed record shall contain the printed name of the signer, the date and time of the signature execution, and the type associated with the signature. The type can be a review, approval, responsibility, or authorship. This information shall be included in every readable form of the electronic record.

Closed Systems

The definition of a closed system is an environment where the system access is controlled by a person responsible for all content of electronic records saved on the system.
People who are using a closed system have the requirement to apply procedures and controls which shall guarantee the authenticity, integrity, and confidentiality of electronic records, and that a signer cannot revoke a signed record as not authentic. This also includes the validation of systems for accuracy, reliability, performance, to be able to recognize not valid or altered records, and operational system checks. For the protection of records, limited access is only granted to authorized persons. Therefor the system requires an authority check and be able to generate an audit trail which contains the date and time of logins and executed actions on records.
External devices, like terminals, are required to be checked if they are still able to deliver valid data input or operational instructions. The people who develop, maintain or use these systems must have the education, training, and experience to perform their tasks. This also includes an appropriate use of controls for documentation which contains controls for distribution, access, and use of the documentation and also revision and change control procedures.

Open System

An open system is the opposite of a closed system. For an open system the system access is controlled by a person who is not responsible for the content of electronic records.
The open system has the same requirements as the closed system with additional requirements.

These additional requirements include the usage of encrypted documents and appropriate digital signature standards, which secure the authenticity, integrity, and confidentiality of a record.

Electronic Signatures

Electronic signatures are a data combination of any symbol or multiple symbols, which are executed, adopted, or authorized to be a legal equivalent of the handwritten signature of a person.

The general requirements for electronic signatures are that every electronic signature to be unique to a person. An electronic signature shall not be reused by another person or reassigned to another person. A person who desires to use an electronic signature has to certify to the FDA that the electronic signature on the system is used to be the legal equivalent of his handwritten signature. The certification has to be submitted in paper form and signed with a handwritten signature. The FDA can request a person to provide additional certification to ensure that a specific electronic signature is his legal equivalent of his handwritten signature.

For biometrics it is required to ensure only the usage of the owner and cannot used by any other person. Biometrics is a method verifying a person's identity through measuring his physical features or actions which can be measured and are unique for the person.

It is also required if the electronic signature is not based on biometrics, that there are at least two distinct identification components. These components can be a code and a password. If a user executes a number of signings during a single system access, only the first signing shall use all electronic signatures. The following signings are able to be signed by at least one component which is only feasible by the person. In general all electronic signatures are only be used by their owner.

For codes or passwords it is required to have controls which ensure the security and integrity. This includes the maintaining of unique combinations of code and password and that they are periodically checked, recalled, or revised. This shall pretend old still working passwords or that a password can get known by other people. Another requirement is to have a procedure for handling the loss of devices which are generating identification code or password information. This includes a temporary replacement. All unauthorized uses of passwords and/or identification codes are required to be detected and reported to the organizational management. This unauthorized access requires also periodical checks of devices to be sure the devices functional

properly and have not been compromised.

2.3 International Standards

2.3.1 EN ISO 13485

The International Organization for Standardization (ISO) EN 13485 was first published in 2003. In the current version EN ISO 13485:2012 published in 2012, the standard adopts the European Medical Devices Directive 93/42/EEC.

EN ISO 13485 is specifying the requirements for a quality management system on which an organization has to demonstrate its ability to provide medical devices and related services. This quality management system systematically follows customer requirements and regulatory requirements that are applicable to medical devices and related services.

The basic aim of the standard is to allow harmonized medical device regulatory requirements for quality management systems. This resulted in the adoption of ISO 9001 process-based model to EN ISO 13485 standard for a regulated medical device manufacturing environment.

Some of ISO 9001 standard requirements which are not suitable for medical devices were excluded in EN ISO 13485. For these other requirements were added which are appropriate for medical devices.

This means that a medical device manufacturer which complies with EN ISO 13485 standard does not automatically comply with ISO 9001 standard. The quality management must also accomplish to the excluded regulation requirements.

The EN ISO 13485 standard gives support for medical device manufacturers to develop a quality management systems, which helps to establish and maintain processes effectively. It also ensures consistency for designing, developing, producing, installing, and delivering medical devices to be safe for their intended usage.

If a medical device manufacturer wants to obtain EN ISO 13485 certification the company needs to implement the following procedures [ISO]:

- Document and record controls
- Internal auditing procedures

- Controls for non-conformance
- Corrective and preventative actions
- Process and design controls
- Record retention
- Accountability and traceability

After developing all of these written policies the company receives the following benefits from EN ISO 13485 certification [ISO]:

- Access to markets that recognize or require the certification including Canada and Europe.
- Reduction in operational costs by highlighting process deficiencies and improving efficiency
- Increase in customer satisfaction by consistently delivering quality products and systematically addressing complaints
- Proven commitment to quality through an internationally recognized standard
- Added transparency to the way complaints, surveillance or product recalls are handled

2.3.2 EN ISO 14971

The EN ISO 14971, Risk Management Requirements for Medical Devices, contains details of the risk management principles and practices. These are referenced in a number of medical device standards.
The referenced device standards can be found in the 3rd edition of IEC 60601-1 (electrical safety), ISO 13485 (quality management systems), IEC/EN 62366 (Usability of medical devices), ISO 10993 (biological evaluation) and IEC 62304 (medical device software).
The risk management process described in EN ISO 14971 includes the following tasks:

- The identification of hazards and hazardous conditions on a medical device which could harm patients or healthcare workers

- The rating on the potential occurrence of such a risk and evaluating the dimension of the consequences
- The developing and implementation of active safeguards into the device or the production process to prevent or control risks
- The periodic reviewing and monitoring of the risk management controls and process to evaluate the effectiveness

Each process aspect of the risk management system is completely documented to supply proof of the manufacturer's commitment to controlling risk consistently the whole life of the medical device design.
The process steps defined in EN ISO 14971 are listed below and should be performed in the following order:

- Implementing of sub-classes to identify characteristics which are related to safety and identify hazards and hazardous situations
- Estimate the identified hazards and hazardous situations
- Evaluate the identified hazards and hazardous situations
- Conduct tests and measurements to prevent hazards and hazardous situations
- Verify the risk controls for the hazards and hazardous situations with a reference to the conducted tests and measurements
- Estimate and evaluate the residual risk after performing the risk controls

To obtain regulatory approval in the United States, European, Japanese, Australian and other major markets, it is necessary to conduct a risk management of the device which can be done in compliance with the EN ISO 14971 standard.

2.4 Differences

The USA and Europe have both working regulations for putting a medical device on the market but there are still differences. The major difference is that in Europe the government is not

directly involved in the process of market approval for a medical device. There are third party companies "Notified Bodies" which certify for working as a notified body. They verify if all the regulations are met and they are only monitored by the government. There can be several different notified bodies in one member state of the European Union. In the US there is only the FDA which monitors, verifies, and grant market access for medical products.

Another difference is that the USA has only three classes which assess the level of risk for the medical device. All devices are categorized in one of these classes. The European system has three different regulations for special definitions of medical devices which have their own classification system. General medical devices have four classes, in vitro devices have two classes and for active implantable devices there is only one class for all of them.

All of these classes need to ensure their safety against users, patients and other people. To ensure this safety all manufacturers in both countries shall use the ISO 14971 to fulfill their risk management for the medical devices. In Europe all of the classified medical devices, except class I devices, need to ensure their safety. In the US system only class III devices need to provide a full risk management. For class II devices it is sufficient if the medical device can ensure the same safety as an already approved device on the market which has the same technological characteristics or usage.

There are also class II medical devices which have to provide a full risk management but this only applies if there is no substantial identical device on the market and the de novo process has to be done.

Another difference in both systems is the surveillance system. In Europe it is only necessary to report incidents which have a serious or potential serious safety problem. These reports need to be send to the country surveillance system and the European vigilance system. The US handles this differently. A manufacturer is recommended to send a report with the full information of every incident to the FDA. The FDA decides which countermeasure is appropriate for this incident. This countermeasure can be a replacement, a pay back or a proscription of the device.

In the USA the FDA offers the possibility to submit applications for medical devices in a digital form if the manufacturer complies with the regulations of the CFR 21 Part 11. The European Council has published a comparative directive for digital signatures and the certification process to be able to digitally sign a digital document. This directive is 1999/93/EC in which

the necessary requirements are specified. In the process of placing a medical device on the European market, there is no general implementation to use digital signatures to submit the documentation to the Notified body in a digital form. Only for some special medical devices, the process was changed to be able to submit a digital documentation.
Additionally to the fact that the USA only has the FDA as institution for market access, the regulations made by the European Union are not completely implemented in the legislation for medical devices of every member state. Only some member states have a conformity with the European regulations.

3 Data Privacy for Medical Device Records

3.1 Europe

The three directives for medical devices define that the member states shall ensure that all gained information during the application process are bound to be observe confidential. The gained information includes all the information of the application process, including the technical documentation. This is required for all parties involved in the process. The communication between member states and notified bodies is excluded to be confidential for the release of warnings and information about persons who publish information under a criminal law.
In general the European Council published the directive 95/46/EC to regulate the protection of individuals for privacy with regard to processing of personal data. This is important for the requirement of clinical data for medical devices. Most of clinical data are reports which describe details about a medical device which are used on a patient. It is necessary to protect the individual patient from being retrospectively identified. It is also important to fulfill this data protection requirement for data which is not saved in a database with special security specifications for personal data. Nowadays there are technologies for data mining and database linkage which increases the possibility of unlawful retrospective identification of patients, these security specifications keep patients information safe.
Regarding to these requirements for information and data records, specialized software which is used to generate these information need to fulfill the requirements too. This is extremely important for software which sends data over a network connection. Therefor the connection shall be highly secured, allowing no on except authorized users of the software the ability to read, write, and delete data.

3.2 USA

The FDA generally protects all records which contain data about individuals. This policy is defined in the 21 CFR Part 21.10. For those records about individuals the FDA is required to collect, maintain, use, and disseminate them to "protect the right to privacy of the individual" (Administration). This protected data records are for example clinical data records which are used in the PMA process.

The general approach for the premarket notification, premarket approval and the de novo process is that all the information contained in the application are considered as confidential before the approval or decline from the FDA is given. After the approval or refusal of the application, the FDA will publish a summary about the information in their public system. In 21 CFR Part 20.61, the FDA defines that a person who submits records can define parts or the complete information to be excluded from publication. This right is defined in the Freedom of Information Act. In general, the Freedom of Information Act defines that everyone has the right to access information which is held by the US government agencies. If the person who submits records does not define parts or the complete information to be excluded, the FDA will automatically exclude specific information. The FDA will exclude information with substantial reasons for exclusion or the information belonging to trade secrets and commercial or financial information.

Trade secrets and commercial or financial information are generally excluded from publishing. The FDA notifies the submitter about information which are excluded from publishing by the submitter but they are necessary to be published. On the other hand the FDA also notifies the submitter if the FDA decided to exclude information from the publication. These notifications always contain an explanation for the decisions of the FDA.

For the premarket approval process the premarket notification procedure defines that the submitter has to write a statement in the application that he agrees on publishing a duplicate of the premarket notification which includes all of the safety and effectiveness information but excludes all patient identifiers, trade secrets, and confidential commercial information.

In general this means that all of the information contained in the process of approval of a medical product are confidential until the FDA publishes safety and effectiveness information in their system. Software, which is used to conduct the information, are encouraged to be not

vulnerable by third parties and all the information that is stored is safe and can only be read by the people who have to authorization to read them.

3.3 Summary

Both countries have defined regulations which mark the conducted data during both processes of approving a medical device and getting a confidential medical device on the market. This means that all the data, including the risk assessment of the medical device are confidential data during the process. For this reason software which helps conducting this data needs to be highly secure. This insures no data can be retrieved from a third party. This high security also includes that the software is not vulnerable to others.

4 Challenges

The API design and implementation has to face several different requirements. These are necessary to not influence the current capabilities of the software. The following challenges are required to be fulfilled:

- Authorization
- Authentication
- Data Transfer Security
- Reliability
- Generalization
- Performance
- Digital Record requirements FDA 21 CFR Part 11
- Mapping request URL with generalized code
- Run API as a windows service

The Authorization and Authentication challenges are also included in the FDA CFR 21 Part 11 requirements. It is necessary that only authorized and authenticated users are able to access to the data. No other user shall be able to access data from the API. All data requests to receive data from the database or save data onto the database have to check the authentication on the system. It is necessary to check if the requested action is authorized for the authenticated user. This is one of the major challenges for the API because the transferred and processed data will be confidential and by the law must be handled as such.
This leads to the second major challenge, that all of the transferred data shall be encrypted and secured from being read by others during the transfer from the client to the API. This

requires a secure protection to ensure that no one can read the authentication information in a request.

The API shall also deliver and store reliable data. This means that all the data delivered shall be the correct data which was previously requested and data shall be stored as it was delivered in the request. The challenge of generalization is coupled with the reliability and the third major challenge. The API shall be able to work with different software and shall adopt the data structure without adding or changing the code. This adoption shall also be able to work with changes in the current business logic. That means if the code of the business logic changes, like new classes are added or code is changed, the API shall still be able to use this code without the necessity to change code of the API. The prototype will be implemented for using the business logic of the Qware Risk Manager but it shall be capable to work also correctly with other business logics.

Another big challenge of the .NET Web API system is to find the best URL mapping for the API to insure that the API correctly understands what a user who sends a request wants to receive.

The general design of the .NET Web API is based on the Microsoft Internet Information Service (IIS), which is a web server from Microsoft for .NET web applications. In our case, the API shall also be able to be run as a windows service without using the IIS web server. This challenge originates from the Qware Risk Manager software which is not used as a web based application. The software is a windows application which runs in a Microsoft Windows operating system environment.

Microsoft Windows operating systems offer the possibility to let console applications be run as a background service. This leads to the requirement to implement the API as a console application and not as a web service. This challenge needs to be fulfilled by the API, that it has no requirement to be run using a web server.

All the challenges of Authorization, Authentication, Reliability, Generalization, Performance, Digital Record requirements of the FDA 21 CFR Part 11, the mapping request URL with generalized code, and that the API can be run as a windows service need to be carefully attended for the design and implementation of the API. The API needs to achieve all of them to be securely and economically usable for the environment of medical devices.

5 State of the Art API

The current state of the art API for Microsoft .NET is the ASP.NET Web API. ASP.NET Web API is a web based framework which is built over Microsoft .NET 4.0 and above. It implements the Hypertext Transfer Protocol (HTTP) specifications. With this implementation the framework can be used to build or ingest a HTTP service, unlike to Simple Object Access Protocol (SOAP) or Windows Communication Foundation (WCF) services.
Another beneficial aspect of the framework is, that it already contains a dependency for the Newtonsoft JavaScript Object Notation (JSON) library. The Newtonsoft JSON library is an open source library which implements JavaScript Object Notation (JSON). JSON is a compact text form data format which is easily readable and used for transmitting data between applications. JSON objects are independent from the programming language because every common programming language has a parser for it.
The ASP.NET Web API is designed to build a connection between the HTTP programming model and the .NET Framework programming model. It can be used in very different modern architectures.

5.1 ASP.NET Web API Techniques

There are three different techniques to implement and use the Web API framework, which are the general usage of HTTP specifications, SOAP over HTTP and Representational State Transfer (REST).

5.1.1 HTTP

The HTTP technique is based on the general specifications of the HTTP protocol for communication over the internet. The protocol is commonly used for messages between web browsers

and web servers. The specifications include the Uniform Resource Identifier (URI), HTTP Methods and HTTP Status Codes. In ASP.NET the common form for this API technique are HTTP Handlers.

URI

The general usage of the URI is to uniquely identify resources. These resources are not only web sites in the World Wide Web rather all the pieces of information which can be unmistakably identified by the URI.

```
1 <scheme name> : <hierarchical part> [ ? <query> ] [ # <fragment> ]
```

Listing 5.1: HTTP scheme URI syntax

The URI consists of a unique syntax to identify resources. The syntax is build up starting with a URI scheme name, which can be http or https, followed by a colon character and scheme specific part. This scheme specific part can vary between different scheme names but for http and https they are equal. The specific part of http and https contains a hierarchical part and an optional query and fragment part. Listing 5.2 shows an example URI from google.com.

```
1 https://www.google.de/webhp#q=fda+21+cfr+part+11
```

Listing 5.2: URI query and fragment example

The hierarchical part identifies for example a folder structure or a web site hierarchy. With the optional query and fragment part it is possible to define additional parameters. An example for the query and fragment can be seen in Listing 5.2.

HTTP Methods

The HTTP protocol uses different request methods so called "verbs". The most common used method is the GET method. The GET method tells the server which resource the client want to receive from the server. Listing 5.3 shows the syntax of an HTTP GET method.

```
1 GET / HTTP/1.1 http://www.google.com
```

Listing 5.3: GET Method Syntax

Another very common used method is POST, which allows to send data to the server. For example this is used to submit the data of form. There are several other request methods which are not used by this technique. The GET method is the general used one.

HTTP Status Codes

The HTTP Status codes are very useful for the HTTP technique and are very important in the HTTP protocol as well.
The status codes are parted into five classes [UK13]:

- 1xx Informational

  Status codes beginning with a 1 indicate a temporary response, which has only a status line and can optionally contain a header.

- 2xx Successful

  Status codes beginning with a 2 indicate that the request was received successfully, understood and accepted.

- 3xx Redirection

  Status codes beginning with a 3 indicate that it is necessary to do further action to complete the request. This can be a redirect to another URI.

- 4xx Client Error

  Status codes beginning with a 4 indicate that it seems the client has done something wrong.

- 5xx Server Error

  Status codes beginning with a 5 indicate that the server recognized an internal error or is not able to process the request.

All of the status code classes are contain several other status codes which indicate an explicit status of the class.

5.1.2 SOAP

The SOAP technique is using the Simple Object Access Protocol. This is the specification for the exchange of structured data messages using web services that are built on top of the

HTTP protocol. The SOAP programming model represents the Remote Procedure Call (RPC) paradigm where a client calls a method or procedure on the remote system by sending an input message with parameters and receiving a response message. The general design of SOAP is to build web services to connect different platforms and systems. It also enables the possibility to implement services according to Service-Oriented Architecture (SOA). It uses Extensible Markup Language (XML) as the structured data. SOAP follows the same request and response model as HTTP where the request and response messages are represented by XML.
Listing 5.4 shows a SOAP request message for calling a method GetCustomer with a customer name as parameter.

```
1  <?xml version="1.0"?>
2  <soap:Envelope
3  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
4  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
5    <soap:Body xmlns:m="http://www.example.org/customer">
6      <m:GetCustomer>
7        <m:CustomerName>John Doe</m:CustomerName>
8      </m:GetCustomer>
9    </soap:Body>
10 </soap:Envelope>
```

Listing 5.4: SOAP Request

The message contains header information and the SOAP envelope which encloses the body of the message. The method of the service that should be processed is GetCustomer with a required parameter CustomerName which is John Doe. The messages in SOAP are send with the HTTP POST method because the message contains data and according to the HTTP specifications data is send with the POST method. Listing 5.5 shows the POST request header of SOAP.

```
1  <?xml version="1.0"?>
2  <soap:Envelope
3  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
4  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
5    <soap:Body xmlns:m="http://www.example.org/customer">
6      <m:GetCustomer>
7        <m:CustomerName>John Doe</m:CustomerName>
8      </m:GetCustomer>
9    </soap:Body>
10 </soap:Envelope>
```

Listing 5.5: SOAP HTTP POST Request Header

After a successful processing, the server returns a response message with the demanded data of the request message. Listing 5.6 shows this response message.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml
charset=utf-8
Content-Length: 390
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/customer">
    <m:GetCustomerResponse>
      <m:CustomerId>111</m:CustomerId >
      <m:CustomerName>John Doe</m:CustomerName>
    </m:GetCustomerResponse>
  </soap:Body>
</soap:Envelope>
```

Listing 5.6: SOAP Response Message

SOAP also contains definitions for messages which have the status code 200 but tell as response message that the given data cannot be processed or that there is no result found. This can be seen in Listing 5.7.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml
charset=utf-8
Content-Length: 390
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/customer">
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>Customer could not be found.</faultstring>
      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Listing 5.7: SOAP Error Response

5.1.3 REST

REST in Theory

The architecture style of REST is using HTTP as the application protocol and hypermedia. The term of REST was introduced by Roy T. Fielding in the year 2000 and means REpresentational State Transfer. The REST approach is constraint driven instead of other approaches which are requirements driven. Constraint driven means that the REST approach is sensible of the eight fallacies of distributed computing and the architecture has defined constraints which are protecting an application from effects of those fallacies.
The eight fallacies of distributed computing are [UK13]

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.
- Topology does not change.
- There is one administrator.
- Transport costs are zero.
- The network is homogeneous.

REST defines six constraints which are needed to be applied by a RESTful architecture to cover the eight fallacies of distributed computing. Table 5.1 shows these six constraints including the benefits a RESTful application receives.

Constraint	**Benefits**
Client/Server	Evolvability, portability of clients, scalability
Stateless	Reliability, visibility, scalability
Cacheable	Performance, efficiency, scalability
Layered system	Manageability, scalability
Code on demand (optional)	Managing complexity, extensibility
Uniform interface	Evolvability, visibility

Table 5.1: Six REST Constraints and Benefits [UK13]

Client/Server

The Client/Server constraint improves the separation of concerns with separating the server from the client. This means that the client and the server are allowed to evolve independently because the client does not have to concern data storage and the server does not need to take care of user interfaces. It leads to a more simple and scalable server-side code.

Stateless

The stateless constraint makes sure that none of the client context is saved on the server. This takes from the client that every request contains all the needed information for the server to process the request. Session states have to be saved on the client. The stateless constraint handles reliability, visibility, and scalability.

Cacheable

Clients are able to cache responses received from web servers. When a client shall not cache a certain response because of using data which has changed or is not actual anymore, the server has to define responses implicitly or explicitly to be not cacheable or cacheable. If this is implemented correctly, caching makes a lot of communication between server and client unnecessary which improves as a result the performance and scalability of an application.

Layered System

Systems can improve the overall scalability through adding intermediary servers, load balancers or shared caches. Through the enforcement of security policies the security can be improved.

Code on Demand

Servers are able to modify or add functionality to the client with sending executable code. An example for that is client-side JavaScripts.

Uniform Interface

The uniform interface consists of four concepts [UK13]:

- Identification of resources
- Manipulation of resources through these representation
- Self-descriptive messages
- Hypermedia as the engine of the application state

The identification of resources intends that resources are identified by URIs. Resources are separated from their representation, which means that a server transmits an XML or JSON

fragment that represents a resource including its properties. Manipulation of resources through these representation means that a client is able to, if he has the permission to do that, modify or delete the resource. This is only possible if the client has a representation of a resource including the metadata. Self-descriptive messages intends that all messages contain all the necessary information to give a description on how to process the message. Hypermedia as the engine of the application state means that a client is able to traverse through an application after receiving a representation of the resources of a fixed entry point URI. The client does that by dynamically generated hypermedia.

Usage of REST

A RESTful architecture uses the HTTP methods directly to receive or manipulate resources. Resources are identified by a unique URI and can be received by calling the URI with the GET method, updated by calling the PUT method, deleted by calling the DELETE method, and creating a new resource by calling the POST method.

```
1 GET /customer/John%20Doe HTTP/1.1 http://www.example.org
```

Listing 5.8: Example REST request for GET

Therefor it is only necessary to send a request body for creating, updating, and deleting a resource. Listing 5.8 shows an example header for receiving a customer resource.
It also uses the HTTP status codes to inform the client that something has not worked or everything went successful. For example, if a resource is not found the server returns the status code 404 Not Found. If there is an internal server error, the server returns the status code 500 Internal Server Error. Sending this status codes instead of sending a complete body with an error message needs less bandwidth because the total message size gets smaller.

```
1  POST /customers/ HTTP/1.1
2  Content-Type: application/vnd.app.customer+xml
3  Accept: application/vnd.app.customer+xml
4  <customer xmlns:xsd="http://www.w3.org/2001/XMLSchema"
5  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6  xmlns="http://www.example.org">
7    <name>Jon Doe</name>
8    <address>1 Doe Street</address>
9    <city>Doe City</city>
10 </customer>
```

Listing 5.9: Request for Creating a New Customer

The POST request shown in Listing 5.9 shows a POST request message for saving a new customer. The server knows that the request contains a customer data through the content-type definition in the header. The content-type contains several parts which define the format of this request. The same takes effect in a response message. The part "application/vnd" specifies that the Internet media type is defined by its vendor specific; app is the name of the application, and "customer+xml" shows that it represents a customer resource based on the XML format. It is also possible to change the representation to JSON with exchanging the XML with JSON.

```
1  HTTP/1.1 201 CREATED
2  Location: http://www.example.org/customer/2
3  <customer xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5  xmlns="http://www.example.org">
6    <id>2</id>
7    <name>John Doe</name>
8    <address>1 Doe Street</address>
9    <city>Doe City</city>
10   <link rel="self" href="http://www.expample.org/customer/3" />
11   <link rel="previous" href="http://www.example.org/customer/1" />
12 </customer>
```

Listing 5.10: Response After Creating new Customer

For the response, shown in Listing 5.10, the server uses the 201 CREATED status instead of 200 OK status to indicate that the creation was processed successfully. It also contains a content-type definition identical to the one from the request message. The request message is different because the server sends a location with the exact URI for the resource and all the resource data, including an id and hypermedia links for the next and previous customer resource. The location tells where to find the new customer and the client can store this URI.

5.2 Summary

The ASP.NET Web API offers three different approaches to implement an API. These three approaches use all the HTTP protocol to communicate but have differences in their usage and their structure. The general HTTP API delivers general web content in the html format. For example this is used to load data asynchronous on a web site or build a paging bar which does not reload the whole web page. The SOAP approach in general is used to call remote methods on a server from multiple platforms and technologies. SOAP can only be used with the XML format for send and receive messages. The REST approach is used to address a specific data

part with a specific URI and get a response in the XML or JSON format.
The most Web APIs on the internet are using the REST approach for example the Google Geocode API and the Facebook Graph API. Additionally some of them like the Google Geocode API support the JSON and XML format as opposed to the Facebook Graph API which only supports JSON.

6 Generalization

Microsoft .NET offers three different options to generalize code. The first option .NET offers is the use of the object type, which can be used as a reference for every other data type. Listing 6.1 shows a code part which uses the object type for a variable and assign different other types to the variable. Every type, predefined or custom type, in .NET inherits directly or implicitly from the Object class. The lower case notation object defines an alias for the Object class. Any type can be referenced with the object type. This does not only work with variables but can be used in methods too to generalize their usage. A method can return any type if the return type is specified as the object type. Even parameters of a method can be defined as object type if there is the possibility to forward different types to the same method and it is not necessary to define for each type the same method which leads to duplicate code.

```
1 object c;
2 c = 1;
3 c = new CustomClass();
```

Listing 6.1: .NET object type

The Object type has one disadvantage. It is still needed to cast the object into a specific data type if public attributes and/or properties shall be accessed. The object type does not offer the special attributes and/or properties of a specific types which it is assigned to. It only offers its own methods and capabilities but none of the special type characteristics.

The second option to generalize code are so called Generics. Generics allow the definition of type-safe data structures, without binding them to actual data types. This leads to a high level of generalization and high quality code because the same code can be reused. Generics are more specific than the object type because it is possible to dynamically bind a specific type to a method or class. Listing 6.2 shows how a method generic method looks like. The specific type is set in angle brackets after the method name when the method is called.

```
1 public static T Load<T>()
2 {
3   return (T)DB.Load();
4 }
```

Listing 6.2: .NET generic method to retrieve a property

Generics are not only usable on methods but can also be used for generic classes. These generic classes can define methods which use the generic type of the class definition. Listing 6.3 shows an example of a generic class.

```
1 public class GenericClass<T>
2 {
3   public static T Load()
4   {
5     return (T)DB.Load();
6   }
7 }
```

Listing 6.3: Example of a generic class with a generic method in .NET

This generic class and the defined method can be used as shown in Listing 6.4.

```
1 CustomClass variable = GenericClass<CustomClass>.Load();
```

Listing 6.4: Code example for the use of a generic class

To be able to use generics it is necessary to know the specific type at the compile time, because it needs to be set with the method call or the usage of the class.

The third type of generalization are reflections. A reflection is called the process of type discovery during the runtime of a program. With the usage of the reflection service, it is possible to retrieve all types contained in a *.dll assembly file. This file can be loaded dynamically during the runtime of the program and do not have to be defined at the compile time. With reflections, it is also possible to call methods, get fields, properties, and events for a type including their parameters and other details (base classes, namespace information).

Table 6.1 shows some of the important members of the System.Reflection namespace.

Type	Meaning
Activator	Holds methods to create local or remote object types
Assembly	Contains static methods which allow to load, investigate, and manipulate an assembly
AssemblyName	Allows to discover several details from an assembly‘s identy
EventInfo	Holds information about an event
FieldInfo	Holds information about a field
MemberInfo	Abstract base class which defines common behaviours
MethodInfo	Contains information of a method
Module	Allows to access a module within a multifile assembly
ParameterInfo	Holds information about a parameter
PropertyInfo	Holds information about a property

Table 6.1: Members of the System.Reflection namespace

Using System.Reflection members opens up the possibility to dynamically call methods and retrieve objects of a specific type during the runtime without the necessity to bind a variable to a known specific type.

Figure 6.5 shows the code for instantiating a class which the code only knows through the class name as a string value and the assembly information which are configured in the app.config file respectively in the web.config file.

```
1 var myObject = Activator.CreateInstance("AssemblyName", "TypeName");
```

Listing 6.5: Create Object using .NET reflections

More possibilities with .NET reflections are the call of methods for such an object and the set of a property. Figure 6.6 shows how it is possible with reflections to call a method of an object with only knowing the type name and the method name as a string value.

```
1 Type type = <your object>.GetType();
2 MethodInfo method = thisType.GetMethod("MethodName");
3 method.Invoke(this, new object[]);
```

Listing 6.6: Call a method with parameters using .NET reflections

Figure 6.7 shows how to set a property of an object using reflections and knowing the property

name of the object.

```
1 Type type = target.GetType();
2 PropertyInfo prop = type.GetProperty("propertyName");
3 prop.SetValue (target, propertyValue, null);
```

Listing 6.7: Set a property of an object using .NET reflections

Reflections use the reference of the object type because all reflection methods which return specific types which are retrieved from a dynamically loaded assembly can be referenced with the object type and that makes it possible to use them with every specific type. Only methods which return a reflection type or the general type class with the definition of the specific type of an object in C# return them as their specific type.

The System.Reflection namespace already defines general classes which are representing the specific types of an object and it is possible to work with them without the necessity to define the specific type of an object in the implementation. .NET reflections define and load object types dynamically in the runtime of the program. This makes it possible to search dynamically in a third part assembly for a specific type during the runtime. The option to define the object type dynamically during the runtime enables the advanced possibility to change code of the assembly and recompile it without the necessity to change the code of the API where the specific type is used.

For the challenge of implementing a generalized API which can be used with different data structures and doesn't need an adaptation if the data structure changes over time, only the usage off the first and the second option of generalization in .NET are not useful. The API will not be able to know any specific type which are used with it. If the API is implemented using specific types of a specific business logic, the API is not fulfilling the challenge of being generalized and is able to work with different business logics as well.

The third option for generalization uses reflections in combination with the first option is the best way to retrieve the generalization which is needed to accomplish the challenge.

For this reason the design and implementation will be based up on using reflections to load business logic assemblies and determine specific types which are defined in the request during the runtime of the API.

7 Design

The design of the API shall be in a generalized way to make it possible to reuse the API with many different software business logics. For the case that the API shall work with the Qware Risk Manager, the Business Logic of the Risk Manager needs adaptations to work with the API. These adaptations for the prototypical implementation are described in the implementation chapter but are not covered by the design chapter. This design chapter only concentrates on the generalized design of the API. As the starting point of the design chapter we begin with the use case of the API.

7.1 Use Case

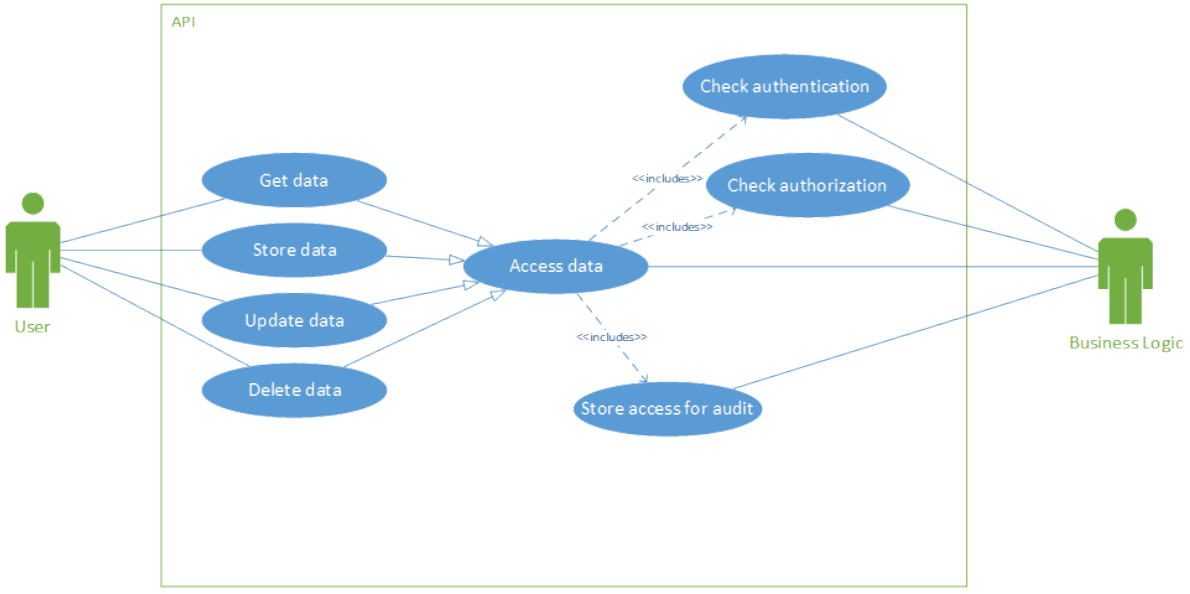

Figure 7.1: Use Case Diagram of the API

The general use case for the API is that a user is able to access a data base system through the API. This usage of the API includes requests for loading, storing, updating and deleting of

data using the business logic of a software. It is necessary for the API to check the configured permissions of the software through the business logic if the user can authenticate himself and the authenticated user is authorized to process the request through the API. The access to the data and the changes which are done by the request shall be stored in an audit system to be able to backtrack the change to the user who requested it.
7.1 shows the use case diagram for this short use case description.

7.2 Requirements

The use case diagram defines the following requirements for the API:

- Access data and be able to get, store, update and delete it
- Check the authentication of the user
- No processing of the data without the authorization for the user
- Store access information for an audit (CFR 21 Part 11)
- Use business logic for authentication and authorization check
- Use the business logic to process the data request
- Use the business logic to save the access information
- Generalized process for every data object

Additionally the challenges define the following requirements:

- Secure (encrypted) connection to the API
- The API has to have the same response time as the regular software
- URL Mapping design for request type
- The API has to run as a windows service

7.3 Request Activities

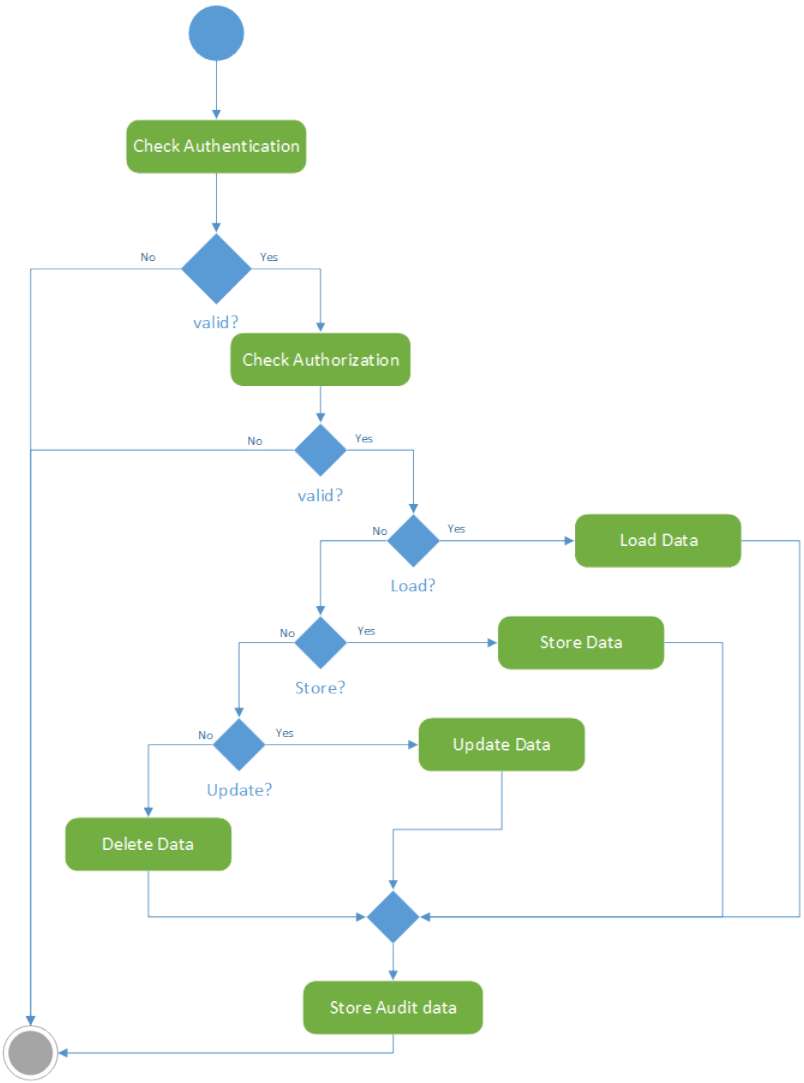

Figure 7.2: API Activity Diagram

The general activities of the API, which are done by processing a request, are first to check the authentication of the user who sends the request. The username and password have to be included into the header of the HTTP request. The API checks the username and password using the Business Logic, then the Business Logic provides a method to check the authentication. If the authentication fails the API is terminates. If the username and password is valid, the API performs the authorization check of the current user. The authorization check includes if the current user is allowed to process the request he sends. For the authorization check and the authentication Business Logic provides a method of authentication. After the

API checked the authorization of the user for the current request, it is the same process as for the authentication. If the user is not authorized to process the request, the API will terminate. These two checks are necessary to fulfill the security requirements of the FDA CFR 21 Part 11 to be still able to digitally sign documents conducted by the software. Additionally this security feature is necessary to retain the confidentiality of the processed and stored data of the database.

The next step is to decide if the current request is to load, store, update, or delete data. The difference between storing and updating data is, storing is the insert command and updating only updates an existing object from the database.

The last activity of the API is to store the audit trail data to be able to make a back trace of the changes. All these activities are processed with every request which is send to the API.

7.4 Request and Response Design

The general design of the API will follow the Restful design which was introduced in the chapter 5. With using the Restful design, the requests and responses of the API are designed to follow the XML or JSON language. Due to the reason that the object structure of the Risk Manager business logic has a very nested inheritance, this API will use the XML language. This means that requests for storing, updating, and responses for getting data are send in the XML format.

The surrounding node will have the name of the object type which shall be stored or updated. This object type root node contains child nodes for every necessary property for the object type. These child nodes are named after the name of the property and contain the value.

Listing 7.1 shows the general structure of an XML object which is sent in the message body from to the API.

```
1  <objectType>
2    <Property1>Content</Property1>
3    <Property2>Content</Property2>
4    <Property3>Content</Property3>
5    <Property4>Content</Property4>
6    <Property5>Content</Property5>
7    ...
8  </objectType>
```

Listing 7.1: XML format object design for request and response

The design also includes the JSON format. The challenge is that the API shall be designed in a generalized way, the API is required to support both formats. Listing 7.2 shows the message structure in the JSON format.

```
[
  "property":value,
  "property":value,
  "property":value,
  ...
]
```

Listing 7.2: JSON format object design for request and response

7.5 URL Mapping

To be able to cover all the functionality which is claimed in the challenges and requirements, the API needs a good URL Mapping to be able to process all the necessary requests.
The API is able to handle four types of HTTP methods, which are GET, POST, PUT and DELETE. The URL shown in Listing 7.3 is for handling the default POST method. POST messages contain all the necessary information about the object which shall be saved. The URL only needs to specify the object type.

```
https://hostname/{type}
```

Listing 7.3: POST URL

The second URL definition shown in Listing 7.4 is necessary for the HTTP GET and POST method. It defines the object type and a specific id of the object which has to be loaded by the API and returned to the client or updated.

```
https://hostname/{type}/{id}
```

Listing 7.4: GET, PUT, and DELETE URL

The URL definition of Listing 7.4 is also necessary for the HTTP DELETE method. To be able to delete an object, it is necessary for the API to know the object type and the specific id of the object which shall be deleted.
Another possibility opens up for the HTTP GET method if a client wants to call a specific method of a type. Therefor the URL shown in Listing 7.5 defines all the necessary information to call the method. The API gets the information about the type from which the method name

from the URL shall be called from the URL. The URL also includes three placeholders after the method name for defining up to three parameters for the method call. These three placeholders are defined as optional parameters and the number depends on the number of parameters a method needs to be called with. For example, if a method has no parameters, none of the optional parameter values are set in the URL. All of the methods have to be static methods because they will not be invoked on an instance of the type.

```
1 https://hostname/{type}/{id}/method/{methodname}/{param1}/{param2}/{param3}
```

Listing 7.5: Specific object method call for GET method

The API cannot just call methods on types but it can also read special properties. If a type has a special property, for example like a one to many list or many to many list, which is not serialized, a client is able to retrieve those properties. Listing 7.6 shows the URL for the retrieving of a property value.

```
1 https://hostname/{type}/{id}/property/{propertyname}
```

Listing 7.6: HTTP GET to retrieve property of specific object

For the list type properties it is in some cases also necessary to add or delete objects from the list. The case in which this is important is if list properties define a many to many relation which is defined through a relation table in the database, but not within the code. This case is covered by the URL mapping in Listing 7.7.

```
1 https://hostname/{type}/{id}/property/{propertyname}/{objectid}
```

Listing 7.7: HTTP POST and DELETE for a list property

These five URL mappings cover all the necessary parameters to process the four HTTP request method types and are able to get data, store data, update data, and delete data. Additionally these URL mappings cover the call of a method type and can retrieve special properties from an instance of an object. These URL mappings fulfill the requirement for the API to be completely able to process the same operations for working with the data, as the Qware Risk Manager Software can do. In addition, these URL mappings need to handle all possible processes with an object and object instance related to the generalized API challenge.

7.6 Security

7.6.1 Authentication

The authentication process is done by the API using the default user type of the business logic. The user type needs to define a method for checking if a user exists. This method gets as parameters the username and the password which is stored in the HTTP header.

7.6.2 Authorization

The authorization handling is done by the business logic itself. The API only calls the requested method wants to perform and the business logic checks if the current user is authorized to perform the request.

7.6.3 Message Security

To provide complete security for the whole process of handling data, it is necessary to use more than just an authentication and authorization check. The API uses the HTTP protocol for messages to communicate with a client. Due to this reason the usage of the HTTP protocol, the API needs to secure the communication with the client too. This is necessary to complete the security features for obtaining the confidentiality of the data which is one major challenge. The authentication including the username and password are send in the HTTP request header which are not encrypted and transmitted in plain text.

If the communication is not secured, the API would be vulnerable for man-in-the-middle and eavesdropping attacks. A man-in-the-middle attack is an attack, when a person catches the request send by the client and modifies the message before he forwards the message to the recipient. Eavesdropping is the attack where third parties read a private conversation without the consent of the sender or recipient. For both attacks it is easy to read the username and password, send with every request, because both of them are not encrypted and written in an encoded format in the HTTP header which can be easily resolved to produce them in plain text.

To prevent this scenario, the HTTP protocol has the possibility to secure the message exchange. HTTPS is designed to secure the communication messages between two parties. It offers two

important aspects. The first is that HTTPS ensures that a client communicates with the service it intends to communicate with and the service does not masquerade itself as a service the client trusts but it not wants to communicate with. The second important aspect is that HTTPS completely encrypts the entire message which are exchanged by a client and a service. This complete encryption includes the request URI (URL), the headers, and the message body. With the encryption of the complete message, it is not possible that a man-in-the-middle or eavesdropping attack can happen.

7.7 Request Sequence

Derived from the request activities, the API uses four different designs of sequences to process the four different HTTP request methods, which can be used with the API. Figure 7.3 shows the sequence diagram for the HTTP GET method if a client sends a request to the URL https://hostname/type/id.

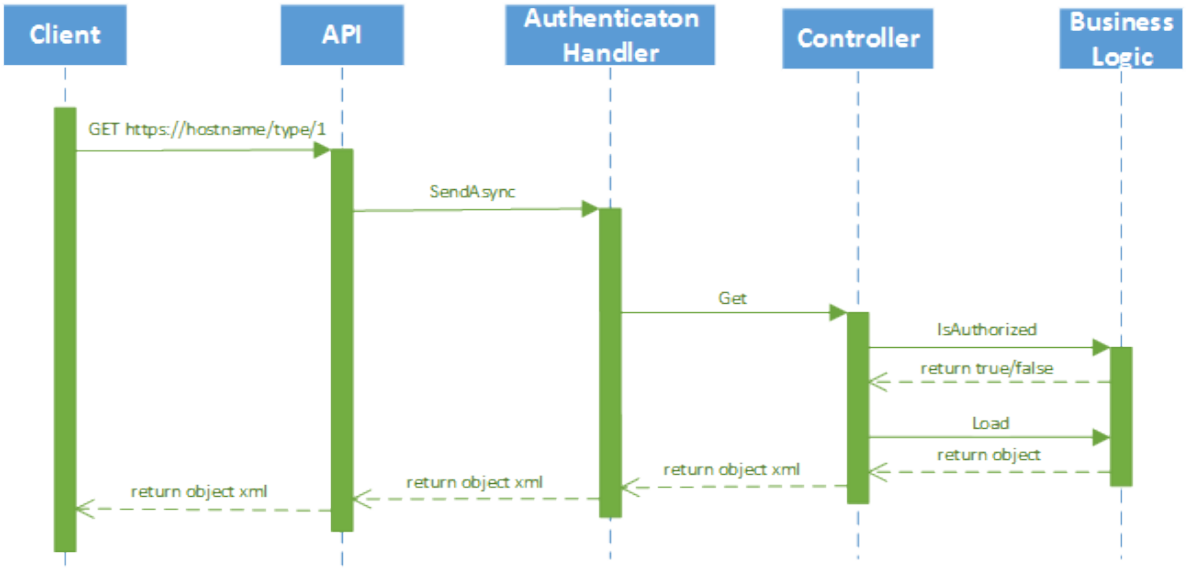

Figure 7.3: Sequence Diagram for API Process of the HTTP GET method

Figure 7.4 shows the sequence diagram for the HTTP POST method which stores a new object of a type into the database.

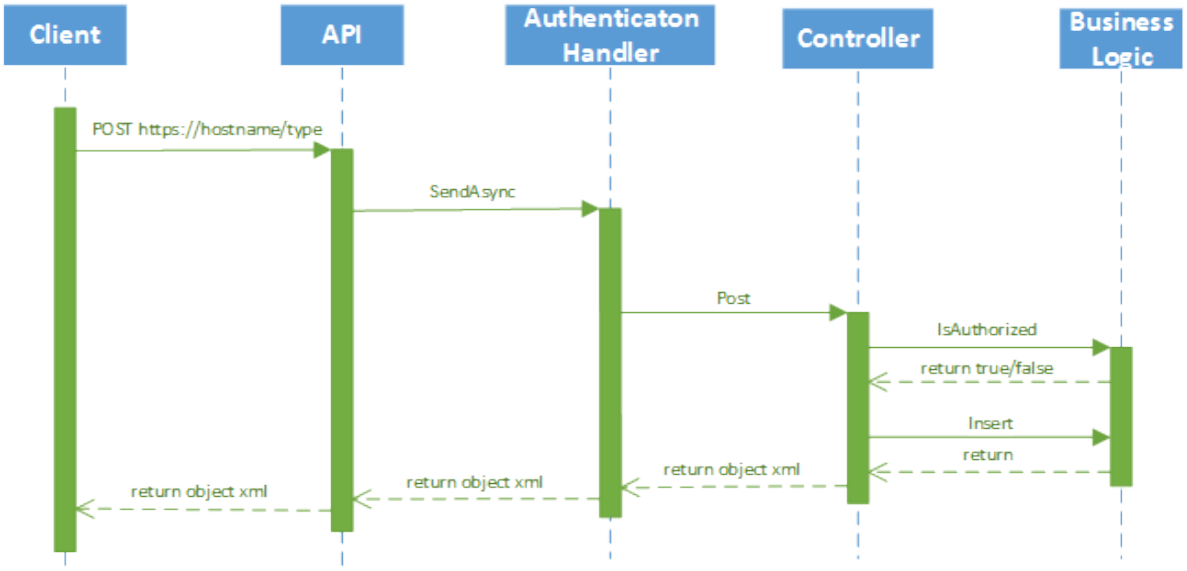

Figure 7.4: Sequence Diagram for API Process of the HTTP POST method

Figure 7.5 shows the sequence diagram of the HTTP PUT method which updates an existing object.

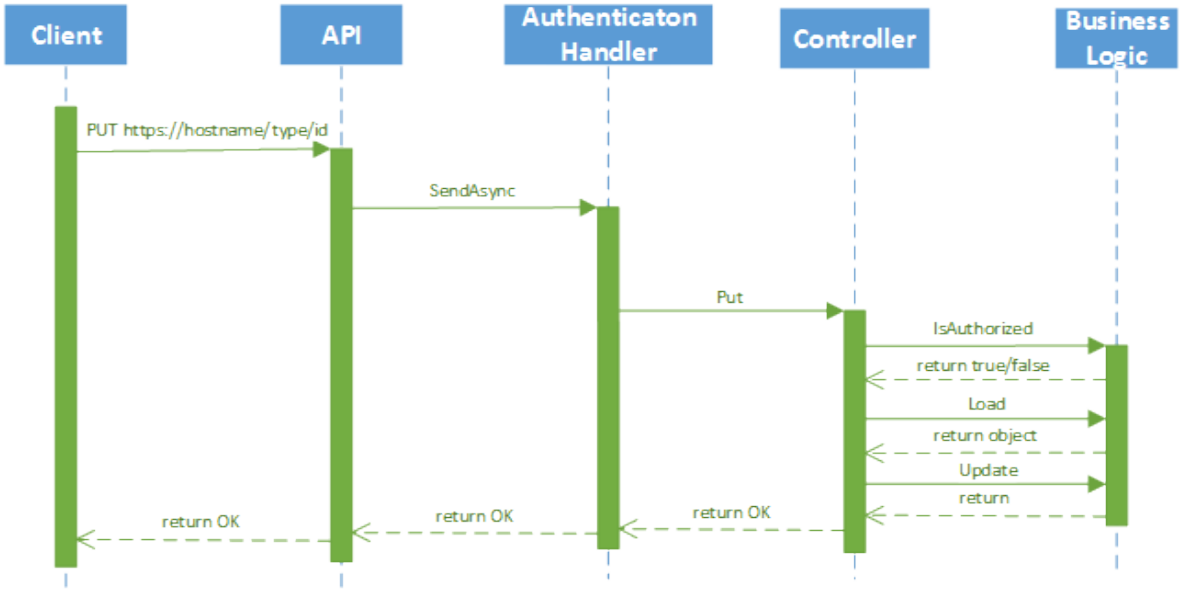

Figure 7.5: Sequence Diagram for API Process of the HTTP PUT method

Figure 7.6 shows the sequence diagram for the HTTP DELETE method to delete an object of a specific type with an id.

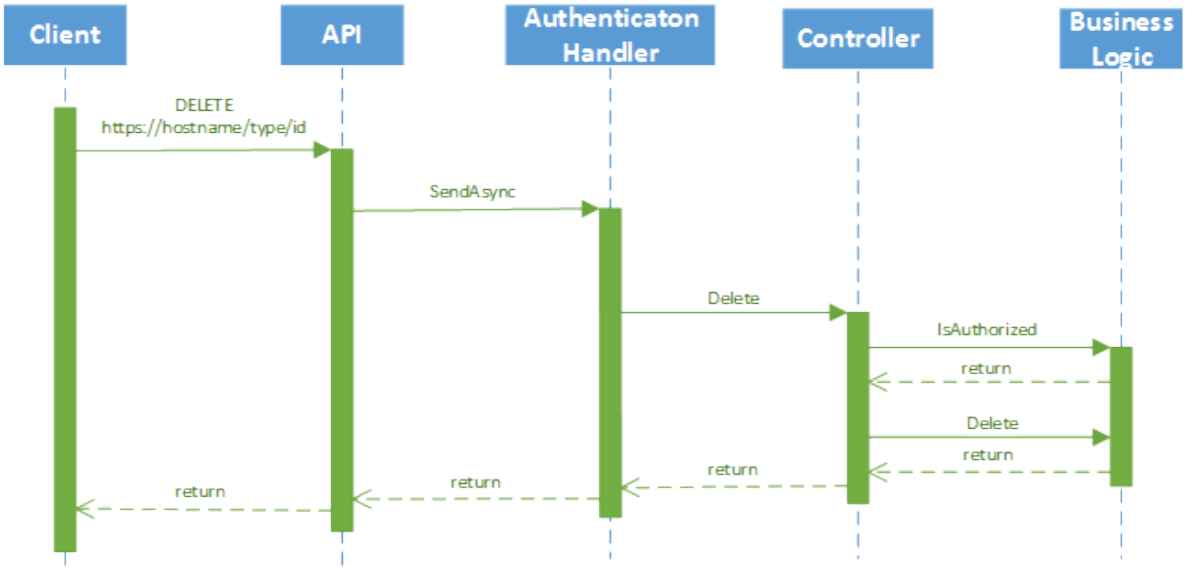

Figure 7.6: Sequence Diagram for API Process of the HTTP DELETE method

In general, the four sequence diagrams are not different in the process of authentication and authorization. The only differences are the different methods in the controller and which are called in the business logic to process the specific type of operation. These four sequence diagrams for the HTTP GET, POST, PUT, and DELETE methods show the sequence of processing data and need to be implemented to fulfill the requirements.

8 Implementation

The prototypical implementation of the API is done with the business logic of the Qware Risk Manager. This chapter goes through the process of handling data and describes the single steps of a request processed by the code. The adaptations, which are necessary for the business logic of the Risk Manager are described after that. The prototype is implemented in two different versions. One version can be run as a web application on an IIS web server and the other version can be run as a windows service with a self-hosting web server.
To be able to use the API with a business logic, it is necessary that the business logic provides a user class with a method to check the username and password, the load, insert, update, and delete methods, the "IsAuthorized" methods to check the current users permission to process the request, and if the business logic provides the possibility of an audit trail corresponding to the CFR 21 Part 11, a method to initialize it.
The API is implemented using ASP.NET 4.5 and uses the ASP.NET MVC technology for configuration of the routes and bind them to a specific controller. The starting point of this chapter is the authentication, which is the first step on processing a request. After this, the chapter continues with controllers which process the request and return the data to the client. That every controller can work properly needs a configuration of the API which includes the registration and initialization of the audit trail. After these sections this chapter describes the different code which is necessary to run the API as a Windows Service. The last section of this chapter is the adaptations which were done to the Risk Manager business logic, to enable it to function with the API.

8.1 Authentication

The starting point of a request sent to the API is the authentication, which is done by the APIAuthenticationHandler. The authentication check is the first processing of a request. This

handler checks if the request, which is sent to the API, contains the authentication header value and if the scheme of the authentication header is "basic". If both of them are included, the code extracts the parameter of the authentication header value and encodes it into "ISO-8859-1". The resulting string is the credentials from the header value and containing the username and the password which are separated with a colon. The string is separated by the code into two strings with the split command and the colon as the indicator where the method shall split the string. After this there are two strings of which one contains the username and the other one the password.
The code then invokes the method to retrieve a user object with the username and password as the parameters from the user class type defined in the business logic. The user class name and the get user method are defined in the application configuration. If the username and password are correct and the get user method returns the user object. The user object is saved in a new AuthenticationIdentity object. This object inherits from the GenericIdentity class. The authentication identity object is then assigned to current user property of the current thread and if the current http context is available, to the current http context user.
These assignments are necessary to be able to retrieve the current user object and check the authorization using the current user object in the later process.

8.2 Controller

As described in section 8.3.1, the API defines three different controllers to handle requests. These three controllers are the default controller, the method controller, and the property controller. All of these controllers use .NET Reflections to work with the objects and types of the configured business logic.

8.2.1 Default Controller

The default controller is the standard controller for processing the requests of the HTTP GET, POST, PUT, and DELETE method. All these methods are defined in the DefaultController class.

GET Method

The GET method of the DefaultController class is used to retrieve objects with a specific id and type. To be able to call this method the URL needs to contain the type name and the id of the specific object.

The code gets the Type class object of the type name string with the APIHelper GetTypeWithString method. The Type class object is used to find the load method which is defined in the configuration parameters and invoke this method with the specific id to get the object. After the object is loaded, the authorization is checked. This is done by the CheckAuthorization method of the AuthorizationHelper class. This method calls the defined authorization method of the configuration parameters on the object instance with the current user object, the HTTP method type as a string and the id as parameters. Details about the authorization method is described in section 8.6.2.

If the authorization returns the value true, the object instance is serialized and a response message with the HTTP status code "OK" is created. This is done in the SerializeObjectAndCreateOkResponse method of the APIHelper. The serialization is done to the XML format or the JSON format. This depends on the configured format.

Both serializer need the Type class object of the object instance. This is needed because the business logic assembly is loaded dynamically during the runtime and is not known during the compile time, also the serializer does not know the type of the object instance. Because of this it is not possible to use the general formatter of the HTTP configuration which is automatically used by the GenerateResponse method of the request object. It was necessary to use the JSON and XML serializer directly and first serialize an object to a string and create the response message afterwards.

Listing 8.1 shows the APIHelper method for the serialize object and create response message. Listing 8.2 and 8.3 show the code of serializing an object into the XML and JSON format.

```
/// <summary>
/// Serializes the object.
/// </summary>
/// <param name="item">The item.</param>
/// <returns></returns>
public static HttpResponseMessage SerializeObjectAndCreateOkResponse(object item, Type type, HttpRequestMessage request)
{
  HttpResponseMessage response = request.CreateResponse(HttpStatusCode.OK);
  if (ConfigurationManager.AppSettings["SerializationType"].Equals("XML", StringComparison.InvariantCultureIgnoreCase))
    {
      response.Content = new StringContent(XmlHelper.SerializeObjectToXml(item, type), Encoding.UTF8, "application/xml");
    }
    else
    {
      response.Content = new StringContent(JSONHelper.SerializeObjectToJSON(item, type), Encoding.UTF8, "application/json");
    }
    return response;
}
```

Listing 8.1: APIHelper code to serialize object and create HTTP Ok response message

```
/// <summary>
/// Serializes the object in XML.
/// </summary>
/// <param name="objectInstance">The object instance.</param>
/// <param name="type">The type.</param>
/// <returns></returns>
public static string SerializeObjectToXml(Object objectInstance, Type type)
{
  StringWriter writer = new StringWriter();
  XmlWriter xmlWriter = XmlWriter.Create(writer);
  XmlSerializer serializer = new XmlSerializer(type);
  serializer.Serialize(xmlWriter, objectInstance);
  return writer.ToString();
}
```

Listing 8.2: XMLHelper code to serialize object into the XML format

```
/// <summary>
/// Serializes the object to json.
/// </summary>
/// <param name="objectInstance">The object instance.</param>
/// <param name="type">The type.</param>
/// <returns></returns>
public static string SerializeObjectToJSON(Object objectInstance, Type type)
{
  MemoryStream stream = new MemoryStream();
  DataContractJsonSerializer serializer = new DataContractJsonSerializer(type);
  serializer.WriteObject(stream, objectInstance);
  StreamReader reader = new StreamReader(stream);
  return reader.ReadToEnd();
}
```

Listing 8.3: JSONHelper code to serialize object into the JSON format

The client gets the HTTP message with the status code "OK" and the serialized content of the object returned.

If the user is not authorized, a response message with the status code "Forbidden" is created and returned to the client.

The complete code of the method is surrounded by a try catch block. This block catches and handles exceptions which can occur in the code. If an exception occurs, the code creates a response message with the internal server error status code. The exception is logged into a log file.

Listing 8.4 shows the code of the Get method. The Get method retrieves an object specified by the type name and the object id. It includes checks for authorization, error logging, and serializes the object to send it as a response.

```
1  /// <summary>
2  /// Gets the specified type.
3  /// </summary>
4  /// <param name="type">The type.</param>
5  /// <param name="id">The identifier.</param>
6  /// <returns></returns>
7  public HttpResponseMessage Get(string type, int id)
8  {
9    try
10   {
11     Type objectType = APIHelper.GetTypeWithString(type);
12     object objectInstance = APIHelper.LoadObject(id, type);
13     if (AuthorizationHelper.CheckAuthorization(objectType, objectInstance, "GET"))
14     {
15       return APIHelper.SerializeObjectAndCreateOkResponse(objectInstance, objectType, Request);
16     }
17     else
18     {
19       return APIHelper.GetNotAuthorizedResponseMessage(Request);
20     }
21   }
22   catch (Exception e)
23   {
24     Log.Error("An internal error occurred", e);
25     return APIHelper.GetInternalErrorResponseMessage(Request);
26   }
27 }
```

Listing 8.4: DefaultController GET method code

POST Method

The Post method of the DefaultController class is able to insert a new object into the database. The data of the new object is sent in the XML or JSON format depending on the configuration of the API. The code first reads the body of the request message as a string value and retrieves the type object of the type name string.

After that the new object instance of the type is filled with the data and the special authorization value is read from the string body value. This depends on which format is configured. For the XML format the code runs the FillObjectFromXml method of the XMLHelper class.

This method creates an XmlDocument object from the body string and a new instance from the

type object. Then it runs through the XML document and stores the XML nodes value to the corresponding property. This node and property matching is done with the XML node name and the property name. The XML node needs to have the same name as the property. The value of the node is directly converted into the specific type of the property. The conversion is possible for every standard type in C#, byte arrays, and nullable types. The id name is ignored because it is not necessary. An updated object already contains an id and a new object must not have one. The return value is the object instance with the filled properties. If the body string is empty or the main XML node is not named as the type name, the method throws an argument exception respectively an argument null exception.

```
1  /// <summary>
2  /// Fills the object from XML.
3  /// </summary>
4  /// <param name="xml">The XML.</param>
5  /// <param name="type">The type.</param>
6  /// <returns></returns>
7  /// <exception cref="System.ArgumentException">Type not matching with XML node.</exception>
8  /// <exception cref="System.ArgumentNullException">Request body not containing necessary xml object.</exception>
9  public static object FillObjectFromXml(String xmlBody, Type type)
10 {
11   XmlDocument xml = new XmlDocument();
12   xml.LoadXml(xmlBody);
13   object objectInstance = Activator.CreateInstance(type);
14   if (xml.ChildNodes.Count > 0)
15   {
16     XmlNode node = xml.FirstChild;
17     if (node.Name.Equals(type.Name, StringComparison.InvariantCultureIgnoreCase))
18     {
19       if (node.HasChildNodes)
20       {
21         foreach (XmlNode childNode in node.ChildNodes)
22         {
23           if (!String.IsNullOrEmpty(childNode.InnerText) && !childNode.Name.Equals("id", StringComparison.InvariantCultureIgnoreCase))
24           {
25             PropertyInfo property = type.GetProperty(childNode.Name);
26             if (property != null)
27             {
28               if (property.PropertyType == typeof(byte[]))
29               {
30                 property.SetValue(objectInstance, APIHelper.GetBytes(childNode.InnerText));
31               }
32               else
33               {
34                 if (property.PropertyType.IsGenericType && property.PropertyType.GetGenericTypeDefinition() == typeof(Nullable<>))
35                 {
36                   property.SetValue(objectInstance, Convert.ChangeType(childNode.InnerText,
                           Nullable.GetUnderlyingType(property.PropertyType)));
37                 }
38                 else
39                 {
40                 property.SetValue(objectInstance, Convert.ChangeType(childNode.InnerText, property.PropertyType));
41                 }
42               }
43             }
44             else
45             {
46               throw new ArgumentException("Property not found.");
47             }
48           }
49         }
50       }
51     }
52     else
53     {
54       throw new ArgumentException("Type not matching with XML node.");
55     }
56   }
57   else
58   {
59     throw new ArgumentNullException("Request body not containing necessary xml object.");
60   }
61   return objectInstance;
62 }
```

Listing 8.5: XmlHelper class method to fill object from xml

After the new object instance is filled, the code retrieves the custom XML value for authorization. For this the code calls the method GetXmlValueForAuthorization of the XmlHelper class. Therefore the code iterates through the XML nodes and check if the node name equals

with the configured value for the "CustomValueForAuthorization". If the value is found, the iteration stops and the string value is returned. Listing 8.6 shows the code of this method.

```
/// <summary>
/// Gets the XML value for authorization.
/// </summary>
/// <param name="xml">The XML.</param>
/// <returns></returns>
public static string GetXmlValueForAuthorization(String xmlBody)
{
  XmlDocument xml = new XmlDocument();
  xml.LoadXml(xmlBody);
  string xmlCustomValue = "";
  if (xml.ChildNodes.Count > 0)
  {
    XmlNode node = xml.FirstChild;
    foreach (XmlNode childNode in node.ChildNodes)
    {
      if (!String.IsNullOrEmpty(childNode.InnerText))
      {
        if (childNode.Name.Equals(ConfigurationManager.AppSettings["CustomValueForAuthorization"]))
        {
          xmlCustomValue = childNode.InnerText;
          break;
        }
      }
    }
  }
  return xmlCustomValue;
}
```

Listing 8.6: XmlHelper code for retrieving the custom value for authorization

If the configured format is JSON instead of XML, the code executes the FillObjectFromJSON method of the JSONHelper class. Within this method it is the same procedure as for the method for XML. The body string is parsed into a JSON object which contains children of the type KeyValuePair. The code iterates through the children and stores the value in the property specified in the key string. The values are directly converted into the property type. The conversion is also possible for all standard types in C#, byte arrays, and generic nullable types. The id key is ignored because it is not necessary. An updated object already contains an id and a new saved object must not have one.

The return value is the filled object instance. Listing 8.7 shows the code of the FillObject-FromJSON method.

```
1  /// <summary>
2  /// Fills the object from json.
3  /// </summary>
4  /// <param name="jsonBody">The json body.</param>
5  /// <param name="type">The type.</param>
6  /// <returns></returns>
7  /// <exception cref="System.ArgumentNullException">Request body not containing necessary json objects.</exception>
8  public static object FillObjectFromJSON(String jsonBody, Type type)
9  {
10   object objectInstance = Activator.CreateInstance(type);
11   JObject json = JObject.Parse(jsonBody);
12   if (json.Count > 0)
13   {
14     foreach (KeyValuePair<string, JToken> childNode in json)
15     {
16       if (!String.IsNullOrEmpty(childNode.Value.ToString()) && !childNode.Key.Equals("id",
             StringComparison.InvariantCultureIgnoreCase))
17       {
18         PropertyInfo property = type.GetProperty(childNode.Key);
19         if (property != null)
20         {
21           if (property.PropertyType == typeof(byte[]))
22           {
23             property.SetValue(objectInstance, APIHelper.GetBytes(childNode.Value.ToString()));
24           }
25           else
26           {
27             if (property.PropertyType.IsGenericType && property.PropertyType.GetGenericTypeDefinition() == typeof(Nullable<>))
28             {
29               property.SetValue(objectInstance, Convert.ChangeType(childNode.Value.ToString(),
                     Nullable.GetUnderlyingType(property.PropertyType)));
30             }
31             else
32             {
33               property.SetValue(objectInstance, Convert.ChangeType(childNode.Value.ToString(), property.PropertyType));
34             }
35           }
36         }
37         else
38         {
39           throw new ArgumentException("Property not found.");
40         }
41       }
42     }
43   }
44   else
45   {
46     throw new ArgumentNullException("Request body not containing necessary json objects.");
47   }
48   return objectInstance;
49 }
```

Listing 8.7: JSONHelper class method to fill an object instance from JSON

After the filling of the object instance, the code gets the custom authorization value from the JSON string. Therefore the code calls the method GetJSONValueForAuthorization from the JSONHelper class. The method parses the JSON string into a JObject. Then the code iterates through the KeyValuePair children and stops if the key matches the configured "CustomValueForAuthorization" parameter and returns the value of the node.

```
/// <summary>
/// Gets the json value for authorization.
/// </summary>
/// <param name="jsonBody">The json body.</param>
/// <returns></returns>
public static string GetJSONValueForAuthorization(String jsonBody)
{
  string xmlCustomValue = "";
  JObject json = JObject.Parse(jsonBody);
  if (json.Count > 0)
  {
    foreach (KeyValuePair<string, JToken> childNode in json)
    {
      if (childNode.Key.Equals(ConfigurationManager.AppSettings["CustomValueForAuthorization"]))
      {
        xmlCustomValue = childNode.Value.ToString();
        break;
      }
    }
  }
  return xmlCustomValue;
}
```

Listing 8.8: JSONHelper code for retrieving the custom value for authorization

The implementation of both methods to be able to get the values from XML and JSON are important for the challenge to implement a generalized API. This opens the possibility to be not compelled to use the XML format but to also use the JSON format.

After the object instance is filled with the body content and the custom authorization value is extracted, the code checks the authorization of the user to do an insert of the new object. The code calls the CheckAuthorization method of the AuthorizationHelper class with the object instance, the object type, the value for authorization, and the string "POST" as parameters.

If the user is authorized, the code gets the method info of the configured insert method. On the other hand the code returns a forbidden response message. If the method info is found the code calls the InvokeMethodOnObject method of the APIHelper class with the object instance, method info, and empty list of types as parameters. The InvokeMethodOnObject method checks if the method, which shall be called is defined, as a public method. If the method is not public, the method throws an exception. The code also throws an exception if the insert method is not found for the type.

After the object is inserted, the object instance is serialized and sent as the response message body. The method of serialization is the same as described in the previous chapter. The complete code of the Post method is surrounded by a try catch block like the Get method. If an exception occurs, the client receives an internal error response message and the exception is logged in the API log file.

Listing 8.9 shows the code of the Post method. As a conclusion, the Post method handles request for inserting new objects into the database. Therefore it parses the XML or JSON format into the code object and inserts it in the database. If any exceptions occur or the insert method is not found, the code throws an exception and returns an internal server error response. If the user is not authorized, a not authorized message is returned.

```
1  /// <summary>
2  /// Posts the specified type.
3  /// </summary>
4  /// <param name="type">The type.</param>
5  /// <returns></returns>
6  public async Task<HttpResponseMessage> Post(string type)
7  {
8    try
9    {
10     string body = await Request.Content.ReadAsStringAsync();
11     Type objectType = APIHelper.GetTypeWithString(type);
12     object objectInstance = null;
13     string valueForAuthorization = "";
14     if (ConfigurationManager.AppSettings["SerializationType"].Equals("XML", StringComparison.InvariantCultureIgnoreCase))
15     {
16       objectInstance = XmlHelper.FillObjectFromXml(body, objectType);
17       valueForAuthorization = XmlHelper.GetXmlValueForAuthorization(body);
18     }
19     else
20     {
21       objectInstance = JSONHelper.FillObjectFromJSON(body, objectType);
22       valueForAuthorization = JSONHelper.GetJSONValueForAuthorization(body);
23     }
24     if (AuthorizationHelper.CheckAuthorization(objectType, objectInstance, valueForAuthorization, "POST"))
25     {
26       MethodInfo method = objectType.GetMethod(ConfigurationManager.AppSettings["InsertMethod"]);
27       if (method != null)
28       {
29         APIHelper.InvokeMethodOnObject(method, objectInstance, new List<object>());
30         return APIHelper.SerializeObjectAndCreateCreatedResponse(objectInstance, objectType, Request);
31       }
32       else
33       {
34         throw new Exception("Insert method not found");
35       }
36     }
37     else
38     {
39       return APIHelper.GetNotAuthorizedResponseMessage(Request);
40     }
41   }
42   catch(Exception e)
43   {
44     Log.Error("An internal error occurred", e);
45     return APIHelper.GetInternalErrorResponseMessage(Request);
46   }
47 }
```

Listing 8.9: DefaultController POST method code

PUT Method

The Put method of the default controller is able to update properties of an object of the database. The client has to send the type string of the object and the specific id. The Put

method works and does the same as the Post method described in the previous chapter. There are only two differences between the Put and the Post method logic. The first difference is, that the Put method loads the object of the type with the specific id and updates the properties with the new values instead of creating a complete new instance. The second difference is that the Put method gets the method info object of the update method with the update method name from the configuration parameters.

Listing 8.10 shows the code of the Put method.

```
1  /// <summary>
2  /// Puts the specified type.
3  /// </summary>
4  /// <param name="type">The type.</param>
5  /// <param name="id">The identifier.</param>
6  /// <returns></returns>
7  public async Task<HttpResponseMessage> Put(string type, int id)
8  {
9    try
10   {
11     string body = await Request.Content.ReadAsStringAsync();
12     Type objectType = APIHelper.GetTypeWithString(type);
13     object dbObject = APIHelper.LoadObject(id, type);
14     object objectInstance = null;
15     if (ConfigurationManager.AppSettings["SerializationType"].Equals("XML", StringComparison.InvariantCultureIgnoreCase))
16     {
17       objectInstance = XmlHelper.FillObjectFromXml(body, dbObject, objectType);
18     }
19     else
20     {
21       objectInstance = JSONHelper.FillObjectFromJSON(body, dbObject, objectType);
22     }
23     if (AuthorizationHelper.CheckAuthorization(objectType, objectInstance, "PUT"))
24     {
25       MethodInfo method = objectType.GetMethod(ConfigurationManager.AppSettings["UpdateMethode"]);
26       if (method != null)
27       {
28         APIHelper.InvokeMethodOnObject(method, objectInstance, new List<object>());
29         return APIHelper.GetOkResponseMessage(Request, "");
30       }
31       else
32       {
33         throw new Exception("Update method not found");
34       }
35     }
36     else
37     {
38       return APIHelper.GetNotAuthorizedResponseMessage(Request);
39     }
40   }
41   catch (Exception e)
42   {
43     Log.Error("An internal error occurred", e);
44     return APIHelper.GetInternalErrorResponseMessage(Request, e);
45   }
46 }
```

Listing 8.10: DefaultController PUT method code

DELETE Method

The Delete method of the default controller class handles HTTP DELETE requests which contain the type name string and an id of the specific object which shall be deleted. The method code loads the object with the id with calling the LoadObject method of the APIHelper class. After this, the code calls the CheckAuthorization method of the AuthorizationHelper class to check the authorization of the current user to delete the current object.
If the authorization is true, the code prepares the parameter list for the configured delete method of the object. The delete method has the integer id value as parameter. Then it is searched for the method information object with the configured delete method name and the parameter array as parameters of the GetMethod method of the type object. If the method information are found the method is invoked with the parameter array containing the id. The result of the method is, that a response message with the HTTP OK status is returned.
The delete method also contains error handling if the type name cannot be found in the business logic or the current user is not authorized to delete the current object. The code is additionally surrounded with a try catch block to catch all exceptions which can occur during the runtime of the code.
With this code it is possible to send delete requests for specific objects of a type. Listing 8.11 shows the code of the delete method of the default controller class.

```
/// <summary>
/// Deletes the specified type.
/// </summary>
/// <param name="type">The type.</param>
/// <param name="id">The identifier.</param>
public HttpResponseMessage Delete(string type, int id)
{
  try
  {
    Type objectType = APIHelper.GetTypeWithString(type);
    object deleteObject = APIHelper.LoadObject(id, type);
    if (objectType != null)
    {
      if (AuthorizationHelper.CheckAuthorization(objectType, deleteObject, "DELETE"))
      {
        // set method paramter types
        List<Type> methodTypeList = new List<Type>();
        methodTypeList.Add(id.GetType());
        MethodInfo method = objectType.GetMethod(ConfigurationManager.AppSettings["DeleteMethod"], methodTypeList.ToArray());
        if (method != null)
        {
          List<object> methodParamList = new List<object>();
          methodParamList.Add(id);
          method.Invoke(deleteObject, methodParamList.ToArray());
          return APIHelper.GetOkResponseMessage(Request);
        }
        else
        {
          throw new Exception("Delete method not found!");
        }
      }
      else
      {
        return APIHelper.GetNotAuthorizedResponseMessage(Request);
      }
    }
    else
    {
      return APIHelper.GetBadRequestResponseMessage(Request, "Type not found!");
    }
  }
  catch (Exception e)
  {
    Log.Error("An internal error occurred", e);
    return APIHelper.GetInternalErrorResponseMessage(Request);
  }
}
```

Listing 8.11: DefaultController DELETE method code

8.2.2 Method Controller

The method controller handles all the HTTP GET requests for the registered route URL. This URL contains the type name, the object id, the static string method, the method name, and three optional parameter string values. To be able to handle all the four different variations of the URL, the method controller class has four different methods to handle the requests. The difference between the four methods is, that one can handle an URL with three parameter string values, one with two parameter string values, one with one parameter string value, and one which does not have any parameter string values from the URL.

These methods make it possible to call a specific method on a specific object including at most three method parameters. It is not possible to have complex objects as a parameter.
The code of these methods loads the specific object with the integer id value and loads the type information object from the business logic. After this all methods check the authorization of the current user to access the object which includes access to the method of the object.
If the authorization is granted, the difference between the four methods appears. The three methods which can handle the different number of parameters for the method are now setting up the parameter list for the method. The method without parameters for the invoked method sets up an empty list. To invoke the method on the object, the method InvokeMethodOnObject of the APIHelper class is called with the method information, the object instance, and the parameter list as parameters. The returned result object is then serialized and included into a HTTP Ok status response message. This serialization and message is created with the SerializeObjectAndCreateOkResponse of the APIHelper class. The code of this method is already described in section 8.2.1.
The method controller get methods also contain error handling as all other methods described in the previous chapter include it. Listing 8.12 shows the code of the Get method for an URL which contains three parameter values. The difference to the code of the other Get methods of the method controller is described above.
As a conclusion the method controller get methods can handle get requests which match the method controller route configuration described in section 8.3.1. They are able to handle methods which have up to three parameters. These processing includes error handling to get sufficient responses to the client and log any occurred exception into the API log file.

```
1  /// <summary>
2  /// Gets the specified type.
3  /// </summary>
4  /// <param name="type">The type.</param>
5  /// <param name="method">The method.</param>
6  /// <param name="param1">The param1.</param>
7  /// <param name="param2">The param2.</param>
8  /// <param name="param3">The param3.</param>
9  /// <returns></returns>
10 public HttpResponseMessage Get(string type, int id, string method, string param1, string param2, string param3)
11 {
12   try
13   {
14     object objectInstance = APIHelper.LoadObject(id, type);
15     Type objectType = APIHelper.GetTypeWithString(type);
16     if (AuthorizationHelper.CheckAuthorization(objectType, objectInstance, "GET"))
17     {
18       MethodInfo methodInfo = objectType.GetMethod(method);
19       if (methodInfo == null)
20       {
21         throw new Exception("Method not found!");
22       }
23       List<object> paramList = new List<object>();
24       paramList.Add(param1);
25       paramList.Add(param2);
26       paramList.Add(param3);
27       object result = APIHelper.InvokeMethodOnObject(methodInfo, objectInstance, paramList);
28       return APIHelper.SerializeObjectAndCreateOkResponse(result, result.GetType(), Request);
29     }
30     else
31     {
32       return APIHelper.GetNotAuthorizedResponseMessage(Request);
33     }
34   }
35   catch (Exception e)
36   {
37     Log.Error("An internal error occurred", e);
38     return APIHelper.GetInternalErrorResponseMessage(Request);
39   }
40 }
```

Listing 8.12: MethodControler Get method code to execute an objects method and retrieve the return value

8.2.3 Property Controller

The property controller class handles HTTP GET, POST, and DELETE method requests to retrieve a specific property of an object. List properties are not serialized with the general get request described in section 8.4 because the serialization of a list property can cause a serialization loop. In general, all of these properties can be every type of object and are not limited to list types.

GET Method

The request URL for the HTTP GET method contains the string name of the type, the id of the specific object and the string property name of the property which shall be loaded and

returned. The code first loads the Type object of the type name string and gets the property info with the property name string from the type object. After this the specific object with the provided id is loaded. The code checks if the current user is authorized to access the returned object. Then the property is loaded from the object. To send the property object as a response message, the property object is now serialized and a HTTP response message is created using the string value of the serialized object.

The property controller get method includes an error handling to return useful messages and status codes to the client as all other controllers and there methods. Exceptions are also logged into the API log file.

The general usage of the property controller is to get properties of a specific object which are not serialized by the get method to receive the specific object. This is necessary to prevent loops in the serialization process. Therefore the property controller is needed to read the properties, which are not serialized with the object.

Listing 8.13 shows the code of the Get method of the property controller class.

```
/// <summary>
/// Gets the specified type.
/// </summary>
/// <param name="type">The type.</param>
/// <param name="id">The identifier.</param>
/// <param name="property">The property.</param>
/// <returns></returns>
public HttpResponseMessage Get(string type, int id, string property)
{
  try
  {
    // load object type
    Type objectType = APIHelper.GetTypeWithString(type);
    if (objectType != null)
    {
      // property info
      PropertyInfo propertyInfo = objectType.GetProperty(property);
      if (propertyInfo != null)
      {
        object objectValue = APIHelper.LoadObject(id, type);
        if (AuthorizationHelper.CheckAuthorization(objectType, objectValue, "GET"))
        {
          object propertyValue = propertyInfo.GetValue(objectValue);
          return APIHelper.SerializeObjectAndCreateOkResponse(propertyValue, propertyInfo.PropertyType, Request);
        }
        else
        {
          return APIHelper.GetNotAuthorizedResponseMessage(Request, "Not authorized to process request.");
        }
      }
      else
      {
        return APIHelper.GetBadRequestResponseMessage(Request, "Property not found!");
      }
    }
    else
    {
      return APIHelper.GetBadRequestResponseMessage(Request, "Object type not found!");
    }
  }
  catch (Exception e)
  {
    Log.Error("An internal error occurred", e);
    return APIHelper.GetInternalErrorResponseMessage(Request, e);
  }
}
```

Listing 8.13: PropertyController Get method code to retrieve an objects special property

POST and DELETE Method

The methods of the property controller for the HTTP POST and DELETE method are able to handle the command to add and remove a specific object of a generic list property. The type of the object has to be a concrete type. The code cannot handle interface or abstract types. These two methods get the property value of the object and load the generic type of the property list. The object, which has to be added or deleted from the list, is retrieved by the generic type of the list and the id through executing the APIHelper method to load an object of a specific type with the id.

The next step is to load the method information to add an object to the list type or to remove an object from the list. This method is then executed on the property value and the value is written back to the property of the object. These steps also include error handling as each other of the methods for all controllers.

```
1  /// <summary>
2  /// Posts the specified type.
3  /// </summary>
4  /// <param name="type">The type.</param>
5  /// <param name="id">The identifier.</param>
6  /// <param name="propertyname">The propertyname.</param>
7  /// <param name="objectid">The objectid.</param>
8  /// <returns></returns>
9  public HttpResponseMessage Post(string type, int id, string propertyname, int objectid)
10 {
11   try
12   {
13     // load object type
14     Type objectType = APIHelper.GetTypeWithString(type);
15     if (objectType != null)
16     {
17       // property info
18       PropertyInfo propertyInfo = objectType.GetProperty(propertyname);
19       if (propertyInfo != null)
20       {
21         object objectValue = APIHelper.LoadObject(id, type);
22         if (AuthorizationHelper.CheckAuthorization(objectType, objectValue, "POST"))
23         {
24           object propertyValue = propertyInfo.GetValue(objectValue);
25           Type genericType = propertyInfo.GetType().GetGenericArguments().First<Type>();
26           if (genericType != null)
27           {
28             object addObject = APIHelper.LoadObject(objectid, genericType.Name);
29             MethodInfo addMethod = propertyInfo.GetType().GetMethod("Add");
30             List<object> addParamList = new List<object>();
31             addParamList.Add(addObject);
32             addMethod.Invoke(propertyValue, addParamList.ToArray());
33             propertyInfo.SetValue(objectValue, propertyValue);
34             return APIHelper.GetOkResponseMessage(Request);
35           }
36           else
37           {
38             throw new ArgumentException("Property is not a list type.");
39           }
40         }
41         else
42         {
43           return APIHelper.GetNotAuthorizedResponseMessage(Request, "Not authorized to process request.");
44         }
45       }
46       else
47       {
48         return APIHelper.GetBadRequestResponseMessage(Request, "Property not found!");
49       }
50     }
51     else
52     {
53       return APIHelper.GetBadRequestResponseMessage(Request, "Object type not found!");
54     }
55   }
56   catch (Exception e)
57   {
58     Log.Error("An internal error occurred", e);
59     return APIHelper.GetInternalErrorResponseMessage(Request, e);
60   }
61 }
```

Listing 8.14: PropertyController Post method code to add an object to a list property

Listing 8.14 shows the code of the Post method. Both methods look very similar but the difference between the Post and Delete method is that the Post method uses the Add method of the list property and the Delete method the Remove method of the list property. Additionally the authorization HTTP method string is a different string value which is delivered. Listing 8.15 shows the code of the property delete method.

```
/// <summary>
/// Deletes the specified type.
/// </summary>
/// <param name="type">The type.</param>
/// <param name="id">The identifier.</param>
/// <param name="propertyname">The propertyname.</param>
/// <param name="objectid">The objectid.</param>
/// <returns></returns>
public HttpResponseMessage Delete(string type, int id, string propertyname, int objectid)
{
  try
  {
    // load object type
    Type objectType = APIHelper.GetTypeWithString(type);
    if (objectType != null)
    {
      // property info
      PropertyInfo propertyInfo = objectType.GetProperty(propertyname);
      if (propertyInfo != null)
      {
        object objectValue = APIHelper.LoadObject(id, type);
        if (AuthorizationHelper.CheckAuthorization(objectType, objectValue, id, "DELETE"))
        {
          object propertyValue = propertyInfo.GetValue(objectValue);
          Type genericType = propertyInfo.GetType().GetGenericArguments().First<Type>();
          if (genericType != null)
          {
            object addObject = APIHelper.LoadObject(objectid, genericType.Name);
            MethodInfo removeMethod = propertyInfo.GetType().GetMethod("Remove");
            List<object> addParamList = new List<object>();
            addParamList.Add(addObject);
            removeMethod.Invoke(propertyValue, addParamList.ToArray());
            propertyInfo.SetValue(objectValue, propertyValue);
            return APIHelper.GetOkResponseMessage(Request);
          }
          else
          {
            throw new ArgumentException("Property is not a list type.");
          }
        }
        else
        {
          return APIHelper.GetNotAuthorizedResponseMessage(Request, "Not authorized to process request.");
        }
      }
      else
      {
        return APIHelper.GetBadRequestResponseMessage(Request, "Property not found!");
      }
    }
    else
    {
      return APIHelper.GetBadRequestResponseMessage(Request, "Object type not found!");
    }
  }
  catch (Exception e)
  {
    Log.Error("An internal error occurred", e);
    return APIHelper.GetInternalErrorResponseMessage(Request, e);
  }
}
```

Listing 8.15: PropertyController Delete method code to remove an object from a list property

8.3 Web API Configuration

The general configuration is done in two places. One place is the initialization of the application where the HTTP configuration parameters are set and the other place is the configuration file of the application.

8.3.1 HttpConfiguration

The HTTP configuration is done when the application is initialized and the HttpConfiguration object is registered. The first step is to call the method to map the http attribute routes of the configuration object.

The second step that is done in the configuration code is to register the HTTP routes. There are four routes which are registered in total. The first route, which is registered, is the default route to the DefaultController class. The URL for the default route defines the URL for the HTTP GET, POST, PUT, and DELETE method containing the type string and an optional id, which is defined in subchapter 7.5. The optional id parameter is mandatory for the GET, PUT, and DELETE method. The code of the configuration is shown in Listing 8.16.

```
1 // Default route for Get, Post, Put, and Delete
2 config.Routes.MapHttpRoute(
3   name: "DefaultApi",
4   routeTemplate: "{type}/{id}",
5     defaults: new { controller = "Default", id = RouteParameter.Optional }
6 );
```

Listing 8.16: HTTP configuration of default controller route

The second route is the method routes to the MethodController class. The second route URL contains the type string, an id string, the string method, and three optional string parameters. This URL can be used to call methods of a specific object with an id. The method controller can be deactivated with the "ActiveMethodController" configuration parameter. Listing 8.17 shows the code of the described configuration step.

```
if (ConfigurationManager.AppSettings["ActiveMethodController"].Equals("True", StringComparison.InvariantCultureIgnoreCase))
{
// Get path for method calls of specific object with id
config.Routes.MapHttpRoute(
  name: "MethodApi",
  routeTemplate: "{type}/{id}/method/{method}/{param1}/{param2}/{param3}",
  defaults: new { controller = "Method", param1 = RouteParameter.Optional, param2 = RouteParameter.Optional, param3 =
       RouteParameter.Optional }
);
}
```

Listing 8.17: HTTP configuration of method controller routes

The third and fourth route are the property routes to the PropertyController class. This URL contains the type string, the string property, the id string for a specific object, and the property name which shall be loaded. The code is shown in Listing 8.18.

```
if (ConfigurationManager.AppSettings["ActivePropertyController"].Equals("True", StringComparison.InvariantCultureIgnoreCase))
{
  // Get path for property calls of a specifig object with id
  config.Routes.MapHttpRoute(
    name: "PropertyApi",
    routeTemplate: "{type}/property/{id}/{property}",
    defaults: new { controller = "Property" }
  );
  // Post and Delete path for adding and removing an object from a property list with a specific object with id
  config.Routes.MapHttpRoute(
    name: "PropertyApi",
    routeTemplate: "{type}/{id}/property/{propertyname}/{objectid}",
    defaults: new { controller = "Property" }
  );
}
```

Listing 8.18: HTTP configuration of property controller route

After the route mapping configuration, the code sets the general serialization format for the configuration. The code reads the value for the "SerializationType" key from the web.config or app.config file. If it is set to the "XML" value, the code removes the JSONFormatter from the configuration formatter list and sets for the XMLFormatter to use the XMLSerializer. If the value for the "SerializationType" key is "JSON", the XMLFormatter is removed from the configuration formatter list. To be able to fulfill the requirement to be a generalized API, the API has to be able to work with the XML and JSON format. This made it necessary to implement the whole serialization process for both possibilities.

```
if (ConfigurationManager.AppSettings["ActivePropertyController"].Equals("True", StringComparison.InvariantCultureIgnoreCase))
{
  // Get path for property calls of a specifig object with id
  config.Routes.MapHttpRoute(
    name: "PropertyApi",
    routeTemplate: "{type}/property/{id}/{property}",
    defaults: new { controller = "Property" }
  );
  // Post and Delete path for adding and removing an object from a property list with a specific object with id
  config.Routes.MapHttpRoute(
    name: "PropertyApi",
    routeTemplate: "{type}/{id}/property/{propertyname}/{objectid}",
    defaults: new { controller = "Property" }
  );
}
```

Listing 8.19: HTTP configuration of default formatter

The last steps of the code for the configuration is to register the APIAuthorizationHandler as a handler for this HTTP configuration, initialize the audit trail, and set the configuration for the logging. Listing 8.20 shows the code of these steps. It is possible to deactivate the general authentication with the ActiveAuthentication configuration parameter. Details about the authentication process are described in the section 8.1.

```
if (ConfigurationManager.AppSettings["ActiveAuthentication"].Equals("True", StringComparison.InvariantCultureIgnoreCase))
{
  // Add the handler for authentication
  config.MessageHandlers.Add(new APIAuthenticationHandler());
}

AuthorizationHelper.InitializeAuditTrail();

// logging configuration
XmlConfigurator.Configure();
```

Listing 8.20: HTTP configuration register authentication message handler

The complete process of HTTP configuration is done with this code and contains all necessary handlers, formatters, routes, and the audit trail initialization to access the database to retrieve, save, update, or delete data and store the audit trail information.

8.3.2 Configuration File

The configuration file contains the general configuration parameters for the API. It contains two parts. One part are the app settings and the second part is the connection string. The connection strings part only contains one configuration value for the database connection. Listing 8.21 shows this connection string. The connection string is the database connection string for the generic data mapper assembly of the Risk Manager business logic.

```
<connectionStrings>
  <add name="Common.Util.GenericDataMapper.Properties.Settings.dbConnectionString" connectionString="Data
      Source=hostname\dbinstance,port;Initial Catalog=dbname;uid=username;password=*******" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Listing 8.21: App config connection strings value

The app settings part contains the following configuration values:

- ActiveMethodController
- ActivePropertyController
- BusinessLogicFileName
- BusinessLogicNamespace
- UserClass
- GetUserMethod
- ActiveAuthentication
- UserAuthorizationMethod
- ActiveAuthorization
- CustomValueForAuthorization
- LoadMethod
- InsertMethod
- UpdateMethod
- DeleteMethod
- ActiveAuditTrail
- AuditTrailClass
- AuditTrailDll
- AuditTrailNamespace

- AuditTrailMethod

The first two configuration parameters set the value if the method controller and the property controller shall be active or not. The value for being active is "True" and to deactivate both or one of the controllers is to set the value as an empty string.
The configuration parameters for the business logic file name and the business logic namespace are set with the assembly file name of the business logic and the specific namespace. These are important to access the objects of the business logic and the API to process the requests.
The UserClass parameter configures the user object name of the business logic for the authentication process. With the usage of the specified GetUserMethod parameter, the user can be loaded from the business logic. In general it is possible to deactivate the whole authentication process with the ActiveAuthentication parameter.
The configuration parameters also specify the UserAuthorizationMethod which is used for authorization. It is also possible to deactivate the whole authorization process with the ActiveAuthorization configuration parameter. It is also possible to configure a custom value for the authorization process. This value can be used if there is a specific value necessary for authorization method to process.
The LoadMethod, InsertMethod, UpdateMethod, and DeleteMethod configuration parameters define the names of the load, insert, update, and delete methods of the business logic. These have to be implemented for all class types the API shall work with.
The last five configuration parameters ActiveAuditTrail, AuditTrailClass, AuditTrailDll, AuditTrailNamespace, and AuditTrailMethod specify all parameters to initialize the recording of the audit trail for each request. The parameters AuditTrailDll and AuditTrailNamespace are optional and only necessary if the audit trail logic is not contained in the business logic assembly and have an own assembly file with a different namespace from the business logic assembly file. If the ActiveAuditTrail configuration parameter is set to "True", the audit trail initialization is active. The audit trail can be deactivated completely if the configuration parameter is set to "False".
All of these configuration parameters are necessary to let the API run as the specified requirements and be able to access the business logic file of the software it is connected to. The option of configurability makes the API a generalized API.

8.4 Audit Trail CFR 21 Part 11

The initialization of the audit trail is defined as a static method in the AuthorizationHelper class. This method first checks if the "ActiveAuditTrail" configuration parameter is set to "True". If the value is not set to "True" the audit trail is not activated and will not be loaded. After the check is true, the code checks if the necessary configuration parameters for the audit trail class and the audit trail class namespace are set. These two are necessary to load the type information from the assembly. If they are set, the code checks if the audit trail needs a special assembly file. This is configured with the "AuditTrailDll" configuration parameter..

If the audit trail initialization needs a special assembly file and is not included in the business logic assembly, the code first tries to get the assembly from the already loaded assemblies using the AssemblyName object of the assembly. This is done by the GetAssemblyByAssemblyName method of the APIHelper class. If the assembly is not found in the list of loaded assemblies, the code loads the assembly into the current application domain and retrieves the type object of the audit trail class.

On the other hand, if the audit trail class is included in the business logic assembly file, the code retrieves the audit trail type object from the GetTypeWithString method of the APIHelper using the audit trail class name as a string as a parameter.

After the Type object of the audit trail class is loaded, the code searches for the method info object of the method to initialize the audit trail. This is done using the GetMethod method of the Type object with the method name as a string and an array of Type objects as parameters. The Type array defines the types of the parameters of the method for which is searched for.

If the method is found, the method is executed using the Invoke method of the MethodInfo object. Listing 8.22 shows the code of the InitializeAuditTrail method.

The code of the Risk Manager needed some adaptations for the audit trail initialization to work with API. These adaptations are described in section 8.6.4.

```
1  /// <summary>
2  /// Initializes the audit trail.
3  /// </summary>
4  public static void InitializeAuditTrail()
5  {
6    try
7    {
8      if (ConfigurationManager.AppSettings["ActiveAuditTrail"].Equals("True", StringComparison.InvariantCultureIgnoreCase))
9      {
10       if (!String.IsNullOrEmpty(ConfigurationManager.AppSettings["AuditTrailClass"]) &&
               !String.IsNullOrEmpty(ConfigurationManager.AppSettings["AuditTrailNamespace"]))
11       {
12         Type auditTrailType = null;
13         if (!String.IsNullOrEmpty(ConfigurationManager.AppSettings["AuditTrailDll"]))
14         {
15           atAssembly = APIHelper.GetAssemblyByAssemblyName(AssemblyName.GetAssemblyName(AppDomain.CurrentDomain.BaseDirectory +
                   "bin\\" + ConfigurationManager.AppSettings["AuditTrailDll"]));
16           if (atAssembly == null)
17           {
18             atAssembly = AppDomain.CurrentDomain.Load(AssemblyName.GetAssemblyName(AppDomain.CurrentDomain.BaseDirectory + "bin\\" +
                     ConfigurationManager.AppSettings["AuditTrailDll"]));
19           }
20           auditTrailType = atAssembly.GetType(ConfigurationManager.AppSettings["AuditTrailNamespace"] + "." +
                   ConfigurationManager.AppSettings["AuditTrailClass"], true, true);
21           }
22           else
23           {
24         APIHelper.GetTypeWithString(ConfigurationManager.AppSettings["AuditTrailNamespace"] + "." +
               ConfigurationManager.AppSettings["AuditTrailClass"]);
25         }
26         if (auditTrailType != null)
27         {
28           MethodInfo auditTrailMethod = auditTrailType.GetMethod(ConfigurationManager.AppSettings["AuditTrailMethod"], new
                   List<Type>().ToArray());
29           if (auditTrailMethod != null)
30           {
31             auditTrailMethod.Invoke(null, new List<object>().ToArray());
32           }
33         }
34       }
35     }
36   }
37   catch(Exception ex)
38   {
39     Log.Error("An internal error occurred", ex);
40   }
41 }
```

Listing 8.22: AuthorizationHelper method for initializing the audit trail

8.5 Windows Service

The API is implemented in two versions. One version can be used with the Microsoft web server IIS and the other version can be run as a Microsoft Windows service. To be able to install and run the API as a Windows server, the code needed some adaptations. The biggest change is, that the API needs to be run in a self-hosted environment.

The configuration file app.config has three more configuration options. The first is the possibility to set, if the service shall use the encrypted HTTPS protocol. The second and the third one are the hostname and port which are registered for the self-host web server configuration.

The initialization of the self-hosted web server is done by the new class APIService, which inherits from the ServiceBase class to enable the API to be run as a Windows service. When the APIService class is instantiated, the first step the code executes is the initialization of the component. The code of this method sets the service name of the component and if the service can be stopped, paused and continued, and the auto log functionality. Listing 8.23 shows the code.

```
1  /// <summary>
2  /// Initializes the component.
3  /// </summary>
4  private void InitializeComponent()
5  {
6    this.ServiceName = "APIService";
7    this.CanStop = true;
8    this.CanPauseAndContinue = false;
9    this.AutoLog = true;
10 }
```

Listing 8.23: APIService method for initializing the windows service

The APIService class also overrides the ServiceBase class method for the starting and stopping of the service. The starting code is overwritten with the OnStart method. This method instantiates the configuration for the self-hosted web server. This configuration is the SelfHostConfiguration class which inherits from the HttpSelfHostConfiguration. The SelfHostConfiguration class overrides the OnConfigurationBinding method of the parent class. This is necessary to set the HttpBingingSecurityMode to transport to activate the HTTPS protocol. Listing 8.24 shows the code of the overwritten OnConfigureBinding method.

```
1  /// <summary>
2  /// Called to apply the configuration on the endpoint level.
3  /// </summary>
4  /// <param name="httpBinding">The HTTP endpoint.</param>
5  /// <returns>
6  /// The <see cref="T:System.ServiceModel.Channels.BindingParameterCollection" /> to use when building the <see
       cref="T:System.ServiceModel.Channels.IChannelListener" /> or null if no binding parameters are present.
7  /// </returns>
8  protected override BindingParameterCollection OnConfigureBinding(HttpBinding httpBinding)
9  {
10   if (BaseAddress.ToString().ToLower().Contains("https://"))
11   {
12     httpBinding.Security.Mode = HttpBindingSecurityMode.Transport;
13   }
14
15   return base.OnConfigureBinding(httpBinding);
16 }
```

Listing 8.24: SelfConfiguraton class method for activating the HTTPS protocoll

After the instantiation of the configuration class for the self-hosted web server, the code does the same configuration as the regular web server version does. The code sets the route in-

formation, sets the serialization formatter, registers the authentication handler, initialized the audit trail, and configures the log file. At the end of the configuration, the code instantiates the HttpSelfHostServer with the configuration instance and calls the OpenAsync and Wait method. Listing 8.25 shows the code of the OnStart method.

```
1  /// <summary>
2  /// When implemented in a derived class, executes when a Start command is sent to the service by the Service Control Manager (SCM) or
       when the operating system starts (for a service that starts automatically). Specifies actions to take when the service starts.
3  /// </summary>
4  /// <param name="args">Data passed by the start command.</param>
5  protected override void OnStart(string[] args)
6  {
7    base.OnStart(args);
8    string protocoll = ConfigurationManager.AppSettings["UseSSL"].Equals("True", StringComparison.InvariantCultureIgnoreCase) ?
           "https://" : "http://";
9    SelfHostConfiguration config = new SelfHostConfiguration(protocoll + ConfigurationManager.AppSettings["URL"] + ":" +
           ConfigurationManager.AppSettings["Port"]);
10
11 ...(Configuration Code)
12
13   HttpSelfHostServer server = new HttpSelfHostServer(config);
14   server.OpenAsync().Wait();
15 }
```

Listing 8.25: APIService class OnStart method of the Windows service

Another necessary method of the APIService class is the public static method Main, which registers the current APIService class as the service which is run by this application.

```
1  /// <summary>
2  /// Defines the entry point of the application.
3  /// </summary>
4  public static void Main()
5  {
6    System.ServiceProcess.ServiceBase.Run(new APIService());
7  }
```

Listing 8.26: APIService class static Main method to register service

The service class needs an installer too. This installer is generated with Visual Studio and in the properties of the installer, it is set, that the service is run by the Local System.

To install the service on Microsoft Windows, it is necessary to run the following line in a command line window, which is run as an administrator:

"C:\Windows\Microsoft.NET\Framework\v4.0.30319>InstallUtil.exe
C:\Users\Emmanuel\Desktop\Service\API\bin\Release\API.exe"

If the API is configured to use HTTPS, the self-hosted web server needs a certificate too. This certificate has to be registered for the application and port. The registration is done with the following command:

"netsh http add sslcert ipport=0.0.0.0:50269 certhash=certificate fingerprint appid=26b143b0-

1f4e-414b-ae03-cd65dc6a05b3"
The certhash can be retrieved from the fingerprint information of the certificate and the appid is found in the AssemblyInfo file of the application.
After the service is installed, it can be started from the Windows Service Manager window.

8.6 Risk Manager Adaptations

The Qware Risk Manager needed some adaptations to run with the API. These adaptations include the authorization management, load, insert, update, and delete methods, and the audit trail initialization. The authorization is a very important part because the Risk Manager software uses a structure of roles and rights for users to authorize them for special controls.

8.6.1 Object Type Right Resource File

The most important part of the adaptations is the conjunction between a database table and the user rights of the Risk Manager. Before the adaptations take place, the Risk Manager software has the user right check coded into the logic of the GUI. The API cannot use this code. For that reason it was necessary to create resource files which connects the business object name with the user right. Therefore the Risk Manager has eight resource files, one global and one local version rights file for each of the HTTP methods.
Due to the reason that a user has complete access to all objects of a project and version if the user is part of the version users, the version user resource file for the HTTP GET method showed in Table 8.1 only contains the audit trail objects. The audit trail objects need special rights to be accessible.
In these resource files the key value of each entry represents the object class name and the value is the name of the corresponding right, which is necessary to access the object. If an object can be accessed with several rights, the right names are separated by a semicolon. For example, for a HTTP GET method request of the object AuditTrail, the returing string of the get method resource file for global user rights is ViewAuditTrail.
The current implementation for the prototype only considers the objects for creating a new project, a new version, and importing the analysis tree for the version.

Table 8.1 shows the definition of a user right resource file. The data of the other resource files can be found in the Appendix A.1.

Class Name	Right
AuditTrail	ViewAuditTrail
AuditTrailChange	ViewAuditTrail

Table 8.1: GET method version user rights

All the resource files are included in the business logic project and compiled into the business logic assembly file.

8.6.2 IsAuthorized Method

The Risk Manager business logic needs a global authorization method for each object class which is loaded with the API to check the authorization for the current user. This method is added to the QRMAbstractBusinessObject class from which every other object class inherits. There are two IsAuthorized methods. One has the current user object and the HTTP method as a string as parameters. This method checks the authorization only for the HTTP GET, PUT, and DELETE method. The code of the method sets the current user id to the AuditTrailUserId property which is used by the audit trail.

The method gets the Type object of the current generic type T and loads the object which is send to the client. The code checks if the object contains a property "Version". This is necessary to determine if the code has to check the global right or the version right information. Then the code calls the corresponding authorization method for the HTTP method.

If the HTTP method is the GET method, the code calls the IsAuthorizedGet method. Within the method, the code checks if a version object is set. The code then searches for the version user of the current user id on the version. The user can access all objects belonging to a version if he is a version user of the version. Otherwise the global rights resource for the HTTP GET method is accessed to get the right name and checked if the current user has the right to read the object.

If the HTTP method is the PUT method, the code calls the IsAuthorizedPut method. Within the method, the code checks if the version object is set to determine if the version rights resource

is used to get the right name, or the global rights resource. The code uses the special resource files for the HTTP PUT method.

For the HTTP DELETE method it is the same process as for the HTTP PUT method. The code only uses the version and global right resource file for the DELETE method.

Listing 8.27 shows the main IsAuthorized method code and Listing 8.28 till 8.30 shows the code of the single IsAuthorized methods for the HTTP GET, PUT, and DELETE method.

```
1  /// <summary>
2  /// Determines whether the specified identifier is authorized.
3  /// </summary>
4  /// <param name="user">The user.</param>
5  /// <param name="method">The method.</param>
6  /// <returns></returns>
7  public virtual bool IsAuthorized(User user, String method)
8  {
9    this.AuditTrailUserId = user.UserId;
10   Type genericType = typeof(T);
11   bool isAuthorized = false;
12   T genericObject = QrmAbstractBusinessObject<T>.Load(this.GetId().Value);
13   PropertyInfo versionProperty = genericType.GetProperty("Version");
14   BusinessLogic.Version version = null;
15   if (versionProperty != null)
16   {
17     version = (BusinessLogic.Version)versionProperty.GetValue(genericObject, null);
18   }
19   else if (typeof(T) == typeof(BusinessLogic.Version))
20   {
21     version = QrmAbstractBusinessObject<BusinessLogic.Version>.Load(this.GetId().Value);
22   }
23   if (method.Equals("GET"))
24   {
25     return IsAuthorizedGet(user, version);
26   }
27   else if (method.Equals("PUT"))
28   {
29     return IsAuthorizedPut(user, version);
30   }
31   else if (method.Equals("DELETE"))
32   {
33     return IsAuthorizedDelete(user, version);
34   }
35   return false;
36 }
```

Listing 8.27: Global is authorized method for checking the authorization of the current user using the object id

```
/// <summary>
/// Determines whether [is authorized get] [the specified identifier].
/// </summary>
/// <param name="id">The identifier.</param>
/// <param name="user">The user.</param>
/// <returns></returns>
public virtual bool IsAuthorizedGet(User user, BusinessLogic.Version version)
{
  bool isAuthorized = false;
  T genericObject = QrmAbstractBusinessObject<T>.Load(this.GetId().Value);
  if (version != null)
  {
    if (BusinessLogic.VersionUser.GetVersionUserByVersionIdAndUserId(version.VersionId, user.UserId) != null)
    {
      return true;
    }
  }
  if (!isAuthorized)
  {
    isAuthorized = CheckUserRightGlobal(GETGlobalClassRight.ResourceManager.GetString(genericType.Name), user);
  }
  return isAuthorized;
}
```

Listing 8.28: IsAuthorized method for HTTP Get requests

```
/// <summary>
/// Determines whether [is authorized put or delete] [the specified user].
/// </summary>
/// <param name="user">The user.</param>
/// <returns></returns>
public virtual bool IsAuthorizedPut(User user, BusinessLogic.Version version)
{
  Type genericType = typeof(T);
  bool isAuthorized = false;

  if (version != null)
  {
    isAuthorized = CheckUserRight(PUTClassRight.ResourceManager.GetString(genericType.Name), user, version);
  }

  if (!isAuthorized)
  {
    isAuthorized = CheckUserRightGlobal(PUTGlobalClassRight.ResourceManager.GetString(genericType.Name), user);
  }
  return isAuthorized;
}
```

Listing 8.29: IsAuthorized method for HTTP Put requests

```
1  /// <summary>
2  /// Determines whether [is authorized delete] [the specified user].
3  /// </summary>
4  /// <param name="user">The user.</param>
5  /// <returns></returns>
6  public virtual bool IsAuthorizedDelete(User user, BusinessLogic.Version version)
7  {
8    Type genericType = typeof(T);
9    bool isAuthorized = false;
10   if (version != null)
11   {
12     isAuthorized = CheckUserRight(DELETEClassRight.ResourceManager.GetString(genericType.Name), user, version);
13   }
14
15   if (!isAuthorized)
16   {
17     isAuthorized = CheckUserRightGlobal(DELETEGlobalClassRight.ResourceManager.GetString(genericType.Name), user);
18   }
19   return isAuthorized;
20 }
```

Listing 8.30: IsAuthorized method for HTTP Delete requests

The other global IsAuthorized method has an additional string type id parameter, which represents the custom authorization value. This custom authorization value is for the Risk Manager business logic the version id. This authorization method is used only for HTTP POST method requests.

The code sets the current user id to the audit trail user id property of the current object. If the string id can be parsed to an integer value, the code tries to load the version object if the id is greater than zero. If the version object is loaded, the version right resource for the HTTP POST method request is used to get the right name. Otherwise the global user right resource. If the parsing of the string value fails, the code directly uses the global right resource file.

Listing 8.31 shows the code of the method.

```
1  /// <summary>
2  /// Determines whether the specified identifier is authorized.
3  /// </summary>
4  /// <param name="id">The identifier.</param>
5  /// <param name="user">The user.</param>
6  /// <param name="method">The method.</param>
7  /// <returns></returns>
8  public virtual bool IsAuthorized(string id, User user, String method)
9  {
10   this.AuditTrailUserId = user.UserId;
11   Type genericType = typeof(T);
12   int versionId;
13   BusinessLogic.Version version = null;
14   if (Int32.TryParse(id, out versionId))
15   {
16     if (versionId > 0)
17     {
18       version = QrmAbstractBusinessObject<BusinessLogic.Version>.Load(versionId);
19     }
20     if (version != null)
21     {
22       return CheckUserRight(POSTClassRight.ResourceManager.GetString(genericType.Name), user, version);
23     }
24     else
25     {
26       return CheckUserRightGlobal(POSTGlobalClassRight.ResourceManager.GetString(genericType.Name), user);
27     }
28   }
29   else
30   {
31     return CheckUserRightGlobal(POSTGlobalClassRight.ResourceManager.GetString(genericType.Name), user);
32   }
33 }
```

Listing 8.31: Global POST is authorized method for checking the authorization with custom parameter

All of the authorization methods use two private methods CheckUserRight and CheckUserRightGlobal. The CheckUserRight method gets as parameters the right name from the resource file, the user object, and the version object. The method checks if the right string contains more than one right through splitting the string into single strings at a semicolon. After that for each right name string the code calls the user object method HasRightForVersion with the right name string and the version object as parameter. Listing 8.32 shows the code for the CheckUserRight method.

```
/// <summary>
/// Checks the user right.
/// </summary>
/// <param name="rights">The rights.</param>
/// <param name="user">The user.</param>
/// <param name="version">The version.</param>
/// <returns></returns>
private bool CheckUserRight(string rights, User user, BusinessLogic.Version version)
{
  bool isAuthorized = false;
  if (!String.IsNullOrEmpty(rights))
  {
    string[] rightList = rights.Split(new char[] { ';' }, StringSplitOptions.RemoveEmptyEntries);

    foreach (string right in rightList)
    {
      if (!isAuthorized)
      {
        if (version != null)
        {
          isAuthorized = user.HasRightForVersion(right, version);
        }
      }
    }
  }
  return isAuthorized;
}
```

Listing 8.32: Check User Right method

The CheckUserGlobalRight method gets only the right name from the resource file and the user object. The code is exactly the same, only the user object method, which is called, is not HasRightForVersion but HasRight. This method only gets the right name string as parameter. Listing 8.33 shows the code.

```
/// <summary>
/// Checks the user right global.
/// </summary>
/// <param name="rights">The rights.</param>
/// <param name="user">The user.</param>
/// <returns></returns>
private bool CheckUserRightGlobal(string rights, User user)
{
  bool isAuthorized = false;
  if (!String.IsNullOrEmpty(rights))
  {
    string[] rightList = rights.Split(new char[] { ';' }, StringSplitOptions.RemoveEmptyEntries);

    foreach (string right in rightList)
    {
      if (!isAuthorized)
      {
        isAuthorized = user.HasRight(right);
      }
    }
  }
  return isAuthorized;
}
```

Listing 8.33: Check User Right Global method

All of the authorizing methods are virtual methods and can be overwritten by classes which inherit from this class. This is necessary because some objects need special rights and checks

for authorization.

8.6.3 Load and Delete

The Risk Manager business logic needs an adaptation for the load and delete method of the QRMAbstractBusinessObject class. The class inherits from the AbstractBusinessObject class of the Generic Data Mapper, which defines a load method with an object type value as parameter. This method is not accessible using Reflections in the API code. The GetMethod method of the object type does not find the Load method of the AbstractBusinessObject. This is the reason why the QRMAbstractBusinessObject class needed an adaptation for the load method to make it accessible for the API. The new virtual load method of the QRMAbstractBusinessObject has an integer id and the current user object as parameter. The current user object is necessary to make the method unique because if the parameter of the new Load method is only the integer id value. It is equivalent to the Load method of the AbstractBusinessObject because an integer type is an object type too. This new method is only used by the API. The user object is not directly used by the method code. This load method calls the defined load method of the AbstractBusinessObject to load the object with the id.
Listing 8.34 shows the code of the new load method of the QRMAbstractBusinessObject.

```
1  /// <summary>
2  /// Loads the specified identifier.
3  /// </summary>
4  /// <param name="id">The identifier.</param>
5  /// <returns></returns>
6  public virtual T Load(int id, User user)
7  {
8    return AbstractBusinessObject<T>.Load((object)id);
9  }
```

Listing 8.34: QRMAbstractBusinessObject load method for the API

Another method which had to be defined is the delete method of the QRMAbstractBusinessObject class. The new delete method takes an integer id as parameter. The code of the method checks if the current object has the same id as the provided id and if this is true, the delete method of the current object is called. If the current object has a different id, the code loads the object of the current type with the provided id and calls the delete method using reflection. Listing 8.35 shows the code of the new delete method.

```
1  /// <summary>
2  /// Deletes the specified identifier.
3  /// </summary>
4  /// <param name="id">The identifier.</param>
5  public virtual void Delete(int id)
6  {
7    if (this.GetId() == id)
8    {
9      this.Delete();
10   }
11   else
12   {
13     List<Type> array = new List<Type>();
14     T deleteObject = QrmAbstractBusinessObject<T>.Load((object)id);
15     Type type = deleteObject.GetType();
16     MethodInfo method = type.GetMethod("Delete", array.ToArray());
17     method.Invoke(deleteObject, new List<Object>().ToArray());
18   }
19 }
```

Listing 8.35: QRMAbstractBusinessObject delete method for the API

8.6.4 Audit Trail Initializiation

The initialization of the audit trail is defined in the QwareSync class of the DbSync assembly. This class needed three new methods. The first method is the initialization of the audit trail. Currently the audit trail is initialized with the multi user configuration of the Risk Manager. The multi user configuration is not used by the API and for this reason the API needed a static method where only the audit trail is initialized.

This method first checks if the usage of the audit trail is set in the global settings of the Risk Manager. If this is true, the code instantiates a new QwareSync object which is assigned to the static SingleInstance property. After this, the code sets the synchronization context of the current thread and instantiates the AuditTrailWriter object including the assignment to the ATWriter property of the QwareSync instance. It is also necessary to register a new event handler to the DataMapper ItemModified event. The API cannot use the default event handler for the data mapper because the code does more than writing the audit trail of the modified object. Therefore a new event handler method APIDataMapper_ItemModified was needed.

The code of the event handler method reads the AuditTrailUserId property of the current sender object to get the current user id. This id is set in the authorization process described in section 8.6.2. After that if the local property UseAuditTrail is set to true, the code writes the audit trail entry using the AuditTrailWriter. The UseAuditTrail property is set to true in the initialization method.

Listing 8.36 shows the code of the event handler method.

```
1  /// <summary>
2  /// Handles the ItemModified event of the APIDataMapper control.
3  /// </summary>
4  /// <param name="sender">The source of the event.</param>
5  /// <param name="e">The <see cref="EntityModifiedEventArgs"/> instance containing the event data.</param>
6  private void APIDataMapper_ItemModified(object sender, EntityModifiedEventArgs e)
7  {
8    this.CurrentUserId = (int)sender.GetType().GetProperty("AuditTrailUserId").GetValue(sender, null);
9
10   if (this.UseAuditTrail)
11   {
12     this.ATWriter.WriteNewAuditTrailEntryFromDataMapper(e, this.CurrentUserId ?? 0);
13   }
14 }
```

Listing 8.36: QwareSync Data Mapper item modified event handler code

The last step of the initialization of the audit trail is to register the call back method for the current page title of the audit trail writer. Therefore a new method APIPageCallback is implemented, which returns the string API. This indicates that the written audit trail was processed by the API.

Listing 8.37 shows the APIPageCallback method code.

```
1  /// <summary>
2  /// APIs the page callback.
3  /// </summary>
4  /// <returns></returns>
5  public static String APIPageCallback()
6  {
7    return "API";
8  }
```

Listing 8.37: QwareSync API Page Callback method code

The complete code of the audit trail initialization method is shown in Listing 8.38.

```
1  /// <summary>
2  /// Starts the synchronize thread for the API.
3  /// </summary>
4  public static void InitializeAPIAuditTrail()
5  {
6    if (GlobalSetting.GetSettings().UseAuditTrail)
7    {
8      QwareSync.SingleInstance = new QwareSync();
9      QwareSync.SingleInstance.SyncContext = System.Threading.SynchronizationContext.Current;
10     QwareSync.SingleInstance.ATWriter = new AuditTrailWriter();
11     DataMapper.ItemModified += new System.EventHandler<EntityModifiedEventArgs>(QwareSync.SingleInstance.APIDataMapper_ItemModified);
12     QwareSync.SingleInstance.UseAuditTrail = true;
13     QwareSync.SingleInstance.ATWriter.SetCurrentPageTitleCallback(APIPageCallback);
14   }
15 }
```

Listing 8.38: QwareSync audit trail initialization method

All these methods are necessary to let the audit trail work properly with the API. The audit trail writes an entry for an insert, update, or delete of an object and shows all the updates

done to properties of the object.

9 Economic Value

For every company it is important to have a high economic value software which is implemented and sold by the company. This is because the company has to pay employees, rents for the rooms, electricity, etc. The designed and prototypically implemented API needs to fulfill that too.

The prototypical adaptation of the business logic of the Risk Manager is not completely finished. This still has to be done by the project developers. The adaptations which are needed to be done with the business logic are that all of the objects need to be enabled to be serialized and the completion of the authorization resource files. For the authorization the relations between the rights of a user and the database objects are not completely fulfilled. These needs to be done before the API is fully functional with the Risk Manager business logic.

If the adaptation is finished, the API can be used for several different scenarios. The most common scenario for which the API shall be used, is to import data into the software from other third party software and/or systems. This cost of adapting the business logic of the Risk Manager is the most what is needed to be done. The development time of prototype API was around three weeks that already included the prototypical adaptations of the Risk Manager business logic. To complete the whole adaptation, it will take at least three full work weeks. This includes to add the serialization parameters into the class code and extend the authorization code for the corresponding objects.

The current implementation of the API is already capable to handle all of the necessary requests. For this reason the prototype is already able to process the necessary requests to import and export data of a database. This is possible for every business logic which implements the necessary methods for handling authorization, authentication, the common methods for objects (load, save, update, delete), and the configuration for the serialization for objects, which have to be serialized by the API. The time to adapt a business logic varies between adding all of the mentioned requirements or only a selected part. Most of the business logics already

implement comment methods of objects. For this reason the serialization, authorization, and authentication need to be adapted. Additionally to the time for adaptations, the setup time for the API, which includes the configuration parameters, deployment, and testing, have to be added.

The benefits of the implementation are that the API can be used on different business logics. The implementation is not a specific one for the Risk Manager business logic but it is a generalized one which can be used with other business logics too. This leads to a great benefit for the API and a high value of reusability. The API fulfills the requirements for software which is used in the area of placing a medical device on the market (EU) and the approval of a medical device for the market (USA). It even fulfills the requirements for digital records of the CFR 21 Part 11 to be able to digitally sign documents and reports which are generated by the software the API is connected with.

Currently there are no APIs available on the market which can be used in the way the developed API can be used. This increases the benefit to be the first company, which offers such a product. Another benefit of the API for the Qware Risk Manager is, that the API can be used as an import helper to import data from other software systems, for example Microsoft Excel. The company can sell a license for every API which is used to import data and the customers could develop an import client using their own IT department. This development can also be done by the bayoonet AG. With the license it is also possible to sell service contracts which can include the fixing of bugs in the software.

These are only the benefits which are already thought the API shall be used for. Other types of use are possible and are even increasing the benefit. One of these other types of use can be that the API is the main connection between the frontend software and the database. In this case, the API can be used to connect the Risk Online software which can be developed for many different environments together to one database. An example of these different environments can be a mobile tablet version.

This analysis shows that the cost of inventing and implementing the API are reasonable because the benefit of using the API are outweigh the costs. The many benefits of the API generates a high economic value and is a valuable new invention which can be used as a commercial product.

10 Conclusion

The designed and implemented prototype of the generalized API is able to be used with the Qware Risk Manager. The API fulfills the defined requirements which are necessary for a software used in the environment of the application of a medical device and putting a medical device on the market. The API is capable on the task of importing the data of an analyze tree into the Risk Manager software. This includes creating a new project, adding a new version to the project, setting a user as version user, and importing the analysis tree. It is also capable to register itself for the usage of an audit trail. This is necessary for the CFR 21 Part 11 requirements to be able to digitally sign documents. Further the requirements of the CFR 21 Part 11 include authentication and authorization requirements which are implemented as well. In general the API is able to load, insert, update, and delete objects of a database. This is done in a generalized implementation which enables the API to work with different business logic assemblies. One of the requirements of the Risk Manager developer was, that the API is capable to be run as a Microsoft Windows service. Therefore the API exists in two versions. One can be run within a Microsoft IIS web server environment and the other one as a windows service which delivers its own web server.

The implementation of the API was not the only implementation respectively adaptation. The Qware Risk Manager business logic had to be adapted to be able to work with the API. This adaption is not done for the whole business logic. It was only fulfilled for the requirement of being able to add a new project, add a version and version user to the project, and import the analysis tree into the version. Only the used classes define the serialization parameters and are configured for the authorization check. All other classes are not configured to be able to be used with the API. These have to still be completed. The adaptations also include implementing authorization and authentication methods.

Overall the API is completely ready to be run and has no implementation left which is not already implemented.

A Appendix

A.1 User Right Resource Data

Class Name	Right
Cause	ViewAllProjects
Function	ViewAllProjects
FunctionHazard	ViewAllProjects
FunctionHazardCause	ViewAllProjects
FunctionHazardCauseMeasure	ViewAllProjects
FunctionSet	ViewAllProjects
FunctionSetFunction	ViewAllProjects
Hazard	ViewAllProjects
Project	ViewAllProjects
Version	ViewAllProjects
VersionUser	ViewAllProjects

Table A.1: GET method global user rights

Class Name	Right
Cause	PerformAnalysis;ImportExportVersion
Filter	ManageProjectData;ImportExportVersion
Function	PerformAnalysis;ImportExportVersion
FunctionHazard	PerformAnalysis;ImportExportVersion
FunctionHazardCause	PerformAnalysis;ImportExportVersion
FunctionHazardCauseMeasure	PerformAnalysis;ImportExportVersion
FunctionSet	PerformAnalysis;ImportExportVersion
FunctionSetFunction	PerformAnalysis;ImportExportVersion
GraphDefinition	ManageGraph
Hazard	PerformAnalysis;ImportExportVersion
LifeCycle	CreateProject;ImportExportVersion
Probability	ManageGraph
Project	CreateProject;ImportExportVersion
Severity	ManageGraph
Version	CreateProject;ImportExportVersion
VersionUser	ManageProjectTeam

Table A.2: POST method version user rights

Class Name	Right
Cause	ImportExportVersion
Filter	CreateProject;ImportExportVersion
Function	ImportExportVersion
FunctionHazard	ImportExportVersion
FunctionHazardCause	ImportExportVersion
FunctionHazardCauseMeasure	ImportExportVersion
FunctionSet	ImportExportVersion
FunctionSetFunction	ImportExportVersion
GraphDefinition	CreateProject;ImportExportVersion
Hazard	ImportExportVersion
LifeCycle	CreateProject;ImportExportVersion
Probability	CreateProject;ImportExportVersion
Project	CreateProject;ImportExportVersion
Severity	CreateProject;ImportExportVersion
Version	CreateProject;ImportExportVersion
VersionUser	CreateProject;ImportExportVersion

Table A.3: POST method global user rights

Class Name	Right
Cause	PerformAnalysis
Filter	CreateProject;ImportExportVersion
Function	PerformAnalysis
FunctionHazard	PerformAnalysis
FunctionHazardCause	PerformAnalysis
FunctionHazardCauseMeasure	PerformAnalysis
FunctionSet	PerformAnalysis
FunctionSetFunction	PerformAnalysis
GraphDefinition	CreateProject;ImportExportVersion
Hazard	PerformAnalysis;ImportExportVersion
LifeCycle	CreateProject;ImportExportVersion
Probability	CreateProject;ImportExportVersion
Project	ManageVersionData
Severity	CreateProject;ImportExportVersion
Version	CreateProject;ImportExportVersion
VersionUser	CreateProject;ImportExportVersion

Table A.4: PUT method version user rights

Class Name	Right
Project	ManageVersionData
Version	ManageVersionData

Table A.5: PUT method global user rights

Class Name	Right
AuditTrail	ViewAuditTrail
AuditTrailChange	ViewAuditTrail
Cause	PerformAnalysis
Function	PerformAnalysis
FunctionHazard	PerformAnalysis
FunctionHazardCause	PerformAnalysis
FunctionHazardCauseMeasure	PerformAnalysis
FunctionSet	PerformAnalysis
FunctionSetFunction	PerformAnalysis
Hazard	PerformAnalysis
LifeCycle	CreateProject;ImportExportVersion
Project	DeleteProjectsAndVersion
User	ManageUserPermissions
Version	DeleteProjectsAndVersion
VersionUser	ManageProjectTeam

Table A.6: Delete method version user rights

Class Name	Right
Project	DeleteProjectsAndVersion
Version	DeleteProjectsAndVersion

Table A.7: DELETE method global user rights

Bibliography

[DIRa] *Council directive 90/385/eec.* http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:31990L0385&from=EN. last visited on 04/17/2015.

[DIRb] *Council directive 93/42/eec.* http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:31993L0042&from=EN. last visited on 04/14/2015.

[DIRc] *Directive 98/79/ec of the european parliament and of the council of 27 october 1998 on in vitro diagnostic medical devices.* http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:31998L0079&from=en. last visited on 04/14/2015.

[DvdZ] Delaney, Helen and Zande, Rene van der: *A guide to the eu active implantable medical devices directive.* http://gsi.nist.gov/global/docs/EUGuide_ActiveImpMedicalDvceDirective.pdf. last visited on 04/17/2015.

[Fac] *Graph api.* https://developers.facebook.com/docs/graph-api. last visited on 08/27/2015.

[FDAa] *Classify your medical device.* http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview/ClassifyYourDevice/ucm2005371.htm. last visited on 04/21/2015.

[FDAb] *De novo classification process (evaluation of automatic class iii designation) draft guidance for industry and food and drug administration staff.* http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/UCM273903.pdf. last visited on 04/22/2015.

[FDAc] *Classify your medical device > device classification panels.* http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview/ClassifyYourDevice/ucm051530.htm. last visited on 04/21/2015.

[FDAd] *Guidance for industry part 11, electronic records; electronic signatures — scope and application.* http://www.fda.gov/downloads/RegulatoryInformation/Guidances/ucm125125.pdf. last visited on 02/16/2015.

[FDAe] *Mandatory reporting requirements: Manufacturers, importers and device user facilities.* http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/PostmarketRequirements/ReportingAdverseEvents/default.htm. last visited on 04/25/2015.

[FDAf] *21 cfr part 11 electronic records; electronic signatures; final rule.* http://21cfrpart11.com/files/library/government/21cfrpart11_final_rule.pdf. last visited on 02/16/2015.

[FDAg] *Regulatory controls (medical devices) > general controls for medical devices.* http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/Overview/GeneralandSpecialControls/ucm055910.htm. last visited on 04/22/2015.

[FDA94] *21 cfr 20.61 - trade secrets and commercial or financial information which is privileged or confidential.* https://www.law.cornell.edu/cfr/text/21/20.61, 1994. last visited on 05/12/2015.

[FDA14a] *21 cfr 21.10 - policy concerning records about individuals.* https://www.law.cornell.edu/cfr/text/21/21.10, 2014. last visited on 05/12/2015.

[FDA14b] *Title 21 food and drugs chapter i food and drug administration department of health and human services subchapter a general part 58 good laboratory practice for nonclinical laboratory studies.* http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfcfr/CFRsearch.cfm?CFRPart=58, 2014. last visited on 04/25/2015.

[FDA14c] *Fda cfr 21 part 807 subpart e–premarket notification procedures.* http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfcfr/CFRSearch.cfm?CFRPart=807&showFR=1&subpartNode=21:8.0.1.1.5.5, 2014. last visited on 04/17/2015.

[Goo] *The google geocode api.* https://developers.google.com/maps/documentation/geocoding/intro. last visited on 08/27/2015.

[HEN09] HENNING, MICHI: *Api design matters.* Communications of the ACM, 52(5):46 – 56, 2009, ISSN 00010782. http://search.ebscohost.com.ezproxy.uwplatt.edu/login.aspx?direct=true&AuthType=ip,uid&db=buh&AN=39363005&site=ehost-live&scope=site.

[ISO] *What is ido 13485 certification? iso 13485 overview.* http://www.emergogroup.com/resources/articles/what-is-iso-13485-certification. last visited on 05/07/2015.

[JP11] Jugel, Uwe and Preußner, André: *A case study on api generation.* In Kraemer, FrankAlexander and Herrmann, Peter (editors): *System Analysis and Modeling: About Models*, volume 6598 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2011, ISBN 978-3-642-21651-0. http://dx.doi.org/10.1007/978-3-642-21652-7_10.

[Lak13] Lakshmiraghavan, Badrinarayanan: *Securing asp.net web api.* In *Pro ASP.NET Web API Security.* Apress, 2013, ISBN 978-1-4302-5783-7. http://link.springer.com.ezproxy.uwplatt.edu/book/10.1007%2F978-1-4302-5783-7.

[MSD] *Msdn developer network.* https://msdn.microsoft.com/en-us/library/9k985bc9(v=vs.110).aspx. last visited on 07/11/2015.

[Süda] Süd, Tüv: *Ce marking of in vitro diagnostic medical devices - ivdd (98/79/ec).* http://www.tuv-sud.com/industry/healthcare-medical-device/market-approval-certification-for-medical-devices/ce-marking/ivdd. last visited on 04/18/2015.

[Südb] Süd, Tüv: *Directive 98/79/ec on in vitro diagnostic medical devices.* http://www.tuev-sued.de/uploads/images/1384776345177985880609/ivd-directive-98-79-ec.pdf. last visited on 04/17/2015.

[Sin] Singh, Jasminder: *Web api self-hosting using windows service: Part 1.* http://www.c-sharpcorner.com/UploadFile/b1df45/web-api-self-hosting-using-windows-service/. last visited on 07/11/2015.

[tto] *Certification and registration medical devices on the european market.* http://de.slideshare.net/ttopstart/certification-and-registration-medical-devices-on-the-european-market. last visited on 08/27/2015.

[UK13] Ugurlu, Tugberk; Zeitler, Alexander and Kheyrollahi, Ali: *Http web services in asp.net.* In *Pro ASP.NET Web API.* Apress, 2013, ISBN 978-1-4302-4726-5. http://link.springer.com.ezproxy.uwplatt.edu/book/10.1007/978-1-4302-4726-5/page/1.

[WA] *Digital signatures for selected european medicines agency submissions - wainwright associates.* http://www.wainwrightassociates.co.uk/industry-news/digital-signatures-for-selected-european-medicines-agency-submissions/. last visited on 05/20/2015.

[Was12] Wasson, Mike: *The asp.net site.* http://www.asp.net/web-api/overview/older-versions/self-host-a-web-api, 2012. last visited on 07/11/2015.

[Was14] Wasson, Mike: *The asp.net site.* http://www.asp.net/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api, 2014. last visited on 07/11/2015.