Tampere University

Ville Saarinen

DESIGN AND IMPLEMENTATION OF A WEB-BASED DATA TRANSFER MODULE IN MICROSCADA X DMS600

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiner: Prof. Hannu-Matti Järvinen, Teemu Leppälä
December 2023

ABSTRACT

Ville Saarinen: Design and Implementation of a Web-Based Data Transfer Module in MicroSCADA X DMS600
Master of Science Thesis
Tampere University
Master's Degree Programme in Information Technology
December 2023

Electrical network distribution system operators are required to export their infrastructure and construction plan data to centralised data point for co-operative joint construction. The amount of data to be transferred per operator is large, and the data must be kept up to date. An automated programmatic solution would reduce workload, be consistent, and ensure that information is cohesive. This research aims to design, implement, and validate the first version of such software that exports construction plan data to the centralised information point via the provided web interface.

Multiple stakeholders affect the motives, objectives, and restrictions set for the software directly or indirectly. Direct stakeholders include the Finnish Transportation- and Communication Agency (*Traficom*), Hitachi Energy, and distribution system operators that are Hitachi Energy's customers. Indirect stakeholders are the European Union and the Parliament of Finland.

This research begins by addressing the directives, laws, and orders regarding joint construction. Firstly, the European Union has set a directive about joint construction and usage, which led to Finnish law on the same subject. Traficom was ordered to implement the centralized information point. They then constructed an act regarding Finnish distribution system operators to send network data and construction plans to the information point.

Some of these operators are customers of Hitachi Energy. Their needs are the basis for the feature design of the software. Most web communication-related restrictions on technologies are set by Traficom since they operate the information point. Internal restrictions on the design of the software are set by Hitachi Energy. These internal restrictions dictate used technologies regarding mainly implementation and validation of the software.

The main concentration of research in used technologies is on relevant web technologies. These include RESTful application programming interfaces (*REST API*), hypertext transfer protocol (*HTTP*) communication, and authentication using JSON web tokens (*JWT*). Another researched area of software development is validation and more precisely automatic testing of software. Different principles and concepts are examined, such as two schools of unit testing: Classical and London. These principles are then used in researching practical testing methodologies, mainly unit testing.

The practical section of this research consists of designing, implementing, and validating the first version of the software. The practical section is bound with the theory researched earlier. The design of the software consists of architecture and deeper module design, fulfilling the set requirements and constraints. Implementation of the software is done using C#. Validation of the program is made with automated unit testing using MSBuild, and static scanning using SonarQube. Software project is then added to the product portfolio in version control and build system using continuous integration methodologies.

Technologies regarding the whole system are analysed upon suitability and security. Improvements to these are suggested in the evaluation, although some of the used technologies were dictated by Traficom. The first version of the software solution is also evaluated: difficulties of the project are addressed, deficiencies are noted, and suggestions regarding follow-up and next steps are given.

Keywords: software design, unit testing, application programming interface

TIIVISTELMÄ

Ville Saarinen: Web-Pohjaisen Tiedonsiirtorajapinnan Suunnittelu ja Toteutus MicroSCADA X DMS600 Tuotteeseen
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Joulukuu 2023

Sähköverkon jakeluverkonhaltijoiden on toimitettava infrastruktuuri- ja rakentamissuunnitelmatietonsa keskitettyyn tietopisteeseen yhteisrakentamista varten. Siirrettävän tiedon määrä operaattoria kohden on suuri, ja tiedot on pidettävä ajan tasalla. Automatisoitu ohjelmallinen ratkaisu vähentäisi työmäärää, takaisi johdonmukaisuuden ja varmistaisi tiedon yhtenäisyyden. Tämän tutkimuksen tavoitteena on suunnitella, toteuttaa ja validoida ensimmäinen versio sellaisesta ohjelmistosta, joka vie rakennussuunnitelmatiedot tähän keskitettyyn tietopisteeseen ohjelmallisen rajapinnan kautta.

Useat sidosryhmät vaikuttavat ohjelmistolle asetettuihin motiiveihin, tavoitteisiin ja rajoituksiin suoraan tai välillisesti. Suoria sidosryhmiä ovat Liikenne- ja viestintävirasto (Traficom), Hitachi Energy sekä Hitachi Energyn asiakkaita olevat jakeluverkonhaltijat. Välillisiä sidosryhmiä ovat Euroopan unioni ja eduskunta.

Tämä tutkimus alkaa käsittelemällä yhteisrakentamiseen liittyviä direktiivejä, lakeja ja määräyksiä. Euroopan unioni on asettanut yhteisrakentamisesta ja yhteiskäytöstä direktiivin, joka johti Suomen lakiin samasta aiheesta. Traficom määrättiin toteuttamaan keskitetty tietopiste. Sen jälkeen he asettivat Suomen jakeluverkonhaltijoita koskevan asetuksen verkkotietojen ja rakentamissuunnitelmien lähettämisestä tietopisteeseen.

Jotkut näistä operaattoreista ovat Hitachi Energyn asiakkaita. Niiden tarpeet ovat tämän luotavan ohjelmiston ominaisuussuunnittelun perusta. Suurin osa verkkoviestintään liittyvistä teknologioiden rajoituksista on Traficomin asettamia, koska he ylläpitävät tietopistettä. Hitachi Energy asettaa sisäiset rajoitukset ohjelmiston suunnittelulle. Nämä sisäiset rajoitukset sanelevat käytettävät tekniikat lähinnä ohjelmiston toteutuksessa ja validoinnissa.

Työssä teknologioiden tutkimus keskittyy pääasiassa asiaankuuluviin verkkoteknologioihin. Näitä ovat RESTful-sovellusohjelmointirajapinnat (*REST API*), hypertekstinsiirtoprotokolla (*HTTP*) -viestintä ja autentikaatio JSON-verkkotunnuksilla (*JWT*). Toinen ohjelmistokehityksen tutkittu alue on ohjelmistojen validointi ja tarkemmin sanottuna automaattinen testaus. Työssä tarkastellaan erilaisia periaatteita ja käsitteitä, kuten kahta yksikkötestauksen koulukuntaa: Klassinen ja Lontoo. Näitä periaatteita hyödynnetään sitten tutkittaessa käytännön testausmenetelmiä, lähinnä yksikkötestausta.

Tämän tutkimuksen käytännön osa koostuu ohjelmiston ensimmäisen version suunnittelusta, toteutuksesta ja validoinnista. Käytännön osuus on sidottu aiemmin työssä tutkittuun teoriaan. Ohjelmiston suunnittelu koostuu arkkitehtuurista ja syvemmästä moduulisuunnittelusta, joka täyttää asetetut vaatimukset ja rajoitukset. Ohjelmiston toteutus tapahtuu C#:lla. Ohjelman validointi tehdään automaattisella yksikkötestauksella käyttäen MSBuild:a ja staattisella skannauksella käyttäen SonarQubea. Ohjelmistoprojekti lisätään sitten tuotevalikoimaan jatkuvan integroinnin menetelmillä versionhallinta- sekä rakennusjärjestelmässä.

Koko järjestelmää koskevia teknologioita analysoidaan soveltuvuuden ja tietoturvan perusteella. Arvioinnissa ehdotetaan parannuksia näihin, vaikka osa käytetyistä teknologioista on Traficomin sanelemia. Ohjelmistoratkaisun ensimmäinen versio myös arvioidaan: projektin vaikeudet käsitellään, puutteet todetaan ja ehdotuksia jatkotoimiksi annetaan.

Avainsanat: ohjelmistosuunnittelu, yksikkötestaus, ohjelmistorajapinta

PREFACE

This Master of Science thesis was conducted at Tampere University in 2022 and 2023. Hitachi Energy commissioned it. I had worked for Hitachi Energy for multiple years before this thesis. An upcoming project had an interesting part, so I suggested combining this research with it.

I would like to thank my examiner, Professor Hannu-Matti Järvinen from Tampere University, for valuable feedback and guidance. I would also like to thank my examiner from Hitachi Energy, Teemu Leppälä, for guidance with this research and the constructive part related to the mentioned project. Lastly, I would like to thank Hitachi Energy for allowing me to conduct this research.

Helsinki, 15 December 2023

Ville Saarinen

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
DMS	Distribution Management System
DSO	Distributed System Operator
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-curve Digital Signature Algorithm
EU	European Union
HATEOAS	Hypermedia as the engine of application state
HMAC	Hash-Based Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task Force
IP	Internet Protocol
JOSE	JavaScript Object Signing & Encryption
JSON	JavaScript Object Notation
JWA	JSON Web Algorithms
JWE	JSON Web Encryption
JWS	JSON Web Signature
JWT	JSON Web Token
MAC	Message Authenticated Code
MIME	Multipurpose Internet Mail Extensions
MVP	Minimum viable product
NaN	Not a Number
NIS	Network Information System
QUIC	Quick UDP Internet Connections
REST	Representational State Transfer, Web API architecture
SCADA	Supervisory Control and Data Acquisition
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
SUT	System Under Test
SWT	Simple Web Token
TCP	Transmission Control Protocol
TLS	Transport Layer Security
Traficom	Finnish Transportation- and Communication Agency
TSO	Transmission system operator
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifiers, superset of URL
URL	Uniform Resource Locator
URN	Uniform Resource Name
UTC	Coordinated Universal Time
WWW	World Wide Web
W3C	World Wide Web Consortium, main international standards organization for world wide web
XML	Extensible Markup Language

1. INTRODUCTION

This research aims to create the first version of software that transfers electrical network construction plan data from distribution system operators to a centralized information point, enabling joint construction of electrical networks. The project involves investigating required technologies, designing the software, and implementing it by programming and validating it with automated testing.

This research is conducted using a constructive methodology. The main principle of it is to solve a practical problem while producing theoretical research. Usual parts of constructive research are acknowledging the problem, understanding the study area, designing a solution for the problem, demonstrating the feasibility of the solution, linking the theory to the solution, and investigating the generalisability of the research. [1]

The first part of the research focuses on shareholder objectives and different motives for this work. There are two direct shareholders: Hitachi Energy, which is providing this software to its customers, and the customers, which are using the software to export required data. Motives for this project stem from European Union directives, which have led to local legislation. Other motives are Hitachi Energy's internal; aim to expand knowledge about web technologies and software development methodologies.

The next part of the research is about the used technologies. They are mainly dictated by the receiving party of the data transfer, Traficom. General web data transfer technologies are investigated to provide background information on the subject. Internal systems provide their constraints for software design, but these internal technologies are not the focus.

The implementation part of the research consists mainly of architectural design. Programmatic choices are presented when they are relevant to the research, such as parts that correlate with the technologies investigated.

Automated software testing is one of the main points in implementing the software. Quality assurance is a necessary part of any legitimate software project. The base ground of automated testing is researched, and a testing suite for the software is implemented.

Lastly, the project is evaluated. Technologies are discussed to the point that is possible. Design and implementation difficulties are addressed. Next steps and follow-ups are then recommended.

2. BACKGROUND AND MOTIVATION

This project has many stakeholders, each providing different perspectives and motives for this work, the most important one for business being Hitachi Energy's customers. The main motive is that this project aims to upgrade their software to comply with current standards and regulations that are set for handling construction plans and joint construction.

Hitachi Energy's internal motives are another primary consideration. Providing software to customers is essential, but providing secure and reliable quality software provides the most value for customers and the company. While creating the module, this project investigates practices of creating quality software and integrates them into this project. The knowledge gained will provide improving guidelines and practices for teams regarding other software products and projects.

2.1 Directives, laws, and ordinances

European Parliament and the Council of European Union (*EU*) has signed a directive (2014/61/EU) [2] to reduce the cost of deploying high-speed electronic communications network. The directive is a minimum regulation, allowing countries to enact it under stricter conditions. The directive targets teleoperators, but some articles also contain regulations concerning other network operators, such as distribution system operators.

The directive's description for the term 'network operator' contains an undertaking providing physical infrastructure for electricity (Article 2.1 point a.ii); therefore, the directive has requirements on electricity network companies. Those articles in the directive that set requirements for member states about electricity network companies or their information are presented below:

1. Article 4.1 states that undertakings providing or authorized to provide communication networks have the right to request and access minimum information about the existing physical infrastructure of any network operator. That information includes the location, route, type, current use, and contact point of that infrastructure.

2. Articles 4.2 and 4.3 state that member states can require that every public sector body that has network operator's information defined in Article 4.1 for its tasks

must make it available via a single electronic information point. When also receiving updates to that information, public sector bodies must make those available via the same information point.

3. Article 4.4 states that if there is no single information point available for the information described in Article 4.1, the network operators must provide this information upon a specific request addressed to them by an undertaking providing or authorized to provide public communications networks.
4. Article 6.1 defines the minimum information of ongoing or planned civil works related to the network operator's physical infrastructure that must be made available upon specific request from an undertaking providing or authorized to provide public communications networks. This information contains, at minimum, location and type, network elements involved, estimated start date and duration, and contact point of the works. Article 6.3 states that the information above must be accessible via a single information point.
5. Article 10.4 requires that member states appoint one or more competent bodies to perform the single information point functionalities referred to above. Article 7.1 also states that member states shall ensure that relevant information regarding conditions and procedures for granting permits for civil works be available via the information point.

Member states may fulfil these minimum requirements from the directive as they see best suited. Finland has legislated the joint construction and joint use of network infrastructure law, the joint construction act (276/2016) [3] based on the EU directive. Below are listed relevant parts of the law:

1. 3 § states that the network operator must hand over the right to use its physical infrastructure to another network operator if requested with fair and reasonable terms. Network operators can refuse to grant the right to use the physical infrastructure in case
    a. the infrastructure is not suited for joint usage
    b. network operator itself uses or will use it
    c. general or national safety is compromised
    d. other services in the same physical infrastructure are compromised.

2. 4 § states that the network operator is obligated to consent to joint construction of physical infrastructure and electrical network when requested by another network operator with a fair and reasonable request. The network operator must agree to the request unless it
    a. increases network operators' costs compared to stand-alone construction
    b. concerns a minor construction project
    c. compromises network safety or designated usage.
3. 5 § states that the Finnish transportation- and communication agency Traficom must ensure that there is an easy-to-use and secure centralized information point. Information concerning networks’ physical infrastructure, planned construction works, permit procedures related to construction, and locations of cables, pipes, and comparable active network parts must be given in digital form without undue delay. Information may not be given if it compromises networks’ information security, general or national safety, or if it contains company and business secrets.
4. 7 § states that the network operator must give information described in point 3 above and updates concerning it to be available via the centralized information point. The network operator must also hand over the same information directly to another network operator if requested with fair and reasonable terms.

The Electricity Market Act (588/2013) [4] also contains regulations concerning network operators' physical infrastructure. Relevant parts in the context of joint construction, physical information infrastructure, and the electrical information point from the act are listed below:

5. 110 § states that the network operator must provide, free of charge, information about its electrical cables near a site where earthworks, waterworks, forestry, or other work is taking place to the person preparing or performing the work. The network operator must provide this information in digital form with other information and instructions relevant to avoid danger. This information must be handled and retained so that information security is not compromised, and only authorized personnel can access it.

The Joint Construction Act also states in 13 § that Traficom may give extensive technical regulations regarding the minimum content of information described in point 3 above, information’s digital format, handling and transferring, and system interoperability and information security. Based on the joint construction law, Traficom has given an order on the delivery of network data and network construction plans [5].

The order from Traficom aims to ensure the accuracy and interoperability of the information used in the centralized information point. The order addresses the information’s digital format, minimum content, and interface system’s interoperability. The order is to be applied only to physical infrastructure that is suited to host other network parts and to construction plans which enable joint construction of the network. [5]

2.2 Hitachi Energy objectives

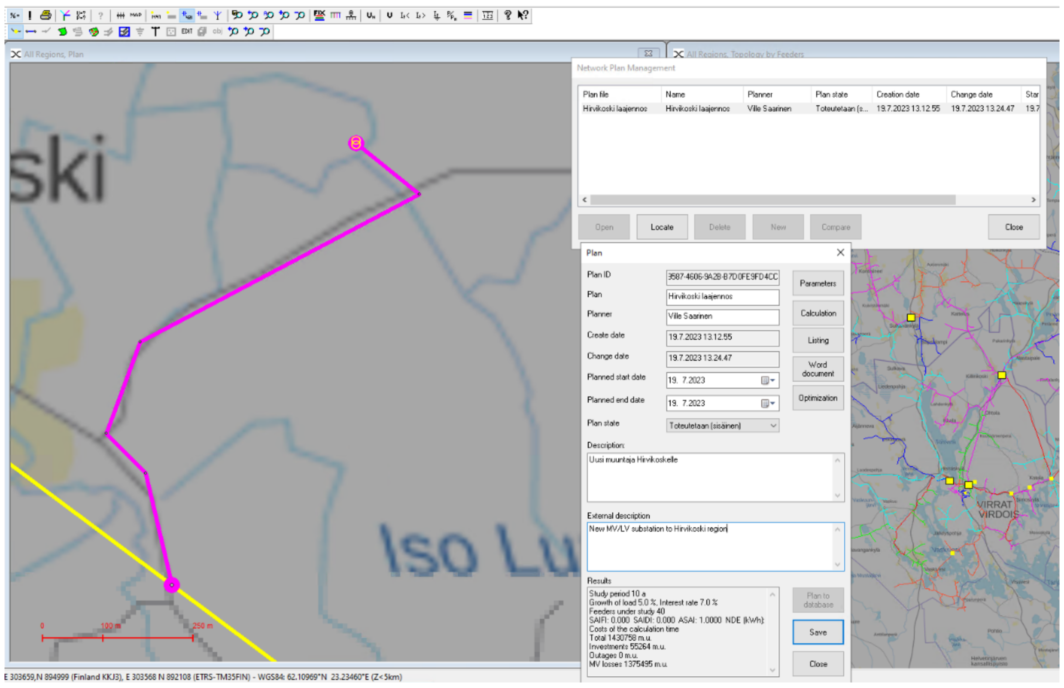
Hitachi Energy provides MicroSCADA X DMS600 distribution management system software for network assets, controlling, and distribution management. This product is used by distribution system operators (*DSOs*) worldwide to handle their electrical network components and electricity usage. DSOs are network operators responsible for providing and operating regional low, medium, and high-voltage electrical networks that distribute electricity directly to customers [6].

MicroSCADA X DMS600 product consists of two main parts: DMS600 Network Editor and DMS600 Workstation. Network Editor is a network information system (*NIS*) that manages network asset data such as transformer and conductor properties, different installation dates, component locations, and construction plans. Workstation is a distribution management system (*DMS*) for electrical network control and management. Where Workstation is used in real-time network operations, Network Editor is used to design the network. These two collaborate mainly via database and some files on the server so that they can be run alone or in parallel.

DMS600 contains information about DSO's electrical network and its components. Some information can also be in third-party applications imported from there to the DMS600 system. Data can also be sent to other systems from DMS600 using an integrated DMS Service platform or other separate applications connected to the DMS600 system.

The first objective for Hitachi Energy in this project is to create a solution for the DMS600 system that can send the required electrical network information to the Traficom information point. Hitachi Energy is not directly required to do so, but providing a programmatic and semi-automatic interface for data transfer improves the product’s functionality and makes clients’ work more efficient and reliable. Since the data must be updated within a reasonable time when it is changed, this becomes a repeatable action that can be helped by automation.

The data sent by this module are construction plans. They are required for joint construction organized by Traficom and are managed in verkkotietopiste.fi centralized information point. Construction plans are created and managed in DMS600 Network Editor. They consist of modified electrical network, planning and construction dates, and other meta information. Construction plan editing is shown in Figure 1 below.



Plan file	Name	Planner	Plan state	Creation date	Change date	Star
Hirvikoski laajennos	Hirvikoski laajennos	Ville Saarinen	Toteutetaan (s...	19.7.2023 13.12.55	19.7.2023 13.24.47	19.7

Plan ID	3587-4606-9A28-B7D0FE9FD4CC
Plan	Hirvikoski laajennos
Planner	Ville Saarinen
Create date	19.7.2023 13.12.55
Change date	19.7.2023 13.24.47
Planned start date	19. 7.2023
Planned end date	19. 7.2023
Plan state	Toteutetaan (sisäinen)

Description:
Uusi muuntaja Hirvikoskelle

External description
New MV/LV substation to Hirvikoski region

Results
```
Study period 10 a
Growth of load 5.0 %, Interest rate 7.0 %
Feeders under study 40
SAIFI: 0.000  SAIDI: 0.000  ASAI: 1.0000  NDE (kWh):
Costs of the calculation time
Total 1430758 m.u.
Investments 55264 m.u.
Outages 0 m.u.
MV losses 1375495 m.u.
```

Figure 1*: Construction plan editing in DMS600 Network Editor.*

In this project, the second objective for Hitachi Energy is to learn and integrate new technologies into the main product. These new technologies relate mainly to modern programmatic web interfaces and technologies used within them. By learning about these technologies, Hitachi Energy and its employees gain information that can be used in the future to create and use different interfaces but also to assess the quality of created or used interfaces.

In this project, the third and final main objective for Hitachi Energy is to improve software quality within the DMS600 product environment. By investigating and advancing, for example, software testing methodologies, products become more reliable with less technical debt in further development and a smaller workload in customer service. This improves productivity and drives the workforce to more meaningful tasks.

2.3 Objectives of a distribution system operator

Distribution system operators are bound by laws to give information about their physical infrastructure as described in Section 2.1. Their main objective is to fulfil the requirements set by the laws.

Another main benefit of joint construction is that it can severely lower the participants' construction costs. For example, repeated opening of road segments can be reduced since multiple networks can be built once via joint construction. The Joint Construction Act also states that the prerequisite for joint construction is that construction costs will not be more than when constructing separately, so the act will not make matters worse financially. Sharing costs will be determined equally between participants so that the entity benefiting the most from the joint construction bears a more significant part of the costs. Especially, small network operators might get more opportunities to its operation from joint constructing. [7]

Finland’s Ministry of Transport and Communications conducted a study about the technical and economic effects of joint construction [8]. Part of the study was based on a survey sent to about 40 network operators, from which about 40% answered. The most prominent areas for joint construction and operation of electrical and communication networks are high voltage networks owned and operated by electrical transmission system operators (*TSO*). TSOs are responsible for national or regional, often high voltage level distribution of electricity [6]. Fingrid Oyj is Finland’s only TSO [9].

Transmission systems are the most prominent subject of joint construction because most new constructions suited for joint construction with a communication network are done in a high-voltage network. According to the study, construction in medium and low-voltage networks suitable for joint construction must be renewal construction, mainly changing overhead lines to underground cables. Since only rare occasions of renewal work in medium and high-voltage networks generate purposeful entities for building communication networks, the most suitable areas for joint construction in the middle and low-voltage networks are in the construction of urban areas and some renewals of entire 20 kV lines in sparsely populated areas. [8]

Challenges for joint construction arise from the differences in the schedules of electricity and telecommunication network projects and the lifetimes of the networks. Generally, electrical network planning is more long-term than telecommunication network planning. Planning and construction of the telecommunication network are guided by changes in the subscriber network and not in the backbone network. [10]

Transport and Communications Committee states in its report (LiVM 3/2016 [7]) regarding the government proposal (HE 116/2015 [10]) that the centralized information point will allow location data of physical infrastructure to be used in the development of operational reliability. It can also be used to prevent unintentional damage occurring in connection with the construction of networks. This will reduce indirect costs of network operation. [7]

On the other hand, the Committee states that the centralised information point makes the data more vulnerable to criminal or security-compromising activity. Therefore, technical implementation, data disclosure, and other data utilisation must be planned and implemented to meet high data security requirements, increasing the costs of the information point. The Committee also states that there will be costs for network operators from delivery and maintenance of data into the centralised data point. [7]

2.4 Existing centralized electrical network data systems

In Finland, Traficom has become responsible for the centralized electrical information point. It has two separate services: verkkotietopiste.fi for network areas and plans, and sijaintitetopiste.fi for physical network infrastructure. Both offer a graphical interface and a programmatic interface, although the programmatic interface for sijaintitietopiste.fi is estimated to be working starting in the second quarter of 2024.

Few operators were providing centralized network information services in Finland before Traficom. These were Keypro Oy, Johtotieto Oy, and the Association of Finnish Local and Regional Authorities (Kuntaliitto) in cooperation with some municipalities and cities. Some of these provided services were self-usable electronic information points, and some were supervised and controlled services. [10]

Similar laws regarding joint construction have been in force at least in Germany, France, Sweden, Denmark, Norway, the United States of America, and South Korea. Centralized information points are in use, at least in Germany, France, Sweden, and Denmark, but only some are electrical, and only some are required to be used by law. This means that the EU directive of joint construction will create or advance the legalization and implementation of joint construction and centralized information points in most of Europe. [10]

3. WEB TECHNOLOGIES

World Wide Web Consortium (*W3C*), which is the leading international standards organization for the World Wide Web (*WWW*, or simply *Web*), describes the Web to be an information space in which items, referred to as resources, are identified by global identifiers called Uniform Resource Identifiers (*URI*) [11]. On the other hand, the Internet is a global system of interconnected computer networks that interchange data by packet switching using a standardized Internet Protocol Suite [12]. Thus, the Web is an information space or collection of resources which can be accessed via the Internet.

For these resources on the Web to be available and usable, many technologies must be used. Three core technologies that make it are URIs, HyperText Transfer Protocol (*HTTP*), and HyperText Mark-up Language (*HTML*). Tim Berners-Lee invented these and their first use case was the sixth of August in 1991 on the Web's first page, created by Berners-Lee. [13]

3.1 URI

Uniform Resource Identifiers (*URIs*) provide a way to identify a resource in the WWW. Each specific URI points to only one resource. The specification does not define resources identified by URIs, so they can be anything: an electronic document, a service, a collection of resources, or a bound book in a library, for example. [14]

URIs can be classified further as locators, names, or both. Uniform Resource Locators (*URLs*) are a subset of URIs that identify the resource and describe its primary access mechanism and way of locating it. Uniform Resource Names (*URNs*) have historically been used to identify resources under the "urn" *scheme*. General URIs can be divided into five hierarchical sections, as shown in Figure 2. The significance of the sections decreases from left to right. These five sections are scheme, authority, path, query, and fragment. Only the scheme and path parts are mandatory, but the path can be empty. [14]

```
  foo://example.com:8042/over/there?name=ferret#nose
  \_/   \______________/\_________/ \_________/ \__/
   |           |            |            |        |
scheme     authority       path        query   fragment
```

Figure 2. *general URI's hierarchical sections.* ***[14]***

The scheme is the first part of URI. It refers to the specification that the scheme and URIs within it use. URI general syntax is extensible by the schemes, making generic syntax a superset of syntax in all current and future schemes. A few well-known schemes are “http”, “ftp”, “file”, and “mailto”. [14]

The authority part defines governing authority for the rest of the namespace defined by the remainder of the URI. The authority part consists of the registered name or server address distinguishing the author, optional user information, and port parts. The authority section starts with a double slash (“//”) and ends with either a single slash (“/”), the number sign (“#”), or a question mark (“?”). Figure 3 shows this format for the authority section. [14]

```
authority   = [ userinfo "@" ] host [ ":" port ]
```

Figure 3. *Authority section in general URI syntax.* ***[14]***

The user info part of the authority section may contain a username and optionally schema-specific details concerning authorization to the resource. The often-used format “username:password” for user info is deprecated, and it should not be used since it presents the password in clear text. This is a security risk, and any text after the colon (“:”), in other words, the password part, should be rejected and not be used, at least in an unencrypted way. [14]

The host part in the authority section is used to identify the authority. It can be an Internet Protocol (*IP*) address literal encapsulated within square brackets, an IPv4 address in dotted-decimal format, or a registered name. Although a URI contains specific authority, it does not mean the used scheme requires access to the given host. Host names are often reused to avoid the creation and registration process of new ones. [14]

The Port part of the authority section is an optional definition for the port number to be used. It follows the host part in URI, separated from it with a single colon (“:”). Schemes can address default ports. For example, “http” scheme uses port number 80, and “https” scheme uses port number 443. If the URI’s port number is the same as the scheme’s default port number, it should be omitted from the URI. [14]

The third section of URI, *path*, is used along the query part to identify a resource within the scheme and possible naming authority. The path part consists of a sequence of path segments in hierarchical order, separated from each other by a single slash (“/”). The path can also be empty. The path starts with a single slash (“/”), which separates it from the possible authority part of the URI. Some path segments are intended to be used for specific cases. For example, so-called dot-segments "." and ".." are used for relative reference at the beginning of a relative path. [14]

The query section of a URI contains non-hierarchical information. It is used along with the path section to identify a resource within the scheme and authority of a URI. Often used format for query information is "key=value" pairs separated by ampersand ("&") from each other. The query part starts with a question mark ("?") and ends with a number sign ("#") or at the end of the URI. [14]

The fragment section of a URI is used to give additional information to identify secondary resources by referencing the primary resource. The secondary resource might be some section or portion of the primary resource or a representational view of the primary resource. The fragment section starts with a number symbol ("#") and is terminated by the end of the URI. [14]

3.2 HTTP

HyperText Transfer Protocol (*HTTP*) is a request/response-based application-level protocol in the Internet Protocol suite model. It is used in collaborative, distributed, hypermedia information systems to transfer data between different WWW services using TCP/IP connections. A Scheme specification called "http" is used in HTTP URLs, which syntax is shown in Figure 4, with the default port number being 80. HTTP protocol is stateless and object-oriented, and it allows for systems to be built independently regardless of the data being transformed. [15]

```
http_URL = "http:" "//" host [ ":" port ] [ abs_path [ "?" query ]]
```

Figure 4. *URL format in HTTP/1.1 scheme.* ***[16]***

The first version of the HTTP protocol, known as HTTP/0.9, was defined in 1991 by Tim Berners-Lee. It is simple and does not transfer client information with the query. Requests in HTTP/0.9 consist of the word "*GET*" and the resource's URI address. Response to the GET request is a message, a byte stream of American Standard Code for Information Interchange (*ASCII*) characters in HTML format. [17], [18]

HTTP/1.0 specification was defined in 1996. Practical information-based systems require more functionality than just retrieval of data with GET requests. Methods defined in HTTP/1.0 are *GET*, *HEAD*, and *POST*. Messages are passed in a similar format as Internet mail and Multipurpose Internet Mail Extensions (*MIME*). Complete requests consist of a request line, headers, and entity body. Complete responses consist of status-line, headers, and entity body. HTTP/0.9 one-line syntax is also allowed. Headers allow metadata to be transferred in both requests and responses. HTTP/1.0 also allows for different content types other than HTML with the use of a Content-Type header. [15], [17]

HTTP/1.1 was defined in 1997 and updated with improvements in 1999, 2014, and 2022. HTTP/1.0 was still insufficient in functionalities that practical information systems require. HTTP/1.0 did not take into consideration caching, virtual hosts, hierarchical proxies, and the need for persistent connections between clients and servers. In HTTP/1.0, each request-response pair requires a new connection, but HTTP/1.1 allows the same connection to be used for multiple request-response exchanges. HTTP/1.1 also defines pipelined connections, allowing new requests to be sent on the same connection before receiving previous responses. Even though HTTP/1.1 allows request pipelining, it still suffered from application-layer head-of-line blocking, not allowing total concurrency. Methods defined in HTTP/1.1 are *GET*, *HEAD*, *POST*, *PUT*, *DELETE*, *TRACE* and *OPTIONS*. [17], [19]

HTTP/2 protocol was standardized in 2015. Web pages became increasingly complex, leading to more data being transmitted over more HTTP requests. This caused overhead for HTTP/1.1 connections. HTTP/2 supports the same core features as HTTP/1.1, but its purpose is to be more efficient. HTTP/2 is a binary, multiplexed protocol, allowing for parallel requests over the same connection and better optimization techniques. Headers are often verbose and similar between requests, so by compressing them, HTTP/2 removes overhead and duplication of transmitted data. HTTP/2 also allows the so-called server push mechanism, allowing servers to use client caches for data. [17], [20]

HTTP/3 is the newest version of HTTP. It was first defined in 2016 with the name “HTTP/2 Semantics Using the QUIC Transport Protocol”, later renamed “HTTP-over-QUIC”. It was renamed by the IETF in 2018 as HTTP/3. It has the same semantics as HTTP/2 but uses Quick UDP Internet Connections (*QUIC*) instead of TCP/IP for the transport layer. This allows for lower latency on HTTP connections. HTTP/2 runs on one TCP connection, so it is possible that all streams are blocked by packet loss detection and retransmission. QUIC uses multiple streams over User Datagram Protocol (*UDP*), and each stream has independent packet loss detection and retransmission, so in error cases, only one of the streams is blocked. [17], [21]

Hypertext Transfer Protocol Secure (*HTTPS*) is an encrypted version of HTTP. It uses Secure Socket Layer (*SSL*) or its successor Transport Layer Security (*TLS*) for bidirectionally encrypted communication between client and server. HTTPS has the same semantics as HTTP, with minor differences in URIs. HTTPS uses “https” scheme instead of “http” and the default port number is 443. [22]

HTTP/0.9 has been deprecated since 2014 on servers that support HTTP/1.1 [23]. Use of HTTP/1.0 should be minimal because of the newer HTTP/1.1 version. A growing number of servers are adapting to HTTP/2 and HTTP/3 protocols. HTTP/2 is used on 35.5% of the top the ten million web pages and HTTP/3 on 27.0%. HTTPS protocol is used as a default on 84.5% of those ten million web pages. These percentages are from November 2023. [24]

Methods are a central part of HTTP protocol. They are used to define the action type of a request. Methods can optionally be safe and/or idempotent. A safe method should not have any action other than data retrieval. GET and HEAD methods are, by definition, considered safe. Idempotent methods have the feature that multiple identical requests have the same side effects as a single request would have. In other words, re-execution of idempotent methods should not change the result. GET, HEAD, PUT, DELETE, OPTIONS, and TRACE are idempotent by definition. However, it is important to note that even if specification defines these methods as safe or idempotent, implementation of a service might still have errors leading to side effects or unwanted behaviour. [16]

HTTP protocol defines a common set of methods, which can be extended with custom methods defined separately by clients and servers. HTTP/1.1 specification contains all currently defined methods: [16]

- The **OPTIONS** method is used to request information about communication options. It allows the client to know options and requirements related to the resource or capabilities of the server without action or retrieval of the resource.
- The **GET** method retrieves the resource pointed by request URI. The get method can be conditional, depending on whether any of the “Modified-Since”, “If-Unmodified-Since”, “If-Match”, “If-None-Match”, or “If-Range” header fields have been included in the request. GET method can also be partial if the “Range” header field is included.
- The **HEAD** method retrieves metainformation about the resource defined in the request URI. The response should be identical to GET except that HEAD does not return message-body. This reduces network usage if the only relevant information about the resource is the meta information.
- The **POST** method requests the server to accept an entity in the request as a new subordinate of the resource in the request URI. This means that either a new

entity is created into resource collection identified by the request URI, or a resource identified by the URI is updated. The request URI should point to an existing resource, and the server decides how the request is handled, usually depending on the request URI.

- The **PUT** method requests a new resource to be stored in the request URI. The new resource is provided as an enclosed request entity. If the resource pointed out by the request URI already exists, the new entity should be considered as a new version of it. This means that the PUT method should either create an entirely new resource to the URI or replace the resource in the URI.
- The **PATCH** method is not part of the original HTTP/1.1 specification. It was defined in 2010 in a separate specification, but it has become an important part of used HTTP methods. It is used to partially update resources defined in the request URI based on a set of changes described in the request entity. If the resource pointed by the request URI is non-existent, the server may create a new resource to that location if it is logically possible upon the given patch entity. [25]
- The **DELETE** method deletes resources defined by the request URI. The action may be overridden by human intervention or by other means. The client cannot be sure that the action is carried out even if the response's status is successful. The server should not return successful status unless at the time it intends to delete or move the resource to an inaccessible location. No extra action should happen if the resource defined in the request URI is non-existent.
- The **TRACE** method performs a message loop-back along the path to the designated resource. The final recipient of the request should reflect the message to the client. This can be used for debugging to determine what the end participant receives. TRACE method should not have an entity in the request body.
- The **CONNECT** method establishes a tunnel to a destination defined by the request URI. After a successful connection, the server restricts the tunnel's behaviour to forwarding data bidirectionally until the tunnel is closed. Tunnels are often used to create end-to-end virtual connections through proxies. [19]

POST, PUT, and PATCH methods are similar but have fundamental differences regarding different meanings of request URI and actions depending on it. In the POST method, the URI identifies the resource responsible for handling the entity in the request. In contrast, in the PUT method, the URI identifies the resource that is the request entity. PATCH is similar to PUT but contains only changes regarding the resource in the entity of the request. Most importantly, PUT is idempotent; POST and PATCH are not. Using the PUT method provides improved reliability and reduces side effects. [16]

HTTP status codes are another central part of HTTP communication. Status codes are three-digit numbers that are part of the response message. They describe the result of the request and the semantics concerning the response. Status codes are divided into five classes that categorize responses by the first number. Two latter numbers do not have any categorization meaning. Valid status codes are within the range of 100 to 599, although the client and server can extend them. Status code classes are presented below: [19]

- **1xx (Informational)** status codes provide an interim status for communication connection or request processing before completing the requested action and sending a final response to the client.
- **2xx (Successful)** status codes indicate that the request was received, understood, accepted, and handled.
- **3xx (Redirection)** status codes indicate that further actions are required to be taken by the user agent to fulfil the request. An example reason for redirection status is that the resource might be moved to a different URI. Redirections may lead to a cyclical path, which the client is required to detect and intervene.
- **4xx (Client Error)** status codes indicate that the client sending the request has made an error. The response should contain an explanation of the error situation.
- **5xx (Server Error)** status codes indicate that it is aware that an error has occurred or is incapable of performing the requested action. The response should contain an explanation of the error situation.

Each class of status codes contains more specific codes defined by the latter two numbers. Detailed status code gives exact reasoning for a response, even without a response message.

3.3 Application programming interfaces

The most central web architecture concept regarding this work is the application programming interface (*API*). They are a way to create machines that communicate with each other using HTTP. Data sent and read by these machines must also be encrypted so that no malicious entities can access sensitive data by hijacking the data traffic between services. Different cryptographic technologies are a way to achieve this. APIs and their data must be available for use by the correct entities, and here, different authentication and authorization technologies are used. Authentication is used to determine who the user is and if they have access right in the first place, and authorization determines what parts of data provided by the API they have access to [26].

The Web provides a way to view its resources or data via services: search engines, weblogs, online stores, and much more. These services can be accessed, used, and viewed through a web browser on an end device, *client*, so that the data is presented in a human-readable way through a User Interface (*UI*). This human-readable data is presented by Hypertext Markup Language (*HTML*) pages. [27]

In the same manner, different programs and services can access these data services and use the data provided by them. This part of the Web is called programmable Web. The difference between these is that in the latter, data is in raw format, meant to be read by programs and not humans. Programmable Web presents data mainly in Extensible Markup Language (*XML*) -format, but other formats such as JavaScript Object Notation (*JSON*), plain text, and binary documents are also being used. [27]

Different web services communicate via application programming interfaces or shortly APIs. APIs work as interfaces connecting other web components to their services by handling requests and responses, as shown in Figure 5. [13]

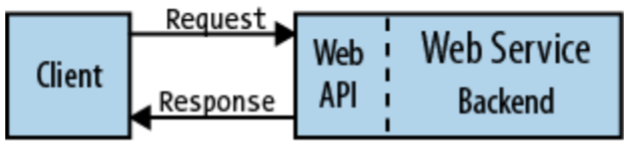


Figure 5. *Web API within client-server interaction.* ***[13]***

API architectures have properties that are used to differentiate, classify, and evaluate them. There are numerous amounts of these properties, but some commonly used when it comes to network-based applications are *performance*, *scalability*, *simplicity*, *modifiability*, *visibility*, *portability*, and *reliability*. [28]

Architectural style can significantly affect performance since component interactions can be a dominant factor in efficiency. However, applications cannot avoid basic costs in achieving application functionality. For example, if the same data is handled in different systems, the application cannot avoid transferring the data between them. [28]

Performance can be further divided into two categories: *network performance* and *user-perceived performance*. Network performance is measured by communication attributes:

throughput, overhead and bandwidth. Application architectures have an impact on network performance by the number of interactions required per user action and the granularity of the data transmitted. [28]

On the other hand, user-perceived performance is measured by the impact that actions have on the person using the application. It is primarily measured by latency and action completion time. Architecture styles often affect these, but most importantly, they impact each other. Latency optimization often has side effects that reduce completion time or vice versa. Therefore, trade-offs are required when considering architectural design. [28]

Scalability means architecture's ability to handle many components and interactions between them. It is also impacted by the frequency of interactions, how evenly they are distributed over time and the requests' handling. Scalability can be improved by decentralizing interactions to many components, simplifying the components, monitoring the system, and controlling interactions upon that. [28]

Simplicity measures the complexity, understandability, and verifiability of a system. The main principle for architectural styles to improve simplicity is the separation of concerns. It allows for functionality allocation within components. When individual components and functionality allocations can be made smaller, components will be easier to understand and implement, improving simplicity. The principle of generality applied to architectural components also enhances simplicity by decreasing variation in the architecture. [28]

Modifiability is one of the critical properties in network-based architectures, more importantly, dynamic modifiability, where changes and updates are made to applications without stopping the entire system. Modifiability is the easiness with which developers can make changes to the application. In an ideal utopia, created applications match the requirements perfectly, but even then, the software needs to be modified when requirements change. In network-based systems, components are often distributed, requiring fragmented and gradual changes, after which old and new implementations must work in coexistence. Overall modifiability can be divided into *reusability*, *configurability*, *customizability*, *extensibility*, and *evolvability*. [28]

Evolvability measures how much the implementation of the component can be changed without negatively impacting other components. Static evolution depends on how well application implementation enforces architectural abstraction, but dynamic evolution can be influenced by architecture style through maintenance constraints and application state location. [28]

Extensibility is the ability to add new functionality to the existing application system, with dynamic extensibility meaning adding functionalities to already deployed systems. Application architecture style can improve extensibility by reducing coupling between different components. [28]

Customizability means the architectural element's ability to temporarily specialize its behaviour and service for a single client component without impacting other client components. Architectural styles that support customization can help with scalability and simplicity. Service components must implement the most frequently used functionalities directly, and the client will define special services. [28]

Configurability means post-deployment modification of application components or their configuration. It allows the use of new services or data types, for example. Configurability is related to extensibility and reusability. [28]

Reusability means that some components, data elements, or connectors can be used in other applications as they are without modification. It is often achieved in architectural styles by reducing coupling between components. [28]

Visibility, in the case of network-based applications, is the ability of a component to mediate or monitor the interaction of two different components. Visibility can improve system's performance, scalability, reliability, and security. Architectural styles may influence visibility by restricting interfaces or allowing for monitoring. [28]

Portability is the ability of software to be run in a different environment. Architectural styles that move the code with data being processed often induce portability. [28]

Reliability, when considering architectural styles, is the ability of architecture to withstand partial failures on components, data, or connectors without leading to failure at the system level. Architectural styles can improve reliability by reducing the scope of failure, allowing monitoring, avoiding singular failure points, and enabling redundancy. [28]

3.4 Representational state transfer

In late 1993, the usage of the Web began to expand from scientists and researchers to regular people, and commercial use gained more interest. The rapid growth of Web's usage combined with poor characteristics of early HTTP and limitations in deployed architecture would outgrow the Internet's capacity. There was a need to design an architecture style that would be built on existing Web protocols and would allow the Web to grow. Roy Fielding designed *the Representational State Transfer* (REST) architectural style to fulfil that. [28]

Roy Fielding defined the Representational State Transfer, shortly REST or RESTful, in his dissertation “Architectural Styles and the Design of Network-based Software Architectures” in 2000 at the University of California, Irvine [28]. REST is not a web technology but rather an architectural style describing principles and constraints to define an ideal model of interactions within the Web. It attempts to maximize components' independence and scalability while minimizing latency and network communication. REST is defined initially as a set of coordinated architectural constraints.

The first constraint of REST architecture is *a client-server* constraint that is based on the separation of concerns principle. It means that user interfaces are separated from data storages. This separation allows for improved portability of user interface between different platforms and better scalability of servers because of simplified structures. It also allows for Web components to be evolved separately. Figure 6 shows web architecture with the first constraint used. [28]

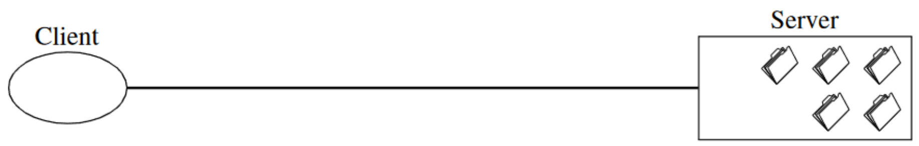


Figure 6. *Client-server system.* ***[28]***

The second constraint for REST is having *stateless* interaction between the client and server. Each request must contain all necessary information for the server to understand it. This improves the reliability, visibility, and scalability of the architectural style. It is easier for the system to recover from partial failures, which improves reliability. It enhances visibility because possible monitoring mediators do not need to know anything before the request is monitored to understand it fully. Statelessness also improves scalability since servers do not have to manage resources across multiple requests, which makes the system simpler. As a trade-off, the stateless constraint might reduce performance since repetitive data has to be sent to the server, resulting in per-interaction overhead. Figure 7 shows web architecture with the first two constraints in use. [28]

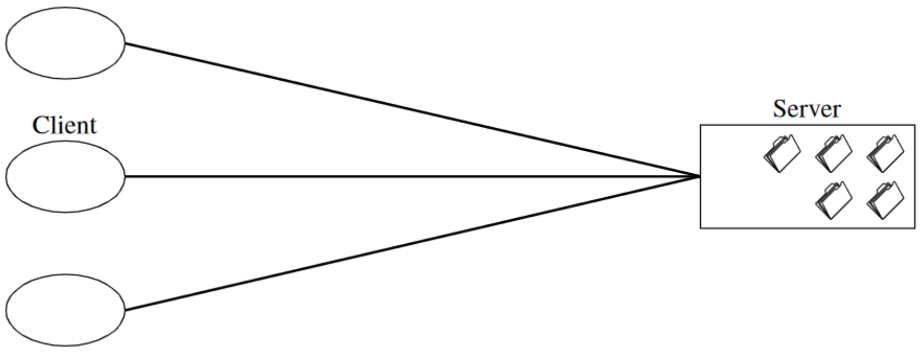


Figure 7. *Stateless client-server system.* ***[28]***

The cache is the third constraint of the REST architecture. It adds the possibility of cache storages that can reuse earlier response data for a new request equivalent to the earlier one, resulting in a speedup response. Response data must be labelled as either cacheable or non-cacheable. Caching possibly improves scalability and performance since it can reduce the average latency of a series of interactions and partially or completely eliminate some interactions. As a trade-off, it can reduce reliability since the data might be different already on the server compared to the cache. Figure 8 shows web architecture with the first three constraints in use. [28]

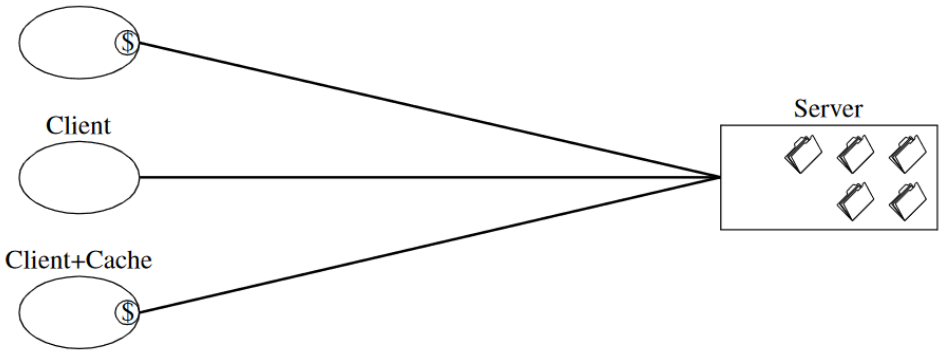


Figure 8. *Stateless client-cache-server system.* ***[28]***

The fourth constraint of REST architecture is *a uniform interface*, in which different data services' interfaces are uniformly designed. This behaviour distinguishes REST from other architectural styles. The generality principle applied to component interfaces improves the visibility of interactions and simplifies overall system architecture. The uniform design also means that implementations are separate from the services they provide, improving evolvability. The trade-off with uniform interfaces is that they reduce efficiency because standardized data form is used rather than application specific. REST is designed to be optimized for common Web cases, being large-grain hypermedia transfers. Figure 9 shows web architecture with the first four constraints in use. [28]

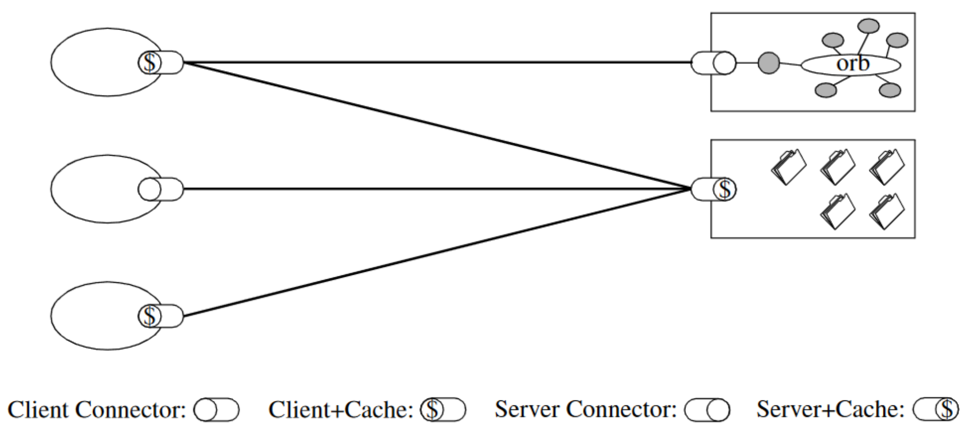


Figure 9. *Stateless and uniform client-cache-server system.* ***[28]***

The fifth constraint of REST further fulfils requirements for Internet-scale architecture. *Layered system* constraint allows the system to be created from multiple hierarchical layers, from which each component cannot see further than the immediate layer they are interacting. This layering reduces overall system complexity and improves scalability through intermediary components that handle infrequently used functionalities. Layered systems, on the other hand, increase the latency and overhead of data processing. This can be countered in network-based systems with shared cache intermediates. At organizational domain boundaries, they can provide improved performance and security. Figure 10 shows web architecture with the first five constraints in use. [28]

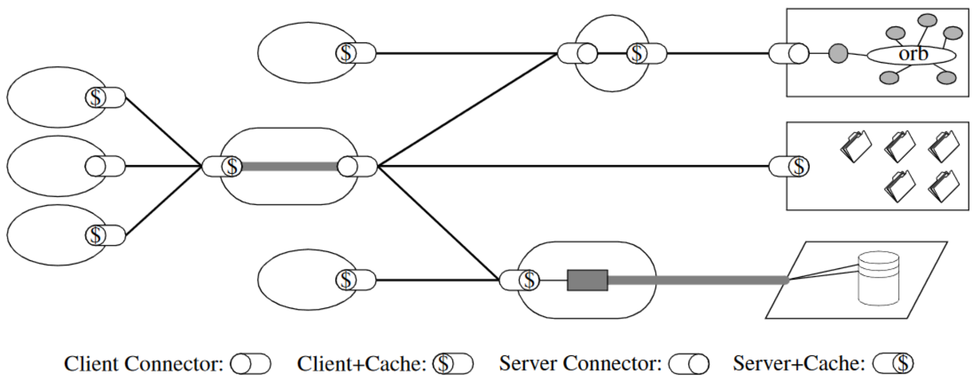


Figure 10. *Layered, stateless, and uniform client-cache-server system.* ***[28]***

The sixth and final constraint of REST is *code-on-demand*. It allows clients to download scripts and applets, extending their functionality. It improves the extensibility and simplicity of clients but reduces overall visibility. Therefore, it is the only optional constraint of REST architecture. Optionality allows, for example, code-on-demand to be used only within an organizational domain, and a general case would still have the desired behaviour. This way, the benefits and disadvantages of code-on-demand can be restricted to a known realm in the overall system. Figure 11 shows RESTful web architecture with all of its constraints. [28]

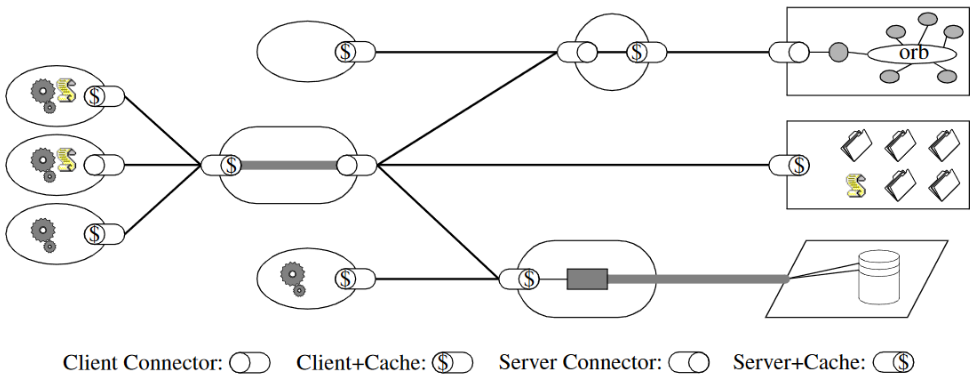


Figure 11. *Layered, stateless, and uniform client-cache-server system with code-on-demand. Full REST architecture style.* ***[28]***

The fourth constraint, the uniform identifier constraint, consists of four sub-constraints [28]. The first one is *the identification of resources*, which states that each web-based resource can be addressed by a unique identifier, for example, URI. [13]

The second sub-constraint is named *manipulation of resources through representation*, meaning a resource can be represented differently depending on the client. Users can view a resource on a browser as an HTML page, and the same resource can be a JSON object in programmatic interaction. The main idea is that representation creates a way to interact with the resource but is not the resource itself. [13]

The third of the sub-constraints is *self-descriptive messages*. It states that the interaction messages, requests, and responses must contain all the necessary information. A client can represent the desired state of resources in request, and the server can represent the current state of resources in response. Additional metadata regarding resources can be transferred inside message headers. [13]

The fourth sub-constraint is *hypermedia as the engine of application state* (*HATEOAS*), which states that a resource's representation includes links to related resources. It means that all subsequent requests that may be made are discovered within the response containing links. [13]

REST APIs represent their resources with URIs. Communication messages are sent in HTTP format. Most modern-day programmatic APIs use XML or JSON as a data representation format. Loads of design rules further address how REST APIs should work and represent their data, but those are out of the scope of this research. [13]

3.5 Authentication and authorization

Authentication is a process to recognize the user or client of an API. Its target is to discover the client's identity so that only legitimate and intended parties can access the data. Authentication can be done in multiple ways such as certificates, once retrieved API keys or user credentials and therefor it affects API design greatly. [29]

Authorization is a process to determine which data entities the recognized client has been granted access to, if any. If the client is not authenticated, no authorization can happen since the client is unknown to the API and access to the data is declined. Authorization is not necessary if there are no different levels of access for users, for example, visitor and admin data routes. [29]

Authorization can be implemented as pre-emptive or just-in-time. Pre-emptive authorization grants access to all required APIs during authentication, and that access grant, such as a token, is then used with all APIs. Pre-emptive authorization imposes a privacy issue that APIs can possibly see access grants related to other APIs. Just-in-time authorization combats this by granting access to each API individually when required. Just-in-time authorization encloses privileges to be viewed by relevant APIs only, but this could create loads of noise, decreasing performance if there are many APIs in use. [29]

There are many authentication methods available to choose from, and they influence API interface, implementation, overall performance, privacy statements, and environment infrastructure. Therefore, it is essential to analyse different authentication methods while designing the API interface since the chosen method needs to be respected in API interface design. [29]

Some most used authentication and authorization methods are:

1. **API Key**. It is a unique identifier to authenticate the client. They are typically used to identify a project within the API rather than a human user. [30]
2. **Bearer token**. It is an opaque string without meaningful information to its user, the client. Used tokens can be unstructured, such as hexadecimal characters, or structured, such as JSON web tokens. [30]
3. **Basic auth** and **Digest auth**. They work based on username and password information. The difference is that basic auth sends username and password joined by a single colon together encoded with base64 encoding, whereas digest authentication applies a hashing function to the authentication information before sending it to the API. [30]

4. **OAuth**. OAuth 1.0 and OAuth 2.0 are industry standard protocols for authorization developed by the Internet Engineering Task Force (*IETF*) OAuth Working Group. They provide a way for applications to access protected data from API without the need for users to disclose their credentials to the consumer service. [30]

5. **Certificates**. Certificates are digitally signed records that attest to the truth of something or ownership of something. Certificates can be used to secure services and APIs. [31]

Many more technologies are available for securing APIs and their data, such as OWASP, OpenID, and Tupas. This work is going to concentrate on the most relevant authentication technology considering this project: Bearer token with JSON Web Tokens.

3.6 Web Tokens

Web tokens are a way to provide authentication and optionally authorization to a private resource on the Web. A token is a piece of data that has no use or meaning on its own, but combined with some tokenization system, they provide a secure way to authorize the user requesting access. [32]

Web tokens are not dependent on the system they are used upon. They can be used in request headers, GET or POST requests, or sessions. It also allows separate servers or third-party services to handle token signing and verification, providing versatility and scalability to the system. Tokens do not require a username and password, which removes crucial security risks. They may be granted for only a period, which helps manage resource access. Tokens are also stateless, which provides scalability. User roles and permissions can be transmitted within the token payload, so they can also handle authorization. [32], [33]

3.6.1 Simple Web Token

Simple Web Tokens (*SWT*) are the most rudimentary web token format. For example, they are designed to transmit simple assertions formatted and compact, added to the HTTP header. The assertion can be represented as name-value data pairs. The only mandatory name-value pair is “HMACSHA256”, which must be the base64 encoded SHA 256 HMAC value of other pairs in the SWT. This verification value prevents the SWT from being tampered with by third parties since the private key used in the SHA 256 HMAC process is agreed upon and known only between data-exchanging parties. Other name-value pairs have also been defined, and whilst not being mandatory, they have proven their utility in a variety of assertion frameworks. [34]

1. **Issuer** identifies the party that issued the SWT.
2. **ExpiresOn** defines the moment when the SWT is not accepted anymore. This timestamp is recorded as the number of seconds passed 1970-01-01T0:0:0Z (Midnight 1.1.1970) in Coordinated Universal Time (*UTC*), also called the Unix time [35], [36].
3. **Audience** identifies the SWT's intended audience. The intent is that if a consumer receives a SWT that has a different audience than expected, the consumer will reject the SWT.

3.6.2 JavaScript Object Notation

JavaScript Object Notation (*JSON*) is a textual syntax representing structured data used to interchange between programs. It uses braces, brackets, colons, and commas to structure representable data into JavaScript's object-like entities. It uses a uniform character set for all its data. JSON syntax is not a complete specification of data interchange, and meaningful communication between producer and consumer requires further agreement on semantics regarding the particular use of JSON. Therefore, JSON can be viewed as a framework on which such semantics can be built upon. [37]

JSON provides notation for seven types of values, which are shown in Figure 12: object, array, string, number, true, false, and null. The first two of these are collections of more values. The rest are singular values, with the last three being absolute values, meaning they cannot be anything else. A string is a sequence of Unicode characters wrapped inside quotation marks. A number is defined as a sequence of decimal digits without leading zeros in the base 10 system. The possible fractional part is separated using the decimal point. [37]

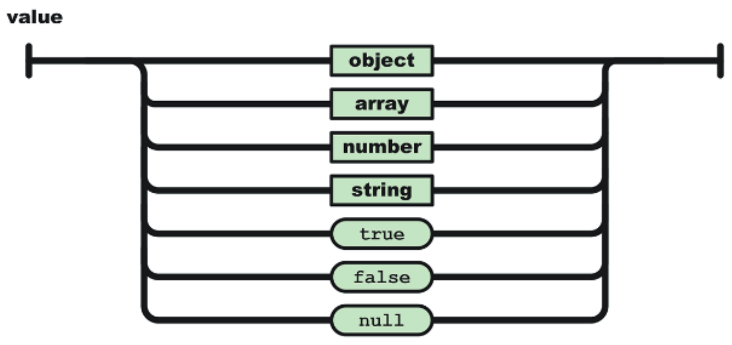


Figure 12. *JSON values* ***[37]***

Objects are represented by curly brackets with zero or more name-value pairs inside, separated by commas. The name of a pair is separated from the corresponding value by a colon. All names must be strings, but there are no other restrictions on names. They do not need to be unique, and they do not have significance in ordering pairs. Values can be any of the ones belonging to JSON syntax. The whole format of JSON objects is shown in Figure 13 below.

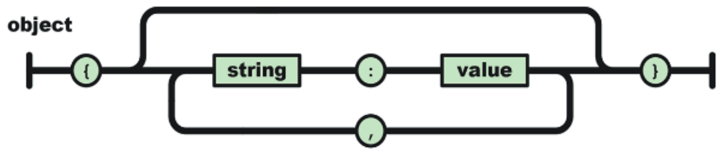


Figure 13. *JSON object format* ***[37]***

Array structure represents a list of values, which can be any belonging to the JSON syntax. An array is defined by square brackets surrounding zero or more values. The values are separated from each other using single commas. The JSON syntax does not define any meaning regarding the order of values in the array, but arrays are often used in situations where there is some semantics on the ordering. Full presentation about the array format is in Figure 14 below. [37]

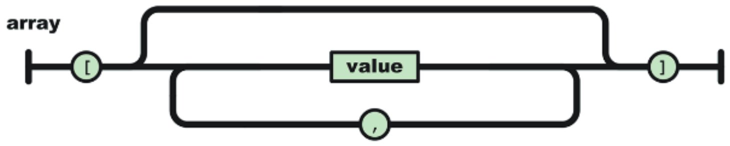


Figure 14. *JSON array format* ***[37]***

Complete JSON structures can present any data object. They are simple in format, which means they are easy to understand and handle.

3.6.3 JSON Web Token

Encoded JSON syntax structures are used as the basis for JSON Web Tokens (*JWT*), which are compact, URL-safe ways of representing *claims* transferred between two communicating parties. JWT headers describe cryptographic operations that have been used on JWT claims. A complete JWT is represented by base64url encoded values that are separated using period characters. A JWT may be enclosed to another JWT-based structure to create a nested JWT. This structuring enables nested signing and encryption to be used. [38]

Claims are pieces of information represented in name-value pairs. The claim's name is always a string, and the value is an arbitrary JSON syntax type value. Claim names within the same JWT must be unique. No claim is strictly required for JWT to be valid by official specification, so requirements are application and context dependent. Claim names are grouped into three classes: registered, public, and private claim names. [38]

Internet Assigned Numbers Authority (*IANA*) JSON Web Token Claims registry defines and maintains registered claim names and their definitions. All claim names are short because JWTs are designed to be compact. The registered claims are: [38]

- **"iss"** (Issuer) claim is identification for the issuer of the JWT. The value of this claim is a case-sensitive string.
- **"sub"** (Subject) claim identifies the subject of the JWT. Claims in a JWT are typically statements about the subject. The value of this claim is a case-sensitive string.
- **"aud"** (Audience) claim identifies the intended recipient of the JWT. If the principal processing the JWT does not identify itself from the audience claim, it must reject the JWT. The value is an array of case-sensitive strings or, in the case of the JWT having only one audience, a case-sensitive string.
- **"exp"** (Expiration time) claim states the expiration time for the JWT, after which the JWT must not be accepted. Implementations may use some leeway with the expiration time to account for clock skew. The value must be a number in Unix time.
- **"nbf"** (Not before) defines the time before the JWT must not be accepted. The implementation may use some leeway. The value must be a number in Unix time.
- **"iat"** (Issued at) states the timestamp when the JWT was issued. It can be used to calculate the age of the JWT. The value must be a number in Unix time.
- **"jti"** (JWT Identifier) provides a unique identifier that can be used to prevent the same JWT from being used repeatedly. The value must provide a negligible probability that the same value cannot accidentally be used on another JWT. The format of the value is a case-sensitive string.

The users of JWTs can define public claim names. New claim names should be registered in the IANA JSON Web Token registry or have a value containing collision-resistant names. Either way, the definer of the claim needs to take precautions so that they are in control of the namespace they use for defined claim names. [38]

The producer and consumer of a JWT agree upon private claim names. These claim names are not part of the claim name registry or public claim names. These claim names should be used with caution since they are subject to collisions, unlike registered and public names. [38]

JavaScript Object Signing & Encryption (*JOSE*) is a framework that provides a collection of standardized methods for securing JWTs and their claim sets. JWTs can be represented as JSON Web Encryption (*JWE*) by digitally signing them or JSON Web Signature (*JWS*) objects by encrypting them. All operations related to these also expect JSON Web Key (*JWK*), a cryptographic key in JSON data structure, to be used with them rather than random parameters. [39]

JWTs contain JOSE headers that describe cryptographic operations applied to the JWT and optional additional information. Header data consists of name-value pairs separated by commas from each other. A single colon separates the name from the value. Headers are encoded with base64url. JOSE headers depend on if the JWT is presented as JWS or JWE. [38]

3.6.4 JSON Web Algorithms

JSON Web Algorithms (*JWA*) are cryptographic algorithms used with JSON Web Signatures, Encryptions and Keys. The algorithms considered here are all used with JSON Web Signatures since only they are relevant to this work. [40]

Signing algorithms consist of two defining parts: algorithm function and hashing function. These both can be seen in algorithm short names. For example, RS256 is an RSASSA-PKCS1-v1_5 algorithm with SHA-256 hashing function. [41]

The used hashing function determines the length of the resulting hash. Commonly used functions are SHA-256, SHA-384 and SHA-512, which all are part of hashing algorithm family SHA-2. The level of security that each of them provides is half of the hash's length; for example, SHA-256 provides 128 bits of security, and SHA-512 provides 256 bits of security. SHA-512 is more secure than SHA-256, but also SHA-256 is currently unbreakable. [41]

The Hash-based Message Authentication Code (*HMAC*) is the most basic algorithm function. HMAC is a symmetric function, meaning tokens are issued and validated with the same cryptographic key. It is also deterministic, meaning that the same JWT header and payload will generate the same signature. Symmetric algorithms are not suitable for multi-party web communication. HMAC algorithms are shortened as HS; for example, HMAC with SHA-256 is HS256. [40], [41]

The generally used JWT signing algorithm is RSASSA-PKCS1-v1_5, shortened as RS. It is an asymmetric and deterministic algorithm based on RSA. While it is still safe for encryption, weaknesses in its signature validation have been found, meaning that it is prone to attacks if implemented falsely. In JWA specification, the implementation requirement for the RS256 algorithm is “recommended”. [40], [41]

Compared to RSASSA-PKCS1-v1_5, a similar RSA-based algorithm RSASSA-PSS (shortened as PS) is recommended over it to get increased robustness against the possible attacks [42]. It is asymmetric and probabilistic, meaning that the same header and payload will generate different signatures each time. It is also probabilistic in a good way since randomness used in signing is not critical for security. This makes the algorithm more straightforward to implement successfully. JWA specification sets implementation requirement for PS256 as “optional”. [40], [41]

Compared with RSA algorithms, Elliptic Curve Digital Signing Algorithms (*ECDSA*, shortened as ES) perform better in security but are more complex to implement. They are faster at signature generation than RSA algorithms but are usually slower in signature validation. They are asymmetric and probabilistic, but they are probabilistic in a bad way since their security relies on randomness. Elliptic Curve Cryptography (*ECC*) is more challenging to break than RSA cryptography, so shorter keys can be used. An elliptic curve key of 256 bits provides about the same security as an RSA key of 3072 bits. JWA specification sets implementation requirement for ES256 as “recommended+”, which means its importance will increase in future [40]. [41]

Edwards-curve Digital Signature Algorithm (*EdDSA*) could be better than ECDSA algorithms, but it has yet to be widely supported. A probabilistic nature is required for ECDSA's security, which is not optimal. EdDSA algorithm is deterministic and uses randomness only in private key creation. EdDSA is performant in both signing and validation and avoids many known vulnerabilities. Two of its variants have been added to the JOSE specification [43]. [41]

3.6.5 JSON Web Signature

JSON Web Signatures are JSON-based objects secured using Message Authentication Codes (MACs) or digital signatures. JWS objects are composed of the JOSE header, JWS payload, and JWS signature parts. JOSE header is a union of the JWS-protected header and JWS unprotected header. JWS objects can be serialized using *JWS Compact Serialization* or *JWS JSON Serialization*. In both, all parts of JWS are encoded using base64url, except for the JWS unprotected header in the JWS JSON serialization structure. [44]

JWS compact serialization consists of a protected header in UTF8 format, JWS payload and JWS signature, all encoded with base64url and concatenated with dots. JWS's unprotected header is unused. This structure is shown in Figure 15 below. [44]

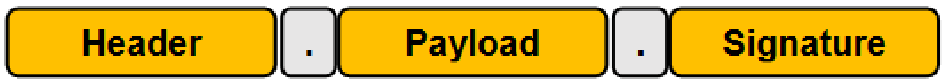


Figure 15. *JSON Web Signature using JWS compact serialization.* ***[45]***

JWS JSON serialization represents the JWS as a JSON object. It consists of one or both: a JWS-protected header in UTF8 format and a JWS unprotected header with JWS payload and JWS signature. JWS unprotected header is not base64url encoded, but other values are. Figure 16 shows the structure with JSON names. [44]

```
{
        "protected": base64uerl( UTF8( JWS Protected Header ) ),
        "header": JWS Unprotected Header,
        "payload": base64url( JWS Payload ),
        "signature": base64url( JWS Signature)
}
```

Figure 16. *JSON Web Signature using JWS JSON Serialization.* ***[44]***

JWS are used to identify the sender of the JWT. Although JWS is encoded, it is not encrypted, so any intruder can construct the plaintext JWT.

Next is an example of JWS in JWS compact serialization. The JWS-protected header describes the type of the object to be a JWT and the algorithm to be HS256, meaning the use of HMAC SHA-256 algorithm for signing. This is shown in Figure 17 below.

```
{
    "typ":"JWT",
    "alg":"HS256"
}
```

Figure 17. *Example JWS protected header in plain text.* ***[44]***

This header is then turned to UTF8 format and encoded using base64url, which results in the following value:

eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9

The JWS payload of the example JWS is shown in Figure 18 below.

```
{
    "iss": "joe",
    "exp": 1300819380,
    "http://example.com/is_root": true
}
```

Figure 18. *Example JWS payload in plain text.* ***[44]***

Base64url encoding this payload results in the following value:

eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ

The encoded header and payload in ASCII format are used as input for the signing algorithm, concatenated using a dot in the format of "encoded-header.encoded-payload". The symmetric key used in this example is shown in Figure 19 below.

```
{
    "kty":"oct",
    "k": "AyM1SysPpbyDfgZld3umj1qzKObwVMkoqQ-EstJQLr_T-1qS0gZH75
          aKtMN3Yj0iPS4hcgUuTwjAzZr1Z9CAow"
}
```

Figure 19. *Example symmetric key used in HMAC SHA-256 algorithm.* ***[44]***

The yielded value from running the signing on the given input encoded with base64url is:

dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk

Combining the full JWS in the format shown in Figure 15 results in the following value:

eyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9

.

eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ

.

dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk

This full JWS can now be used to authenticate an API, for example.

3.6.6 JSON Web Encryption

JSON Web Encryption represents encoded JSON-based data structure. They consist of six possible parts depending on the serialization. These are JOSE Header, JWE Encrypted Key, JWE Initialization Vector, JWE AAD, JWE Ciphertext, and JWE Authentication Tag. JOSE Header has three sub-parts: JWE Protected Header, JWE Shared Unprotected Header, and JWE Per-Recipient Unprotected Header. JWEs can be consisted of using either *JWE Compact Serialization* or *JWE JSON Serialization*. [46]

JWE Compact Serialization consists of JWE Protected Header in UTF8 format, JWE Encrypted Key, JWE Initialization Vector, JWE Ciphertext, and JWE Authentication Tag, all separated with dots from each other. Each part is separately base64url encoded. This serialization is shown in Figure 20 below. [46]

```
Base64url( UTF8( JWE Protected Header ) )
.
Base64url( JWE Encrypted Key )
.
Base64url( JWE Initialization Vector )
.
Base64url( JWE Ciphertext )
.
Base64url( JWE Authentication Tag )
```

Figure 20. *JSON Web Encryption in JWE Compact Serialization format.* ***[46]***

In JWE JSON Serialization format, one of the JOSE headers' sub-parts must be present with JWE Encrypted Key, JWE Initialization Vector, JWE Ciphertext, JWE Authentication Tag, and JWE AAD. All but JWE Shared Unprotected Header and JWE Per-Recipient Unprotected Header are base64url encoded. These parts are presented in JSON-style objects. Figure 21 below shows this format with names and their corresponding values. [46]

```
{
    "protected": Base64url( UTF8( JWE Protected Header ) )
    "unprotected": JWE Shared Unprotected Header
    "header": JWE Per-Recipient Unprotected Header
    "encrypted_key": Base64url( JWE Encrypted Key )
    "iv": Base64url( JWE Initialization Vector )
    "ciphertext": Base64url( JWE Ciphertext )
    "tag": Base64url( JWE Authentication Tag )
    "aad": Base64url( JWE AAD )
}
```

Figure 21. *JSON Web Encryption in JWE JSON Serialization format.* ***[46]***

JWE structures are not signed, but they are encrypted. It means that the receiver cannot identify the sender. Only the sender and the receiver can decrypt the contents of the JWE.

4. AUTOMATED SOFTWARE TESTING

Failure is unwanted behaviour or lack of wanted behaviour in software. The most significant cause of software failures is product specification. It can be deficient, contain errors, be inaccessible, or there might not be one to start with. The next most significant cause is software design. It can also be insufficient, poorly documented and communicated, or contain errors. The third most significant cause of failures is programmatic errors. Often, these are caused by complex software, rushed schedules, insufficient documentation, or plain mistakes. Many failures seem to be programmatic errors on the surface, but they can be traced down to be errors in design or implementation. [47]

One main objective of automated software testing is to find failures, fix them and make the software work as intended. Failures waste developers' time by requiring focus on tasks that do not progress the software. Tests that are run often and, most importantly, before integration and deployment phases provide early warnings of broken functionalities. [48]

It is essential to find failures as soon as possible since failures in production affect customers and can be very costly to the company by losing customers' money and time, affecting the company's reputation. The cost of failures grows logarithmically as the software goes from specification through design, coding and testing to release [47]. Although initial investment in using automated testing can be notable, it will improve quality and reduce maintenance costs in the long run. [49]

Finding failures leads to the bigger goal of testing: enabling sustainable growth of the software project. It is easy to create growth at the beginning of the project, but when the code base grows, progress becomes slower. Each row of code is a liability because it could contain failures. By probability, the more code there is, the more failures the code contains, them being noticed or unnoticed. These start to stack up and take time away from driving the software forward with new features. Making progress becomes slower or stops completely. [48]

This decreasing development speed phenomenon is also known as *software entropy*. Entropy, on a general level, is used to describe disorder in systems. Software entropy can be seen in the form of code that deteriorates. Each change in code base increases its entropy, in other words, complexity and disorder, if not taken care of properly. Such caretaking can be, for example, cleaning and refactoring. Entropy growth eventually leads to situations called *regression*, where modifying one part breaks some other parts.

Fixing failures and unwanted behaviours becomes a never-ending task, and no real progress can be made. The code base becomes unreliable, and regaining stability becomes more difficult. [48]

Tests help with this since they work as a safety net, providing stability and insurance when creating new functionalities or refactoring some parts of the code. When done correctly, they ensure that new functionality works as expected and does not affect already existing functionalities or have other side effects. They ensure that the software is sustainable and scalable. [48]

Tests can also ensure the quality of the software design. Tests and underlying production code they test are highly intertwined. It is practically impossible to create valuable tests if the code base they cover is not well designed. Tests and the code base are developed together, requiring significant effort to be put into the code base for tests to be valuable. A side effect of this is that the necessity of creating tests for production code often leads to better code design. Developing valuable tests also provides a deeper understanding of the functional requirements that the system under testing has. Although this is not the primary goal of testing, it is a good side effect. [48]

Testing can also indicate poor-quality code, especially tightly coupled code. If some parts are challenging to test, the code might need more modularity. It does not work vice versa; easily testable code does not automatically mean good-quality code. [48]

On the other hand, it is vital that testing is done correctly and efficiently. Each line of code can contain failures, which also applies to the testing code. The more testing code there is, the more failures there are in the tests, and the more they increase the upkeep and maintenance costs of those tests and the project. Tests must be maintained the same way as production code, so tests must be designed, implemented, and conducted in a way that they provide maximum value. Often, it would be better to not write a test rather than write an inefficient test. [48]

Inefficient tests also have other effects. They might give false positive alarms if coupled heavily with production code implementation. False positive alarm is a situation where a test case informs of a found failure even though there is not one. A significant number of false positives lowers the trustworthiness of the tests, leading to lowered motivation for implementing and using the tests. Inefficient tests might not reveal failures they are designed to reveal due to incorrect implementation. It leads to failures passing through to production deployments. Complexity and slowness decrease the value of the tests since they require more maintenance and running time. Higher running time decimates the number of times the tests are run, which decreases their potential of catching failures early. [48]

In contrast to inefficient tests, good tests maximize the benefits of testing while minimizing the effort put into them. They reveal failures effectively without giving false alarms. They are integrated into every part of the development cycle to ensure early discovery of failures. They are automated, repeatable, easy to implement, quick, and easy to run. They are readable, maintainable, and trustworthy, so they get used to their maximum potential. Good tests provide a safety net against regression, and they have high resistance to refactoring. [48], [50]

4.1 Automated testing concepts

There are different fundamental concepts on how software tests should be constructed. Black box and white box testing are two opposite software testing concepts. They provide a frame of how tests are developed and constructed. Black box testing implements tests without knowing the inner implementation of the code. These tests are generated purely based on the public interface, so they do not test how something is done but what the code does. These tests are derived from software requirements and specifications. In the opposing white box testing, the application's inner workings are tested, the how-part. These tests derive from source code. [47]–[49]

Both concepts have their pros and cons. White box testing could be tightly coupled with implementation, which generates brittle tests that might provide false positive alarms and do not resist refactoring. Brittle and fragile tests tend to fail even if the testable functionality has changed only internally, or the change is in some other functionality. On the other hand, they provide a deeper understanding of the implementation and may spot errors not visible from the external specification. Black box testing has the opposite pros and cons, meaning tests resist refactoring well and are not fragile but do not provide a more profound vision of the code under testing. There could remain scenarios and hidden effects that are not verified by tests. The optimal testing suite is created by combining both concepts in the right way. [47]–[49]

Test pyramid is a software testing concept that is used when composing a general structure of tests. It represents the proportions of different testing types in a software project. It often consists of three parts, shown in Figure 22: unit tests, integration tests, and end-to-end tests. The pyramid displays the relative quantity of given test types. Figure 22 provides a general model, which can be modified to the needs of a project. Integration tests might have greater value if the project is smaller and more straightforward, for example, a basic CRUD (Create, Read, Update and Delete) interface. On the other hand, if the project contains a lot of algorithmic, complex business-critical utility code, the number of unit tests might be greater. End-to-end tests test the system in the most production-like situation possible. It means that all shared dependencies with testing possibility are not mocked out but instead used in the tests. End-to-end tests ensure that the whole application is working together. End-to-end tests can also interact with the user interface. Depending on the available user interface, these tests are sometimes separated into own, fourth group, UI or GUI tests. It is important to note that the broader and more consuming the tests are, the fewer there should be those in the testing suite; therefore, the pyramid-shape. [48]

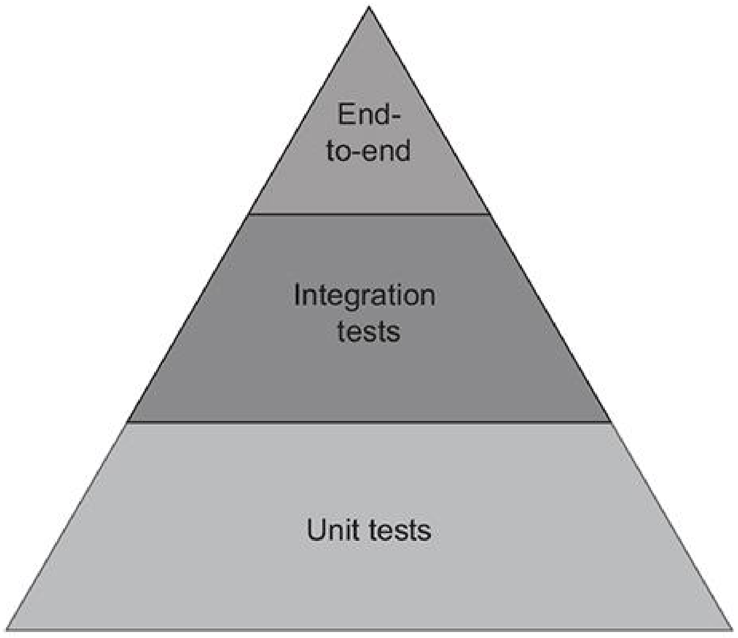


Figure 22. *Test pyramid consisting of the three primary testing types.* ***[48]***

Testing concepts are utilised when testing suites are designed and implemented. Multiple types of automated tests can be used to ensure quality on different levels of the software from different perspectives. They are differentiated by their objective, broadness, target, behaviour, and other properties. The most used ones are unit testing, integration testing, and acceptance testing. Other testing types are, for example, end-to-end testing, regression testing, stress testing, penetration testing, smoke testing, performance testing, static analyses and many more. It is important to recognise which types are necessary and provide value to the project under testing. [48]

Unit tests are the smallest and most basic type of tests. There are a lot of different definitions and nuances when defining unit tests. Essential features of unit tests are that they are quick, automated tests that verify a small piece of code called *a unit* in an isolated manner. [48]

Integration tests are the next type of automated tests. The simplest definition is that integration tests are tests that do not fit into the unit test definition. Another definition could be that integration tests verify the behaviour of multiple units working together, integrated into each other. Properties that integration tests often have are that they verify systems behaviour with external dependencies, work with multiple units of code and are not isolated or as fast to run as unit tests. Many of the other testing types fall under the definition of integration testing, but they have other more accurate definitions. [48]

End-to-end testing, also known as E-to-E, E2E or system testing, is one of these types of integration testing. The objective is to test the software behaviour from a client's perspective. They are the most comprehensive type of automated testing since it goes through the whole software. E2E tests are often the costliest tests because they are complex, challenging to maintain, and require lots of initialization work to be run. On the other hand, they provide the best insight into the software's behaviour from the business point of view. [48]

Acceptance testing or user testing is intended to test against requirements set for the software. These tests are planned based on the requirements that the user or customer has set and accepted for the software. Actual users or customers often perform acceptance testing. This way, the testing enforces the black box style: the user can use anomalous actions that reveal some unexpected software behaviour. This way, acceptance testing also tests for human factors affecting the software. Acceptance testing is usually the last step before handing the software to the customer. [51]

Regression testing is a particular type of testing. Its purpose is to verify that changes to one part of the software have not invalidated some other part, in other words, testing against regression. Unexpected side effects can be relatively common when changes to the code base are made. Coding errors cause some side effects, but some stem from subsystem integrations. [51]

Stress testing is another particular type of testing. Its purpose is to test how the system behaves when it is pushed to or over its designed capabilities. One example of stress testing is to test for database connection capability. If the software can handle *n* number of simultaneous connections, what happens when *n + 1* connections are made? What happens when a thousand or million users connect? [47], [51]

One part of stress testing is special situations. These can be low disk space, no internet connection, or slow CPUs. An example of a special situation is the turn of the millennium, also known as the *Y2K problem* and the handling of unexpected formats in years. [47], [51]

In multithreaded, concurrent software, race condition testing is valuable. It ensures the software works successfully even though some tasks or threads fail. It also ensures that synchronization between different tasks and threads is successful. [47]

The last particular type of testing is repetition testing. It is mainly used to find memory leaks by doing the same operation over and over again, such as loading and saving a file. [47]

4.2 Testing measurements

There are some ways to measure the effectiveness of tests in a project. By addressing that test can have four different outcomes, tests can be classified and measured according to the results that they provide. These four outcomes are:

- There is a failure, and the test finds it.
- There is a failure, and the test does not find it (*false negative*).
- There is no failure, and the test finds a failure (*false positive* / *false alarm*).
- There is no failure, and the test does not find a failure.

The first and last ones are those that tests aim towards. The second and third outcomes are flaws often produced by bad-quality tests. Tests that produce false results undermine trust towards the tests and make testing a burden rather than an asset. [48]

A good test can find failures without generating false alarms. A test property called *test accuracy* measures how well a test achieves this. The number of failures, or signals, is divided by the number of false alarms, or noise, that the test produces to calculate test accuracy, as shown in Figure 23. Both variables in that formula are critical; a test that can find no failures is useless, as is a test that produces only false alarms. Test accuracy property is problematic in the case that there are no false alarms or there are no failures found. [48]

$$\text{Test accuracy} = \frac{\text{Signal (number of bugs found)}}{\text{Noise (number of false alarms raised)}}$$

Figure 23. *Test accuracy formula.* ***[48]***

Some measurements do not concentrate on the test's results but on the analysis of tests and their execution in relation to the system under testing. One of these metrics is cyclomatic complexity, which describes code complexity. It is calculated by number of branches in a program or method. It can be used to describe the different paths the code can flow. This number is tied to unit testing metric branch coverage since the cyclomatic complexity value is the number of atomic tests required to test the given code. Cyclomatic complexity is calculated with formula:

$$Cyclomatic\ complexity = 1 + x$$

where x is the number of branching points. [48]

A similar metric with cyclomatic complexity is cognitive complexity. It tells how difficult the code is to read and understand rather than how difficult it is to test. Cognitive complexity is more difficult to calculate since it is human-related rather than purely mathematical, but the basic idea is to [52]:

- Ignore structures that allow readable short-handing of statements, such as the null-coalescing operator.
- Increase with each break in the code's linear flow, for example, loop structures.
- Increase with nested structures that are flow breaking. For example, two consecutive if-clauses (cognitive complexity of 2) are easier to understand than two nested if-clauses (cognitive complexity of 3).

The next measure purely used for the testing suite is code coverage, also known as test coverage, a testing metric that tells how many lines of the production code are tested by the automated tests. It is calculated by dividing the number of code lines that are executed at least in one test case by the total number of production code lines, as shown in Figure 24. [47], [48]

$$\text{Code coverage (test coverage)} = \frac{\text{Lines of code executed}}{\text{Total number of lines}}$$

Figure 24. *Code coverage formula.* ***[48]***

Branch coverage is another similar metric used with unit tests. It measures code branches traversed compared to the total number of branches, as shown in Figure 25. This metric focuses on control structures within the code, such as *if* and *switch* statements. [47], [48]

$$\text{Branch coverage} = \frac{\text{Branches traversed}}{\text{Total number of branches}}$$

Figure 25. *Branch coverage formula.* ***[48]***

Code coverage can be a treacherous metric because it is easily manipulated by refactoring the code into more compact lines, for example, by turning if-else statements into ternary operators. Compared to the code coverage way of measuring raw code lines, branch coverage provides greater insight into how well the tests test the code behaviour. Branch coverage is more challenging to manipulate with refactoring since, for example, with ternary operator usage, the same branches still exist. Even so, this metric also suffers from the same treacherousness. Figure 26 shows an example of this refactoring with ternary operator. [48]

```
// Test case
public void Test()
{
      bool result = IsStringLong("abc");
      Assert.Equal(false, result);
{

// Testble function before refactoring
// Code coverage 80%, branch coverage 50%
public static bool IsStringLong(string input)
{
      if (input.Length > 5)
            return true;

      return false;
}

// Testable function after refactoring
// Code coverage 100%, branch coverage 50%
public static bool IsStringLong(string input)
{
      return input.Length > 5 ? true : false;
}
```

Figure 26. *Refactoring effects on code and branch coverages.* ***[48]***

There are also a few other pitfalls when it comes to the use of coverage metrics. The first of them is that the tests are not guaranteed to test all outcomes. Code coverage counts all executed code lines, not tested. There can be, for example, some class member assignment that is not tested but only executed, and only the function return value is tested with assertion. Coverage metrics can hide these within them. [48]

Another pitfall is the so-called pesticide paradox: the more the code is tested, the more immune it becomes to the tests. The same tests will not reveal more failures after some time, so new tests or different testing methodologies must be implemented to find more failures. Therefore, coverage metrics generated by the tests can be good, but the efficiency of the tests can be poor. [47]

The third pitfall is that without assertion, or in other words, testing, the tests usually pass and provide code coverage, given that no exceptions are thrown. This behaviour can be achieved, for example, with function calls within the test case, but those calls or functionalities not being tested with assertions. [48]

The last pitfall is external libraries used within the code under testing. Even though code and branch coverages would be hundred per cent, there is still the possibility of different outcomes from hidden branches inside the external library. For example, a function of the external library can throw an unexpected exception with a specific, untested input value. Therefore, there can still be outcomes of the testable function that are untested and uncounted for, even with full coverage metrics. [48]

Code and branch coverages can give insight into test quality but should be used with reservation. If they are low, it most certainly means that the testing needs to be improved, but this does not work vice versa. A conclusion that unit test suites are sufficient cannot be drawn straight from the fact that these metrics provide high values since all tests can still be inefficient. Therefore, these values provide one more measure which can be used to help with unit testing development. [48]

4.3 Unit testing

Unit tests are the most basic form of software testing. They are small and fast, and they usually create the base of a testing suite because most tests are in that category. The most basic definition is that a unit test is a test that verifies the correct functioning of a small piece of code in an isolated and quick manner. [48]

Important functional concepts regarding testing are dependencies and test doubles. These two concepts are highly coupled to each other, and they are used in most other areas of unit testing. [48]

4.3.1 Dependencies and testing doubles

Testing doubles are used in test suites to replace dependencies of the code under testing. There are different types of dependencies: private, shared, volatile and external. [48], [50]

- Private dependencies are only referenced by the unit under testing.
- Shared dependencies are shared between different units under testing.
- External dependencies are not in-memory of the program, such as a database. They are often also shared, but not always; for example, the database instance could be run in its own Docker container for each test, making it external but not shared.

- Volatile dependencies require installing, setting up or configuring the machine, or they have non-deterministic behaviour. Shared and volatile dependencies often overlap; for example, a database is both, but a file system is shared but not volatile since it exists ready by default. An example of non-deterministic volatile dependency is a private class that generates random numbers.

Test doubles are most used to replace dependency classes in the system under testing, allowing for accurate testing and monitoring. There are a few types of test doubles depending on their composition and usage. These are mocks, spies, stubs, dummies, and fakes. [53]

Stubs are test doubles that are used to inject indirect inputs into the system under testing. Their production code counterparts are only used to retrieve data, and therefore stubs cannot be asserted against. Stubs can be further classified into two groups: *responders*, which inject valid data and *saboteurs*, which inject invalid data, such as errors and exceptions. [53]

Spies are fundamentally like stubs. The difference is that they can be used to observe the system's outputs under testing by capturing indirect outputs of the system under testing while being exercised. These observations are saved, and they can be used later for verification. [53]

Mocks are used to verify the indirect outputs of the system under testing while it is being exercised. They also often have the functionality of stub since they have to return values to the system under testing, but the main weight is on the verification of indirect outputs. [53]

Fakes are used to replace the dependencies for reasons other than verification of indirect inputs or outputs of the system under testing. They often have the same functionality as the production object but are much more straightforward. While being built for testing purposes, they are not used for control or verification. Usual usages are to replace non-existing production objects, too slow objects, or to eliminate deleterious side-effects from the tests. [53]

Dummies are objects that are used when neither test nor the system under testing care about it. They are often as simple as possible, for example an instance of object-class or a null object. In a sense dummies are not test doubles, but a way to satisfy the interface of the system under testing. [53]

4.3.2 Testing concepts

There are different fundamental concepts regarding unit testing. The first one is regarding the verification process of the tests. There are generally three ways to verify the correct functioning of the system under test: output-, state- / result-, and communication- / interaction- / action-based testing. They provide different approaches to viewing the correct functioning of the code. [48], [50]

Output-driven testing means the tests verify that the expected output is received with given inputs. This style works well with the functional programming style since the systems under test should not contain side effects and invariants but only input and output, which can then be solely used. This style of testing is decoupled from the implementation and has resistance to refactoring since only the public interface is used. This decoupling leads to lower maintenance costs when refactoring or fixing the production code. [48]

State- or result-based testing verifies that the system under testing and its collaborators are in the correct state after the operation under testing. State-based testing is often used with an object-oriented programming approach to ensure that objects function correctly. The downside is that state-based tests verify a more significant portion of the system's public interface under testing alongside dependencies compared to output-based tests and, therefore, are more easily coupled to the implementation. State-based tests are also often less maintainable because they contain more assertions than output-based tests. [48], [50]

Communication-, interaction-, or action-based testing is the third way to test the code. It uses test doubles to verify that different parts of the program communicate correctly. This style tends to couple heavily with the implementation since it dictates how the system under test should communicate. This coupling leads to brittle tests. Overusing this style might lead to shallow tests since everything is mocked out, and only a small portion of the program is tested. Communication-based tests are not maintainable since they require setting up the test doubles and interaction assertations, and they do not resist refactoring because of coupling with implementation. [48]

The vague and argumentative definition of unit testing has created two schools of unit testing. These groups are classical / Boston and London / Mockist. They view unit testing principles, objectives and definitions from different perspectives, so they provide different ways of creating unit testing suites. [48]

Unit tests defined on the fundamental level are tests that verify a small unit of code in a quick and isolated manner. The separation between classical and London concepts stems from the different interpretations of this definition, which leads to different styles of creating unit testing suites. [48]

In classical style, the unit is defined as a unit of behaviour. In London style, the unit is a unit of code. The interpretation of isolation also differs; the classical school has an isolation of the unit tests, and the London school has an isolation of the code under testing. The main resulting difference from this is the use of testing doubles, such as mocks and stubs. In some cases, the way to verify the results of the test differs fundamentally. [48]

Classical concept enforces the correct behaviour of the unit under testing. It is often done by using output- and state-based assertations. Only shared dependencies should be mocked to create isolated unit tests that emulate the production behaviour. This means that the production code should be used when it is possible. The use of production code provides great testing coverage and effectively reveals failures. It also verifies that the unit works as intended from the client's perspective. It resists refactoring well since the tests are not coupled with the implementation of the unit under testing. On the other hand, these tests provide alarms if a dependency is erroneous, leading to alarms in tests not directly related to the error, which leads to possible difficulties finding the failure location. With larger projects, testing might become complicated and laborious if the program consists of a large number of interconnected dependencies. [48]

The London style of testing incorporates test doubles to isolate the system under testing. This means using test doubles for all shared and mutable private dependencies, meaning only immutable private dependencies will not be mocked out. In the London system, the goal is to ensure the correct working of the unit under testing, separated from other parts of the program. This separation often leads to output- and communication-based assertation style. This style produces granular tests that accurately pinpoint and delimit the failure location. However, with extensive use of test doubles and communication-based testing, the tests might get tightly coupled to the implementation and be over-specificized, which leads to brittle tests that do not stand against refactoring [53]. This coupling leads to false positive alarms and increased maintenance costs. [48]

4.3.3 Test-driven development

A more abstract approach to unit testing is test-driven development, which can be used with both classical and London testing styles. Test-driven development is a way to emphasise the importance of testing in quality software production. It can mean a few things from a practical view: test-first development, which means creating tests before the actual code; test-driven design, where testing also drives the design of the code; or something in between. [50]

In classical programming workflow, the production code is written first, and then the tests are written to ensure the correct functioning. This is illustrated in Figure 27. [50]

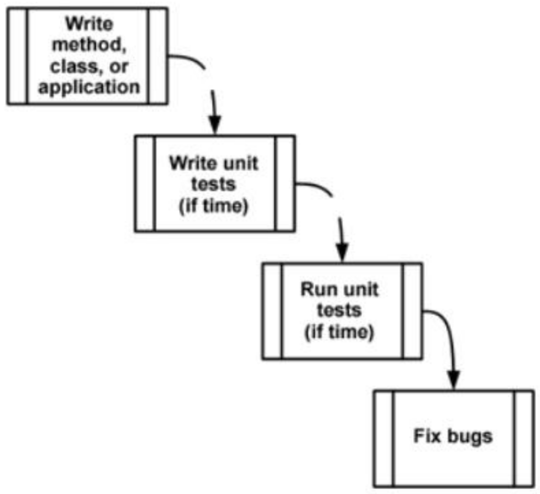

Write method, class, or application
Write unit tests (if time)
Run unit tests (if time)
Fix bugs

Figure 27. *Classical flow of software development.* ***[50]***

The cycle is flipped around in test-driven development, as seen in Figure 28. The tests are written first to fail since no production code fulfils their requirements. Then, minimum, passable solutions are written so that the tests pass. Then the code is refactored to be suitable and ensured to be still working by the tests. [50]

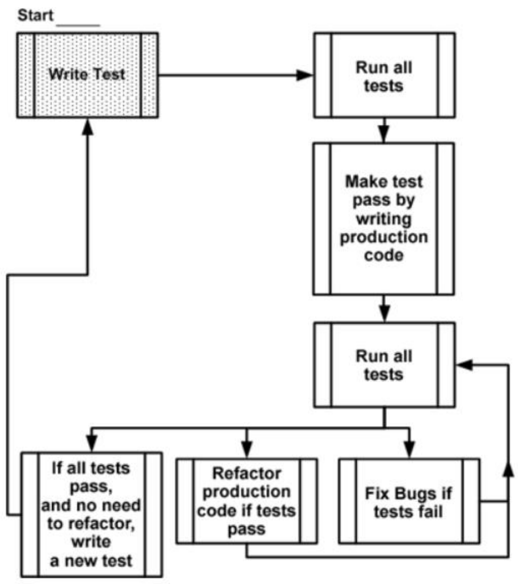

Start
Write Test
Run all tests
Make test pass by writing production code
Run all tests
If all tests pass, and no need to refactor, write a new test
Refactor production code if tests pass
Fix Bugs if tests fail

Figure 28. *Test-driven development flow of software development.* ***[50]***

The idea is that small, incremental steps taken in cyclic test-driven development allow for controlled software growth. The tests work as a safety net for the production code, emphasising the connection between testing and quality code. It must be noted that test-driven development does not ensure better quality but rather provides ways to incorporate effective practices. [50]

4.3.4 Test case design

When creating testing suites, every possible input-output combination, state, and situation cannot be tested separately because there would be infinite test cases. Therefore, equivalent partitioning must be used to group scenarios together. Equivalent partitioning is the process of narrowing the number of test cases into smaller but effective sets. Test cases are divided into equivalent classes. Each test in a class tests the same thing, possibly revealing the same failure. Therefore, each equivalent class can be reduced into one test case. There is always a risk with equivalent partitioning; too much and failures will go unnoticed, or too little and the testing suite will be inefficient and costly to maintain. [49]

Data that is used in the software is often the basis for equivalent partitioning. It is divided into valid, null, and invalid or garbage data. Valid data can be expected; for example, an integer field expects all whole numbers. Null data for that field could be null or Not-a-Number (NaN). Invalid data could be a text string or something else unexpected. Null and invalid data groups create their equivalent classes as they are, but valid data is divided into different partitions based on boundary conditions. [49]

Boundary conditions are set restrictions for the software; for example, the integer field could be limited to have values inclusively between 0 and 1000. This field's boundaries would be values 0 and 1000, creating equivalent classes for negative integers, integers within the range of 0-1000, and integers over 1000. Each boundary condition is usually also its equivalent class, so there would be five equivalent classes from these boundary conditions. Each class would have one to few test cases. [49]

There are also sub-boundary or internal boundary conditions which can be used to further partition tests. These are boundaries created by the internal specification of the software. The integer field could be in C++ type int, which is 32 bits with a maximum value of 2147483647 and minimum value of -2147483648. These could be considered as boundary conditions for testing, creating four more equivalent classes. [49]

Another separation method for testing suites is to divide tests into positive, negative, and exception tests. This separation provides aids for clear names and defined structures for the tests. Combined with parametrizing the tests, it creates clear groups of efficient tests. Parametrizing allows for a test to cover multiple inputs with one test case, which provides more thorough tests. [48]

4.3.5 Test practicalities

Creating unit tests has a lot of practicalities, which are used to improve readability and maintainability. The classical definition of unit testing emphasises the isolation of test cases, meaning tests should also have independent modification and execution. Testing classes can have constructors, they can be derived from other testing classes, or the test cases can have initialiser methods. All of these can be used to combine common functionalities between test cases, such as test double creations. Combining does decrease code replication and helps to adjust test cases if there are changes in the arrangement classes; for example, if their constructors change [50]. On the other hand, using initialization functions and test class constructors can result in high coupling of the tests since test cases are no longer independent. They might also decrease the readability of test cases since code related to them is scattered into multiple places [48].

The consistent structure of all test cases makes it easier to follow the flow of the tests and understand them. It also eases cognitive load when reading unfamiliar test cases. The typical structure is the AAA pattern (Arrange, Act, Assert). It consists of three parts [48]:

- **Arrange** is used to get the system under testing and its dependencies into the state required for the test case.
- **Act** section is used to call desired methods on the system under testing and capture the resulting output.
- **Assert** part is where the correct behaviour, output, or state of the system is ensured.

Another equivalent pattern is the Given-When-Then pattern, which works the same way as the AAA pattern. It might be easier for non-programmers to understand, but other than that, there are no differences. [48]

Another principle that improves readability and maintainability is consistently naming test cases. The commonly used naming structure consists of the method under testing, scenario, and expected result combined with underscores [50]. For example, if the method name is GetAuthorizationHeaderValue, the scenario is that the authorization header is requested multiple times and the expected result is the use of the previous token with latter requests, the resulting test case name would be ”GetAuthorizationHeader-Value_MultipleCalls_UsePreviousToken”.

Another structure for naming test cases is plain English sentences. For example, the test case above could be named “Use_previous_token_with_consecutive_authentica-tion_calls”. This way, the behaviour of the test and the system under testing is easier to understand, even for non-programmers. This style reduces the cognitive payload required to understand the test case. It also highlights the testing of behaviour rather than implementation. The use of the method name in the test case name also couples the test to the implementation. [48]

Although plain English naming provides many benefits, the three-part naming system is widely used. It also constructs each test case name similarly, unifying test cases and improving readability and understandability. Because this system is widely used, it allows new persons to quickly understand the tests and get to work quicker, which reduces maintenance costs.

Another widely used naming convention is naming the system under testing “SUT” inside the test case. This convention provides a unified, simple, and understandable way of recognizing the most essential object in the test case, which improves readability and maintainability. [48]

4.4 Integration testing

Integration tests are automated tests that do not fulfil some of the requirements for being a unit test. They often combine many software units to test that the parts work together and the software works as a whole. They are often slower, more complex to create and automate, broader than unit tests, and they often need more configuring. [48], [50]

Integration tests often collaborate with out-of-system dependencies, such as databases, to ensure that they work as expected in production environments. Only unmanaged out-of-process dependencies should be mocked to verify that the communication stays the same between the systems. [48]

4.5 Continuous integration

Continuous integration is a software development process in which each team member frequently integrates their local work to a single, shared, and controlled source code repository. It consists of a set of practices and procedures that are triggered by usually a change in source code and lead to either producing a tested and analysed working build or giving fast feedback on broken functionality. [54], [55]

The first of these practices is maintaining a single, shared repository. Everything necessary for building the program must be there, but other essential things, such as configurations or documentation, can also be stored there. The repository should have a mainline where most of the work is done. Different branches are functional, but overused can cause problems. [54]

The next practice is to have an automated, fast, and self-testing build job run on a separate machine. Building the program should happen with a single action from the user, for example, running a script or pushing a button. The build should include everything required to get a clean machine to have a working instance of the software. Automated environments are often used, such as MSBuild with .NET. [54]

The process should automatically run automated tests to ensure the working of the build. Automated tests provide fast feedback about the state of the build. Different analyses can also be run, such as SonarQube static analysis. These can reveal vulnerabilities, inaccuracies, and weaknesses. [54]

Keeping the build fast is vital so developers get feedback on their work and can continue different tasks. A multipart build pipeline can help with this. The pipeline consists of many stages, which run in sequence. Example pipeline could consist of three parts: building, running unit tests, and running slower tests such as integration tests, end-to-end tests, and static analyses. By dividing the process into these stages, developers get fast feedback from the quick-running stages, and the slower stages ensure that the build is viable in the production environment. [54]

An essential part of continuous integration is to fix the build immediately if it breaks. One main principle is to be able to develop the program on a known stable base, a working build. When the build breaks, this stable base of development also breaks. [54]

The last practices of continuous integration relate to the visibility of the build. Everyone should be able to see what is happening in the development and build process so that the progress rate and possible challenges are known across teams. The latest executable should also be readily available for everyone involved with the software so that it can be efficiently run and tested by anyone involved. [54]

Values that continuous integration creates are reduced risks, reduced repetitive processes, generation of deployable software, enhanced project visibility, and enhanced confidence of the development team. The downside is the maintenance costs of the integration workflow. [56]

Continuous integration helps discover defects earlier, allowing them to be fixed earlier. It also allows for the health of the software to be measured and monitored over time. Reducing repetitiveness saves time and effort. Continuous integration ensures that the build process is automatic or easy to run and runs the same way every time. This frees developers to do higher-value work. [56]

Deployable software is the most tangible part of the process for clients and customers. The importance of creating a deployable software version quickly at any time cannot be overstated. Continuous integration pipeline detects problems immediately, they can be fixed as soon as possible, and a working deliverable can be created. Without this process, the failures may go unnoticed, they are fixed later, technical dept might accumulate, and releases might be delayed. [56]

Continuous integration allows for measuring and monitoring the health of the project. Project visibility allows noticing trends in build success, software quality, failures, and vulnerabilities. Measurements and monitoring provide temporal and just-in-time data, which can be used to make effective decisions. [56]

Continuous integration can also improve the confidence of the development team. Frequent integration, building, and testing provide fast feedback. This feedback allows developers to know the impacts of their work and reduces the fear of creating regression in different parts of the program. [56]

5. DESIGN AND IMPLEMENTATION

The software design process is either linear, like the waterfall model, or cyclic, like the scrum methodology. The waterfall model handles one step of software development fully at once before moving to the next one. Cyclic models have an iterative nature. When minor improvements are made to any part of the software, changes are reviewed and validated, and the next version is planned upon the previous one. [57]

In all process models, software design begins by defining the requirements and constraints for the program. Requirements come from the problem that the software is trying to solve, often defined by end users. These requirements must be achieved within some constraints that define the solution space. Constraints can be explicit or implicit, and they can be discovered or introduced in the course of the design process. [57]

Other significant steps in software development processes are architecture design, implementation, and testing. Architecture design consists of creating a system-level structure for the software. The implementation part is where the production software is created based on the design. Implementation is then validated and verified using different testing methods. [57]

5.1 Requirements and constraints

The software that this research aims to construct is the verkkotietopiste interface module that sends construction plans to Traficom. The basis for this is defined in Section 2. The module, its design and implementation are influenced by development and production environments and other software working with it.

First, constraints are set by the production environment that is used with the whole MicroSCADA product, which the module will be part of. It is used only on Windows machines. The part of the main MicroSCADA product where the module will be integrated uses C# and .NET Framework version 4.8. Those must be used for this module to be compatible with the main product. Using them allows for a seamless development and deployment process.

The module uses external libraries to achieve the required functionalities and a more fluent development workflow. These libraries must be licensed with an Apache-2.0 license [58] or something less restrictive. The licenses of external libraries must also be copied and sent to the customer with the final product. This way, the license requirements are lawfully fulfilled.

This client and the server communicate using HTTPS, HTTP1.1 and TLS 1.3 versions. Since this module is a client for a REST API, the API specification dictates many of the technologies.

Data transfer to and from the server happens using JSON data format. Authentication is conducted using signed JWTs. The JWT must contain the following claims: *iss*, *sub*, *aud*, *iat*, and *exp*. *Iss*, *sub*, and *aud* values are static ones received from Traficom via separate request. *Iat* and *exp* values are filled dynamically for each authentication request. [59]

The JWS component is created from the JWT using the RSA key with the RS256 algorithm, which is short for "RSASSA-PKCS1-v1_5 using SHA-256"-algorithm [40]. Successful authentication grants the client an access token, which is used with every request to authorize access to the requested data object [59].

Other technical specifications from interface specification are related to data formats. The first one is the data type for different geometries. GeoJSON must be used, which is a JSON-based data type used for geometrical data. Another specification concerns coordinate systems. The ETRS-TM35 coordinate system must be used for all XY-plane coordinate points. Z coordinates are not stored on the server. [59]

Internal continuous integration systems create the rest of the restrictions related to software building and testing. The build server uses Visual Studio 2017 (version 15.0), which means that MsBuild of that version must be used for development. It also restricts the C# language version to C# 7.0. The tests are run with the MsTest framework, so automated tests must use MsTest v2.

Since the created version of the product will be *a minimum viable product* (MVP), there are only a few required functionalities. The module must be able to identify that the Traficom server is live and be able to send construction plans when a customer starts the action.

5.2 System architecture

The overall system architecture consists of the MicroSCADA X DMS600 server, where the DMS600 product is installed. Relevant parts of the installation are shown in Figure 29 below. These consist of a database, DMS600 Core, DMS Service and the Traficom interface module. The module connects via the internet to the Traficom centralized information point's API.

The database is the central place where DMS600-related data is stored. Each component connects to it by itself. DMS600 Core is used to create and handle the construction plans sent to Traficom via the interface. DMS Service handles the life cycles of the modules and is also used to configure, monitor, and control them. Traficom interface is the client module which is created in this research project. It is used to communicate with the Traficom centralized information point REST API.

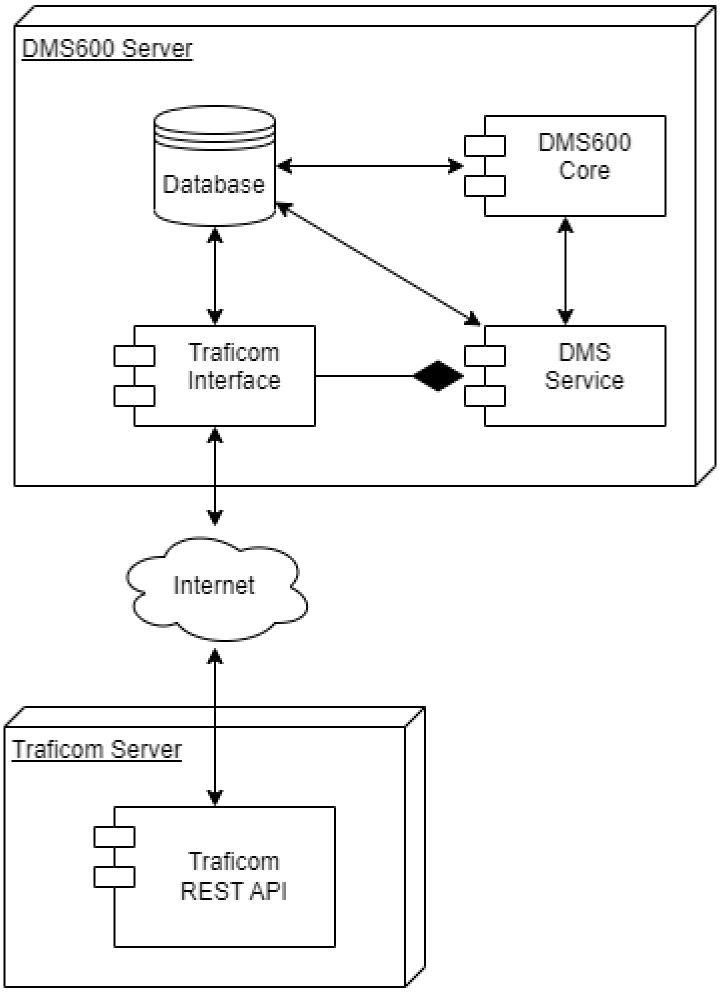


Figure 29. *Overall system architecture.*

The Traficom REST API implements CRUD operations for its data objects and JWT-based access token authentication for them. The base URL for all requests is https://api.verkkotietopiste.fi/api/external/. Relevant paths and operations for this project are [59]:

- **POST /getToken** is used for authentication. *Payload* is the JWS structure described in Section 5.1. If the authentication is successful, an access token is received with the response from the server.

- **PUT /plan** is used to create or update construction plans. The construction plan is transferred in the request's body. If the plan already exists, it is overridden with the new plan, and if not, a new one is created.

The API also has paths for updating and deleting plans. It also has paths for different types of objects. Those are not used since they are not necessary for the implementation of required features.

5.3 Programmatic decisions

The complete module diagram can be viewed in Appendix A. This chapter will contain only the most relevant parts of it.

The overall architecture of the software is not the main point of interest in this work, so it is discussed briefly. The design is influenced by hexagonal architecture, which stems from the layered architecture style combined with the domain-driven design style. The primary design strategy separates domain logic, process control, and service connections into their own classes. This separation creates a modular, maintainable, and testable structure. [48]

The software contains four connections to shared dependencies and one to volatile external dependencies. These are connections to DMS Service, database, Traficom REST API, file system and coordinate transform class. The connection to the parent program, DMS Service and the interface module is shown in Figure 30. DMS Service connects to modules through *DMSServiceComponentBase* interface. *DMSServiceConnector* class implements is. It works as a mediator between the DMS Service and the module. It does not contain any business logic.

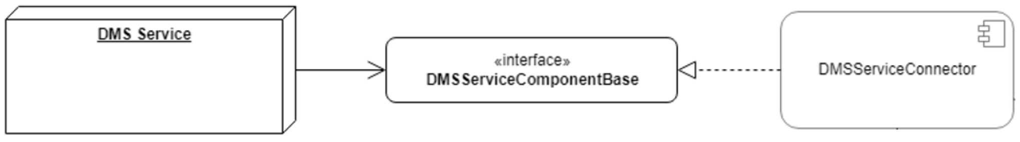


Figure 30. *Connection with MicroSCADA DMS600 DMS Service program.*

Another external dependency is the persistence database, from which the construction plan data is retrieved. Figure 31 shows classes related to that. PlanRepository class provides a simple interface to read construction plans from the database. It returns the data in an enumerable container containing the plans. Connection to the database server is done through the IDbClientWrapper interface, which provides functionality to handle the database. It is a wrapper interface since the actual class is external and static. By using this wrapper method combined with the interface, the database connection can be mocked to allow for easier testing.

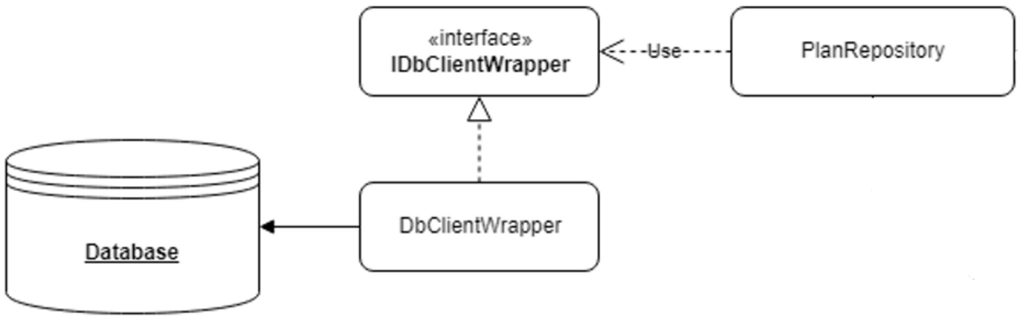


Figure 31. *Module desing for persistency service and database connection.*

The third service to an external dependency is the service to the Traficom REST API, as shown in Figure 32 below. Some parts of the program require authenticated access, and others do not. Therefore, there are two different classes for the connection to separate these use cases. The actual internet trafficking is done using the .NET class "*HttpClient"*. This class is static throughout the whole program, so only one instance is used as recommended by the .NET documentation [60]. HttpClient class is mockable using external libraries. By using the libraries, the testable functionality can be extended the most.

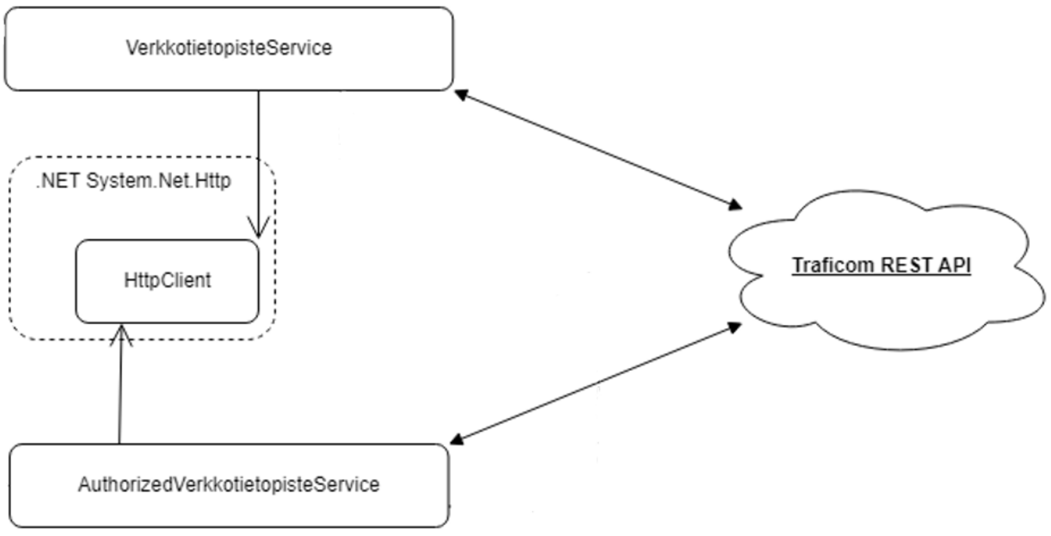


Figure 32. *Module design of service for connecting to the Traficom REST API.*

The server's file system is the fourth and final shared dependency, which stores the RSA key and base JWT information in text files. *IFileSystem* interface from the .NET *System.IO.Abstractions* namespace is used to inject this dependency and make the connected code more testable by mocking that interface in the tests.

The volatile, external dependency to the coordinate transform class is used for coordinate transforming between different projection systems. Figure 33 shows the wrapper interface and class used to access the dependency. The interface is used so that the functionality can be mocked in unit tests.

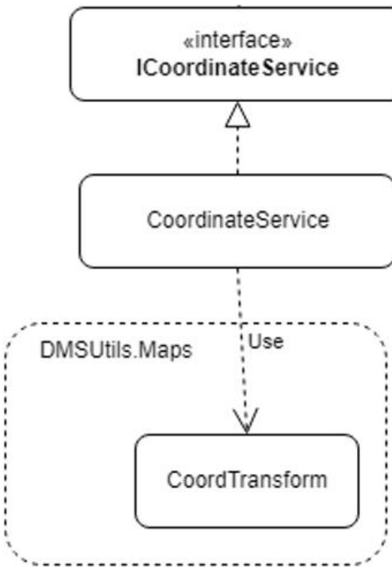
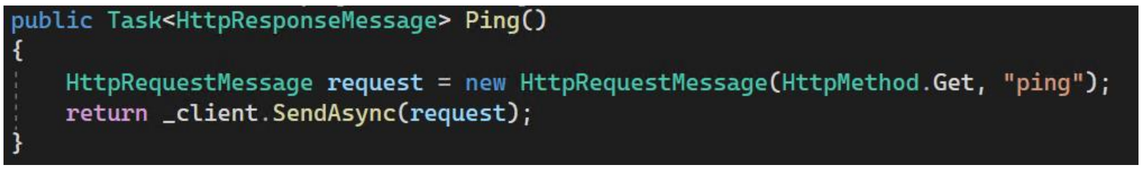


Figure 33. *Design for coordinate transform functionality.*

Essential technologies regarding the software and its requirements are HTTP communication, JSON data structure, authentication with the Traficom server, and cryptography related to the authentication. These are the core elements in web-based data transfer.

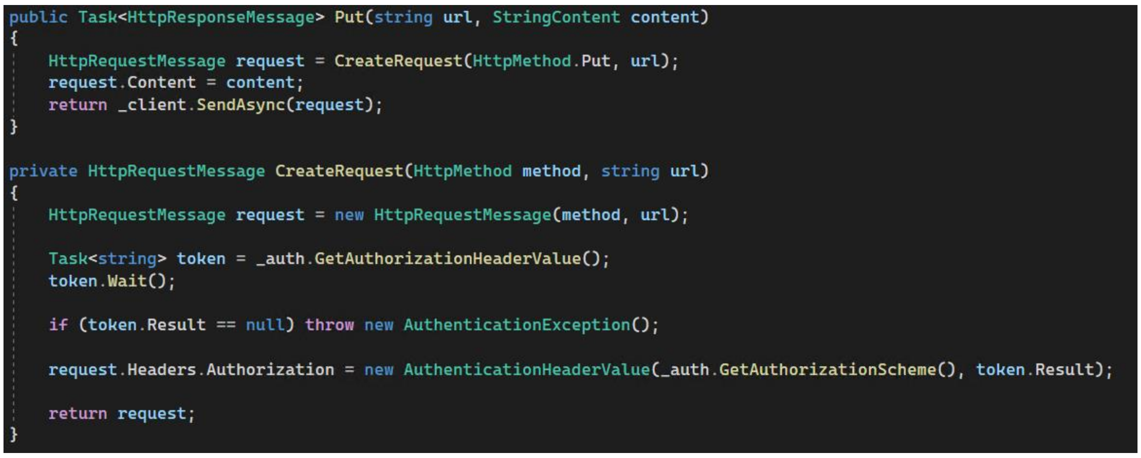
HTTP communication is handled using the HttpClient class. It provides an extensive interface to handle requests and responses. Basic usage of the class is demonstrated in Figure 34, where the implementation for ping requests is shown. The static HttpClient instance, which already has a base URL, is contained in the _client variable. The Ping method creates an HTTP request containing the GET method and “ping” path. No content is delivered in the body of the request. The request is sent asynchronously, and the resulting task is returned to the function caller to be further handled.



```
public Task<HttpResponseMessage> Ping()
{
    HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, "ping");
    return _client.SendAsync(request);
}
```

Figure 34. *Encapsulated basic unauthorized usage of the HttpClient class.*

Authenticated use of the HttpClient is shown in Figure 35 below. The Put method is part of the public interface for the AuthorizedVerkkotietopisteService class. It takes the path in the “url” parameter and payload content in the content parameter. The private CreateRequest method is used to create the HTTP request containing authorization information in the Authorization header. An instance of the class responsible for authentication functionality is contained in the _auth variable.

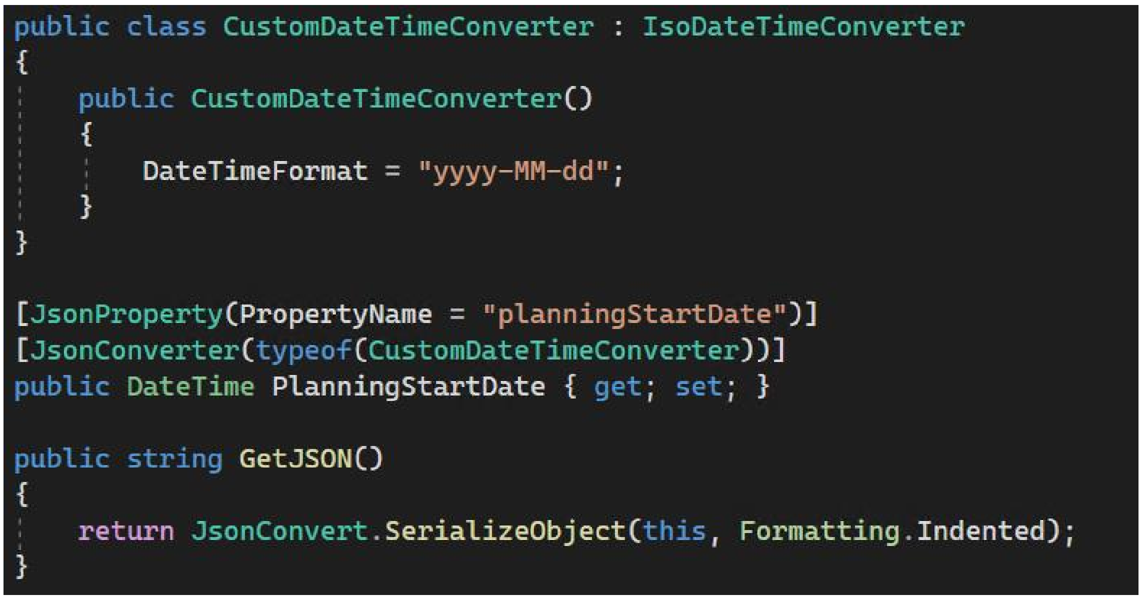


```
public Task<HttpResponseMessage> Put(string url, StringContent content)
{
    HttpRequestMessage request = CreateRequest(HttpMethod.Put, url);
    request.Content = content;
    return _client.SendAsync(request);
}

private HttpRequestMessage CreateRequest(HttpMethod method, string url)
{
    HttpRequestMessage request = new HttpRequestMessage(method, url);

    Task<string> token = _auth.GetAuthorizationHeaderValue();
    token.Wait();

    if (token.Result == null) throw new AuthenticationException();

    request.Headers.Authorization = new AuthenticationHeaderValue(_auth.GetAuthorizationScheme(), token.Result);

    return request;
}
```

Figure 35. *Encapsulatede authenticated usage of the HttpClient class.*

Both client services return responses encapsulated into task objects. This way, asynchronous execution and synchronization can be handled in controller classes.

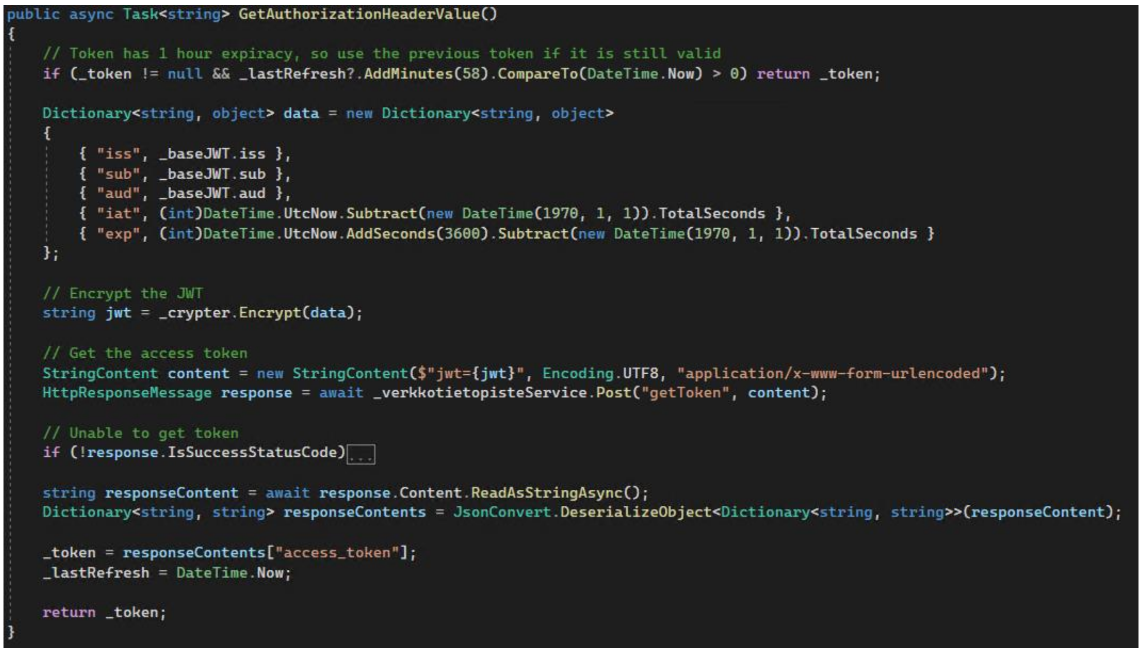
JSON data is created using Newtonsoft.Json external library. Figure 36 shows a simplified example of data creation in JSON format. When the GetJSON method is called, the object’s properties and values are generated into the new JSON object by the JsonConvert.SerializeObject function. In Figure 36, the JSON would contain “planningStartDate” with the property’s value at the time. The value is formatted from the .NET DateTime object into the “yyyy-MM-dd” format using the CustomDateTimeConverter. Using the Newtonsoft.Json library allows model classes to be simple and descriptive. They are directly linked to the sendable JSON objects, which simplifies the data flow of the software, making it easier to understand and maintain.



```
public class CustomDateTimeConverter : IsoDateTimeConverter
{
    public CustomDateTimeConverter()
    {
        DateTimeFormat = "yyyy-MM-dd";
    }
}

[JsonProperty(PropertyName = "planningStartDate")]
[JsonConverter(typeof(CustomDateTimeConverter))]
public DateTime PlanningStartDate { get; set; }

public string GetJSON()
{
    return JsonConvert.SerializeObject(this, Formatting.Indented);
}
```

***Figure 36.** Simplified example of JSON data creation using Newtonsoft.Json library.*

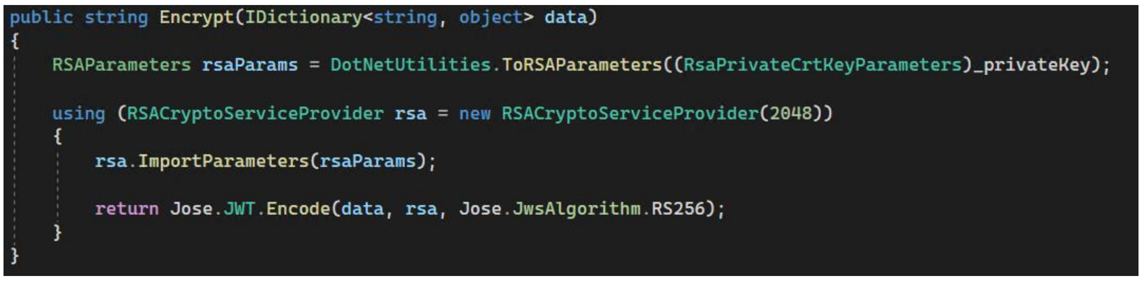
JWT-based authentication is a crucial part of the security for data transfer between internet parties. In Figure 35, the _auth instance and its GetAuthorizationHeaderValue method are used to get the access token for Traficom REST API requests. Figure 37 shows the code for gaining the token. The token is valid for one hour, so the previous token is used if it is still valid. Otherwise, a new token is requested by creating the required JWT structure, signing it using encrypting class instance in the _crypter variable, sending the request with JWS as payload to getToken path with POST method and extracting the new access token from the successful response. This token is then returned for current use and stored for further use.



```
public async Task<string> GetAuthorizationHeaderValue()
{
    // Token has 1 hour expiracy, so use the previous token if it is still valid
    if (_token != null && _lastRefresh?.AddMinutes(58).CompareTo(DateTime.Now) > 0) return _token;

    Dictionary<string, object> data = new Dictionary<string, object>
    {
        { "iss", _baseJWT.iss },
        { "sub", _baseJWT.sub },
        { "aud", _baseJWT.aud },
        { "iat", (int)DateTime.UtcNow.Subtract(new DateTime(1970, 1, 1)).TotalSeconds },
        { "exp", (int)DateTime.UtcNow.AddSeconds(3600).Subtract(new DateTime(1970, 1, 1)).TotalSeconds }
    };

    // Encrypt the JWT
    string jwt = _crypter.Encrypt(data);

    // Get the access token
    StringContent content = new StringContent($"jwt={jwt}", Encoding.UTF8, "application/x-www-form-urlencoded");
    HttpResponseMessage response = await _verkkotietopisteService.Post("getToken", content);

    // Unable to get token
    if (!response.IsSuccessStatusCode)...

    string responseContent = await response.Content.ReadAsStringAsync();
    Dictionary<string, string> responseContents = JsonConvert.DeserializeObject<Dictionary<string, string>>(responseContent);

    _token = responseContents["access_token"];
    _lastRefresh = DateTime.Now;

    return _token;
}
```

Figure 37. *Authorization functionality used to gain access token for Traficom REST API.*

Encryption is another critical part of secure data exchange on the web. The JWT data in Figure 37 is signed using the Encrypt method of _crypter instance. Figure 38 shows this signing functionality. It takes the JWT as a data parameter. First, RSA private key parameters are read using DotNetUtilities.ToRSAParameters function. Next, an in-memory RSA structure is created with the read key parameters. Lastly, Jose.JWT.Encode function from Jose.JWT external library is given the JWT data, the RSA structure and the RS256 algorithm to generate the JWS object. The generated object is then returned in string format, ready to be used.



```
public string Encrypt(IDictionary<string, object> data)
{
    RSAParameters rsaParams = DotNetUtilities.ToRSAParameters((RsaPrivateCrtKeyParameters)_privateKey);

    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider(2048))
    {
        rsa.ImportParameters(rsaParams);

        return Jose.JWT.Encode(data, rsa, Jose.JwsAlgorithm.RS256);
    }
}
```

Figure 38. *Encryption functionality used to sign authentication JWT with RSA key using RS256 algorithm.*

Programmatic decisions are used to aid with the testing of the software. The primary testing concept used in unit testing is the classical style of unit testing. This style leads to minimal use of mocks. Interface classes are only used to separate shared dependencies from the business logic of the module. These interface classes can then be mocked in the testing suite. The created interface classes are shown in Figure 44 in Appendix A.

One interface class separates the database connection from the business logic. Another interface class, ICoordinateService, separates the coordinate transform functionality. Coordinates are transformed using an internal library. The implementation uses a static class, which is underlyingly dependent on the core MicroSCADA X DMS600 software configurations. This dependency makes its functionality volatile and external, and it is easier to wrap behind an interface and mock in tests.

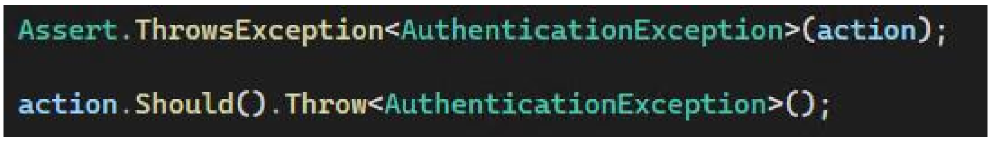
There are also other shared external dependencies shown earlier in this chapter. Those do not require creating separate interface classes since they can be mocked using third-party libraries. This allows for the most extensive testing of the code created for the module and for the communication it has with the other parties.

5.4 Automated testing

Automated testing is an integral part of quality software development. It integrates into the program's design, implementation, quality, and delivery. It should be part of each software project to ensure the successful growth of the software.

The testing aspect partly drives the design of the created data transfer module. The chosen testing strategy is the classical way of testing, emphasising the use of production instances in collaboration and ensuring the correctness of outputs and states received due to actions performed. This strategy leads to minimal use of testing doubles in unit testing.

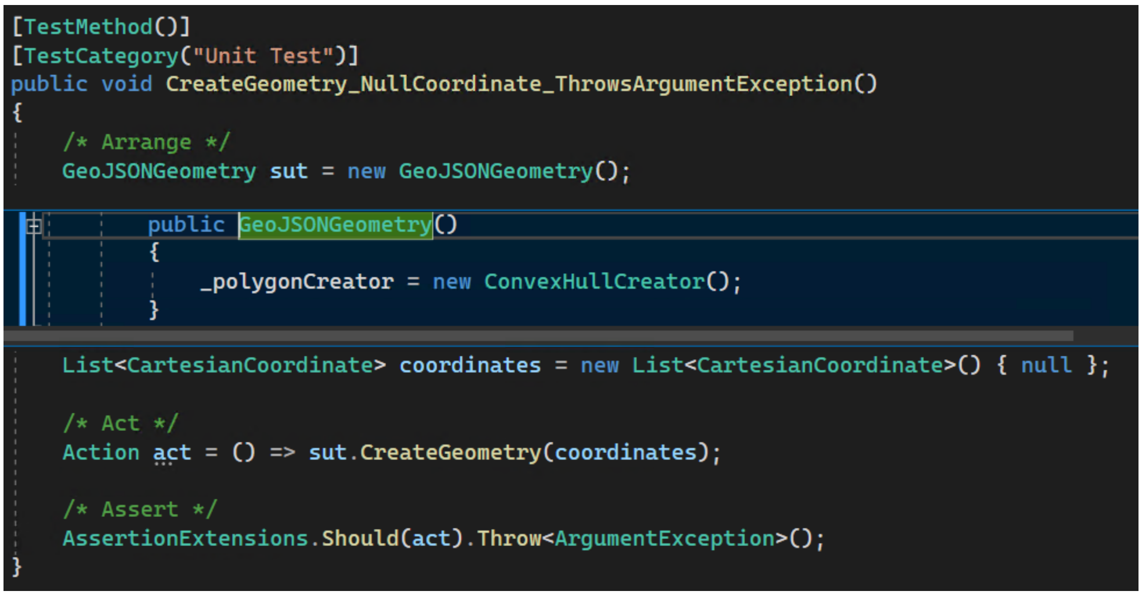
One practice that increases test readability is using assertation libraries, such as *Fluent Assertions* [61]. Assertion libraries help to structure assertions intuitively, as can be seen in Figure 39. The upper assertion is done using default MsTest unit testing tools, and the second one is with Fluent Assertions library. Even with this simple example, the library provides a more sentence-like way of asserting, which helps with the cognitive load of understanding the test cases.



```
Assert.ThrowsException<AuthenticationException>(action);

action.Should().Throw<AuthenticationException>();
```

Figure 39. *Assertion for exception with MsTest default tools and with Fluent Assertion library.*

Figure 40 demonstrates the use of collaborating production instances in a test case. The system under testing, more precisely its function *CreateGeometry,* is being tested when erroneous input is given. The function is expected to throw an *ArgumentException* error. GeoJSONCreator class uses the ConvexHullCreator class to create geometric shape from a list of coordinate points by using a convex hull algorithm. This instance of the ConvexHullCreator class is created in the constructor of the GeoJSONGeometry class. The production version is used since there is no possibility for injections. In the London style of testing, the ConvexHullCreator instance would be injected, for example, through the constructor of the system under testing, so that it could be mocked in tests.



```
[TestMethod()]
[TestCategory("Unit Test")]
public void CreateGeometry_NullCoordinate_ThrowsArgumentException()
{
    /* Arrange */
    GeoJSONGeometry sut = new GeoJSONGeometry();

            public GeoJSONGeometry()
            {
                _polygonCreator = new ConvexHullCreator();
            }

    List<CartesianCoordinate> coordinates = new List<CartesianCoordinate>() { null };

    /* Act */
    Action act = () => sut.CreateGeometry(coordinates);

    /* Assert */
    AssertionExtensions.Should(act).Throw<ArgumentException>();
}
```

Figure 40. *Test case where function is expected to throw an exception when given erroneous input.*

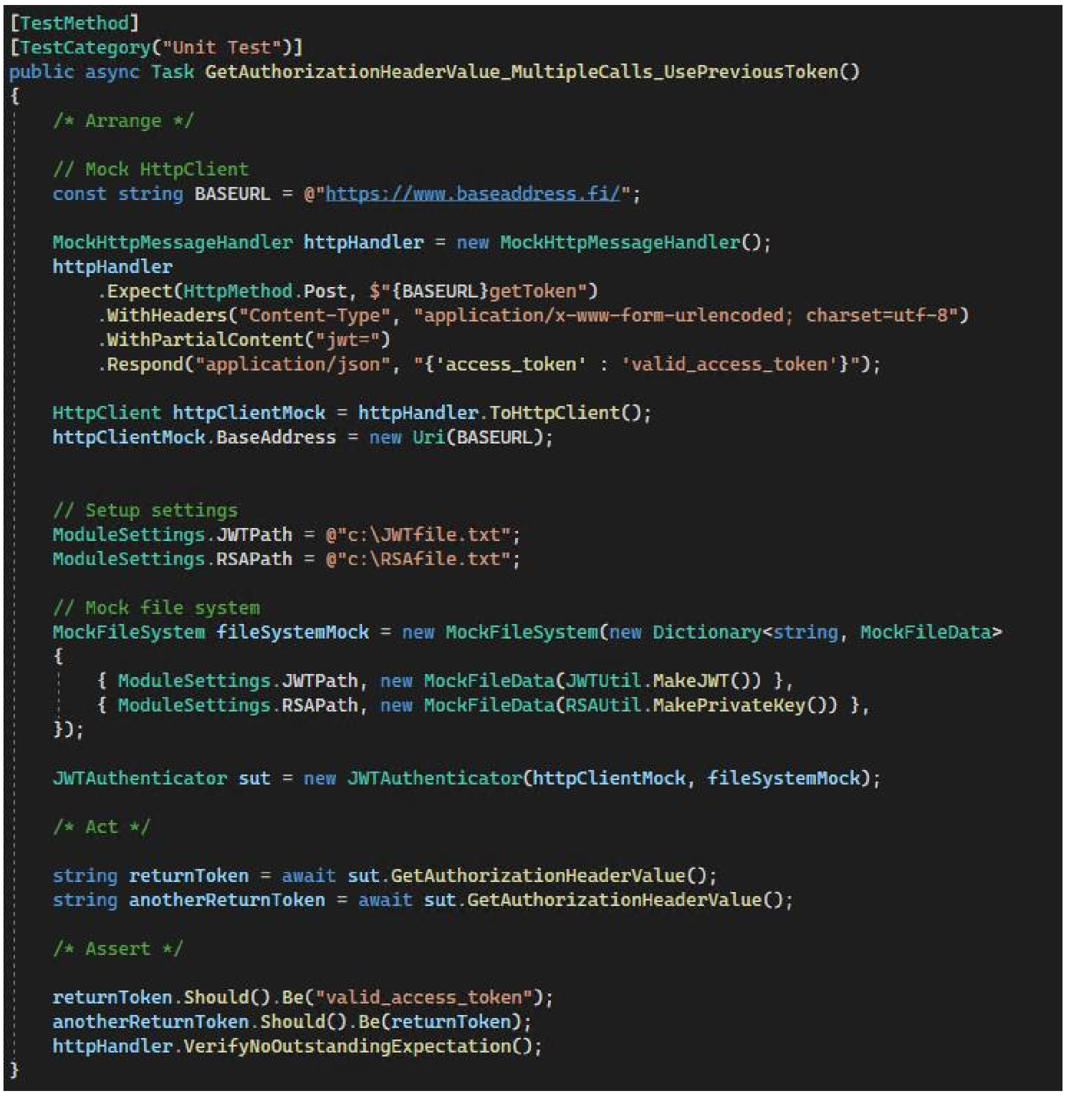
Currently, the testing suite consists solely of unit tests. Those have been the easiest to set up, and they provide the most value in the limited time given for this project. Integration tests would require extensive configuration or mocking since the database schema for the construction plan data is complex.

The most important parts of the software to be tested are domain-critical parts containing business logic. Those are data handling functionalities, which contain geometry creation, coordinate transformation, and authentication flow. Those unit tests provide the most value when integration testing is not considered. With integration testing, data reading and updating with the database, and data sending to the Traficom REST API would be the most critical parts.

Although the classical style emphasises using actual production instances, shared and volatile dependencies are replaced with test doubles to allow isolation of the tests. The current unit tests for this module mocks four dependencies shown in the previous chapter: HttpClient, file system, database, and coordinate transform. DMS Service connector is not worth mocking at this point since the value generated by mocking those would not be sufficient compared to used time and effort. Third-party library “*Moq*” [62] is used when no better solution is available.

Figure 41 shows a test case using both HttpClient and file system mocks. The test case calls the JWTAuthenticator class's GetAuthorizationHeaderValue method multiple times in a row and ensures that the access token is once retrieved from Traficom REST API and other times retrieved from in-memory persistence.

The test case starts by defining a mock for HttpClient. A third-party library, *mockhttp,* created by Richard Szalay[63], is used first to create an HTTP handler, define expected HTTP requests, and create an HTTP client with a random base address for dependency injection. After that, application-specific static in-memory settings are set. After that, the file system mock is set up using *the System.IO.Abstractions* package [64] created by TestableIO. The file system mock allows the creation of a virtual file system, which is used via the injected IFileSystem interface.

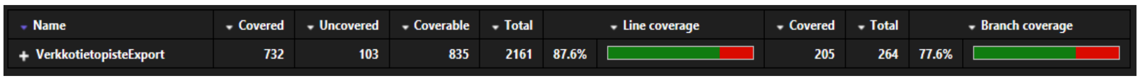


```
[TestMethod]
[TestCategory("Unit Test")]
public async Task GetAuthorizationHeaderValue_MultipleCalls_UsePreviousToken()
{
    /* Arrange */

    // Mock HttpClient
    const string BASEURL = @"https://www.baseaddress.fi/";

    MockHttpMessageHandler httpHandler = new MockHttpMessageHandler();
    httpHandler
        .Expect(HttpMethod.Post, $"{BASEURL}getToken")
        .WithHeaders("Content-Type", "application/x-www-form-urlencoded; charset=utf-8")
        .WithPartialContent("jwt=")
        .Respond("application/json", "{'access_token' : 'valid_access_token'}");

    HttpClient httpClientMock = httpHandler.ToHttpClient();
    httpClientMock.BaseAddress = new Uri(BASEURL);


    // Setup settings
    ModuleSettings.JWTPath = @"c:\JWTfile.txt";
    ModuleSettings.RSAPath = @"c:\RSAfile.txt";

    // Mock file system
    MockFileSystem fileSystemMock = new MockFileSystem(new Dictionary<string, MockFileData>
    {
        { ModuleSettings.JWTPath, new MockFileData(JWTUtil.MakeJWT()) },
        { ModuleSettings.RSAPath, new MockFileData(RSAUtil.MakePrivateKey()) },
    });

    JWTAuthenticator sut = new JWTAuthenticator(httpClientMock, fileSystemMock);

    /* Act */

    string returnToken = await sut.GetAuthorizationHeaderValue();
    string anotherReturnToken = await sut.GetAuthorizationHeaderValue();

    /* Assert */

    returnToken.Should().Be("valid_access_token");
    anotherReturnToken.Should().Be(returnToken);
    httpHandler.VerifyNoOutstandingExpectation();
}
```

Figure 41. *Test case where use of previous access token is ensured.*

Database connection uses the DbDataReader class, which has no worthy third-party libraries. Therefore, the IDbClientWrapper interface is mocked using the Moq library, and the DbDataReader class that it returns is mocked with a manually created class that implements its functionality using in-memory data handling. The coordinate transform dependency is also mocked using the Moq library.

Different measures can help to determine how effective a testing suite is. Static measurements related to testing that are used with this module are the number of tests, code coverage, branch coverage, cyclomatic complexity, and cognitive complexity. Coverage numbers are calculated using the Fine Code Coverage library [65] since only enterprise version of Visual Studio includes testing coverages. Figure 42 shows the coverage results. Complexity metrics are calculated with SonarQube static analysis.



Name	Covered	Uncovered	Coverable	Total	Line coverage		Covered	Total	Branch coverage	
+ VerkkotietopisteExport	732	103	835	2161	87.6%		205	264	77.6%	

Figure 42. *Code coverage and branch coverage generated by Fine Code Coverage library.*

The number of tests in the testing suite is 161, with 210 test cases when parametrized tests are counted as separate. Code coverage of the tests is 87,6 %, and branch coverage is 77,6 %. These are sufficient results, considering that there are no integration tests. Cyclomatic complexity for the whole module is 295, and cognitive complexity is 160.

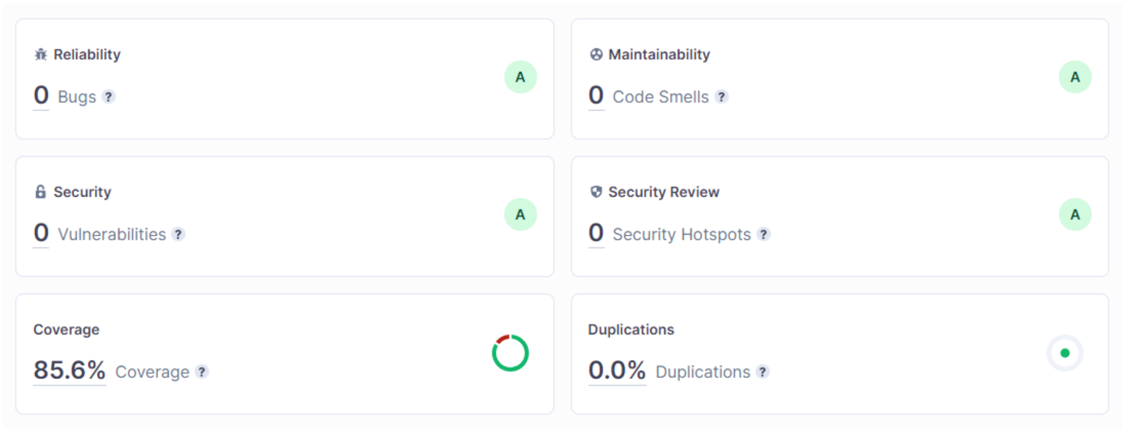
5.5 Continuous integration

The module created in this research incorporates continuous integration workflow into its development process. It consists of actions done in local environments by developers, integrating code into a shared, single repository and automated building, testing, and analysing.

The source code system is a GIT repository hosted in Azure servers. It has a main branch where the current version of the module is. Each development cycle begins with the developer creating a new branch based on the main branch. Development work is then done in that branch. Automated tests are run in the local environment to ensure quickly that the new code works.

New code is then pushed to the repository in this separate branch. A pull request is issued from the development branch to the main branch when development is finished. The pull request is automatically checked against merge conflicts by the Azure server. Before the pull request is merged into the main branch, another team member issues a review of the new code and accepts it or suggests changes. This review process is then repeated until the review is successful and there are no merge conflicts.

After this, the new code is merged into the main branch. The merge automatically triggers an Azure DevOps pipeline. It consists of setting up testing building and testing environment in an Azure agent. Then a SonarQube analysis is prepared, the project is built, automatic tests are run, and finally, SonarQube code analysis is executed. The results are then visible from the SonarCloud web view. The results for this module are shown in Figure 43 below. It shows that the code does not contain code smells, failures (bugs), vulnerabilities, security issues or duplications.



Reliability
0 Bugs ? A

Maintainability
0 Code Smells ? A

Security
0 Vulnerabilities ? A

Security Review
0 Security Hotspots ? A

Coverage
85.6% Coverage ?

Duplications
0.0% Duplications ?

Figure 43. *Results from SonarQube scanning.*

The code coverage results from SonarQube differ from the Fine Code Coverage because of the different ways they count code lines. Some lines that are considered code lines by Fine Code Coverage are not considered that in SonarQube and vice versa.

The Build and automated test job is started manually and done separately on the Jenkins server after running the Azure pipeline. This build job handles building a production version of the software, testing it using automated tests, collecting necessary executables, libraries, and other documents, creating a single zip folder from them, and moving that into shared online storage.

Emails from successful and failed pipelines, builds, and tests are sent to involved personnel. Fast feedback is received, and actions can be taken quickly.

6. EVALUATION AND FOLLOW-UP

This chapter provides an overall retrospective of the project. The use of technologies and implementation of the module regarding requirements and quality are considered, although Traficom specified many of the technologies used.

Upon this retrospective, possible improvements and next steps are considered. These would improve the module's quality and safety, improve workflow cycles, and add features.

6.1 Fulfilment of requirements

The intended functionality for the first version of this module has been implemented successfully. The list of functionalities is short, so the flow of the program is straightforward, which can be seen in the module design. Implementation difficulties raised from database handling, data reading, and creating some of the test doubles.

Database handling was complex because connections are handled in the MicroSCADA X DMS600 environment using the OdbcDataReader class. The more commonly used class, SqlDataReader, could be used, but it would require extra programming and configuration, and it would deviate from other modules in the main software.

Data reading was also complicated to implement because the database's data schema for construction plans is not in object-based, normalized form. This means that extra steps had to be made to be able to read the data into object-oriented models in the program.

Lack of normalization was why the .NET *Entity Framework* (*EF*) could not have been used for database connection and data reading, at least without significant effort. Entity Framework is designed to work well with object-oriented, normalized data.

Quality control of the project was somewhat successful. Most program parts were tested with unit tests since necessary external dependencies were replaced with testing doubles. The lack of integration tests diminishes the level of quality control. SonarQube scanning integrated into development workflow improves failure and vulnerability detection.

Regardless of the challenges, all software requirements were fulfilled within the limits of the given restrictions. Quality control of software cannot be overseen, and there is always something to be improved.

Another objective for this work was to improve knowledge of the technologies within Hitachi Energy and its employees. This objective is partly fulfilled. The project provides basic and some advanced information about the technologies and methodologies used. Distribution of that information among employees is still required for efficient learning.

The last objective of the project was to provide DSOs with software to automate infrastructure data exporting to Traficom centralized systems to enable joint construction. This objective is also partly fulfilled. The software does what is needed to fulfil required functionalities, but distribution, configuration and production deployment to customer systems must be done to take full advantage of those features.

6.2 Suitability of used technologies

The list of used technologies for this project is relatively narrow. In this section, all technologies used in this module and those public ones that affect clients from the Traficom side are evaluated.

Overall infrastructure for the interface, REST API with CRUD operations, suits this application well. Transmitted data is object-oriented, and RESTful principles incorporate authentication fluently. Provided endpoints regarding this module are designed using RESTful principles.

Used authentication, JWT signed with RSA key, is a well-suited method for this application because of its statelessness and presumably relatively sparse use. It allows for a secure but straightforward authentication flow. A more modern, JWT-based OAuth protocol could be used, but it cannot be fully argued one way or another without knowing detailed use cases and user scenarios. It could provide extended features with more flexibility, but it could also create excess overhead and complexity.

When technologies used in the module are considered, few ones could be improved. The first one is the database handling and data reading functionality. Entity Framework would improve simplicity, readability, and maintainability by providing a clean interface and hiding boilerplate code. It would also make testing easier since it has extensive test-doubling capabilities available.

The use of different testing frameworks would be another possible improvement regarding technologies. MsTest v2 is Microsoft’s framework, but there are other widely used ones, such as xUnit or NUnit. By using different testing framework, the test suite could improve in performance, cleanness, maintainability, and extensibility [66]. Differences are minor, but if the testing suite grows a lot in size over time, the improvements might be valuable.

6.3 Security considerations

Security plays an essential part in storing and handling critical infrastructure information. It has been questioned if a centralized information point is secure enough compared to the data being stored in a decentralized manner by the distribution system operators [67]. Data being stored in one place increases the risk of it all being leaked simultaneously. Jukka-Pekka Juutinen, leader of the cyber security centre agency, argued against this [68]. He stated that ensuring the security of all systems providing information would be impossible in a decentralised information system. Many infrastructure owners were not ready to improve their systems to provide the required information automatically. Hence, the other option considered was an email-based system where a mediator party handles data transfer. This system would not have complied with EU directives since information delivery would not have been possible without undue delay. It was also noted that email is not a particularly secure way of transferring information.

The verkkotietopiste.fi service, a centralized information point for construction plans that Traficom has created, has good security from an outside perspective but could have minor improvements. Without knowing implementation details, other similar authentication methods compared to JWT, such as OAuth, cannot be thoroughly discussed. Therefore, the algorithm of the current JWT implementation is the most important part.

The current implementation uses the RS256 (RSASSA-PKCS1-v1_5 with SHA-256) algorithm for JWT signing. Firstly, it must be noted that this algorithm is not broken and is cryptographically safe on today's technology, and it is widely used and defaulted in many implementations. That said, there are better alternatives which are more secure and effective, such as RSASSA-PSS, ECDSA or EdDSA algorithms. [41]

Another way to improve JWT security would be to implement a JWE structure on top of the existing JWS. This would also encrypt the JWT message. Would this be a valuable improvement, considering the costs of it is not discussed here.

Another slight improvement would be the use of better HTTP standard. Currently, the API uses HTTP/1.1. The use of HTTP/2 or HTTP/3 would provide improved performance.

The last security improvement concerns client machines and the created interface module. The client-server currently stores the secrets required for API authentication as plain text files. Encrypting or another secure storage method would improve the security of handling them.

6.4 Further development

The designed and implemented module is the first version created to fulfil minimum requirements. Quality software development is one of the key elements of the project.

Multiple aspects of testing could be improved. As mentioned, there are no integration tests in the current testing suite. By creating them, the value of the testing suite would increase since more comprehensive test cases simulate production use cases better.

Even though there are a significant number of unit tests already, creating more of them for edge cases that still need to be covered would also be beneficial. Increasing branch coverage and creating more comprehensive unit testing on data reading functionality would be beneficial.

Production code can be refactored safely when the testing suite has good coverage, and tests are not fragile. Refactoring would streamline the software and reduce maintenance costs by reducing cognitive and cyclomatic complexities. It would simplify current implementation and make new features more straightforward to implement as part of existing software. It would be easier for new developers to understand the code and get their work started.

The test-driven development style was not used with the development of the first version, but it could be beneficial in the future. It would ensure that testing stays to standard, and the design of new features would support testability.

Related to testing is continuous integration and the build pipeline. Incorporating an automatic run of tests and SonarQube analyses on every pull request to the main branch would be beneficial. It would ensure that new code is always sufficient before it is merged into production code. It is currently up to the developer to manually run branch-specific tests and analyses.

The continuous integration process consists of an automatic Azure pipeline running SonarQube analysis and a manual Jenkins build. It would remove repetitiveness and manual work if the build was automatically triggered upon successful tests and analysis from the Azure pipeline.

When testing and integration workflows are without flaws, it is easy to implement new functionalities. The created interface module is the first version, so it has minimal features. Implementing more functionalities could help clients deliver, handle, and synchronise construction plan data with Traficom systems. An example of functionality could be the possibility to read and present construction plans that are already stored in the Traficom system.

As discussed in the previous section, the module's security could also be improved. Creating better security practices around storing and handling JWT and RSA data required for authentication would be an improvement and would patch some vulnerabilities in case servers get compromised and possibly breached.

After the DSO customers have used the first version of the software, usage feedback could be collected. Upon this feedback, new features could be designed and implemented, and existing ones could be modified to suit DSO needs better. Since this software is entirely for customer usage, it is crucial to communicate with them to fix inaccuracies, provide improvements and support their objectives regarding joint construction. Continuous improvement of software requirements, system design, functionalities, and quality control ensure that the software can grow to its full potential.

REFERENCES

[1] L. Lehtiranta, J.-M. Junnonen, S. Kärnä, and L. Pekuri, *Designs, Methods and Practices for Research of Project Management*. Gower Publishing, Ltd., 2015.

[2] "Directive 2014/61/EU on measures to reduce the cost of deploying high-speed electronic communications networks." European Parliament and the Council, May 15, 2014. [Online]. Available: https://eur-lex.europa.eu/legal-content/en/ALL/?uri=celex:32014L0061

[3] "Laki verkkoinfrastruktuurin yhteisrakentamisesta ja käytöstä 276/2016." Accessed: Jun. 21, 2022. [Online]. Available: https://www.finlex.fi/fi/laki/alkup/2016/20160276

[4] E. P. Oy, "Sähkömarkkinalaki 588/2013." Accessed: Oct. 26, 2022. [Online]. Available: https://www.finlex.fi/fi/laki/ajantasa/2013/20130588

[5] T. Liikenne- ja viestintävirasto, "Määräys verkkotietojen ja verkon rakentamissuunnitelmien toimittamisesta." Finlex, May 04, 2020. Accessed: Jun. 21, 2022. [Online]. Available: https://www.finlex.fi/data/normit/45988/01_maarays_M71.pdf

[6] "Guideline on electricity transmission system operation." Accessed: May 23, 2023. [Online]. Available: https://eur-lex.europa.eu/EN/legal-content/summary/guideline-on-electricity-transmission-system-operation.html

[7] "Liikenne- ja viestintävaliokunnan mietintö LiVM 3/2016 vp." Accessed: Oct. 26, 2022. [Online]. Available: https://www.eduskunta.fi:443/FI/vaski/Mietinto/Sivut/LiVM_3+2016.aspx

[8] Liikenne- ja viestintäministeriö, "Laajakaistan yhteisrakentamisdirektiivi, teknistaloudellinen selvitys." Liikenne- ja viestintäministeriö, Jun. 09, 2015. Accessed: Oct. 30, 2022. [Online]. Available: http://urn.fi/URN:ISBN:978-952-243-457-9

[9] Fingrid. Accessed: May 23, 2023. [Online]. Available: https://www.fingrid.fi/en/

[10] "Hallituksen esitys eduskunnalle laeiksi verkkoinfrastruktuurin yhteisrakentamisesta ja -käytöstä sekä tietoyhteiskuntakaaren muuttamisesta HE 116/2015." Finlex, Nov. 12, 2015. Accessed: Oct. 26, 2022. [Online]. Available: https://www.finlex.fi/fi/esitykset/he/2015/20150116

[11] "Architecture of the World Wide Web, Volume One." Accessed: Oct. 31, 2022. [Online]. Available: https://www.w3.org/TR/webarch/

[12] "Help and FAQ - W3C." Accessed: Oct. 31, 2022. [Online]. Available: https://www.w3.org/Help/#webinternet

[13] *REST API Design Rulebook*. Accessed: Oct. 30, 2022. [Online]. Available: https://learning.oreilly.com/library/view/rest-api-design/9781449317904/

[14] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, “Uniform Resource Identifier (URI): Generic Syntax,” Internet Engineering Task Force, Request for Comments RFC 3986, Jan. 2005. doi: 10.17487/RFC3986.

[15] H. Nielsen, R. T. Fielding, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.0,” Internet Engineering Task Force, Request for Comments RFC 1945, May 1996. doi: 10.17487/RFC1945.

[16] H. Nielsen *et al.*, “Hypertext Transfer Protocol – HTTP/1.1,” Internet Engineering Task Force, Request for Comments RFC 2616, Jun. 1999. doi: 10.17487/RFC2616.

[17] “Evolution of HTTP - HTTP | MDN.” Accessed: Nov. 08, 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP

[18] “The HTTP Protocol As Implemented In W3.” Accessed: Nov. 08, 2022. [Online]. Available: https://www.w3.org/Protocols/HTTP/AsImplemented.html

[19] R. T. Fielding, M. Nottingham, and J. Reschke, “HTTP Semantics,” Internet Engineering Task Force, Request for Comments RFC 9110, Jun. 2022. doi: 10.17487/RFC9110.

[20] M. Thomson and C. Benfield, “HTTP/2,” Internet Engineering Task Force, Request for Comments RFC 9113, Jun. 2022. doi: 10.17487/RFC9113.

[21] M. Bishop, “HTTP/3,” Internet Engineering Task Force, Request for Comments RFC 9114, Jun. 2022. doi: 10.17487/RFC9114.

[22] E. Rescorla, “HTTP Over TLS,” Internet Engineering Task Force, Request for Comments RFC 2818, May 2000. doi: 10.17487/RFC2818.

[23] R. T. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing,” Internet Engineering Task Force, Request for Comments RFC 7230, Jun. 2014. doi: 10.17487/RFC7230.

[24] “Usage Statistics of Site Elements for Websites.” Accessed: Nov. 10, 2023. [Online]. Available: https://w3techs.com/technologies/overview/site_element

[25] L. M. Dusseault and J. M. Snell, “PATCH Method for HTTP,” Internet Engineering Task Force, Request for Comments RFC 5789, Mar. 2010. doi: 10.17487/RFC5789.

[26] Auth0, “Authentication vs. Authorization,” Auth0 Docs. Accessed: Nov. 01, 2022. [Online]. Available: https://auth0.com/docs/

[27] L. Richardson and S. Ruby, *RESTful Web Services*. Accessed: Oct. 31, 2022. [Online]. Available: https://learning.oreilly.com/library/view/restful-web-services/9780596529260/

[28] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” 2000, [Online]. Available: https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[29] *API Development: A Practical Guide for Business Implementation Success*. Accessed: Nov. 01, 2022. [Online]. Available: https://learning.oreilly.com/library/view/api-development-a/9781484241400/

[30] "API Evangelist: An Introduction to API Authentication," Newstex Trade & Industry Blogs. Accessed: Nov. 01, 2022. [Online]. Available: https://www.proquest.com/docview/2346189315/citation/DB3710BD9C2947E1PQ/1

[31] R. Oppliger, *Security Technologies for the World Wide Web*. Norwood, UNITED STATES: Artech House, 2002. Accessed: Nov. 01, 2022. [Online]. Available: http://ebookcentral.proquest.com/lib/tampere/detail.action?docID=227620

[32] "Token Based Authentication Made Easy," Auth0. Accessed: Nov. 17, 2022. [Online]. Available: https://auth0.com/learn/token-based-authentication-made-easy

[33] "Token Based Authentication -- Implementation Demonstration." Accessed: Nov. 17, 2022. [Online]. Available: https://www.w3.org/2001/sw/Europe/events/foaf-galway/papers/fp/token_based_authentication/

[34] Archiveddocs, "Simple Web Token (SWT)." Accessed: Nov. 01, 2022. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/azure/azure-services/hh781551(v=azure.100)

[35] A. D. Olson, P. Eggert, and K. Murchison, "The Time Zone Information Format (TZif)," Internet Engineering Task Force, Request for Comments RFC 8536, Feb. 2019. doi: 10.17487/RFC8536.

[36] C. Newman and G. Klyne, "Date and Time on the Internet: Timestamps," Internet Engineering Task Force, Request for Comments RFC 3339, Jul. 2002. doi: 10.17487/RFC3339.

[36] "ISO/IEC 21778:2017," ISO. Accessed: Nov. 01, 2022. [Online]. Available: https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/07/16/71616.html

[38] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," Internet Engineering Task Force, Request for Comments RFC 7519, May 2015. doi: 10.17487/RFC7519.

[39] "Javascript Object Signing and Encryption (JOSE) — jose 0.1 documentation." Accessed: Nov. 03, 2022. [Online]. Available: https://jose.readthedocs.io/en/latest/#overview

[40] M. Jones, "JSON Web Algorithms (JWA)," Internet Engineering Task Force, Request for Comments RFC 7518, May 2015. doi: 10.17487/RFC7518.

[41] S. Brady, "JWTs: Which Signing Algorithm Should I Use?," Scott Brady. Accessed: Feb. 27, 2023. [Online]. Available: https://www.scottbrady91.com/jose/jwts-which-signing-algorithm-should-i-use

[42] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, “PKCS #1: RSA Cryptography Specifications Version 2.2,” Internet Engineering Task Force, Request for Comments RFC 8017, Nov. 2016. doi: 10.17487/RFC8017.

[43] I. Liusvaara, “CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE),” Internet Engineering Task Force, Request for Comments RFC 8037, Jan. 2017. doi: 10.17487/RFC8037.

[44] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Signature (JWS),” Internet Engineering Task Force, Request for Comments RFC 7515, May 2015. doi: 10.17487/RFC7515.

[45] A. Rahmatulloh, R. Gunawan, and F. M. S. Nursuwars, “Performance comparison of signed algorithms on JSON Web Token,” *IOP Conference Series. Materials Science and Engineering*, vol. 550, no. 1, Jul. 2019, doi: 10.1088/1757-899X/550/1/012023.

[46] M. Jones and J. Hildebrand, “JSON Web Encryption (JWE),” Internet Engineering Task Force, Request for Comments RFC 7516, May 2015. doi: 10.17487/RFC7516.

[47] *Software Testing, Second Edition*. Accessed: Nov. 29, 2022. [Online]. Available: https://learning.oreilly.com/library/view/software-testing-second/0672327988/

[48] *Unit Testing Principles, Practices, and Patterns*. Accessed: Nov. 22, 2022. [Online]. Available: https://learning.oreilly.com/library/view/unit-testing-principles/9781617296277/

[49] *Software Testing and Continuous Quality Improvement, 3rd Edition*. Accessed: Nov. 29, 2022. [Online]. Available: https://learning.oreilly.com/library/view/software-testing-and/9781351722209/

[50] *The Art of Unit Testing: with Examples in .NET*. Accessed: Nov. 22, 2022. [Online]. Available: https://learning.oreilly.com/library/view/the-art-of/9781933988276/

[51] J. Horch, *Practical Guide to Software Quality Management*. Norwood, UNITED STATES: Artech House, 2003. Accessed: Nov. 22, 2022. [Online]. Available: http://ebookcentral.proquest.com/lib/tampere/detail.action?docID=227656

[52] G. A. Campbell and S. Sa, “A new way of measuring understandability”.

[53] *xUnit Test Patterns: Refactoring Test Code*. Accessed: Jan. 22, 2023. [Online]. Available: https://learning.oreilly.com/library/view/xunit-test-patterns/9780131495050/

[54] “Continuous Integration,” martinfowler.com. Accessed: Feb. 22, 2023. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[55] *Continuous Integration in .NET*. Accessed: Feb. 22, 2023. [Online]. Available: https://learning.oreilly.com/library/view/continuous-integration-in/9781935182559/

[56] *Continuous Integration: Improving Software Quality and Reducing Risk*. Accessed: Feb. 22, 2023. [Online]. Available: https://learning.oreilly.com/library/view/continuous-integration-improving/9780321336385/

[57] *Software Design Methodology*. Accessed: Aug. 15, 2023. [Online]. Available: https://learning.oreilly.com/library/view/software-design-methodology/9780750660754/

[58] “Apache License, Version 2.0.” Accessed: Jan. 31, 2023. [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0

[58] “Verkkotietopisteen sähköinen rajapinta.”, Traficom. Accessed: Nov. 10, 2023. [Online]. Available: https://www.traficom.fi/sites/default/files/media/file/Verkkotietopisteen-sahkoinen-rajapinta.pdf

[60] karelz, “HttpClient Class (System.Net.Http).” Accessed: Feb. 08, 2023. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.net.http.httpclient?view=net-7.0

[61] “Fluent Assertions,” Fluent Assertions. Accessed: Feb. 20, 2023. [Online]. Available: http://www.fluentassertions.com/

[62] “moq.” Moq, Feb. 18, 2023. Accessed: Feb. 20, 2023. [Online]. Available: https://github.com/moq/moq4

[63] R. Szalay, “MockHttp for HttpClient.” Feb. 10, 2023. Accessed: Feb. 10, 2023. [Online]. Available: https://github.com/richardszalay/mockhttp

[64] “TestableIO/System.IO.Abstractions.” TestableIO, Feb. 06, 2023. Accessed: Feb. 10, 2023. [Online]. Available: https://github.com/TestableIO/System.IO.Abstractions

[65] F. Ngwenya, “Fine Code Coverage.” Feb. 16, 2023. Accessed: Feb. 21, 2023. [Online]. Available: https://github.com/FortuneN/FineCodeCoverage

[66] “NUnit vs. XUnit vs. MSTest: Comparing Unit Testing Frameworks In C#,” LambdaTest. Accessed: Feb. 25, 2023. [Online]. Available: https://www.lambdatest.com/blog/nunit-vs-xunit-vs-mstest/

[67] “Lukijan mielipide | Kriittisten infratietojen keskittäminen on valtava turvallisuusriski,” Helsingin Sanomat. Accessed: Feb. 27, 2023. [Online]. Available: https://www.hs.fi/mielipide/art-2000009226461.html

[68] “Lukijan mielipide | Keskitetyistä tietovarannoista voidaan tehdä turvallisia,” Helsingin Sanomat. Accessed: Feb. 27, 2023. [Online]. Available: https://www.hs.fi/mielipide/art-2000009234645.html

APPENDIX A: FULL MODULE DIAGRAM

Figure 44 below shows the whole module diagram of the created program. It does not include testing classes but only production classes.

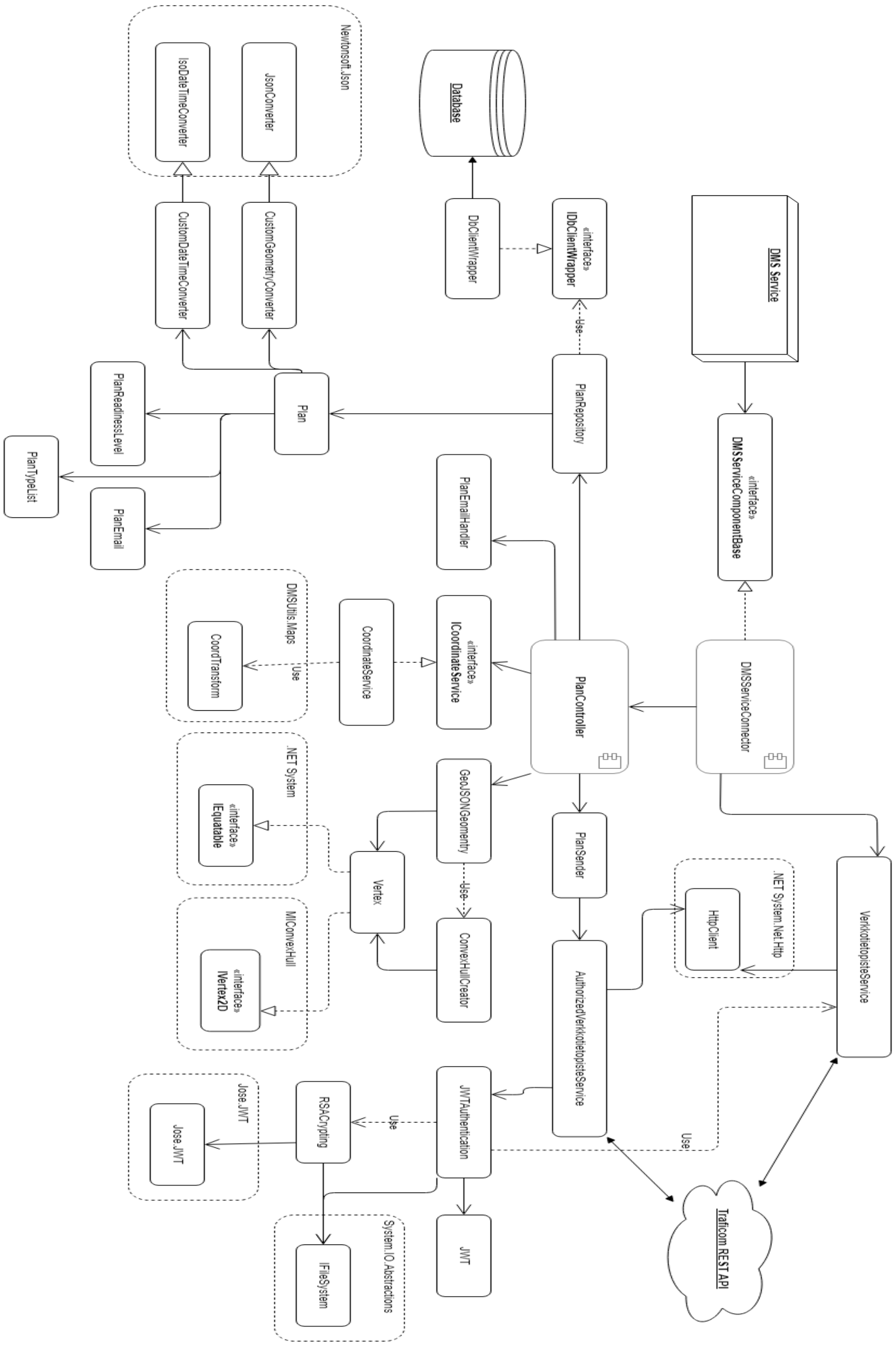


Figure 44. *Full module diagram of the created program.*