



# Architecture

## E-resept Forskrivningsmodul

Thula – Nordic Source Solutions

Date | 13/11/2017

Software version | 177

Document version | 1.1

Document status | Customer Review

Author | Ægir Örn Leifsson

Contributor(s) |

Distribution | Customer

File name | E-resept FM - Architecture.docx

© 2018 thula®

The document is only intended for Thula personnel and customers  
Style and contents are confidential

## Table of Contents

<b>1</b>	<b>Document control .....</b>	<b>4</b>
1.1	Revision tracking .....	4
1.2	Document source, storage and distribution .....	4
1.3	Revision history .....	4
1.4	Related documents.....	4
1.5	Reader comments .....	4
1.6	Glossary .....	4
<b>2</b>	<b>Introduction.....</b>	<b>9</b>
<b>3</b>	<b>Architectural overview .....</b>	<b>9</b>
<b>4</b>	<b>Client architecture .....</b>	<b>11</b>
4.1	Overall architecture .....	11
4.2	UI architecture .....	12
4.3	Service proxies .....	13
4.4	EPJ API .....	14
<b>5</b>	<b>Server architecture .....</b>	<b>16</b>
5.1	Overall architecture .....	16
5.2	Services and request isolation.....	17
5.3	Task scheduling.....	17
5.3.1	Task scheduling model .....	17
5.3.2	Scheduled tasks.....	17
5.4	FEST cache and updates .....	18
5.5	Synchronous communication .....	20
5.6	Asynchronous communication.....	21
5.7	Public service API.....	22
5.8	Clustering .....	23
5.9	Trace events and monitoring .....	25
5.9.1	Trace events .....	25
5.9.2	WMI objects and performance counters .....	26
5.9.3	MS SQL Server.....	26
<b>6</b>	<b>Client-server communications.....</b>	<b>26</b>
6.1	General information .....	26
6.2	Version control.....	27
6.3	Load-balancing and failover .....	28
6.3.1	Server side.....	28
6.3.2	Client side.....	29
6.4	Client authentication and encryption.....	30
6.4.1	Transport security and server authentication .....	30
6.4.2	User authentication .....	31
6.4.3	Limitations in current security model .....	32

6.5	Fault contracts and error handling .....	33
7	Database .....	33
7.1	Database versioning.....	33
7.2	Database sharding .....	33
8	Miscellaneous.....	34
8.1	Components used .....	34
8.1.1	StructureMap .....	34
8.1.2	Rhino Mocks .....	35
8.1.3	NUnit.....	35
8.1.4	LinqToXsd.....	35
8.1.5	LinqToSql.....	35
8.1.6	SharpZipLib .....	35
8.2	Kodeverk .....	35
8.3	Custom tracing for troubleshooting.....	36
8.3.1	WCF logging on the application server .....	36
8.3.2	EPJ API logging on the client .....	37
8.3.3	Logging M30 to disk during FEST updates .....	37
9	Development environment .....	37

# 1 Document control

*This section describes how to version, file, distribute and improve this document.*

## 1.1 Revision tracking

This document is subject to revision control so that after each formal change a new version shall be created with a new data and revision number. At any given time the revision with the highest version number is considered the official and valid version of this document.

## 1.2 Document source, storage and distribution

The source of this document is maintained by Thula. It is stored in the Thula document repository in the following folder:

- E-resept \ System Documentation \ Paper Based

This document shall be distributed in PDF format only.

## 1.3 Revision history

Date	Version	Author/Approved by	Description
<b>Release 4.1.0</b>			
2017-11-13	1.1	Ægir Örn Leifsson	Added referenced to .NET 4.7.1 as the stated required .NET version for the FM.
<b>Release 3.8.0 RC1</b>			
2016-10-07	1.0	Ægir Örn Leifsson	Version 1.0 for customer review.
<b>Release 3.7.0</b>			
2016-02-24	0.1	Ægir Örn Leifsson	Draft of document structure, for customer review.

## 1.4 Related documents

Document	Description
E-resept FM - EPJ API and Technical Specification	A functional and technical description of the FM EPJ API.
E-resept FM - Installation and Configuration Guide	Administrator guide to the FM.

## 1.5 Reader comments

If you have any comments on the contents of this document please send those by e-mail to [thula@thula.is](mailto:thula@thula.is). If a review result in changes, all users of this document should be notified.

## 1.6 Glossary

This section lists some terms used in this document and specifies how they are to be interpreted within the scope of the document.

Abbreviation	Explanation or web reference
Forskrivningsmodul	Prescription module. The software being described in this document.
FM	Forskrivningsmodul.
EPJ	Electronic patient journal. A computerized patient record/journal system that communicates with the FM through the FM EPJ API and import/export of user and patient data.
API	An application programming interface (API) is an interface implemented by a software program that enables it to interact with other software.

	<a href="http://en.wikipedia.org/wiki/API">http://en.wikipedia.org/wiki/API</a> .
RF	<p>Reseptformidleren.</p> <p>Reseptformidleren er et sentralt elektronisk helseregister/database som de aller fleste meldinger i e-resept går gjennom. Her oppbevares den elektroniske resepten og her slettes den, 4 uker etter at den er blitt ugyldig, det vil si ferdig ekspedert eller utløpt på dato.</p>
AR	<p>Adresseregisteret.</p> <p>Adresseregisteret er verktøyet for presis adressering av elektroniske meldinger i helsenettet.</p> <p><a href="https://register-web.test.nhn.no/Ar">https://register-web.test.nhn.no/Ar</a>.</p>
DBMS	<p>Database Management System.</p> <p>A database management system (DBMS) is a computer software application that interacts with the user, other applications, and the database itself to capture and analyze data. A general-purpose DBMS is designed to allow the definition, creation, querying, update, and administration of databases.</p> <p><a href="https://en.wikipedia.org/wiki/Database">https://en.wikipedia.org/wiki/Database</a>.</p>
COM	<p>Component Object Model (COM) is a binary-interface standard for software components introduced by Microsoft in 1993. It is used to enable inter-process communication and dynamic object creation in a large range of programming languages.</p> <p><a href="https://en.wikipedia.org/wiki/Component_Object_Model">https://en.wikipedia.org/wiki/Component_Object_Model</a>.</p>
Web service	<p>A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.</p> <p><a href="https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice">https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice</a>.</p>
SOAP	<p>SOAP (Simple Object Access Protocol) is a protocol specification for exchanging structured information in the implementation of web services in computer networks. Its purpose is to induce extensibility, neutrality and independence. It uses XML Information Set for its message format, and relies on application layer protocols, most often Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.</p> <p>SOAP allows processes running on disparate operating systems (such as Windows and Linux) to communicate using Extensible Markup Language (XML). Since Web protocols like HTTP are installed and running on all Operating systems, SOAP allows clients to invoke web services and receive responses independent of language and platforms.</p> <p><a href="https://en.wikipedia.org/wiki/SOAP">https://en.wikipedia.org/wiki/SOAP</a>.</p>
MVVM	<p>Model–view–view-model (MVVM) is a software architectural pattern. MVVM facilitates a separation of development of the graphical user interface – be it via a markup language or GUI code – from development of the business logic or back-end logic (the data model). The view model of MVVM is a value converter; meaning the view model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented. In this respect, the view model is more model than view, and handles most if not all of the view's display logic.</p> <p><a href="https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel">https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel</a>.</p>

XAML	<p>Extensible Application Markup Language (XAML) is a declarative XML-based language developed by Microsoft that is used for initializing structured values and objects. It is available under Microsoft's Open Specification Promise. The acronym originally stood for Extensible Avalon Markup Language - Avalon being the code-name for Windows Presentation Foundation (WPF).</p> <p>XAML is used extensively in .NET Framework 3.0 &amp; .NET Framework 4.0 technologies, particularly Windows Presentation Foundation (WPF), Silverlight, Windows Workflow Foundation (WF) and Windows Runtime XAML Framework and Windows Store apps. In WPF, XAML forms a user interface markup language to define UI elements, data binding, eventing, and other features. In WF, workflows can be defined using XAML. XAML can also be used in Silverlight applications, Windows Phone apps and Windows Store apps.</p> <p><a href="https://en.wikipedia.org/wiki/Extensible_Application_Markup_Language">https://en.wikipedia.org/wiki/Extensible_Application_Markup_Language</a>.</p>
WPF	<p>Windows Presentation Foundation (or WPF) is a graphical subsystem by Microsoft for rendering user interfaces in Windows-based applications. WPF, previously known as "Avalon", was initially released as part of .NET Framework 3.0. Rather than relying on the older GDI subsystem, WPF uses DirectX. WPF attempts to provide a consistent programming model for building applications and separates the user interface from business logic. It resembles similar XML-oriented object models, such as those implemented in XUL and SVG.</p> <p>WPF employs XAML, an XML-based language, to define and link various interface elements. WPF applications can be deployed as standalone desktop programs or hosted as an embedded object in a website. WPF aims to unify a number of common user interface elements, such as 2D/3D rendering, fixed and adaptive documents, typography, vector graphics, runtime animation, and pre-rendered media. These elements can then be linked and manipulated based on various events, user interactions, and data bindings.</p> <p>WPF runtime libraries are included with all versions of Microsoft Windows since Windows Vista and Windows Server 2008. Users of Windows XP SP2/SP3 and Windows Server 2003 can optionally install the necessary libraries.</p> <p><a href="https://en.wikipedia.org/wiki/Windows_Presentation_Foundation">https://en.wikipedia.org/wiki/Windows_Presentation_Foundation</a>.</p>
Data binding	<p>Data binding is a general technique that binds data sources from the provider and consumer together and synchronizes them. This is usually done with two data/information sources with different languages as in XML data binding. In UI data binding, data and information objects of the same language but different logic function are bound together (e.g. Java UI elements to Java objects).</p> <p>In a data binding process, each data change is reflected automatically by the elements that are bound to the data. The term data binding is also used in cases where an outer representation of data in an element changes, and the underlying data is automatically updated to reflect this change. As an example, a change in a TextBox element could modify the underlying data value</p> <p><a href="https://en.wikipedia.org/wiki/Data_binding">https://en.wikipedia.org/wiki/Data_binding</a>.</p>
WCF	<p>The Windows Communication Foundation (or WCF), previously known as "Indigo", is a runtime and a set of APIs in the .NET Framework for building connected, service-oriented applications</p> <p>WCF is a tool often used to implement and deploy a service-oriented architecture (SOA). It is designed using service-oriented architecture principles to support distributed computing where services have remote consumers. Clients can consume multiple services; services can be consumed by multiple clients. Services are loosely coupled to each other. Services typically have a WSDL interface (Web Services Description Language) that any WCF client can</p>

	<p>use to consume the service, regardless of which platform the service is hosted on. WCF implements many advanced Web services (WS) standards such as WS-Addressing, WS-ReliableMessaging and WS-Security.</p> <p><a href="https://en.wikipedia.org/wiki/Windows_Communication_Foundation">https://en.wikipedia.org/wiki/Windows_Communication_Foundation</a>.</p>
WCF Bindings	<p>Windows Communication Foundation (WCF) separates how the software for an application is written from how it communicates with other software. Bindings are used to specify the transport, encoding, and protocol details required for clients and services to communicate with each other. WCF uses bindings to generate the underlying wire representation of the endpoint, so most of the binding details must be agreed upon by the parties that are communicating.</p> <p><a href="https://msdn.microsoft.com/en-us/library/ms733027(v=vs.110).aspx">https://msdn.microsoft.com/en-us/library/ms733027(v=vs.110).aspx</a>.</p>
NetTcpBinding	<p>A type of WCF Binding.</p> <p>The NetTcpBinding generates a run-time communication stack by default, which uses transport security, TCP for message delivery, and a binary message encoding. This binding is an appropriate Windows Communication Foundation (WCF) system-provided choice for communicating over an Intranet. The default configuration for the NetTcpBinding is faster than the configuration provided by the WSHttpBinding, but it is intended only for WCF-to-WCF communication.</p> <p><a href="https://msdn.microsoft.com/en-us/library/system.servicemodel.nettcpbinding(v=vs.110).aspx">https://msdn.microsoft.com/en-us/library/system.servicemodel.nettcpbinding(v=vs.110).aspx</a>.</p>
Dependency injection (DI)	<p>In software engineering, dependency injection is a software design pattern that implements inversion of control for resolving dependencies. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state.[1] Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.</p> <p>Dependency injection allows a program design to follow the dependency inversion principle. The client delegates the responsibility of providing its dependencies to external code (the injector) . The client is not allowed to call the injector code.[2] It is the injecting code that constructs the services and calls the client to inject them. This means the client code does not need to know about the injecting code. The client does not need to know how to construct the services. The client does not need to know which actual services it is using. The client only needs to know about the intrinsic interfaces of the services because these define how the client may use the services. This separates the responsibilities of use and construction.</p> <p><a href="https://en.wikipedia.org/wiki/Dependency_injection">https://en.wikipedia.org/wiki/Dependency_injection</a>.</p>
Inversion Control (IoC)	<p>of In software engineering, inversion of control (IoC) is a design principle in which custom-written portions of a computer program receive the flow of control from a generic framework. A software architecture with this design inverts control as compared to traditional procedural programming: in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the framework that calls into the custom, or task-specific, code.</p> <p>Inversion of control is used to increase modularity of the program and make it extensible,[1] and has applications in object-oriented programming and other programming paradigms. The term was popularized by Robert C. Martin and Martin Fowler.</p> <p><a href="https://en.wikipedia.org/wiki/Inversion_of_control">https://en.wikipedia.org/wiki/Inversion_of_control</a></p>

**Continuous integration (CI)**

In software engineering, continuous integration (CI) is the practice of merging all developer working copies to a shared mainline several times a day. Grady Booch first named and proposed CI in his 1991 method,[1] although he did not advocate integrating several times a day. Extreme programming (XP) adopted the concept of CI and did advocate integrating more than once per day - perhaps as many as tens of times per day.

[https://en.wikipedia.org/wiki/Continuous\\_integration](https://en.wikipedia.org/wiki/Continuous_integration)



## 2 Introduction

This document describes the architecture of the FM and lists the technologies and libraries used in the FM implementation. Both the overall architecture of the solution is described and the architecture of the client and server components individually.

## 3 Architectural overview

The FM is built on top of version 4 of the .NET Framework on the Microsoft Windows platform, but due to the use of SHA-256 in communication with Kjernejournal, .NET 4.7.1 is the stated required .NET version as of version 4.0.0 of the FM. The FM is deployed in three tiers, a client application, an application server and a database. The DBMS supported is Microsoft SQL Server.

The FM client is a 32-bit Windows client application which exposes a COM API for integration with other client-side applications. The FM client communicates with the FM application server over SOAP services.

The FM application server is a Windows service which runs as a 64-bit application on 64-bit Windows, and as a 32-bit application on 32-bit Windows. It exposes SOAP service APIs to the FM client and external applications, the application server also handles all communication with the FM database and integrations with external services (e.g. RF, AR, etc.).

The FM depends on Microsoft SQL Server for data storage. The FM uses MSSQL compatibility level 100 and supports MSSQL 2008 R2, 2012 and 2014.

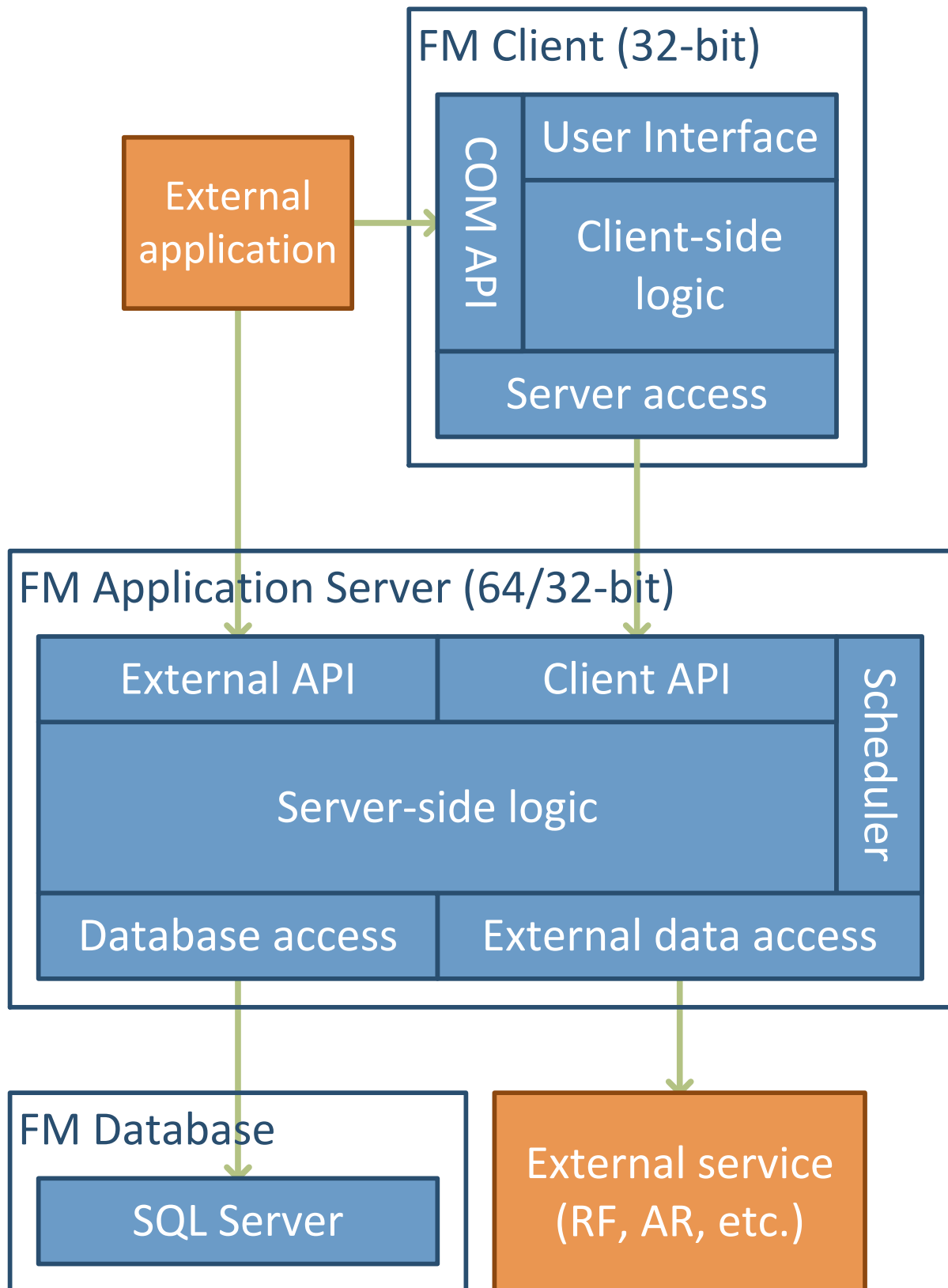


Figure 1 - FM architecture overview.

## 4 Client architecture

Figure 2 shows the FM client application architecture, extracted from Figure 1 above. In Figure 2 the internal architecture of the FM client is somewhat simplified. This section serves to dive deeper into the architecture of the FM client application.

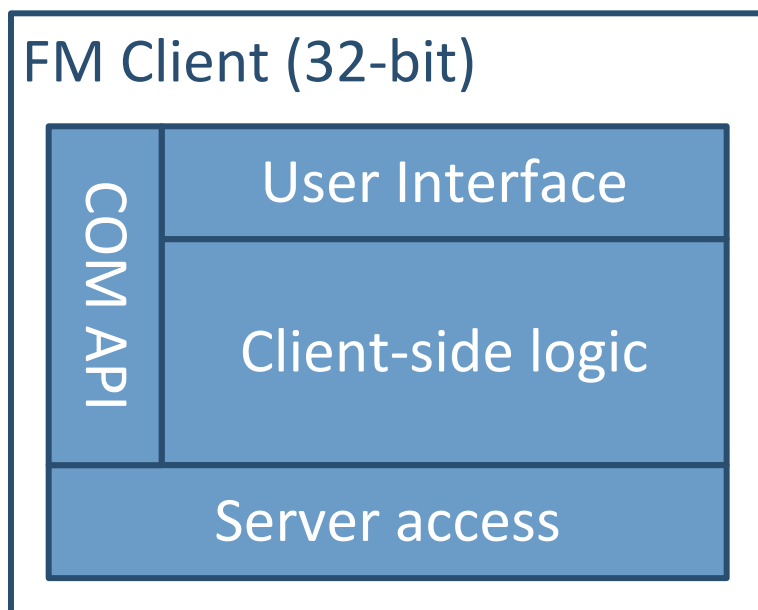


Figure 2 - Simplified FM client application architecture.

### 4.1 Overall architecture

The FM client is a 32-bit .NET 4 application. It is built following the MVVM architectural pattern as shown in Figure 3.

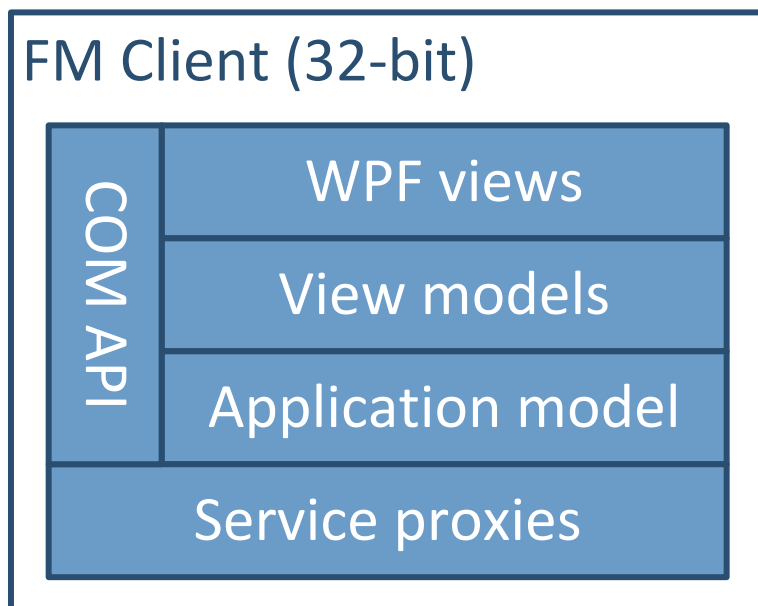


Figure 3 - FM client application architecture.

The “WPF views”, “View models” and “Application model” layers shown in Figure 3 correspond to the view, view model and model in the MVVM pattern. Those are discussed further in section 4.2.

The “Service proxies” shown in Figure 3 handle communication with the FM application server and provide higher layers with an abstraction of the FM application server. The service proxies are described further in section 4.3.

The “COM API” shown in Figure 3 is used by external client applications to interact with the FM. This API is described in section 4.4.

## 4.2 UI architecture

The FM UI is built using WPF. All screens and reports in the FM are implemented as WPF views written in XAML. In terms of MVVM, the FM screens and reports are “views”.

The FM views are data bound to view models, as defined by MVVM. Views and view models come in pairs, so that for each view there exists exactly one view model. Each view model is implemented as a plain .NET class that models the view it supports.

When an MVVM view is constructed and shown, a number of things must happen. Both the view and view model must be constructed, and UI elements in the view must be bound to properties and methods on the view model. The MVVM implementation in the FM uses a *convention over configuration* approach (ref. [https://en.wikipedia.org/wiki/Convention\\_over\\_configuration](https://en.wikipedia.org/wiki/Convention_over_configuration)) to achieve this.

When a view is to be shown in the FM, the programmer requests an instance of a view model class. All view model classes in the FM reside in a folder called “ViewModel” and have a name suffix of “ViewModel”. Similarly, all views reside in a folder called “View” and have a name suffix of “View”. This convention is used by the FM MVVM framework to construct the correct view and bind it to a view model when a view model is instantiated. For example, when a view model of type “MyViewModel” is instantiated in the FM code, a view of type “MyView” is automatically created and bound to the view model.

Standard WPF data binding is used when a view is bound to a view model, but here convention is also used for binding e.g. buttons in the view to methods on the view model. A button called “Save” will thus automatically be bound to a method with the signature “void Save()” on the view model, and the activation of the button will automatically be bound to a property with the signature “bool CanSave” if one exists.

For data binding to work correctly, the view model must notify the view of any data changes that need to be reflected in the view. In WPF the view model raises events for this purpose, such events must be raised when a property value changes, when the contents of a list (or more generically, an enumerable) change, etc. To ensure that this is done correctly and consistently in the FM, view model properties are automatically code generated.

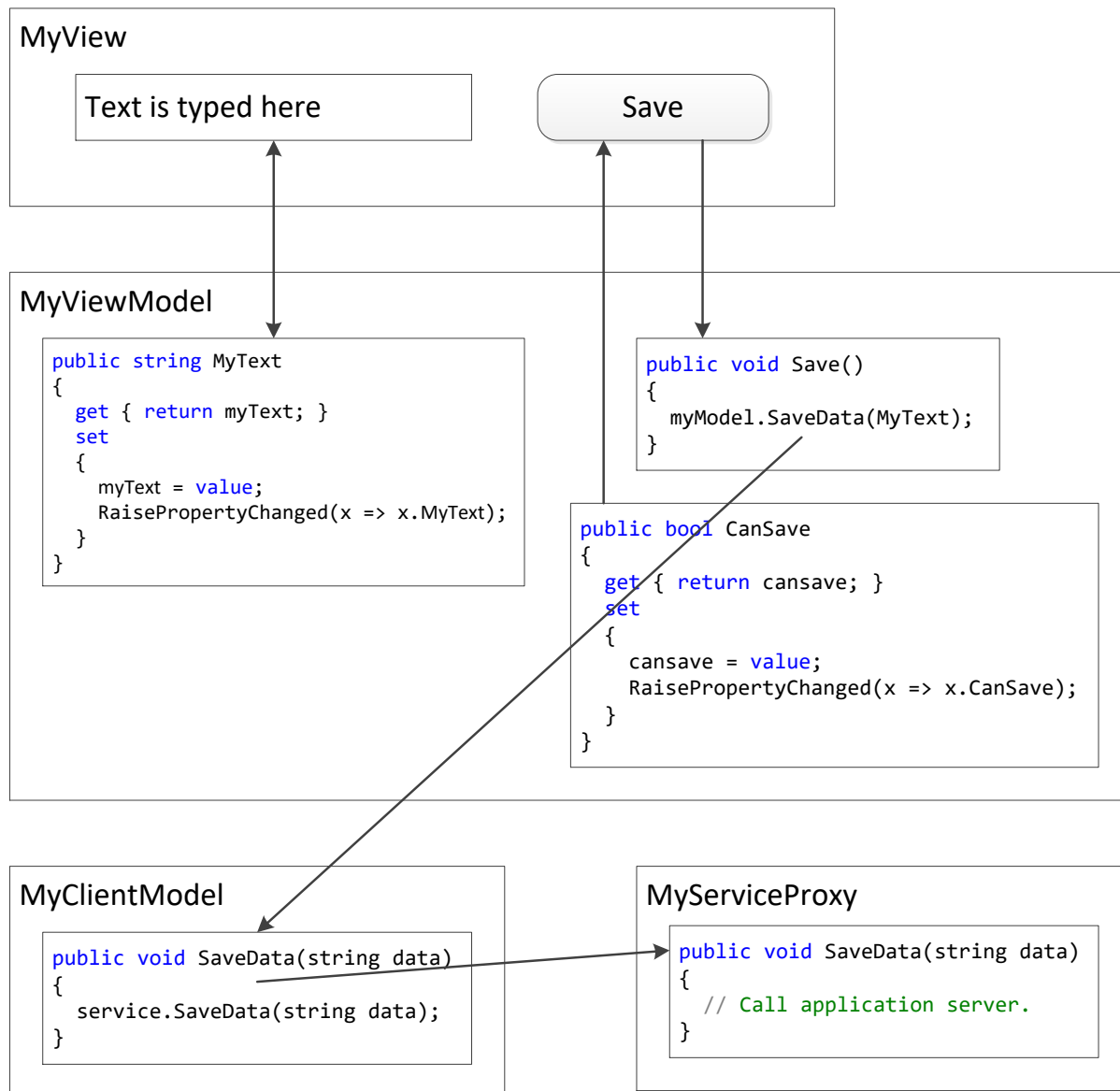
Figure 4 shows a very simple example of the interaction between a view (MyView), view model (MyViewModel), model (MyClientModel) and an application server service proxy.

The view contains an edit box for text and a button. The edit box is bound to a string property on the view model and updates are pushed in both directions, i.e. if the user types into the text box, then the property value on the view model is updated; and if the property value on the view model is updated in code, then the value shown in the edit box in the view is automatically updated.

The “Save” button on the view is bound to a boolean property called “CanSave” which enables and disables the button when set in code. The button is also bound to a void method called “Save” which is called when the button is clicked.

The same method on the view model calls a method called “SaveData” on the model “MyClientModel”. In the example in Figure 4 the model simply forwards the SaveData call to a service proxy, in reality the model may implement some client side logic.

The service proxy (MyServiceProxy) is a client side abstraction for calling a service on the application server. Service proxies are discussed further in section 4.3.



**Figure 4 - Illustration of data binding in the FM client.**

The example above shows a view with editable data (the text box) and behaviour (the button). Reports in the FM are implemented in exactly the same way, but do not contain any behaviour and never update properties on a view model, i.e. the binding is one-way from the view model to the view (report) and once generated the view is static.

## 4.3 Service proxies

Communication between the FM client and application server is built on the .NET WCF framework.

The FM defines a strict contract between the client and application server. This contract is implemented as a set of data contracts, service contracts and operation contracts. See <https://msdn.microsoft.com/en-us/library/ff183866.aspx> for information about contracts in WCF.

The data contracts are simply implemented as .NET classes that do not define any behaviour. The service and operation contracts are defined as .NET interfaces with associated methods and functions. Both the data contract classes and service and operation contract interfaces are shared between the client and application server (i.e. they are housed in the FM.Common assembly which is included both in the client and application server).

On the client side, the service and operation contracts are implemented by service proxy classes that derive from base classes that take care of all aspects of the client-server communication, such as versioning, security, load-balancing, etc. The result is that when writing code, accessing the application server from the FM client is no different from accessing functionality built into the client itself.

The fact that the client-server contracts are defined in code, which is built into both client and server, means that full compile-time checking is performed for those contracts and there is no possibility for the client and server end of the contracts to fall out of sync. Section 6.2 describes how the FM ensures that a given version of the FM client can only make calls to the same exact version of the FM application server.

## 4.4 EPJ API

The FM EPJ API consists of three main parts:

- A COM interface implemented by the FM and exposed to external systems. The COM identifier of the FM is "eResept.Forskrivningsmodul.1".
- A COM interface implemented by external systems and called by the FM when the external system has made a call to the FM COM interface and requested a callback when the FM is closed. The COM identifier is "eResept.ForskrivningsmodulCallback.3".
- An XML schema that defines the data passed between the FM and external systems. This data contract is defined in a namespace that as the time of this writing is called <http://www.kith.no/xmlstds/eresept/forskrivningsmodul/epjapi/2016-08-23>.

All three parts of the EPJ API are strictly versioned. The two interfaces have names that end with an integer that denotes the version of the interface. The XML schema contains a namespace declaration that has a name which ends with a version number in the form of a date. At each time, multiple versions of both the COM interface and XML schema can be supported. As an example, version 3.7.0 of the FM supports version 1 of the API COM interface (eResept.Forskrivningsmodul.1), versions 2 and 3 of the COM callback interface (eResept.ForskrivningsmodulCallback.2 and eResept.ForskrivningsmodulCallback.3), and five different versions of the XML schema (namespaces ending with 2013-03-12, 2014-01-14, 2014-05-02, 2015-09-01 and 2016-08-23).

When an EPJ makes a call to the FM, the FM will always respond with the same XML namespace version as used to make the call. If a callback is requested by the EPJ, the FM will respond on the latest callback interface version that has been registered in Windows.

The fact that COM is used for communicating from an EPJ to the FM means that the EPJ does not need to consider lifecycle management of the FM process, i.e. the EPJ does not have to take care of starting the FM and does not have to know e.g. where the FM has been installed. Since COM is a very mature technology in Windows (it was introduced in 1993) it can also be accessed and used by just

about anything that will run on the Windows platform (a benefit when integrating with older EPJ systems).

The following two lines show how an EPJ implemented in .NET can acquire a reference to the FM:

```
var type = Type.GetTypeFromProgID("eResept.Forskrivningsmodul.1", true);  
var fm = Activator.CreateInstance(type)
```

The first line looks up the FM data type which can be registered with Windows, identified by the ProgId "eResept.Forskrivningsmodul.1".

The second line requests an instance of the FM, using the data type that resulted from the code line above. When the second line runs, Windows will automatically start the FM if it is not already running, or hand the caller a reference to the already running FM process if one has already been started. This ensures that an EPJ cannot start multiple versions of the FM and that an EPJ does not need to have any information about how the FM has been installed, beyond having knowledge of the COM API.

The "fm" variable in the code above points to a dynamic object. This means that once the EPJ has acquired a reference to the FM it can simply make calls to the FM object as follows:

```
var result = instance.StartPasient(startPasient);
```

Where "startPasient" is a string that contains a StartPasient XML document as defined by one of the supported EPJ API schema versions. After this call completes, the "result" variable will point to a string that contains a StartPasientSvar XML document defined by the same schema version used in the call.

The code shown above is exactly the same as the code used by the FM to access the callback API when a callback has been requested. In this case the EPJ should register an implementation of the "eResept.ForskrivningsmodulCallback.3" interface with Windows, which the FM can subsequently request and call.

The code below is another example of how the FM COM API can be accessed and used. This time from a Windows PowerShell script. The first and third lines only serve to read the XML document to be sent to the FM from disk. The second line requests a reference to the FM from Windows (it thus corresponds to the first two lines of .NET code shown above) and the fourth line makes a StartPasient call to the FM (it corresponds to the third line of .NET code shown above).

```
param([string]$parameterFile = "startpasient.xml")  
$fm = New-object -comObject eResept.Forskrivningsmodul.1 -strict  
$startPasient = gc $parameterFile -Encoding UTF8  
$fm.StartPasient($startPasient)
```

Methods available on the FM COM API are of three basic types:

- Methods named Start[x] – these methods open the FM UI (example, StartPasient, StartInbox).
- Methods named Les[x] – these methods return data from the FM (example, LesCave, LesVarslinger).
- Methods named Skriv[x] – these methods write data to the FM (example, SkrivCave, SkrivBrukerInfo).

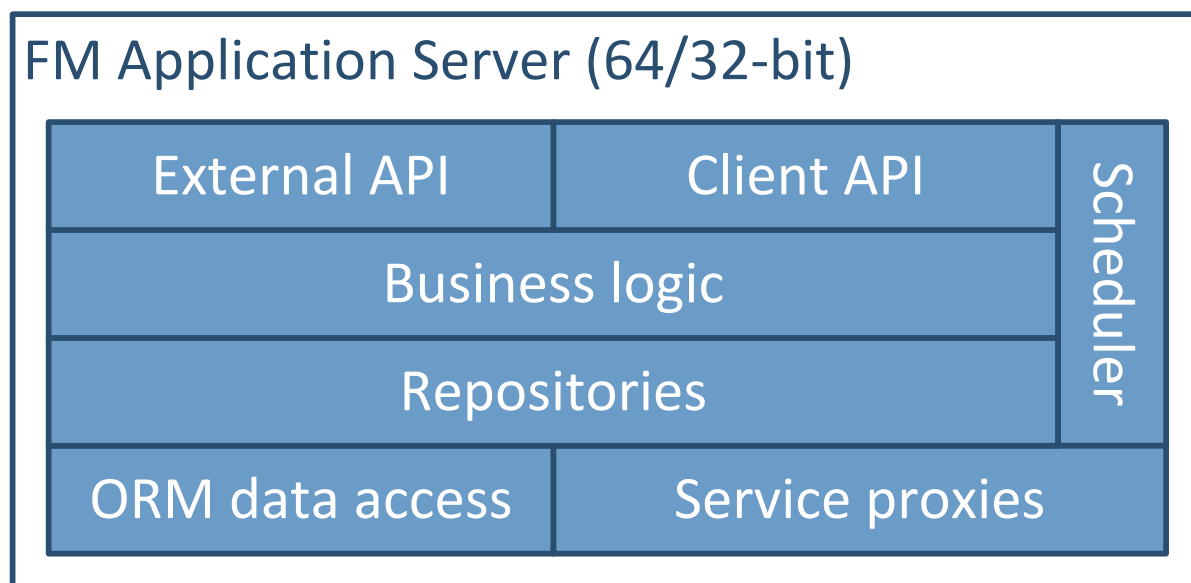
As of version 3.7.0 of the FM, the Les[x] and Skriv[x] methods are available as web service calls on the application server in addition to being available on the COM API. The web service version of the API is described in section 5.7.

Refer to the document "E-resept FM - EPJ API and Technical Specification" for more information about the EPJ API.

## 5 Server architecture

### 5.1 Overall architecture

The FM application server is a Windows service built with the “Any CPU” platform target. This means that the application server will run as a 64-bit application on 64-bit Windows, and as a 32-bit application on 32-bit Windows. Running the application server on 64-bit Windows therefore allows for access to more resources, such as memory, and more effective resource utilization.



**Figure 5 - FM application server architecture.**

The role of the FM application server is to run services and scheduled tasks:

- Services are called by external processes (e.g. the FM client) and offer functionality to those processes. An example of a service is the medication search service, which allows a client to search for medications given a criteria.
- Scheduled tasks are triggered periodically by the application server itself and do not directly offer functionality to other processes. Examples of scheduled tasks are the FEST update and a task that backs up the FM database.

The scheduler component shown in Figure 5 manages scheduled tasks.

The client API shown in Figure 5 is the entry point for the FM client to call services on the application server. The client API is a collection of SOAP services that are only meant to be consumed by the FM client and follow the same versioning as the FM itself.

The external API is an entry point for 3<sup>rd</sup> party software to call services on the FM application server. This API mirrors part of the COM API described in section 4.4 and follows the same versioning. Much like the client API, the external API is a SOAP service.

The business logic layer consists of plain .NET classes that implement the logic needed in the application server.

The repository layer provides data access abstractions for higher layers. The purpose is to allow business logic code to request data from the local database or external services without having knowledge of the structure of those data stores. This allows repositories to be mocked in automated tests of business logic code.



The ORM (object-relational mapping) layer is built on LinqToSql from Microsoft. This layer provides compile-time checked access to the FM database.

The service proxies are automatically generated proxies for communicating with external services, such as Reseptformidleren and Adresseregisteret.

## 5.2 Services and request isolation

When the FM application server starts, it creates instances of the API services and wraps them in WCF service hosts. The services, and all their dependencies are created by dependency injection (ref. section 8.3.3). Requests to services on the FM application server are:

- Transactional – each request runs within a dedicated transaction scope.
- Isolated – each request runs independently of other requests.

Every request made to the API services is transactional, i.e. every request is wrapped in a transaction which ensures that the request is atomic (i.e. it either executes completely or not at all, in terms of updates to the FM database). Every request is also associated with a „request scope“ which is used by the dependency injection framework to ensure proper scoping of instances required to process the request. Three types of DI scopes are in use in the FM application server:

- Per instance request – a new object instance is created each time an interface is requested.
- Singleton – a single object instance is created during the application server process lifetime and is returned each time an interface is requested.
- Request scoped – a single object instance is created for each request scope. This allows a single request to handle an instance like a singleton that is not shared with other requests.

There is usually a 1-1 relationship between transactions and request scopes, but not always.

Before a WCF service host is started, it is configured by the FM. Part of this configuration is to attach „behaviours“ to the service. This is where version control, load balancing and security are implemented for services (ref. section 6). During this step, fault behavior is also configured. Refer to section 6.5 for more information about fault behavior and fault contracts.

## 5.3 Task scheduling

### 5.3.1 Task scheduling model

As described in section 6.3, the FM application server can run in either master or slave mode. When a single application server instance is used, that instance always runs as master.

Only the application server running as master runs scheduled tasks. Schedules in the FM can be of three types:

- Interval – schedules that run on fixed intervals (e.g. every 5 minutes or every 60 minutes).
- Fixed time every day – schedules that run at the same time every day (e.g. at 05:00 in the morning, every morning).
- Fixed time on set days – schedules that run at the same time on selected days (e.g. at 05:00 in the morning on Mondays, Wednesdays and Fridays).

### 5.3.2 Scheduled tasks

The following scheduled tasks have been implemented in the FM:

Task name	Description
ClearM96Cache	Task that clears old M96 cache data from the FM database. Runs daily at a fixed time.
CompressDataTask	Task that compresses XML data stored in the FM database. This task was originally implemented to compress uncompressed data in existing FM installations, but can also be used to re-compress data if a more efficient compression algorithm is introduced in the future. This task runs for a maximum of one hour each time. Runs daily at a fixed time, either at 20:00 or 00:00, depending on when the DatabaseBackupTask runs.
CreateNewDocumentsDatabaseTask	Task that creates a new documents database on SQL Express when the current documents database is full (ref. section 7). Runs daily at 05:00, only runs when the FM database is stored in SQL Express.
DatabaseBackupTask	Task that backs up the FM database. Runs daily at a fixed time, can be disabled.
DatabaseMaintenanceTask	Task that performs database maintenance. Currently this task defragments indexes in the FM database. Runs 4 hours after the DatabaseBackupTask on Sundays.
ExpireMessagesWaitingForAckTask	Transitions asynchronous messages that have been waiting for an ack for over 72 hours from the "Sent" state to the "Retry limit reached" state. Runs every 5 minutes.
FetchNewReferenceNumbersFromRfTask	Task that downloads new reference numbers from RF when needed. Runs every 60 minutes (i.e. the FM checks if new reference numbers are needed every 60 minutes).
SendRfVerifyMessage	When the FM has been updated, a Verify message is sent to RF. If this fails, a schedule is set up to run the SendRfVerifyMessage task after 7 days to retry the Verify message.
UpdateFestTask	Task that checks for, downloads and installs FEST updates. Runs on a fixed time on set days.
UpdateOrganizationHierarchyTask	Task that updates the organization hierarchy, only used by Helse Vest. Runs daily at a fixed time, can be disabled.

## 5.4 FEST cache and updates

The FM application server periodically checks for updated versions of FEST via the FEST web service. The address of the service and the frequency of the checks can be configured, but the FEST updates cannot be turned off completely. A FEST update can also be triggered manually by an administrator.

An FM FEST update happens in three main steps:

- 1) Download FEST in the form of an M30 – here the FM can either download a full M30 or an incremental M30. When an update is triggered manually the user can select if a full or incremental file should be downloaded. When an update is triggered automatically an

incremental file is normally downloaded (examples of when a full M30 is downloaded is during the first FEST download for a new FM database, and when the FM is upgraded to depend on a new version of the FEST specification).

- 2) Write new FEST records to DB and deactivate outdated FEST records in DB – during this step all new records (oppføringer) received in the M30 file from step #1 are written to the FestRecords table in the FM database and outdated records are deactivated in the FM database (note that invalidated records are not deleted, merely flagged as inactive).
- 3) Re-index FEST in the FM database – once the FestRecords table is up to date, following step #2, the FEST data in that table is processed and relational data built, during this step the following happens:
  - a. The FestRecordLinks table is rebuilt – this table is an index on the FestRecords table which allows FEST records to be queried based on data from the records (e.g. ATC code, virkestoff-ID, etc).
  - b. The MedicationSearchTerms and MedicationSearchData tables are rebuilt – these tables are used by the medication search in the FM, to provide fast search access to medication data from FEST.
  - c. Interaction data in the FM database is rebuilt – the interaction data from FEST is rolled out into several tables in the FM database (all of which have names beginning with «Interaction»).
  - d. Kortdose and structured dosing data is updated based on new FEST data.
  - e. Handelsvare trees are built, to be used in the NIB and FIB prescription screens in the FM.

Each FEST update executes as a single transaction, with the ACID properties that brings. This means that the FM can continue to be used during FEST updates without risk of mixing different versions of FEST data during other transactions.

It should be noted that during each FEST update, the FM runs once and only once through steps 1-3 described above (the exception is when no new FEST increment is available from the FEST service, in which case steps 2 and 3 do not run). This has two drawbacks:

- If the FM has for some reason missed one or more incremental updates, only one increment will be installed each time the FEST update runs. It is therefore not guaranteed that the FM has been updated to the latest FEST version following a scheduled FEST update.
- Since step #3 is a relatively expensive step, when compared to the first two, doing a series of incremental updates to bring the FM up to the latest FEST version (without missing any incremental changes, which can happen if a full M30 is downloaded) can take longer than strictly necessary.

Both of these issues could be addressed through the simple measure of downloading successive incremental updates during step one, until no update is returned by the FEST service. Followed by running step 2 for each received increment, and finally step 3 only once. The time it would take for step #3 to run for n+1 incremental updates is exactly the same as it would take to run it for n updated, since all of the locally known FEST data is processed during step #3, not just data from the recently downloaded M30.

Since the FM relies heavily on FEST data, and FEST data size is virtually static (when compared to the size and growth of prescription date), access to FEST data is optimized through caching. In fact, the FM application server caches the entire FEST catalogue when it starts up.

When multiple FM application servers are used in a single installation, as described in section 5.8, only the master server runs FEST updates. However, all instances of the FM application server keep their own copy of the FEST cache. It is therefore necessary for the master application server to notify other instances when FEST updates happen so that all application server instances refresh their FEST cache. This happens through the database.

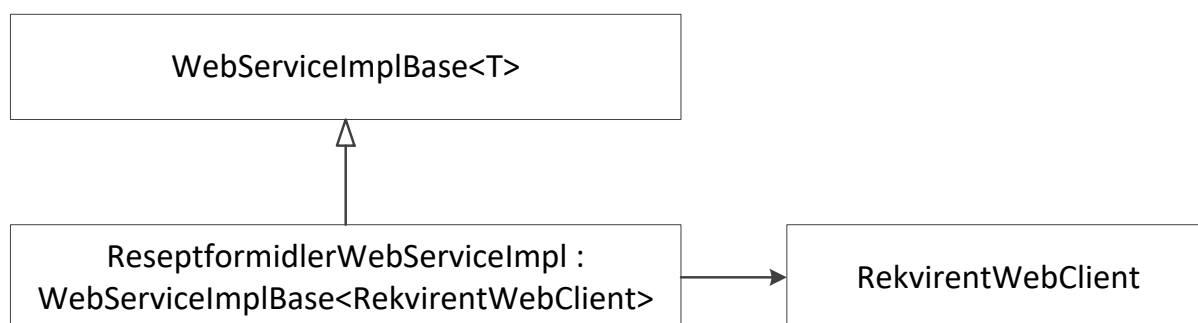
The CacheVersions table in the FM database houses information about the current FEST version. When an application server caches FEST it also reads the current FEST version from CacheVersions. Each time when a new transaction in the application server requires access to FEST, it requests a copy of FEST from a FEST cache manager component (an internal component in the FM application server). When the FEST cache manager receives a request for FEST it checks the version of the currently cached FEST data against the version value stored in CacheVersions. If the two versions match, the requesting transaction is given access to the cached FEST version. If the two versions do not match, a new FEST version is read and a pointer to the newly read FEST version is given to the transaction. This mechanism provides the following benefits:

- Since the cache version is communicated through the database, no direct server to server communication is needed and no complex cache invalidation between the application servers is required.
- Since the FEST cache pointer given to application server transactions is scoped by the DI framework it is guaranteed that each transaction receives only one version of FEST, even if multiple requests from within that transaction are made. This eliminates the risk of errors due to inconsistent FEST data when a transaction starts before a running FEST update completes and ends after the update completes.
- Since the FEST cache manager releases a previous cached FEST version when a new version is read, the previous cached FEST will be garbage collected by the .NET runtime when all transactions using that version have completed.

## 5.5 Synchronous communication

The FM application server is responsible for all communication with external services, the FM client does not directly connect with serviced other than the FM application server.

Figure 6 shows the object model used for web service communication in the FM. In this figure, the "Rekvirent" service of the RF is used as an example.



**Figure 6 - Object model for web service communication in the FM.**

- **WebServiceImplBase<T>** – This is a generic abstract type that serves as a base type for all implementations of web service clients for external services in the FM. The generic type T is a code generated proxy for each service (e.g. “RekvirentWebClient” for the RF rekvirent service and “FestService250Client” for the FEST service). This class handles generic web service communication and logging.
- **RekvirentWebClient** – This is a WCF proxy code generated from the WSDL of the service to be consumed. This class handles the actual web service communication.
- **ReseptFormidlerWebServiceImpl** – This class inherits from **WebServiceImplBase**, providing a WCF proxy (in this example **RekvirentWebClient**) for the generic type T. The role of this class is:
  - To set up the WCF binding (the actual connection), including security and addressing configuration.
  - To implement an interface that is consumed by FM business code that wants to communicate with the RF. In unit tests, this is the class that is mocked to allow RF communication to be unit tested without actually connecting to RF.
  - To provide any message encoding that may be necessary for the web service interface.

Some services utilize messages defined in XML schemas outside the service itself. Examples of this are the e-resept Mx messages exchanged with RF and the M30 message exchanges with FEST. Like described in section 8.1.4, strongly typed classes are generated from these types and used internally in the FM. The result is that interaction with messages, as well as services, is at all times strongly typed and compile-time checked.

As all synchronous communication runs on top of the WCF framework, standard methods of debugging and tracing WCF can be used. This is further described in section 8.3.

## 5.6 Asynchronous communication

All asynchronous communication to and from the FM is handled through folder dropping, i.e. writing and reading text files from a disk directory. All asynchronous communication is handled by the application server.

The FM relies on external mechanisms, e.g. DIPS communicator, for the actual exchange of asynchronous messages with external parties. It is further assumed that those external mechanisms take care of retrying messages that are not answered with an acknowledgement message. The FM does contain a fully implemented retry mechanism, but this is turned off at all times.

When a component of the FM wishes to send an asynchronous message it hands over the message to an internal messaging engine in the FM. There the message is queued up to be sent.

When an asynchronous message is dropped into the folder being monitored by the FM, the messaging engine reads the message, queues it up to be processed and deletes the message file.

Both incoming and outgoing messages that have been queued up for processing, are being processed or have been processed are stored in the FM database. The following tables are used:

- **PendingMessages** – Incoming and outgoing messages that have been queued up for sending or receiving are stored here. Messages usually stop here for a very short time (milliseconds).

- **SentMessages** – Outgoing messages that have been sent, i.e. written to the “outbox” folder used by the FM, are stored here until an acknowledgement has been received or they have timed out waiting for an acknowledgement.
- **RetryLimitReachedMessages** – Outgoing messages that did not receive an acknowledgement before the set timeout. Note that since the retry mechanism in the FM is turned off, so that the FM never retries sending, this table is effectively equivalent to the SentMessages table (when the retry mechanism is active, messages in SentMessages will be retried, but messages in RetryLimitReachedMessages will not). Note that acknowledgements will be received and processed for messages both in SentMessages and in RetryLimitReachedMessages.
- **DoneMessages** – Incoming and outgoing messages that have been fully processed.
- **DuplicateMessages** – Incoming messages that are considered duplicates.
- **ErrorMessages** – Incoming messages that cannot be processed, e.g. because they fail XML schema validation.

When an incoming message is received, it is checked if the message has been received before. If the message has been received before and an acknowledgement sent, then the FM will resent the same acknowledgement.

As no mechanism is in place for the application server to push notifications to the FM client, a running client is not immediately notified when an incoming asynchronous message is processed. An FM client will therefore only become aware of new incoming data when it initiates communication with the application server relating to the incoming data.

## 5.7 Public service API

The FM application server can be configured to offer a web service API that partially mirrors the EPJ API described in section 4.4. As described in that section, methods available on the FM COM API are of three basic types:

- Methods named **Start[x]** – these methods open the FM UI (example, StartPasient, StartInbox).
- Methods named **Les[x]** – these methods return data from the FM (example, LesCave, LesVarslinger).
- Methods named **Skriv[x]** – these methods write data to the FM (example, SkrivCave, SkrivBrukerInfo).

As of version 3.7.0 of the FM, the Les[x] and Skriv[x] methods are available as web service calls on the application server web service API.

The web service API is built using WCF and offers structured service, operation and data contracts. Clients can be automatically generated from the service.

The data contracts used are automatically generated from the XML schema used in the COM EPJ API and thus closely resemble the XML documents exchanged with that API.

The web service API can be enabled and disabled via configuration, further it is possible to select a TCP port number for the service and specify if it should be encrypted. This configuration is done via the FM administration client and is described in the built in FM help.

When communication with the web service should be encrypted, a certificate to use for the encryption must be selected. Further, that certificate must be registered in Windows against the selected TCP port. The FM administration client provides functionality to help with this registration.

The web service API is exposed on the following type of URI:

[http://\[servername\]:\[port\]/FmWebService/\[yyyy-mm-dd\]](http://[servername]:[port]/FmWebService/[yyyy-mm-dd])

Where *servername* is the network name of the Windows server running the FM application server and *port* is the TCP port number where the web service is exposed. [yyyy-mm-dd] is a date that represents the EPJ API version. A concrete example of an address might be:

<http://myFmServerMachine:9999/FmWebService/2016-06-21>

New versions of the EPJ API get a new version number, in the form of a date, and get a new URI with the new version number.

When encryption is enabled, http at the start of the URI is replaced with https.

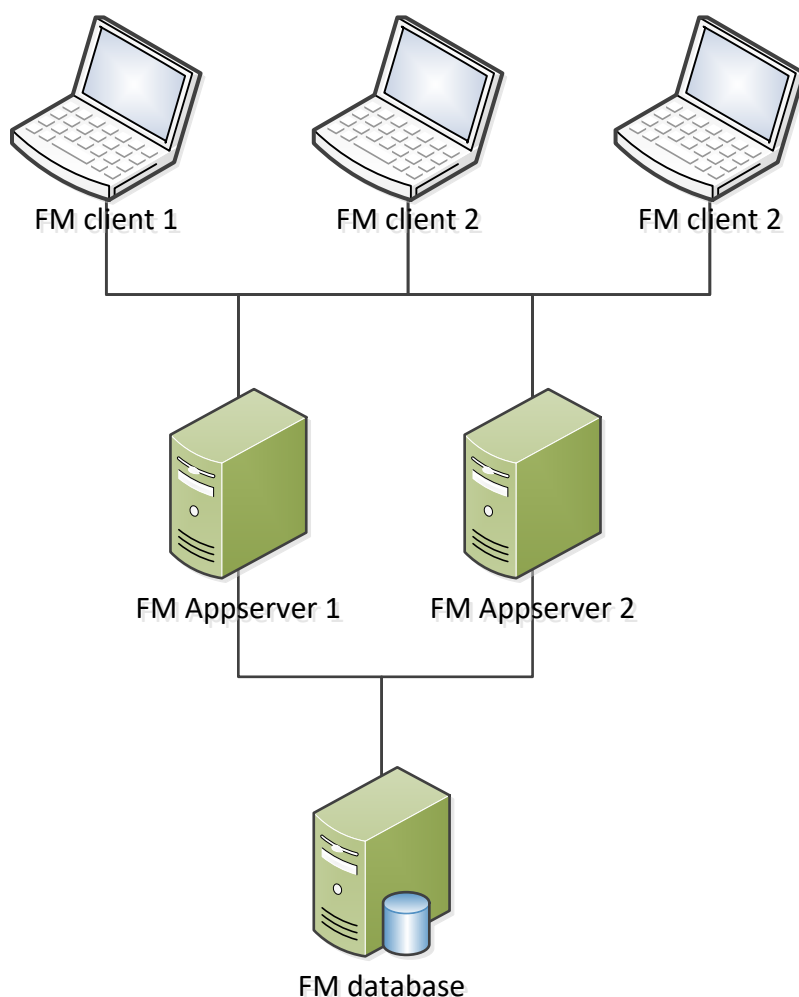
## 5.8 Clustering

All FM installations rely on a single database (see section 7 for information about database sharding) and most installations will support multiple FM clients. In addition, it is possible to run multiple FM application servers in a single FM installation, **Villa! Uppruni tilvísunar finnst ekki.** illustrates an example of this. Although the example shown contains two application servers, there is no set limit on the number of application servers that can be installed.

There are two possible reasons for running more than one application server in an installation:

- Load-balancing: The available application servers will automatically share the load of servicing FM clients.
- Failover: If one application server goes offline, the remaining application servers continue to service FM clients.





**Figure 7 - Two FM application servers sharing client load and providing failover.**

The FM uses the Microsoft SQL Server DBMS which implements standard solutions for providing scalability and reliability. It is therefore outside the scope of FM development to address those things on the database level.

An FM application server can run in the role of either master or slave. At each time, only one application server can assume the master role against each FM database, but any number of application servers can run in the slave role.

An application server in the master role services requests from FM clients, runs scheduled tasks (such as FEST updates) and handles database migrations if needed when started. An application server in the slave role only services requests from FM clients but does not run any schedules etc.

Since an FM application server cannot start without access to an FM database, the role of an application server is controlled through the database. This is implemented via a mechanism where the master application server will write a lock, identified as `System.MasterRoleLock` into the `SystemConfiguration` table. The lock consists of a timestamp and a server identifier, which again consists of the server name and a port number. An example of a lock value is "09/28/2016 12:55:56|myFmServer:8903".

When an FM application server starts, it reads the master role lock from the database, and if the lock exists it checks the timestamp of the lock. If the lock either does not exist or is more than 20 seconds old, the newly started application server will assume the master role and proceed to write a new lock



value into the FM database identifying itself as being in the master role. An application server in the master role will refresh the master role lock every 10 seconds for as long as it remains running.

All running application servers confirm their running status by writing an entry into the SystemServers table in the FM database every 60 seconds. Each entry contains a server identifier, which takes the same form as the server identifier in the master role lock, and a timestamp.

The mechanism for load-balancing and failover is described in section 6.3.

## 5.9 Trace events and monitoring

This section describes mechanisms that can be used to check the status of the FM application server and to automatically monitor it using standard system monitoring applications.

These mechanisms come in three forms, each can be used individually, or they can be used together:

- **Trace events** – The FM application server traces out events of the types Information, Warning and Error. By default these events are traced out to the Windows Event log where they can be manually examined or automatically monitored. Events have unique event type numbers and many warning and error events specify the type number of an information even that clears the warning/error if and when the situation leading to that warning/error is automatically resolved (as an example, error event 20210 reports an error in connecting to the FM database, this event can be automatically cleared by the information event 20211 which reports a successful database connection).
- **WMI objects** – The FM application server publishes WMI (Windows Management Instrumentation) objects that provide generic information about the application server and specific information about scheduled tasks and internal services running in the application server.
- **Performance counters** – Windows Performance Counters are named counters that can be read by the Performance monitor application built into Windows and by system monitoring software. The FM application server provides performance counters that can be used to monitor communications with external services (these can e.g. be used to monitor the number of sent, successful and failed messages exchanged with RF over time). The .NET framework, and WCF specifically, also expose a number of performance counters that can be used to monitor the FM application server.

### 5.9.1 Trace events

Events traced out to the Windows Event Log by the FM application server (and the FM client, as described below) all take the same general format:

- **Event Id** – A unique integer that identifies the event.
- **Event Type** – Information, Warning or Error. A fourth type exists, Verbose, but this is normally not enabled.
- **Message body** – A short textual description of the event.
- **Message details** – A more detailed textual description of the event. This is not included for all events.
- **User action** – Information about what an administrative user can do to resolve a warning or an error. This is not included for all events.
- **Additional data** – Other relevant data. This typically contains a stack trace

An example of an actual trace event follows. An integer within curly braces (e.g. {0}) represents dynamic text that is included in the event. An n preceded by a backslash (\n) represents a line break.

- Event Id – 20000
- Event Type – Warning
- Message body – An error occurred when processing an incoming message from {0}:\n{1}
- Message details – Processing will be retried until it succeeds or the message is manually removed from the input folder. This warning may be automatically cleared by an information event 20086.
- User action – User action:\nIf the error is caused by a temporary network or database problem, the processing of the message will eventually succeed when the problem has been fixed, so no user action is required. If, however, the error is caused other reasons, so reprocessing the message will never succeed ("poison message"), it must be manually removed from the folder. Note: since no NACK message is sent in this case, it is likely that the sender of the message will resend it and this situation will emerge again.
- Additional data – Additional data:\nIncoming message:\n{0}\n\nException thrown:\n{1}

Note that the FM client also traces out errors to the Windows Event Log. Those are traced out on the computer running the FM client, while events from the FM application server are traced out on the application server computer.

### 5.9.2 WMI objects and performance counters

Refer to *E-resept FM - Installation and Configuration Guide*, section 4.4, for information about WMI and performance counters.

### 5.9.3 MS SQL Server

The MS SQL Server used by the FM can be monitored in standard ways. This is outside the scope of this document.

## 6 Client-server communications

This section describes the communication between the FM client and application server. Some of the information in this section has already been covered to some extent in the preceding sections, but is gathered here in one place.

### 6.1 General information

All communication between the FM client and application server is built on the .NET WCF framework. The FM application server exposes a range of SOAP services implemented on top of WCF that the FM client consumes.

As described earlier in this document, the FM defines a strict contract between the client and application server. This contract is implemented as a set of data contracts, service contracts and operation contracts. See <https://msdn.microsoft.com/en-us/library/ff183866.aspx> for information about contracts in WCF.

The data contracts are simply implemented as .NET classes that do not define any behaviour. An example of a simple data contract is shown below:

```
[DataContract]
public class RateInfo
```

```
{
    [DataMember]
    public Guid Id { get; set; }

    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public string RateId { get; set; }
}
```

The service and operation contracts are defined as .NET interfaces with associated methods and functions. Both the data contract classes and service and operation contract interfaces are shared between the client and application server (i.e. they are housed in the FM.Common assembly which is included both in the client and application server). An example of a simple service and operation contract is shown below:

```
[ServiceContract]
public interface IWcfRateService
{
    /// <summary>
    /// Register rates for a patient.
    /// </summary>
    /// <param name="patientId">The id of the patient.</param>
    /// <param name="rates">The rate records that should be saved for the patient.</param>
    [OperationContract]
    void RegisterRate(Guid patientId, IList<RateInfo> rates);
}
```

On the client side, the service and operation contracts are implemented by service proxy classes that derive from base classes that take care of all aspects of the client-server communication, such as versioning, security, load-balancing, etc. The result is that when writing code, accessing the application server from the FM client is no different from accessing functionality built into the client itself.

In the application server, the service and operation contracts are implemented by actual service classes that know how to handle the incoming requests. It should be noted that those classes are registered against the WCF framework and do not themselves implement anything that has to do with the technical aspect of the client-server communication. This means that the server-side service implementations can freely depend on each other without incurring any of the overhead normally associated with accessing a SOAP service.

The fact that the client-server contracts are defined in code, which is built into both client and server, means that full compile-time checking is performed for those contracts and there is no possibility for the client and server end of the contracts to fall out of sync. Section 6.2 describes how the FM ensures that a given version of the FM client can only make calls to the same exact version of the FM application server.

## 6.2 Version control

The FM client and application server share strict service, operation and data contracts. It is important that both the client and the application server use and expect the same version of these contracts at each time. The contracts are versioned together with the FM software, this means that only clients and application servers of the exact same version should be allowed to communicate.

This is ensured through the use of a custom header in the SOAP requests made from the FM client to the application servers. In every call from the FM client, a SystemVersion element is added to the SOAP header which informs the application server of the software version of the calling client. Before processing the request, the application server inspects the SystemVersion header, compares it with its own software version and rejects the call if the versions do not match.

It should be noted that the SystemVersion header contains the entire FM version number, including a revision number at the end. This ensures that even different release candidates of the same FM version cannot communicate.

Below is an example of a SOAP request made from the FM client. The SystemVersion header is highlighted in yellow.

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:u="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <s:Header>
    <a:Action
s:mustUnderstand="1">http://tempuri.org/IWcfAnonymousConfigurationService/IsStagingMode</a:
:Action>
    <a:MessageID>urn:uuid:e65f9183-95c0-4c81-8844-813c21ad83a2</a:MessageID>
    <a:ReplyTo>
      <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
    </a:ReplyTo>
    <SystemVersion
xmlns="http://theriak.com/PrescriptionModule">3.7.0.13713</SystemVersion>
    <a:To
s:mustUnderstand="1">net.tcp://localhost:8903/IWcfAnonymousConfigurationService</a:To>
    <o:Security s:mustUnderstand="1" xmlns:o="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <u:Timestamp u:Id="_0">
        <u:Created>2016-09-26T15:48:37.364Z</u:Created>
        <u:Expires>2016-09-26T15:53:37.364Z</u:Expires>
      </u:Timestamp>
      <c:SecurityContextToken u:Id="uuid-b19abd7f-c268-4250-ad99-7b7fbce2e768-5"
xmlns:c="http://schemas.xmlsoap.org/ws/2005/02/sc">
        <c:Identifier>urn:uuid:2188caae-0ef8-403a-b6d1-b157e2dead50</c:Identifier>
      </c:SecurityContextToken>
      <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        <SignedInfo>
          <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#"/>
          <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-
sha1"/>
          <Reference URI="#_0">
            <Transforms>
              <Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
            </Transforms>
            <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
            <DigestValue>I5M21AkNxfrRhui0B4FtNenpt3Q=</DigestValue>
          </Reference>
        </SignedInfo>
        <SignatureValue>S20SUD4EIg+q58s3UyQMxgzodGM=</SignatureValue>
        <KeyInfo>
          <o:SecurityTokenReference>
            <o:Reference URI="#uuid-b19abd7f-c268-4250-ad99-7b7fbce2e768-5"/>
          </o:SecurityTokenReference>
        </KeyInfo>
      </Signature>
    </o:Security>
  </s:Header>
  <s:Body>
    <IsStagingMode xmlns="http://tempuri.org/">
  </s:Body>
</s:Envelope>
```

## 6.3 Load-balancing and failover

Section 5.8 describes how the FM application server can be clustered. This section builds on the information from that section and describes how the FM client obtains information about running application servers and then utilized that information to provide load-balancing and failover.

### 6.3.1 Server side

If the configuration parameter identified as LoadBalancingEnabled has a value of 1 in the SystemConfiguration table, FM application servers will include information about available application servers in the SOAP header of responses sent back to FM clients. Available application servers are

considered to be those servers who have confirmed their status in the SystemServers table during the last 90 seconds.

This means that when an FM client is started, it is sufficient for the client to have knowledge of one running application server. As soon as the first request is made by the client to that application server, the client will receive information about all currently available application servers. The following section explains how this information is used by the FM client.

The request response example below shows how FM application server addresses are propagated back to clients in the ServerAddresses header:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:u="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <s:Header>
    <a:Action
s:mustUnderstand="1">http://tempuri.org/IWcfRequestService/InvokeResponse</a:Action>
    <ActivityId CorrelationId="5bc5f94d-0c2c-48d5-a681-b64e5585d669"
xmlns="http://schemas.microsoft.com/2004/09/ServiceModel/Diagnostics">10753015-0795-48cd-
9298-135d2b6f372d</ActivityId>
    <a:RelatesTo urn:uuid:fd1e309e-4b25-45d0-8c47-e4abbed28726</a:RelatesTo>
    <ServerAddresses xmlns="http://theriak.com/PrescriptionModule"
xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
      <string
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">applicationServer1:8903<
/string>
      <string
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">applicationServer2:8903<
/string>
      <string
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">applicationServer3:8903<
/string>
      <string
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">applicationServer4:8903<
/string>
    </ServerAddresses>
    <a:To s:mustUnderstand="1">http://www.w3.org/2005/08/addressing/anonymous</a:To>
    <o:Security s:mustUnderstand="1" xmlns:o="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <u:Timestamp u:Id="_0">
        <u:Created>2016-09-28T11:36:05.855Z</u:Created>
        <u:Expires>2016-09-28T11:41:05.855Z</u:Expires>
      </u:Timestamp>
    </o:Security>
  </s:Header>
  <s:Body>
    <!-- Removed-->
  </s:Body>
</s:Envelope>
```

## 6.3.2 Client side

When an FM client has knowledge of multiple running application servers it will round-robin between all available servers when opening connections. Since the same connection can be used for multiple requests, this does not mean that each request will go to a new application server, but whenever a new connection is required, the next server in line is chosen. This results in load from clients being distributed across all available application servers.

Each client maintains two lists of application server addresses:

- Whitelist: Servers that are known to be running (i.e. the server provided a response the last time it was used).
- Blacklist: Servers that are assumed not to be running (i.e. calling the server failed the last time it was used).

These lists are stored in the Windows registry and continuously updated by the FM client. These lists are not meant to be manually maintained by users or administrators.

Additionally, server addresses can be stored in the `ServerAddressAndPort` value in the Windows registry. This value is not automatically updated by the FM client and is meant for configuration by administrators. The value in `ServerAddressAndPort` is assumed to consist of application server name and port numbers, separated by semi-colons. Examples of valid values are: "applicationServer1:8903" and "applicationServer1:8903;applicationServer2:8903". If `ServerAddressAndPort` does not exist, or is empty, it defaults to the value "localhost:8903".

When the FM client first starts, it reads the value of `ServerAddressAndPort`, randomizes the order of addresses contained in `ServerAddressAndPort` and creates an in-memory whitelist of application server addresses. The client then reads the whitelist stored in registry, and adds addresses from the whitelist not present in the in-memory whitelist to an in-memory blacklist. The registry blacklist is then read and added at the end of the in-memory blacklist.

The result is that once started, the FM client has an in-memory whitelist of addresses configured in `ServerAddressAndPort` (in a randomized order) and an in-memory blacklist of addresses from the automatically maintained white- and blacklists from the registry (where the whitelist is given a higher priority by being placed at the start of the in-memory list).

When a connection is to be opened to an FM application server the client tries the whitelist address next in line and, if the connection succeeds, rotates the address to the back of the whitelist so that connections round-robin over the entire whitelist. If a connection to an address from the whitelist fails, the address is moved to the blacklist and the next address from the whitelist tried. If all addresses from the whitelist fail, the client will proceed to try the addresses from the in-memory blacklist before giving up and presenting the user with a connection error.

When a request is successfully made from the client to an application server, the server address list piggybacked in the SOAP header of the response (see the previous section) is used to update the server white- and blacklists. This means that if a server goes offline, it will end up in the blacklist of all clients that try to use that server, but as soon as the application server comes back and announces itself to the FM database, it will be reported as available to the clients again by the other running application servers and will be taken into use again. This mechanism also means that there is no need to configure clients to use new application servers when they are introduced, as they will be taken into use dynamically by all clients.

## 6.4 Client authentication and encryption

This section describes how the network connection between the FM client and application server is secured, both how it is encrypted and how the identity of the end user is communicated to the application server.

### 6.4.1 Transport security and server authentication

The FM client and application server communicate over an SSL encrypted TCP connection. A WCF binding of the type `NetTcpBinding` is used.

A self-signed certificate, built into the FM assemblies is used for encrypting the communication channel and to provide server authentication.

Server authentication means that the FM client will only accept connections with application servers using the pre-approved self-signed certificate.

## 6.4.2 User authentication

When a communication channel is opened, a security context is negotiated between the client and the application server. This follows established WS-Trust standards and is handled by WCF (i.e. this is not custom code written in the FM). During this process the client sends a username and password to the application server (over the SSL connection) and a security context token (SCT) is created which is used for subsequent SOAP requests that pass over the communication channel.

Since the username and password of the actual user is used to authenticate against the FM application server, connecting to the server will fail if the username and/or password are not correct. This means that if an EPJ makes a call to the EPJ API with an incorrect password, the FM client will not be able to open a connection to the FM application server at all. The same holds true when a user tries to log into the FM administration dashboard and types in an incorrect password. The result is that the FM client may return a generic error of a failed connection, or unreachable application server, when user authentication fails.

An example of the SOAP request sent from the client during the security context establishment process is shown below. The standard SCT request action identifier and the UsernameToken are highlighted in yellow.

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:u="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <s:Header>
    <a:Action
s:mustUnderstand="1">http://schemas.xmlsoap.org/ws/2005/02/trust/RST/SCT</a:Action>
    <a:MessageID>urn:uuid:c677364a-59a6-4d16-a4fd-257f5ae7f403</a:MessageID>
    <a:ReplyTo>
      <a:Address>http://www.w3.org/2005/08/addressing/anonymous</a:Address>
    </a:ReplyTo>
    <a:To s:mustUnderstand="1">net.tcp://egir:8903/IWcfRequestService</a:To>
    <o:Security s:mustUnderstand="1" xmlns:o="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
      <u:Timestamp u:Id="_0">
        <u:Created>2016-09-28T11:35:08.029Z</u:Created>
        <u:Expires>2016-09-28T11:40:08.029Z</u:Expires>
      </u:Timestamp>
      <o:UsernameToken u:Id="uuid-afb01793-9a85-4033-b454-37d5b2defc82-13">
        <o:Username>
          <!-- Removed-->
        </o:Username>
        <o:Password>
          <!-- Removed-->
        </o:Password>
      </o:UsernameToken>
    </o:Security>
  </s:Header>
  <s:Body>
    <t:RequestSecurityToken xmlns:t="http://schemas.xmlsoap.org/ws/2005/02/trust">
      <t:TokenType>http://schemas.xmlsoap.org/ws/2005/02/sc/sct</t:TokenType>

      <t:RequestType>http://schemas.xmlsoap.org/ws/2005/02/trust/Issue</t:RequestType>
      <t:Entropy>
        <!-- Removed-->
      </t:Entropy>
      <t:KeySize>256</t:KeySize>
    </t:RequestSecurityToken>
  </s:Body>
</s:Envelope>
```

An example of a response provided by the FM application server is shown below, the standard response action and the SCT data are highlighted:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
xmlns:a="http://www.w3.org/2005/08/addressing" xmlns:u="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <s:Header>
    <a:Action
s:mustUnderstand="1">http://schemas.xmlsoap.org/ws/2005/02/trust/RSTR/SCT</a:Action>
```



```

    <a:RelatesTo>urn:uuid:c677364a-59a6-4d16-a4fd-257f5ae7f403</a:RelatesTo>
    <ActivityId CorrelationId="3824955c-2f9d-43a4-9945-3873f7444833"
    xmlns="http://schemas.microsoft.com/2004/09/ServiceModel/Diagnostics">8581dd48-9dd9-4602-
    beb3-ac3b21c40aba</ActivityId>
    <a:To s:mustUnderstand="1">http://www.w3.org/2005/08/addressing/anonymous</a:To>
    <o:Security s:mustUnderstand="1" xmlns:o="http://docs.oasis-
    open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
        <u:Timestamp u:Id="_0">
            <u:Created>2016-09-28T11:35:08.269Z</u:Created>
            <u:Expires>2016-09-28T11:40:08.269Z</u:Expires>
        </u:Timestamp>
    </o:Security>
</s:Header>
<s:Body>
    <t:RequestSecurityTokenResponse
    xmlns:t="http://schemas.xmlsoap.org/ws/2005/02/trust">
        <t:TokenType>http://schemas.xmlsoap.org/ws/2005/02/sc/sct</t:TokenType>
        <t:RequestedSecurityToken>
            <c:SecurityContextToken u:Id="uuid-a2165cc5-07ea-45bd-a6f5-a8e0e5bc67a9-7"
            xmlns:c="http://schemas.xmlsoap.org/ws/2005/02/sc">
                <c:Identifier>urn:uuid:8daeadd7-65a5-41dc-9f51-
                170676dacb5b</c:Identifier>
            </c:SecurityContextToken>
        </t:RequestedSecurityToken>
        <t:RequestedAttachedReference>
            <o:SecurityTokenReference xmlns:o="http://docs.oasis-
            open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
                <o:Reference URI="#uuid-a2165cc5-07ea-45bd-a6f5-a8e0e5bc67a9-7"/>
            </o:SecurityTokenReference>
        </t:RequestedAttachedReference>
        <t:RequestedUnattachedReference>
            <o:SecurityTokenReference xmlns:o="http://docs.oasis-
            open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
                <o:Reference URI="urn:uuid:8daeadd7-65a5-41dc-9f51-170676dacb5b"
                ValueType="http://schemas.xmlsoap.org/ws/2005/02/sc/sct"/>
            </o:SecurityTokenReference>
        </t:RequestedUnattachedReference>
        <t:RequestedProofToken>

        <t:ComputedKey>http://schemas.xmlsoap.org/ws/2005/02/trust/CK/PSHA1</t:ComputedKey>
        </t:RequestedProofToken>
        <t:Entropy>
            <!-- Removed-->
        </t:Entropy>
        <t:Lifetime>
            <u:Created>2016-09-28T11:35:08.268Z</u:Created>
            <u:Expires>2016-09-29T02:35:08.268Z</u:Expires>
        </t:Lifetime>
        <t:KeySize>256</t:KeySize>
    </t:RequestSecurityTokenResponse>
</s:Body>
</s:Envelope>

```

### 6.4.3 Limitations in current security model

It should be noted that while the encryption used between the FM client and application server is strong and follows accepted standards, the confidentiality of the certificate used is weak. This is due to the certificate being built into the FM client and application server.

A reasonably skilled programmer could extract the certificate from the FM assemblies and either pose as an FM application server, implement a man-in-the-middle attack between the client and application server, or pose as an FM client towards the application server.

The design assumption is that the FM is installed on secure networks, much like the EPJ systems it communicates with, and the purpose of encrypting the communication is simply to ward off basic network snooping attempts.

It should be further noted, that by replacing the self-signed, built in certificate, with an officially issued SSL certificate, the security of the FM client and application server communication can be made strong. This would add complexity and cost to setting up a new FM installation (a certificate would



have to be purchased for the installation, installed on the application server computer and selected in the FM configuration).

## 6.5 Fault contracts and error handling

Exceptions that occur in the application server are of course .NET exception types. Those types are not serializable, i.e. they cannot propagate between the application server and the client. For this reason, a fault contract is defined as part of the data contracts between the FM client and application server. The behavior of the WCF services is such that when an exception that should be propagated to the client occurs server-side, the exception is mapped to a fault object that can traverse the network boundary to the FM client. This fault object is defined as part of the client-server data contract and can be handled by the client.

Fault objects received by the FM client can be managed in specific ways, for example when the application server detects a server mismatch. Or they can be managed generically, essentially leading to a red error dialog in the FM client.

## 7 Database

The FM depends on Microsoft SQL Server for data storage. The FM uses MSSQL compatibility level 100 and supports MSSQL 2008 R2, 2012 and 2014.

### 7.1 Database versioning

The FM database is versioned. The version is stored in the SystemConfiguration table, as a value with the Id System.DatabaseSchemaVersion. When the FM application server starts and connects to the FM database, it starts by checking the database version value stored in the database and compares it with the database version expected by the FM application server:

- If the database version is too high, the application server traces out an error event with the Id 20143 and then stops. This indicates that the database has been updated to a newer FM version than the one being run.
- If the database version is too low, then the application server upgrades the database to the expected version and then starts.
- If the database version matches the expected one, then the application server starts.

This means that when the FM is updated to a new version, it is sufficient to stop the application server, install the new version, and start the application server again. The application server then automatically takes care of updating the database to the latest version. The client of course has to be upgraded also.

Each version of the FM can contain multiple database updates, which each have their own version number. When the database is upgraded, each of those updates is executed as a single transaction. This means that failed database upgrades are always rolled back and can safely be retried. However, when a version update requires the installation of more than one version, and the first one succeeds but one of the later ones fail, there is no way to downgrade to the previous FM version. The error must be resolved, sometimes with help from technical support, and the upgrade retried. For this reason it is recommended to back up the FM database before upgrading.

### 7.2 Database sharding

The FM supports both SQL Server Express and full versions of SQL Server.

When running on a full version of SQL Server, all data is stored in a single database as the database size is only limited by disk space.

Database size in SQL Server Express is limited to 10 GB per database. Each database server can however host multiple databases, the limitation is just that each one cannot grow beyond 10 GB.

Since the FM stores a lot of signed XML documents, the FM database can grow very large. These documents are compressed in the database, but the database can still easily grow much larger than 10 GB over time. To support this on SQL Server Express, the FM automatically does database sharding, i.e. it splits the stored data across multiple databases.

XML documents are all stored in a table called “Documents” in the FM. This is the table that occupies virtually all of the storage space required by the database. The FM monitors the size of the database, and when it nears 9,5 GB in size, a secondary database is automatically created where newly created XML is stored.

On a newly created FM the following databases exist:

- fm (assumption here is that the main database is simply called “fm”, but it can be called by any name).
- fm.Documents.0.

The “fm” database contains a view called “Documents” and insert, update and delete triggers on that view. Thus, to the FM it looks like “Documents” is a single table, while it is in fact a view. The results of the view in the above example returns a join of the contents of a “Documents” table in the “fm.Documents.0” database and a “Documents.0” table in the main “fm” database (this is for backwards compatibility, as database sharding was not part of early FM versions).

When the “fm.Documents.0” database reached 9,5 GB in size, the FM application server automatically creates a new document called “fm.Documents.1”, updates the “Documents” view in the main database to include records from the newly created database and updated the triggers on that view in order to allow documents to be added, updated and deleted correctly.

## 8 Miscellaneous

### 8.1 Components used

This section lists 3<sup>rd</sup> party libraries and components used FM development. Libraries and components that are part of the .NET framework are not listed (e.g. WCF and WPF).

#### 8.1.1 StructureMap

StructureMap is the DI / IoC container used in the FM. It allows implementations to be registered against interfaces in a central location, and business code to list dependencies on those interfaces. The implementation provided for each interface can then differ depending on the context at each time. As an example, when the client requests an instance of a service interface, StructureMap returns a web service proxy that makes calls to the application server, when the application server requests an instance of the same interface, StructureMap returns a reference to a plain old .NET object. In both cases the code written against the interface is exactly the same, but what happens when a call is made is very different. When the same service interface is requested by a class undergoing unit testing, a mocked implementation is provided as described in the following section.

For information about StructureMap refer to:

<http://structuremap.github.io/>

### 8.1.2 Rhino Mocks

Rhino Mocks is a mock object framework used in automated unit tests in the FM. Rhino Mocks is used to provide “mocked” or faked implementations of interfaces during unit testing. For example, when a unit test is written for a business object that depends on a repository interface that gives access to RF, Rhino Mocks is used to predefine the calls that should be made to the repository during the test and the answers that should be sent back. This allows for automatic testing of processing different types of responses RF may give when messages are sent to it, errors that may come up in RF communication and many other things.

For information about RhinoMocks refer to:

<https://www.hibernatingrhinos.com/oss/rhino-mocks>

### 8.1.3 NUnit

NUnit is the automatic unit test framework used in FM development.

For information about NUnit refer to:

<http://www.nunit.org/>

### 8.1.4 LinqToXsd

LinqToXsd is used for providing support for typed XML programming. All e-resept messages are defines in XML schemas and are exchanged as XML messages. .NET code is automatically generated by LinqToXsd for use in the FM based on these schemas. This means that FM developers never directly touch XML exchanged by the FM, all interaction with XML is through strongly typed classes which are automatically code generated.

For information about LinqToXsd refer to:

<https://linqtoxsd.codeplex.com/>

### 8.1.5 LinqToSql

LinqToSql is the ORM used in the FM. LinqToSql is used for generating a strongly typed object model from the FM database.

For information about LinqToSql refer to:

[https://msdn.microsoft.com/en-us/library/bb386976\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/bb386976(v=vs.110).aspx)

### 8.1.6 SharpZipLib

SharpZipLib is a compression library used in the FM to compress and decompress files during data import and export, and during automatic FM database backups.

For information about SharpZipLib refer to:

<https://icsharpcode.github.io/SharpZipLib/>

## 8.2 Kodeverk

Coding systems (kodeverk) that are part of FEST are read from FEST and updated to those coding systems are therefore imported during FEST updates.

Other coding systems are hard coded into the FM.Common assembly and require a new FM version when they are to be updated.

## 8.3 Custom tracing for troubleshooting

### 8.3.1 WCF logging on the application server

WCF contains very powerful tracing options that can be used for debugging. These can be tuned on by editing the “eResept Forskrivningsmodul Server.exe.config” file in the disk folder where the FM application server is installed.

By default this file looks like this:

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <generatePublisherEvidence enabled="false" />
  </runtime>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
</configuration>
```

By modifying the file as follows and restarting the application server service the FM can be made to output detailed logs for all incoming and outgoing service communication:

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <generatePublisherEvidence enabled="false" />
  </runtime>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.0"/>
  </startup>
  <system.diagnostics>
    <sources>
      <source name="System.ServiceModel" switchValue="Information, ActivityTracing"
        propagateActivity="true" >
        <listeners>
          <add name="xml"/>
        </listeners>
      </source>
      <source name="System.ServiceModel.MessageLogging">
        <listeners>
          <add name="xml"/>
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="xml" type="System.Diagnostics.XmlWriterTraceListener"
        initializeData="C:\temp\eResept Forskrivningsmodul - Server Traces.svclog" />
    </sharedListeners>
  </system.diagnostics>

  <system.serviceModel>
    <diagnostics wmiProviderEnabled="true">
      <messageLogging
        logEntireMessage="true"
        logMalformedMessages="true"
        logMessagesAtServiceLevel="true"
        logMessagesAtTransportLevel="true"
        maxMessagesToLog="30000"
        maxSizeOfMessageToLog="999999999"
      />
    </diagnostics>
  </system.serviceModel>
</configuration>
```

The configuration above will produce a file called “C:\temp\eResept Forskrivningsmodul - Server Traces.svclog”. This file can be viewed using the Microsoft Service Trace Viewer, part of the Windows SDK.

### 8.3.2 EPJ API logging on the client

The FM client can be made to log out both timing information related to the EPJ API and the XML documents sent into and returned by the API.

This functionality is turned on as follows:

1. Ensure the following key exists in the Windows registry, on the client PC where the FM client is to be logged:

```
HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Theriak\eResept Forskrivningsmodul\DevTest
```

2. Create the following values in the above key:

Value name	Type	Data
LogEpiApiTimingInformation	REG_DWORD	"1" for enabled. "0" for disabled.
EpiApiTimingInformationFilename	REG_SZ	The file name of the log file, e.g. "C:\temp\fmclientlog.txt".
LogEpiApiXml	REG_DWORD	"1" for enabled. "0" for disabled.
EpiApiXmlLogDirectory	REG_SZ	The directory where EPJ API XML messages should be stored. Each message is stored in a new file. An example of a possible value is "C:\temp\FmClientLog\".

### 8.3.3 Logging M30 to disk during FEST updates

The FM application server can be configured to save a copy of downloaded M30 messages to disk during FEST updates. This is done as follows:

1. Ensure the following key exists in the Windows registry, on the FM application server PC:

```
HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Theriak\eResept Forskrivningsmodul\Server\DevTest
```

2. Create the following values in the above key:

Value name	Type	Data
M30LoggingEnabled	REG_DWORD	"1" for enabled. "0" for disabled.
M30LogFolder	REG_SZ	The directory where the M30 messages should be stored. Each message is stored in a new file. An example of a possible value is "C:\temp".

## 9 Development environment

The FM is written in C# on top of the .NET framework 4 framework. The full framework is used for all components, client and server. Since SHA-256 is not supported in version 4 of .NET, the stated required version for the FM is .NET 4.7.1. SHA-256 is required for communication with Kjernejournal.

The FM client, application server, installers and several utilities and test tools are all bundled in a single Visual Studio solution. Also included in the solution are all required libraries and frameworks, apart from the .NET framework (ref. section 8.1 for a list of those libraries and frameworks).

Since all dependencies are included in the FM solution, the solution can be opened with a fresh Visual Studio Professional installation and built. No components beyond the Visual Studio development environment need to be installed.

Microsoft Visual Studio 2010 and later are supported. At the time of this writing, most development happens on Microsoft Visual Studio 2015.

Note that Microsoft Visual Studio 2010 is required for building the FM installers. This is therefore the version used in the automatic build environment for the FM.

FM code is housed in a Subversion version control repository (for information about Subversion, see <https://subversion.apache.org/>). Whenever a code change is committed to the repository, a continuous integration mechanism powered by TeamCity picks up the code changes, builds the entire FM solution and runs all defined unit tests (for information about TeamCity, see <https://www.jetbrains.com/teamcity/>). At the time of this writing, over 2000 unit tests are defined in the FM codebase.

If the CI build fails, or any of the defined unit tests, the developer responsible is automatically notified and must fix the build before a version can be built for testing.

In addition to providing the CI mechanism, TeamCity also runs “daily builds” of the FM code. Daily builds run every night and leave production-build versions used by FM testers for both daily testing and system testing. The daily builds can also be triggered on-demand when new builds are wanted for testing.

When a build of the FM has been approved for release (or as a release candidate), the same daily build approved by FM testers is published (i.e. official FM builds are produced by exactly the same automatic build mechanism that builds versions for FM testers).

It should be noted that both the version control system used and the build system and build mechanism used are external to the FM and could be replaced by any other version control system or build system without affecting FM functionality or FM code.