Linköping University
Department of Computer and Information Science

Final Thesis

System design for integration of fault isolation and recording tool for airplanes

by

Oscar Tholander

LIU-IDA/LITH-EX-G—13/037-SE

2013-07-02

Supervisor: Lena Buffoni
IDA, Linköping University
Johan Gransten
Saab Aeronautics, Linköping

Examiner: Lena Buffoni
IDA, Linköping University
Abstract

This thesis investigates the possibilities to design enterprise software after the REST architectural style and also look into how these kinds of systems can be developed using the .NET 4.0 Framework, Windows Communication Foundation (WCF) and EntityFramework 4.0. The thesis will in detail go through how to design the database abstraction to transferring the database content over a network, encoded with JSON to a client. The proposed solution is a viable way of designing network based software and is quite simple to implement when you get you head around the concepts of REST. The future in developing REST based applications is getting better and easier in the .NET Framework with added and improved support for in newer releases of the framework.

Linköping, Sweden, May 2013

Oscar Tholander
Acknowledgements

This bachelor thesis was done at Saab Aeronautics in Linköping. I would like to thank my supervisor Johan Gransten for the support he has given me during this thesis. I would also like to thank Torbjörn Fransson for discussions of possible thesis project and for his insight of the current systems at Saab and help with defining the key features of the system.

At last I would like to thank my family and friends for the support and motivation they have given.

*Linköping, Sweden, May 2013*

*OSCAR THOLANDER*
1 INTRODUCTION ............................................................................................................... 5
   1.1 BACKGROUND ........................................................................................................ 5
   1.2 PURPOSE .................................................................................................................. 5
   1.3 GOALS .................................................................................................................... 5
   1.4 SCOPE ..................................................................................................................... 5
   1.5 ABBREVIATIONS .................................................................................................... 6

2 TECHNOLOGIES ............................................................................................................. 7
   2.1 HYPERTEXT TRANSPORT PROTOCOL .................................................................. 7
   2.2 UNIFORM RESOURCE LOCATOR (URL) ................................................................. 8
   2.3 REPRESENTATIONAL STATE TRANSFER ............................................................... 9
       2.3.1 Constraints ................................................................................................... 9
       2.3.2 REST Elements ......................................................................................... 10
   2.4 RESTful WEB SERVICE ....................................................................................... 15
   2.5 SOAP .................................................................................................................... 17
   2.6 PLATFORM SPECIFIC TECHNOLOGIES .............................................................. 20
       2.6.1 Windows Communication Foundation (WCF) .............................................. 20
       2.6.2 Entity Framework 4.0 ................................................................................ 21
   2.7 DEVELOPMENT PROCESSES AND PRINCIPLES ............................................. 21
       2.7.1 Test Driven Development ........................................................................... 21
       2.7.2 Single-Responsibility Principle ................................................................ 22

3 SYSTEM IMPLEMENTATION ....................................................................................... 23
   3.1 SYSTEM DESCRIPTION ....................................................................................... 23
   3.2 SYSTEM DESIGN AND ARCHITECTURE ............................................................. 23
       3.2.1 Databases and Entity Framework ................................................................ 25
       3.2.2 CRUD operations and Database connector classes .................................... 27
       3.2.3 Controllers and Contracts ......................................................................... 32
       3.2.4 RESTful Web Service ................................................................................ 35
       3.2.5 Client API ................................................................................................... 39

4 SYSTEM EVALUATION ................................................................................................. 44
   4.1 BENEFITS AND DRAWBACKS WITH REST AND SOA ....................................... 44
       4.1.1 Drawbacks ................................................................................................... 44
       4.1.2 Benefits ....................................................................................................... 44
   4.2 ABSTRACTIONS OF LAYERS .............................................................................. 45
   4.3 TEST SUITE AND TEST DRIVEN DEVELOPMENT ............................................. 45

5 DISCUSSION AND CONCLUSIONS .......................................................................... 46
   5.1 .NET FRAMEWORK ............................................................................................. 46
   5.2 OTHER FRAMEWORKS AND PLATFORMS ............................................................ 46
   5.3 CONCLUSIONS AND FUTURE WORK .................................................................. 47

6 BIBLIOGRAPHY ............................................................................................................ 48

7 APPENDICES ............................................................................................................... 49
   7.1 FILE STRUCTURE OF THE PROJECT IN VISUAL STUDIO 2010 .......................... 49
1 Introduction

1.1 Background
As technology advances we get new ways to use and interact with software. In recent years there has been a massive change in how we consume software on a daily basis with the introduction of the concept cloud computing (also know as “the cloud”), tablets and smart phones. Tablets and smart phones have relatively low performance and storage capabilities compared to a regular desktop PC so lots of the computation and storage capabilities have moved onto the cloud and remote servers.

With the introduction of cloud computing vast improvements have been done to Service-oriented Architectures (SOA) and Web Services. When mentioning Web Services a lot of people instantly think of webpages but there is much more to it than that, for instance the Spotify Desktop Client and Diablo 3 are good examples of “regular” desktop applications that are using these technologies. Spotify lets you stream music on many different platforms, everything from your PC to your Hi-Fi system, which all connect to the same backend infrastructure.¹

So the question is how can we implement an application that is distributed over a network and embrace technologies used by applications like Spotify in a big corporate infrastructure to allow increased maintainability and user experience?

1.2 Purpose
The purpose of this thesis was to look into how to redesign the current systems for generating the fault isolation logic and recording for airplanes. The fault isolation logic is represented as flowcharts in the application and the recording tool uses these charts to monitor the software and hardware in the airplanes for errors. The thesis also looks into how to increase the collaboration between these two tools (today they are separated) and how this is done.

1.3 Goals
The primary goal was to design a system that is easy to maintain over time and increase the collaboration between existing systems and databases. In the avionics field systems seem to have a long lifespan, as it is a tedious work to get them approved for integration with aircraft that is in use.

1.4 Scope
The intent of the thesis is to give a better understanding about how SOA and Web services work in general and how these architectural concepts can improve

¹ Kreitz, Gunnar. Spotify — Behind the Scenes. 28 April 2011.
maintainability and collaboration between systems. The intent was never to implement a fully working system, just to implement a “proof-of-concept” system that will show the pros and cons of the selected technologies. The theory part of the thesis is focused on the REST architectural concepts, to give the reader a better understanding of these concepts and how these are implemented later on.

1.5 Abbreviations
Following abbreviations and nomenclature will be used throughout the thesis.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA</td>
<td>Service-oriented Architecture</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transport Protocol</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>WCF</td>
<td>Windows Communication Foundation</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, Request, Update, Delete</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>JSON</td>
<td>Java Script Object Notation</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
2 Technologies

This chapter will cover the key features in the technology and concepts used to give the reader a basic understanding before discussing the implementation of the system. The chapter will also cover some basics on technologies that were considered for the implementation of the tool so it will be easier to see the pros and cons of each technology.

2.1 Hypertext Transport Protocol

The Hypertext Transport Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems and is the foundation of data communication of the World Wide Web. The Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) coordinated the development of the HTTP standard and this resulted in series of publications of Request for Comments (RFCs). Most notably is the RFC 2616 - which defines the HTTP/1.1 version that is commonly used today. The main difference between HTTP/1.0 and HTTP/1.1 is that HTTP/1.1 can reuse a connection multiple times to download content instead of a separate connection for each resource request.

In a client-server model, HTTP functions as a request-response protocol. For example take a web browser – the user types in the URL of the webpage they want to visit, i.e. http://www.example.org. The browser then creates a HTTP GET Request and sends it to the webservice (there is also a DNS lookup where the browser finds the IP address of the webserver). A simplified example of a GET Request can be seen in Figure 1.

![Figure 1: A simple HTTP Get Request](image)

The server receives the request and sees that the user requests the webpage and sends back a response with the requested data. How this response header looks is illustrated in Figure 2.

HTTP has a number of predefined request methods, also referred to as HTTP verbs or just verbs, to indicate the desired action to be performed on the identified resource. In our example we use the verb “GET” and our resource is a webpage. The following verbs are the most relevant ones for this thesis:

- **GET** – Requests using GET only retrieve data from the specified source.
- **POST** – Requests the server to accept the data that the client is sending.
- **PUT** – Requests the server to store the enclosed entity, if the entity already exists the current one is modified or if the entity do not exists the sever creates a new entity with the enclosed entity.
• **DELETE** – This request deletes the specified resource.

```
HTTP/1.1 200 OK
Date: Fri, 19 April 2013 13:37:42 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
Last-Modified: Wed, 17 April 2013 13:37:42 GMT
Cache-control: private, max-age=0
Content-type: text/html; Charset=UTF-8
Content-Length: 79

<html>
<body>
<h1>Example</h1>
<p>This is an example</p>
</body>
</html>
```

**Figure 2: HTTP Response with Example homepage**

### 2.2 Uniform Resource locator (URL)

A uniform resource locator (URL) is a specific character string that represents a reference to a resource and is more commonly knows as a web address. URL is often used as a synonym for URI (uniform resource identifier). URL refers to a subset of URIs that in addition to identifying the resource also describing its primary access mechanism (e.g. its network “location”).

```
scheme://domain:port/path?query_string#fragment_id
```

**Figure 3: URL syntax**

The syntax of an URL can be seen in Figure 3. The *scheme* name (or protocol) defines the namespace, purpose and the syntax of the remaining part of the URL. For example a web browser usually dereference the URL http://example.org:80 by performing an HTTP request to the host at example.org using port number 80 while the URL mailto:bob@example.org may start and e-mail composer with the address bob@example.org “To field”. Other examples of scheme names are https or ftp. The *domain* name or IP address gives the location for the URL; the domain section of a URL is not case sensitive as DNS ignores cases (ease the lookup between URLs and IP addresses). The *port* number is optional, if it is left out the default port number for the scheme in question is used, i.e. port 80 for http: and for https: the port number is 443. The *path* is used to specify the resource requested and it is case-sensitive. The *query string* contains additional data to be passed to the software running on the server and the

---

fragment identifier, if present, specifies a part or position within the overall resource or document.

2.3 Representational State Transfer
REST is an architectural concept drafted by Roy Fielding in his dissertation *Architectural Styles and the Design of Network-based Software Architectures*. The REST architecture is a hybrid style combined of several network-based architectural designs, i.e. Client-Server, Stateless and Cache, with additional constraints that define a uniform connector interface.

2.3.1 Constraints
The REST architecture is built upon the following six constraints; Client-Server, Stateless, Cache, Uniform Interface, Layered System and Code-On-Demand. These constraints help improving visibility, reliability, scalability and efficiency.

2.3.1.1 Client-Server
This constraint separates the clients from servers. By separating the user interface concerns from data storage concerns the system get an improved portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

2.3.1.2 Stateless
The client-server constraint added with a Stateless constraint. This means that every request from the clients must contain all necessary information needed by the server to understand the request, and cannot take any advantage of stored contexts about the clients on the server. This constraint includes the properties of visibility, monitoring systems does only need to look at a single request to understand the full nature of the request. Reliability is improved because it eases the task of recovering from partial failures. Scalability is improved as we do not need to store any data about the clients and can free up resources easily and also simplify the implementation as we dont need to manage resource usage across requests.

2.3.1.3 Cache
To improve network performance we add the cache constraint that requires that the data is either labeled as cacheable or non-cacheable. If a response is cacheable the client can store that response in their local cache to reuse it for equivalent requests. The main

---

4 Uniform resource locator. April 24, 2013.


advantage of the cache is that it will remove some of the interactions between the client and server and further improve efficiency and scalability.\(^8\)

### 2.3.1.4 Uniform Interface

This is a central feature of the REST architectural style and distinguishes it from other network-based architectures. In order to get a uniform interface, multiple architectural constrains are needed to guide the behaviour of components, REST is defined by the following four constraints: identification of resources, manipulation of resources through representation, self-descriptive messages and hypermedia as the engine of application state. Further details about these constraints can be read in section 2.3.2.\(^9\)

### 2.3.1.5 Layered System

To further improve scalability we add layered system constraints. This constraint allows the architecture to be composed of hierarchical layers that do not allow components to “see” beyond the immediate layer with which they are interacting. These layers can be used for scalability trough load balancing servers and may also enforce security policies.\(^10\)

### 2.3.1.6 Code-On-Demand

This constraint allows clients to be extended by downloading and executing code in form of applets or scripts. This simplifies clients by reducing the number of features that need to be pre-implemented and also improves the system extensibility. But it also reduces visibility and is therefore an optional constraint to REST.\(^11\)

### 2.3.2 REST Elements

As REST is an abstraction of the architectural elements within a distributed hypermedia system it ignores the details about component implementations and protocol syntax. Instead the focus is on the roles of components, interaction between them and the interpretation of significant data elements. REST encompasses the fundamental constraints over components, connectors and data that define the very basis of Web architecture and therefore the essence of its behaviour as a network-based application.\(^12\)

---


\(^12\) Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architecture*. Page 86
2.3.2.1 Data Elements

REST uses a number of so-called Data elements; see Table 1 for a summary of them. As a distributed hypermedia system only have three processing paradigms and REST provides a hybrid of the following three options:

- Render the data where it is located and send a fixed-format image to the recipient.
- Encapsulate the data with a rendering engine and send both to the recipient.
- Send the raw data to the recipient along with metadata that describes the data type.

These options have their pro and cons; Option 1 – the traditional client-server style, allows all information about the true nature of the data to remain hidden within the sender. This prevents assumption about how the data structure is implemented and makes the client implementation easier, but it also restricts the functionality of the client and locates most of the processing on the server, which can lead to scalability problems.

<table>
<thead>
<tr>
<th>Data Element</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>The intended conceptual target of a hypertext reference</td>
</tr>
<tr>
<td>Resource identifier</td>
<td>URL, URI</td>
</tr>
<tr>
<td>Representation</td>
<td>HTML document, XML, JSON, JPEG image</td>
</tr>
<tr>
<td>Representation metadata</td>
<td>Media type, last-modified time</td>
</tr>
<tr>
<td>Resource metadata</td>
<td>Source link, alternates, vary</td>
</tr>
<tr>
<td>Control data</td>
<td>If-modified-since, cache-control</td>
</tr>
</tbody>
</table>

**Table 1: REST Data Elements**

Option 2 also provides information hiding while enabling specialized processing of the data via the provided rendering engine, but limits the functionality of the client to what is anticipated within the engine and may vastly increase the amount of data transferred.

Option 3 let the sender to be simple and scalable while minimizing the bytes transferred, but looses the advantages of information hiding and requires that both the server and client uses the same data types.

As mentioned before, REST is a hybrid of all the options by focusing on a shared understanding of data types, but limiting the scope of what is revealed to a standardized interface.\(^{13}\)

Components communicate by transferring a representation of a resource in a format matching a standard data type, which is selected dynamically based on the capabilities or desires of the clients and the nature of the resource. Whether the representation is the same format as the raw data or derived from the source, it still remains hidden behind the interface. By using this mobile object style, by sending a representation that consists of

---

\(^{13}\) Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architecture*. Page 87
instructions in the standard data format of an encapsulated rendering engine, we gain a separation of concerns of the client-server style without the server scalability problem, allows information hiding through a generic interface to allow encapsulation and evolution of services and slot provides a diverse set of functionality through downloadable feature-engines.\textsuperscript{14}

\subsection{2.3.2.1.1 Resources, Resource Identifiers and Representations}

The main abstraction of information in REST is a resource; any kind of information that can be named can be a resource, i.e. a document, a temporal service, a collection of other resource, a non-virtual object (e.g. a person), etc. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.\textsuperscript{15}

Some resources are static in the sense when they are examined at any time after their creation they always corresponds to the same value set as when created while others have a high degree of variance in the value over time. The only thing that has to be static for a resource is the semantics of the mapping, since it is what distinguishes one resource from another.\textsuperscript{16}

This abstract definition of a resource enables some key features in Web Architecture:\textsuperscript{17}

- It provides generality by compressing many sources of information without artificially distinguish them by type or implementation.
- Allows late binding of the reference to a representation, enabling content negotiation to take place based on the request.
- Allows the author to reference the concept rather than some singular representation of that concept, thus removing the need to change all the existing links whenever the representation changes.

REST uses resource identifiers to identify the resource in question between components interaction, the REST connectors provide a generic interface for accessing and manipulating the value of the resource regardless of the membership function is defined for the type or the type of software handling the request. REST relies that the author defining the REST system is choosing a resource identifier that best fits the nature of the concept being identified.\textsuperscript{18}

\begin{flushright}
\textsuperscript{14} Fielding, Roy Thomas. \textit{Architectural Styles and the Design of Network-based Software Architecture}. Page 87, 88  
\textsuperscript{15} Fielding, Roy Thomas. \textit{Architectural Styles and the Design of Network-based Software Architecture}. Page 88  
\textsuperscript{16} Fielding, Roy Thomas. \textit{Architectural Styles and the Design of Network-based Software Architecture}. Page 89  
\textsuperscript{17} Fielding, Roy Thomas. \textit{Architectural Styles and the Design of Network-based Software Architecture}. Page 89  
\textsuperscript{18} Fielding, Roy Thomas. \textit{Architectural Styles and the Design of Network-based Software Architecture}. Page 90
\end{flushright}
REST components perform actions on a resource using a *representation* to capture the current or intended state of the resource and sending that representation between components. A representation is a sequence of bytes and the representation metadata to describe those bytes. In a more commonly and less precise names for representation is for example: document, file, and HTTP message entity, instance or variant. Control data defines the purpose of a message between the components such as the action requested or the intended meaning of a response. It is also used to parameterize requests and override the default behaviour of some connecting elements, i.e. cache behaviour, should we store this representation in a cache or not.

### 2.3.2.2 Connectors and Components

REST uses many different connector types to encapsulate the activities of accessing a resource and transferring its representation, these types of connectors are summarized in Table 2 with some examples. The connectors present an abstract interface to enable communication between components and improving simplicity by providing a clean separation of concerns and hiding the underlying implementation of resources and communication. The generality of the interface also enables substitution of the interface, if the users’ only access the system through an abstract interface the current implementation can be replaced without any impact to the users. A connector also manages the network communication for a component; information can be shared across multiple interactions in order to improve efficiency and responsiveness of the system.

<table>
<thead>
<tr>
<th>Connector</th>
<th>Web Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Libwww, libwww-perl</td>
</tr>
<tr>
<td>Server</td>
<td>Libwww, Apache API</td>
</tr>
<tr>
<td>Cache</td>
<td>Browser, Akamai cache network</td>
</tr>
<tr>
<td>Resolver</td>
<td>Bind (DNS lookup library)</td>
</tr>
<tr>
<td>Tunnel</td>
<td>SOCKS, SSL after HTTP CONNECT</td>
</tr>
</tbody>
</table>

**Table 2: REST Connectors**

As mentioned earlier, all REST interactions are stateless and this specific restriction accomplish four limitations of the system:

- Removes any need to store application state between requests
- Allows interactions to be processed in parallel without the processing mechanism to know the interaction schematics.
- Allows an intermediary to view and understand the request in isolation.

---

• Forces all of the information that might factor into reuse of a cached response to be present in each request.

The primary connector types are client and server. The main difference between them is that the client initiates the communication with a request, while the server listens for incoming connections and responds to request in order to supply access to its services. A component can include both client and server connectors.22

A cache connector can be located on the interface of a client or server connector in order to save cacheable responses for future reuse. A cache can be used by a client to avoid repetition of network communication and on a server to avoid repeating the process of generating a response. Some cache connectors can be shared which means that cached responses can be used for other requests then it was originally intended to. Shared cache can be effective at reducing the impact of “flash crowds” on the load of a popular server, especially when the caching is arranged hierarchically to over a large group of users, i.e. users within a company’s network.23

REST attempts to balance the transparency of cache behaviour for efficient use of the network rather than assuming that absolute transparency is always needed. A cache is able to determine if a response is cacheable because of the generic interface, if some form of user authentication is part of the request or if the response indicated it should not be shared then the response is only cacheable by a non-shared cache, i.e. in your browser. The cacheable option can be overridden by including control data that states if the response can be cached, non-cacheable or cached for a limited time.24

A resolver translates partial or complete resource identifier into the network address information needed to establish inter-component connection. Most URI include a DNS hostname as the mechanism for identifying the naming authority for the resource. For example when a Web browser want to go to a Web page it will extract the hostname from the URI and make use of a DNS resolver to get the IP address for that authority.25

The last form of connector type is a tunnel, which relays communication across a connection boundary such as a firewall or a lower-level network gateway. The only reason it is modelled as a part of REST and not abstracted away as part of the network infrastructure is that some REST components may dynamically switch from active component behaviour to that tunnel, an example is when a HTTP proxy that switches to a tunnel in response to a CONNECT method request, which allows the client to directly

communicate to the remote server using an different protocol like TLS that don’t allow proxies.

<table>
<thead>
<tr>
<th>Component</th>
<th>Web Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin server</td>
<td>Apache httpd, Microsoft IIS</td>
</tr>
<tr>
<td>Gateway</td>
<td>Squid, CGI, Reverse Proxy</td>
</tr>
<tr>
<td>Proxy</td>
<td>CERN Proxy</td>
</tr>
<tr>
<td>User agent</td>
<td>Chrome, Firefox</td>
</tr>
</tbody>
</table>

Table 3: REST Components

A REST component is an application that is consuming the REST style; these can be divided into a number of categories, a summarized in Table 3 with some examples of applications.

An origin server uses a server connector to manage the namespace for a requested resource; it is the definitive source of representations of the resource and must be the ultimate recipient of any request that intends to modify the value of the resource in question. The resource implementation is hidden behind the interface.

A gateway (a.k.a. reverse proxy) component is an intermediary imposed by the network or origin server to provide an interface encapsulation of other services, data translations, performance improvements or security enforcements. A proxy is also an intermediary selected by a client to provide the same properties as a gateway but the difference is that the client effectively chooses to use a proxy while a gateway is forced.

A user-agent uses a client connector to initiate requests and becomes the ultimate recipient of the response. A Web browser is a perfect example that provides information services and renders service response according to the application needs.

2.4 RESTful Web Service

As REST is only conceptual architectural style and it can be applied on many different software concepts. A RESTful Web Service is implemented using HTTP and the concepts from REST – this give us a versatile, accessible and scalable service. Let us implement a small blog system to give a simple example on how we can utilize REST, HTTP and URLs together.

---

To begin with we define a base URI for our web API; http://myblogsite.org/post/{id}, so the scheme in use is HTTP, the domain is myblogsite.org and the resource is post and {id} is the unique id of a blog post that is stored in a database somewhere hidden away in the backend. When we want to retrieve a collection of blog posts we can use the following URL: http://myblogsite.org/post/ and the server will receive an HTTP Get Request. The server will then send a collection of blog posts to the client; in Figure 4 there is an example of a response of blog posts using JSON (JavaScript Object Notation).

<table>
<thead>
<tr>
<th>Method</th>
<th>GET (Retrieve)</th>
<th>PUT (Update)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Request header</td>
<td>GET /post/1 HTTP/1.1 Host: myblogsite.org Accept: application/json</td>
<td>PUT /post/3 HTTP/1.1 Host: myblogsite.org Content-Type: application/json Content-Length: 47 {id: 1, title: New Title, Content: New content}</td>
</tr>
<tr>
<td>Description</td>
<td>Fetches the post with id 1 from the server</td>
<td>Updates the post with id 1 with the new information</td>
</tr>
</tbody>
</table>

In Figure 5 there is an example on the CRUD (Create, Retrieve, Update, Delete) operations on the post resource and how they map to their corresponding HTTP verb. This let the web API to be quite general; if we implement a User resource (a resource to handle users of the blog) we would use the same methods to interact with the backend. This is one of the key features in REST and not to reinvent the wheel every time a new resource is added. But what will happen if we try to access a resource that we don’t have? Then the server will respond the client with the HTTP status code 404: Not found or 500: Internal Server Error depending on the implementation.

<table>
<thead>
<tr>
<th>Method</th>
<th>POST (Create)</th>
<th>DELETE (Delete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Request Header</td>
<td>POST /post/ HTTP/1.1 Host: myblogsite.org Content-Type: application/json Content-Length: 47 {id: 0, title: New Title, Content: New content}</td>
<td>DELETE /post/3 HTTP/1.1</td>
</tr>
<tr>
<td>Description</td>
<td>Creates a new post entity</td>
<td>Will delete the post with id 3</td>
</tr>
</tbody>
</table>

Figure 4: Response example of blog posts

Figure 5: Client HTTP Requests
2.5 SOAP

SOAP was designed in 1988 for Microsoft as an object-access protocol. SOAP was originally defined as Simple Object Access Protocol but the acronym was dropped in version 1.2 of the standard and the standard is maintained by XML Protocol Working Group of the World Wide Web Consortium. SOAP has three characteristics – Extensibility, Neutrality and Independence.

As SOAP is basically a stateless, one-way exchange paradigm between SOAP nodes but applications can create more complex interaction patterns (i.e. request/response) by combining such one-way messages with features from underlying protocols (HTTP as a request/response implementation). SOAP is also silent on the semantics of any application-specific data it conveys – this means when the SOAP messages is traversing through the network the data in the SOAP body won’t change during the transmission, instead we will use the SOAP header to add extra information that might be needed for SOAP intermediaries.\(^{30}\)

![SOAP Structure](image)

A message consists of three parts, an envelope, a header and a body that resides within the envelope. The header element is optional and is an extension mechanism that provides a way to pass information in a message that is not application payload. The SOAP header have been designed in anticipation of various uses, many of which will involve the participation of other SOAP nodes along a message’s path from an initial SOAP sender to an ultimate SOAP receiver. These nodes are also known as SOAP intermediaries which will provide value-added services, i.e. security. The SOAP body is the mandatory part within an envelope and is where the main end-to-end information carried in a SOAP message must be put. The content in these elements is based on the application, but let us see how the blog example from the RESTful chapter will be translated into an SOAP powered application.\(^{31}\)


In Figure 7 there is an example of an SOAP message where we call the procedure on the server to fetch all the posts of the blog. This procedure creates the response that can be seen in Figure 8 and we can see that the XML syntax that SOAP utilize is very verbose which will increase the payload of each message. In these two examples the payload is not very much but scale it up a bit, i.e. the response returns 100 posts and this will of course increase the load on the network of the server and as well on clients network. If we compare the content length in the HTTP header of the REST Response, encoded in JSON, and the SOAP Response, which is encoded with XML, there is a quite a big difference, 113 bytes versus 551 bytes. Even if we would have sent our REST Response with XML the REST Response would been smaller as we don’t need the SOAP specific XML tags.

<table>
<thead>
<tr>
<th>POST /post HTTP/1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host: <a href="http://www.myblogsite.org">www.myblogsite.org</a></td>
</tr>
<tr>
<td>Content-Type: application/xml+soap; charset=8 utf</td>
</tr>
<tr>
<td>Content-Length: 242</td>
</tr>
<tr>
<td>SOAPAction: “<a href="http://www.w3c.org/2003/05/soap-envelope%E2%80%9D">http://www.w3c.org/2003/05/soap-envelope”</a></td>
</tr>
</tbody>
</table>

```xml
<?xml version="1.0">
<env:Envelope xmlns:soap="http://www.w3c.org/2003/05/soap-envelope">
  <soap:Header>
    </soap:Header>
  <env:Body>
    <m:GetPosts xmlns:m="http://www.myblogsite.org/post">
      </m:GetPosts>
    </env:Body>
</env:Envelope>
```

**Figure 7: Example message**

The SOAP protocol give us even more overhead in the Request messages we send, as we can see in Figure 7 the content length is 242 bytes as in a RESTful architecture we will only send a HTTP GET Request with the correct resource specified. But with the RESTful architecture we are more or less bound to the CRUD operations while with SOAP we can implement more ”standard” methods we could see in a ”regular” software like GetSubscriberCount, which will calculate how many subscribers a blog has. This is of course feasible in a REST architecture aswell but we need to take an approach by utilizing query strings within the url.

---

32 The content length field in a HTTP header is calculated by summing up the byte size of the JSON/XML message that we send. In this case one character equals one byte as we are transmitting an UTF-8 encoded message.
A SOAP web service is often (if not always) connected to a WSDL document (Web Service Description Language) which is an standarized way to describe your web service in the form of an XML document. The document defines a service as a collection of network endpoints or ports and a WSDL document uses the following items to define a network service:

- **Types** – a container for data type definitions using some type system
- **Message** – an abstract, typed definition of the data being communicated.
- **Operation** – an abstract description of an action supported by the service.
- **Port type** – an abstract set of operations supported by one or more endpoints.
- **Binding** – a concrete protocol and data format specification of the particular port type.
- **Service** – a collection of related endpoints.

This document will let anyone download your definition of your web service interface and can then easily, in some cases automated, consume your service in their applications.

---


http://www.w3.org/TR/wsdl (accessed May 3, 2013)
2.6 Platform specific technologies

The platform the system will be built upon is .NET Framework Version 4.0 and C#. The .NET platform has a big standard library and a big community that develop many new features and plugins for it.

The tool will be developed in Visual Studio 2010 IDE and SQL Server 2008.

2.6.1 Windows Communication Foundation (WCF)

WCF is a framework for building service-oriented applications in the .NET platform. You can send data as asynchronous messages from one service endpoint to another. A service endpoint can be a part of a service hosted by IIS\(^{34}\) or hosted in an application. An endpoint can be a client of a service that requests data from a service endpoint and the message can be as simple as a single character or word, XML or as complex as a stream of binary data.\(^{35}\)

Here are some of the features of WCF that is most interesting for us:

- **Service Orientation** – WCF enables you to create service-oriented applications. The service has general advantage of being loosely coupled instead of hard-coded from one application to another. This means that any client on any platform can connect to any service as long as the essential contracts are met.\(^{36}\)

- **Data Contracts** – A data contract is a formal agreement between a service and a client that abstractly describes the data to be exchanged. WCF includes a comprehensive system for working with data in this manner. Once you have created the classes that represent data, the service automatically generates the metadata that allows clients to comply with the data types that has been designed.\(^{37}\)

- **AJAX and REST Support** – WCF can be configured to process “plain” XML data that is not wrapped in a SOAP Envelope. It can also be extended to support specific XML format, i.e. ATOM (a RSS standard), and non-XML format such as JSON.\(^{38}\)

To kick-start the development process there is an inbuilt project wizard in the Visual Studio 2010 to set up a WCF Service Application, this wizard generates and configures all the files we need to run an empty “Hello World” version of a WCF Service Application.

---

\(^{34}\) Internet Information Services (ISS) is a web server developed by Microsoft.


2.6.2 Entity Framework 4.0

The Entity Framework is a set of technologies in ADO.NET that supports the development of data-oriented software applications. The framework enables developers to work with data in the form of domain-specific objects and properties, such as customers and customer addresses, without having to concern themselves with the underlying database tables and columns where this data is stored. In other words, we do not need to write any database-specific code to access the data in the databases – the framework will handle all that for us and automatically generate corresponding classes for each table.³⁹

Visual Studio 2010 and .NET 4.0 ships with Entity Framework 4.0 and any system with .NET Framework 3.5 SP1 can run an Entity Framework application. Visual Studio 2010 includes rich tool support for generating and maintaining model and mapping files in a Visual Studio application. The Entity Data Model Designer supports creating advanced mapping scenarios such as table-per-type, one class that you create in the designer will correspond to a table in the underlying database and the relation to other classes/tables will also be translated to the database. The framework also lets you generate an Entity Data Model from an existing database as well.⁴⁰

Using the Entity Framework in this project will give us an extra abstraction layer above the databases and we don’t need to write any specific SQL code for the databases that means the system can easily be ported to another SQL database, as we just need to change the database connector for the Entity Framework.

2.7 Development Processes and Principles

2.7.1 Test Driven Development

Test Driven Development (TDD) is a software development process, and just as much as a mind-set of the developer, that relies on the repetition of a development cycle:

1. Add a test.
2. Run all tests and see if the new one fails.
3. Write some code.
4. Run the automated tests and see them succeed.
5. Refactor the code.
6. Repeat.

In TDD, the developer starts with writing a test for each new feature. This test must fail because the test is written before the developer has implemented the feature. To write a test the developer must clearly understand the feature’s specification and requirements.

When the test is correctly written it is run for the first time and it is bound to fail as the developer have not written any code to implement the feature. This is often called in TDD terminology to “Go from Red to Green” as many IDE and tools for testing has that colour mapping of red as a failed test and green as a passed test.

Then the developer writes some code to implement the feature – rerun the test to see if the test pass or any other previously implemented test and feature passes or fails. If anything fails rewrite the code and rerun the tests.

When the tests succeed the developer tries to refactor the code by trying to find duplicated code and move this to a function that can be called instead of rewriting the same segment of code.

This workflow will be used throughout the whole implementation of the FITool-Backend and ClientAPI to ensure a reliable code base and will ease the process of refactoring the code, as you will get instantaneous response if the new feature or refactoring breaks your system. This will also increase the maintainability of the system as it will be easier to implement a new feature to the system – the new developer write a test, implement the new feature, rerun the test suite for the whole project and see if it breaks the current implementation. It also gives new developers an excellent example on how your code works. Also if a bug is found in the system the developers can easily write a test to replicate the bug and refactor the effected code.

One downside with TDD is it that it slows down the development speed a bit as you do your testing while developing the system.

### 2.7.2 Single-Responsibility Principle

This principle implies that “A class should have only one reason to change”. It is important to separate the responsibilities of classes, as each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility among the classes and if a class has more than one responsibility the class will have more than one reason to change.\(^\text{41}\)

Defining classes after this principle also ease the development process of TDD as it will be much more simpler to test the class as it only has one responsibility to achieve and for that we can write a test that verifies it.

Combined with Test Driven Development the Single-Responsibility principle will generate well-authored and robust code for the project. The classes will be easier to understand for new developers as they only have one responsibility and will therefore increase the maintainability of the system.


22
3 System Implementation

This chapter will cover the overall system design, in detail implementation of the specific classes and modules of the "proof-of-concept" prototype of the system.

3.1 System description

The tool will consist of two parts, the FITool-Backend and FITool-ClientAPI. The FITool-Backend will be implemented as a RESTful Web Service to enable access and collaboration between the underlying databases. There is one database for the Recording variables and one for the fault isolation logic.

The FITool-ClientAPI will implement communication and an interface to the server so the frontend developer (or any other developer) doesn’t need to implement these by themselves. This will speed up the development of their applications.

3.2 System Design and Architecture

The architecture of the tool is quite straightforward, seen in Figure 9. It is a classical Client-Server architecture and the backend is an adaptation of the Model-View-Controller (MVC) pattern that is common in web applications, but has the view segment scaled down, as the backend do not return any graphical content.
The Model, in this case the Entity Framework Model classes, encapsulates the databases and generates EntityModel objects of the tables in the databases. Each table has an own connector class, these are called “Table” Connectors in Figure 9. There is an class for each table in the EntityModel and these are then wrapped in the façades FIToolDatabaseConnector and RecordingDatabaseConnector to ease the access to each connector class. The controller classes has instances of the DatabaseConnector classes and converts the EntityModel objects to DataContracts object so they can be converted automatically by WCF to JSON or XML and transported easily over HTTP to the client. In Appendix 7.1 there is a layout of the file and class structure in Visual Studio 2010.
The client then receives the data on JSON or XML and then converts this back to a DataContract object ready to be used in the client code.

In the following chapters there will be a detailed description of each layer in the system and how these where implemented, the order of the chapters is how the system was implemented but minor parts were done more or less in parallel, i.e. the design of the database and the resource identification for the REST URLs but for readability these will be described separately.

The example used in the flowing chapters will be how we implement the Schema entity, from the database to the client API.

3.2.1 Databases and Entity Framework

As mentioned in the beginning of this thesis we model the fault isolation logic as flowcharts, how this looks in the Entity Model Designer in Visual Studio 2010 can be seen in Figure 10. The following models are exported to SQL code for the MS SQL databases that is used in the development and testing environments. The testing database is used for the unit testing of the CRUD operations and is purged between each test so the test has its own environment to execute in, this is important so we do not get any false errors or false passes.

![Figure 10: Entity Model Design of the FITool database](image)

3.2.1.1 FITool Entities

A Schema consists of the following members and relation:

- An id, a 32-bit unique integer with auto increment option enabled.
- A name, a name to describe what the schema describes.
- A date, the date when the schema was created or last updated.
- The relation 1-to-Many to the entity Node, a schema consists of many nodes.
A Node consists of the following members and relations:
- An id, 32-bit unique integer with auto increment option enabled.
- A schema id, acts as a foreign key in our database application – let us know which schema this specific nodes relates to.
- A TrueLinkId, the id of the Link entity that evaluates true, can be null value.
- A FalseLinkId, the id of the Link entity that evaluates false, can be null value.
- The relation 1-to-Many to the entity Expression, a node consists of many expressions.
- The relation 1-to-Many to the entity Link, a node consists of many links.

A Link consists of the following members and relations:
- An id, 32-bit unique integer with auto increment option enabled.
- A node id, acts as a foreign key in our database application – let us know which node the link links to.
- Have the relations TrueLink and FalseLink that is a Zero or One relation which means that a link can be either set or not, i.e. a Start node in the flowcharts will only have one outgoing link which will be a TrueLink and the FalseLink will be set to null in the Node entity.

An Expression consists of the following members and relations:
- An id, 32-bit unique integer with auto increment option enabled.
- A node id, acts as a foreign key in our database application – let us know which node this expression belongs to.
- An ExpressionString, a string that contains the logic expression to be evaluated.
- The relation 1-to-Many to the entity Variable, an expression is built up by variables.

An Variable consist of the following members and relations:
- An id, a 32-bit unique integer with auto increment option enabled.
- An expression id, acts as a foreign key in out database application – let us know which expression the variable belongs to.
- A name, the shorter version of the name of the variable.
- A full name, the long version of the name of the variable.
- Recording variable id, the id to the variable definition in the Recording database.

3.2.1.2 Recording Entities
The recording database is much more vast in reality and for the proof-of-concept prototype we do not need the whole database so we just use two tables from it, one for the Recording variable and one for the Type of the variables.
Figure 11: Entity Model Design of the Recording database

An Variable consist of the following members and relations:
• An id, a 32-bit unique integer with auto increment option enabled.
• A name.
• A parameter name.
• A type id, which type belongs to this variable.

A Type consists of the following members and relations:
• An id, a 32-bit unique integer with auto increment option enabled.
• A name.
• A Base type. I.e. Integer or Boolean.
• A Length. The bit length of the variable.

The Entity Framework will automatically generate classes for us to use in the application. These classes has the same name as the tables in question and a collection of objects are called a set, i.e. a SchemaSet which will be an iterable collection of all Schema objects retrieved from the underlying database.

3.2.2 CRUD operations and Database connector classes

Now that we have a database implementation and a working Entity Model acting as a Object-Relation-Manager to the database, which implies that we do not need to write a single line of SQL code in our application but instead use LINQ (Language-Integrated Query) which enables to use query capabilities to the language syntax of C#. LINQ is quite similar to SQL but much easier to learn and understand. The connector classes will implement entity specific CRUD (Create, Retrieve, Update, Delete) operations.

3.2.2.1 Connector interface

To begin with we implement a generic interface for all the connector classes that includes the definition of the essential CRUD operations for the connector classes. The interface also utilizes the concepts of generics, which means it can be applied on any object (this

will be clearer later on), the keyword T in the Code snippet 1 can be seen as a generic Type called T, in this case these classes will be all the Entity classes that has been generated. If we had to implement this as a specific only interface we would write either Schema or even Integer but the generic syntax let us reuse the same code over and over again, which is good programming standard.

```csharp
namespace FIToolBackend.DatabaseConnectors
{
    interface IConnector<T>
    {
        void Create(T t);
        T Get(int id);
        List<T> Get();
        void Update(T t);
        void Delete(int id);
        int CurrentId { get; }
    }
}
```

**Code snippet 1: Connector interface class**

The Create function will create a new entity of type T in our database. The two Get functions will return a specific entity with the specified id, while the other will return all entities of type T as a List object. The Update function will update the specified entity T and Delete will remove the entity with the specified id in the database. The CurrentId property is only used for testing and checking purposes.

**3.2.2.2 Example implementation of the Schema entity**

All Connector classes look more or less the same way, just some minor alteration for entity specific parameters, most notably the “Sets” – i.e. SchemaSet and NodeSet for the Schema entity and Node entity respectively.
Code snippet 2: Tests for CRUD on Schema entity

As we develop with TDD we start by defining a test that we can run. In Code snippet 2 we can see how the test suite for the SchemaConnector class looks like (some parts are left out because of space). These tests are in reality written in an incremental manner, by starting write the test for the Create function we will only have the instance of SchemaConnector class and a test method trying to create a Schema in the database. This
test will fail to begin with as for the moment don’t have an SchemaConnector class yet, but this is intended with Test Driven Development.

```csharp
namespace FIToolBackend.DatabaseConnectors.FITool
{
    public class SchemaConnector : IConnector<Schema>
    {
        public void Create(Schema t)
        {
            using (var db = new FIEntityModelContainer())
            {
                Schema s = new Schema();
                s.Name = t.Name;
                s.datetime = DateTime.Now;
                db.AddToSchemaSet(s);
                db.SaveChanges();
            }
        }
    }
}
```

**Code snippet 3: Create function for a Schema**

Secondly we start by creating the SchemaConnector class and rerun the test we get a new error, as the Create function isn’t implemented. Then we implement the Create function, which can be seen in Code snippet 3. Here can we also see that the class also inherits the IConnector interface with the type T set as Schema.

The implementation is quite straightforward, we take the data from the Schema we have as in-parameter, extract the information we need – the name in this case. Add it into a new instance of a Schema entity and also the current time at the server, add it to the current SchemaSet and then save it to the database. This repetition and flow of work continuous for all the other CRUD operations and their implementations can be seen in Code snippet 4 and their respective tests in Code snippet 2.

This class is implemented with the single responsibility principle in mind which implies that the class only do one thing and one thing only and in this case handle the CRUD operations to the database – so no checking of the in or out data from its functions. This makes the class very easy to test, as we do not need to account for faulty data.
As the implementation has quite a few Connector classes, one for each table – seven of them in all, we need to utilize the Façade design pattern and “hide” these behind an “interface” for easier access and to minimize the coupling of the code base. There are two Façade classes – FIToolDatabaseConnector and RecordingDatabaseConnector, the first one includes the connector classes for the FITools database while the other one includes the classes of the Recording database.
In Code snippet 5 there is an example how the CreateSchema and GetSchema functions are implemented in the FIToolDatabaseConnector class – also with the single-responsibility principle in mind. All the functions in this class take “raw” data, in the CreateSchema case a raw string as input and then create a Schema object; this is a bit redundant as we also create a new Schema class in the SchemaConnector class. But this is a side effect of the generic interface IConnector. If we had any naming conventions of our schemas it would be here we would implement these checks as well, in the GetSchema function we can see an example of error handling when we try to access a non-existing schema.

```csharp
using FIToolBackend.DatabaseConnectors.FITool;

namespace FIToolBackend.DatabaseConnectors
{
    public class FIToolDatabaseConnector
    {
        private SchemaConnector schemaConnector = new SchemaConnector();
        //More connector classes...

        public void CreateSchema(string name)
        {
            schemaConnector.Create(new Schema { Name = name });
        }

        public Schema GetSchema(int id)
        {
            Schema s = schemaConnector.Get(id);
            if (s != null)
                return s;
            else
                throw new SchemaConnectorError("Schema with id " + id + " do not exist");
        }
        //More functions...
    }
}
```

**Code snippet 5: FIToolDatabaseConnector**

### 3.2.3 Controllers and Contracts

With the single responsibility principle we now need a way to translate our entity objects into something that is easily transported over the HTTP protocol and lives up to the REST restriction – a hypermedia. We can implement our own JSON or XML parser to do this or utilize the inbuilt features of WCF and DataContract classes. The classes that implement these translations are called Controllers, FIToolController and RecordingController respectively. The DataContract classes we are to implement is separated from the FIToolBackend namespace and reside in the namespace FIToolBackend.DataContracts that will let the compiler to compile these into their own binaries so we can utilize these DataContract classes in the Client API as well.
The DataContract classes are real simple to implement – just a regular C# class with some extra attributes, the [DataContract] and [DataMember] are the attributes we use. These attributes allow adjusting how the class is serialized, i.e. the list of NodeContracts is an optional property for us in the SchemaContract class while the Name property is not optional.
The Controller classes, FIToolController and RecordingController, handle the conversion between the Entity classes and DataContract classes. The CreateSchema function just passes on the Name property from the SchemaContract class to the Connector class while in the GetSchema function much more things are going on.

In the GetSchema we start by fetching the schema from the Connector class and then we fetch all the nodes that is related to the schema we want to fetch. We then create a list of NodeContracts and convert the Node entities to NodeContract object and add them to the list. Finally we return an SchemaContract object with all the relevant data from the Schema entity and the newly created NodeContract list.
3.2.4 RESTful Web Service

The REST architectural style fits in very well with this application – a Fault Isolation Schema can easily be translated into a hypermedia format such as JSON or XML and as most of the data in the databases are already in textual form and the conversion is easily done. The hardest part when designing a REST Web Service is not to start thinking in the ways of calling a remote procedure on a server and that it is run there. Instead we work with data as a resource, which we can create new entities, retrieve, manipulate and delete them. For instance it is easily to think about implement a function called “Validate” on the server that validates the schema on the server and then returns True/False to the client but this is not the intended use of REST.

3.2.4.1 Identifying resources

Before we start defining and implementing the REST interface we need to identify which resources we have and how we should interact with these. We start by defining the URL schema for our service, displayed in Figure 12.

<table>
<thead>
<tr>
<th>HTTP Method</th>
<th>URI Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/fidb/schema/1</td>
<td>Requests the schema with the id 1.</td>
</tr>
<tr>
<td>GET</td>
<td>/fidb/schema</td>
<td>Requests all the schemas.</td>
</tr>
<tr>
<td>POST</td>
<td>/fidb/schema</td>
<td>Creates a new schema.</td>
</tr>
<tr>
<td>PUT</td>
<td>/fidb/schema</td>
<td>Updates the schema specified in the payload of the HTTP message.</td>
</tr>
<tr>
<td>DELETE</td>
<td>/fidb/schema/1</td>
<td>Deletes the schema with the id 1.</td>
</tr>
</tbody>
</table>

Figure 12: URL schema for the service

The protocol is HTTP, the network path is a dynamic entity depending if we are developing or in production mode – in development we use localhost and the port number that is assigned by Visual Studio. We have two database, FITools database and the Recording database – they are named as fidb and recording in our schema. We have seven different resources in our REST application and they are the Entity models we have defined earlier. The id field is optional and used when we want to get a specific resource.

In Figure 13 is an example of the resource identification of the schema entity, this is repeated for every entity in the application.

3.2.4.2 Defining the interface

When we run the wizard to create our WCF Service Application project there is a number of files generated for us there is three files, which are important to us:

- **FIToolService.svc.cs** – Implementation of the server interface.
- **IFIToolService.cs** – Definition of the server interface.
- **Web.Config** – A configuration file for our service.
In the IFIToolService.cs file we define an interface class, which will be a ServiceContract and include a number of OperationContract functions. These attributes allows us to define the public interface to our service and which data each function should return.

```csharp
namespace FIToolBackend
{
    [ServiceContract]
    public interface IFIToolService
    {
        [OperationContract]
        [WebGet(UriTemplate = "/fidb/schema/{id}",
            ResponseFormat=WebMessageFormat.Json)]
        SchemaContract GetSchema(string id);

        [OperationContract]
        [WebGet(UriTemplate = "/fidb/schema",
            ResponseFormat = WebMessageFormat.Json)]
        List<SchemaContract> GetSchemas();

        [OperationContract]
        [WebInvoke(Method = "POST",
            ResponseFormat = WebMessageFormat.Json,
            RequestFormat = WebMessageFormat.Json,
            UriTemplate = "/fidb/schema")]
        void CreateSchema(SchemaContract schema);

        [OperationContract]
        [WebInvoke(Method = "PUT",
            ResponseFormat = WebMessageFormat.Json,
            RequestFormat = WebMessageFormat.Json,
            UriTemplate = "/fidb/schema")]
        void UpdateSchema(SchemaContract schema);

        [OperationContract]
        [WebInvoke(Method = "DELETE",
            ResponseFormat = WebMessageFormat.Json,
            RequestFormat = WebMessageFormat.Json,
            UriTemplate = "/fidb/schema/{id}"),
        void DeleteSchema(string id);
    }
}
```

**Code snippet 8: Service interface definition for Schema Entity**

The interface also implements the attributes WebGet and WebInvoke on the functions, these attributes defines which HTTP verbs and which Uri Template they should listen to. The two GET functions returns an SchemaContract object but this object is serialized into JSON format that is defined in the ResponseFormat attribute. In the POST and PUT functions we also define a RequestFormat, which tells our interface that we expect to receive data in this format and then deserialize it into a SchemaContract object.
3.2.4.3 Implementing the interface
In FIToolService.svc.cs we implement what the interface should do with the retrieved data we get on the services interface. This class does not really do that much at all, just passing on the data we get on our interfaces to the Controller classes.

```csharp
namespace FIToolBackend
{
    [AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
    public class FIToolService : IFIToolService
    {
        private FIToolController ficon = new FIToolController();

        public void CreateSchema(SchemaContract schema)
        {
            ficon.CreateSchema(schema);
        }

        public SchemaContract GetSchema(string id)
        {
            int Id = Convert.ToInt32(id);
            return ficon.GetSchema(Id);
        }

        public List<SchemaContract> GetSchemas()
        {
            return ficon.GetSchemas();
        }

        public void UpdateSchema(SchemaContract schema)
        {
            ficon.UpdateSchema(schema);
        }

        public void DeleteSchema(string id)
        {
            int Id = Convert.ToInt32(id);
            ficon.DeleteSchema(Id);
        }
    }
}
```

Code snippet 9: Service implementation of the interface IFIToolService

The attribute AspNetCompatiblityRequirements, seen in Code snippet 9, is needed to enable the AJAX features to the service. AJAX is an feature that enables JavaScript to asynchronously retrieve data from the service. Why the in parameter for the id’s is string is because of the hypermedia restriction of the REST style as an Integer is not an valid hypermedia format and has to be converted to a text string.

3.2.4.4 Configuring the interface
To configure the service we have an XML file that the wizard has generated. The configuration won’t work as a RESTful Web Service out of the box and we need to modify some sections of it, in Code snippet 10 is the whole configuration file for the FIToolBackend Service, excluding the paths to the database files.
There is a lot to digest in the configuration file but the most important segments in this file are between line number 12-20, 23-28, 29-35, 37-38 and 43-48. On lines 12 through 20 define our service and a corresponding behavior configuration for the service, the service endpoint is also defined with the default address (localhost), bind it to the WebHttpBinding class which allows the service to be exposed to HTTP requests instead...
of SOAP messages and also set the endpoints behavior configuration to our own defined behavior “web”.

On lines 23 to 28 define the service behaviors and only have one named “ServiceBehavior” with the <serviceMetadata> tag with the attribute “httpGetEnabled” set to true, which allows the service to distribute a WSDL of the service.

Lines 29 to 35 define the endpoints behaviors, the behavior named “web” and uses the <webHttp> tag to specify the WebHttpBehavior on the endpoint, which enables the Web programming model for a WCF service. It has some attributes set; helpEnable enables the Help page of the service, automaticFormatSelectionEnable allows WCF to automatically determine which is the best format for the responses (this is set to false as we want to choose which format is the best), defaultOutgoingResponseFormat specifies the default message format when the application header in the HTTP header isn’t set.

The <connectionString> tag on line 43 to 48 specifies where the EntityFramework can find the databases and map these the correct EntityModelContainer.

### 3.2.5 Client API

The Client API was developed to enable easy access to the FIToolBackend service in frontend applications so the frontend developers will not need to take network communication in account and can use it like any other “database” framework or API.

The implementation is quite simple – the base of the API consists of the functions Response and Send, which can be seen in Code snippet 11. The Response function has a WebRequest as in parameter that is just a container class for a HTTP web request, either a GET, POST, PUT or DELETE request. The Response function reads the response that the service sends to the client and returns this data. For example the response from a GET request would be a JSON encoded string of the requested resource. The Send function works the other way around – the client sends an HTTP request to the service and retrieves an HTTP status code as response. But we don’t do anything if we get an error response from the server, like an internal server error (we try to access a schema that do not exists for example), and an exception will be thrown instead.

The Client API also shares the same DataContract classes as in the FIToolBackend service, and this allows the Client API to easily to serialize and deserialize the JSON strings that is sent over the network to be serialized into DataContract objects again.
Private string Response(WebRequest request)
{
    HttpWebResponse resp = request.GetResponse() as HttpWebResponse;
    if (resp.StatusCode == HttpStatusCode.BadRequest)
    {
        throw new WebException("Bad Request");
    }
    Stream receiveStream = resp.GetResponseStream();
    StreamReader readerStream = new StreamReader(receiveStream, Encoding.UTF8);
    Char[] read = new Char[256];
    int count = readerStream.Read(read, 0, 256);
    String data = ";
    while (count > 0)
    {
        String str = new String(read, 0, count);
        data += str;
        count = readerStream.Read(read, 0, 256);
    }
    readerStream.Close();
    receiveStream.Close();
    resp.Close();
    return data;
}

Private void Send(WebRequest request, string payload)
{
    //Send the Request (either a POST or PUT)
    using (Stream stream = request.GetRequestStream())
    {
        stream.Write(Encoding.UTF8.GetBytes(payload), 0,
        Encoding.UTF8.GetBytes(payload));
        stream.Flush();
    }
    string response = Response(request);
}

Code snippet 11: Basic HTTP communication functions

In Code snippet 12 is there some generic functions to even more ease up the communication to the backend. Why these are implemented as generic because we want to send many different DataContract classes to the backend and do not want to specific functions for each single DataContract.
private T Get<T>(Uri uri)
{
    request = WebRequest.Create(uri);
    string data = Response(request);
    return JSONHelper.Deserialize<T>(data);
}

private void Post<T>(Uri uri, T obj)
{
    string payload = JSONHelper.Serialize<T>(obj);
    request = WebRequest.Create(uri);
    request.Method = "POST";
    request.ContentLength = Encoding.UTF8.GetByteCount(payload);
    request.ContentType = "application/json";
    Send(request, payload);
}

private void Put<T>(Uri uri, T obj)
{
    string payload = JSONHelper.Serialize<T>(obj);
    request = WebRequest.Create(uri);
    request.Method = "PUT";
    request.ContentLength = Encoding.UTF8.GetByteCount(payload);
    request.ContentType = "application/json";
    Send(request, payload);
}

private void Delete(Uri uri)
{
    request = WebRequest.Create(uri);
    request.Method = "DELETE";
    Response(request);
}

---

The Get function sends a HTTP GET request to the backend service and deserialize the JSON encoded response with the helper class JSONHelper, see Code snippet 13 for implementation. The Post, Put and Delete functions also adjust the WebRequest class to send the corresponding HTTP Request, the Post and Put functions also encode the payload the client is sending into JSON and adds the needed information we need to send the data over the HTTP protocol.
The JSONHelper class is a smooth way of converting theDataContract classes into JSON strings with some inbuilt features in .NET and C# so we do not need to write our own JSON parser and object parsers to do these conversions. There is also ways to do this with XML as well.

```csharp
class JSONHelper
{
    public static string Serialize<T>(T obj)
    {
        DataContractJsonSerializer serializer = new DataContractJsonSerializer(obj.GetType());
        MemoryStream ms = new MemoryStream();
        serializer.WriteObject(ms, obj);
        string returnValue = Encoding.UTF8.GetString(ms.ToArray());
        ms.Dispose();
        return returnValue;
    }

    public static T Deserialize<T>(string json)
    {
        T obj = Activator.CreateInstance<T>();
        MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(json));
        DataContractJsonSerializer serializer = new DataContractJsonSerializer(obj.GetType());
        obj = (T)serializer.ReadObject(ms);
        ms.Close();
        ms.Dispose();
        return obj;
    }
}
```

**Code snippet 13: Helper class for JSON serialization**

**Code snippet 14: Client CRUD operations on the Schema entity**

```csharp
public void CreateSchema(SchemaContract schema)
{
    Uri uri = new Uri(baseUri, "fidb/schema");
    Post<SchemaContract>(uri, schema);
}

public SchemaContract GetSchema(int schemaId)
{
    Uri schemaUri = new Uri(baseUri, "fidb/schema/" + schemaId.ToString());
    return Get<SchemaContract>(schemaUri);
}

public void UpdateSchema(SchemaContract schema)
{
    Uri uri = new Uri(baseUri, "fidb/schema");
    Put<SchemaContract>(uri, schema);
}

public void DeleteSchema(int id)
{
    Uri uri = new Uri(baseUri, "fidb/schema/" + id.ToString());
    Delete(uri);
}
```
In Code snippet 14 is the functions that the frontend developer has access to in the Client API, there are the regular CRUD operations that has been seen in the backend. They all work in a similar fashion. Create the URI to the backend, the baseUri is the base address to the service but we extend it with the specific database and which resource we want to access. CreateSchema and UpdateSchema also take an SchemaContract that will be sent with the request to the service.
4 System Evaluation

This chapter will conclude the benefits and drawbacks of the proposed system implementation. It will focus on three main things, REST and SOA, Abstraction of the code and the Test suite and TDD.

4.1 Benefits and drawbacks with REST and SOA

With every design decision you weigh the pro and cons of the design and there is of course some drawbacks of Service Oriented Architecture but there is also valid benefits of it as well.

4.1.1 Drawbacks

The biggest drawback with a Service Oriented Architecture and application is the complexity of the whole software. You will have to maintain the code of the backend service and as well the frontend application that is built upon the backend. If the network connection is lousy, i.e. slow connector or a congested network, the application will have some performance issues and this will affect the user experience in the frontend application.

As you have a public interface to the surrounding world for your backend you need to implement more security features, i.e. API keys for authenticating users of the backend service and log their actions. We also need to check the user input even more meticulous for invalid data, i.e. malicious SQL injections that can expose the whole database. In a regular desktop application the developer have more control over what kind of call is done to the database and can also sandbox the execution of the application.

4.1.2 Benefits

The main benefit of Service Oriented Architecture is the separation of concerns you get for free when you split up your software in a client-server manner.

All the database transaction will end up in the backend of the system and you can put developers that are really good at developing database application on developing the backend – while you can put developers that are good on developing GUI and user experience driven applications on the frontend.

Another big benefit with a RESTful Web Service is that you get an “platform independent” platform to build your applications around, of course the service will run on a platform specific server, i.e. IIS in this case. But the frontend applications can be run on any platform as long as they can connect to the network and implement interface to connect to the backend. So you can implement a frontend application on Windows, Unix, Android and iOS platform with ease and you can use the programming language you prefer as well as the communication is built on the HTTP protocol as application protocol and not as a transport protocol as it is used in SOAP.
4.2 Abstractions of Layers

There are quite a few abstraction layers in the current design; the Model layer consist of the Entity Model classes, the specific Entity Connector classes and the façades that wrap the Entity Connector classes, the Controller layer consists of the two Controller classes that calls the Connector façades. The last layer is the Service Layer that implements the service and allows for network communication.

This abstraction allows us to easily debug the backend when a bug is found as we know what each specific layer does. If there are problems with the connectivity you will debug the Service Layer to begin with and if there is faulty data in theDataContract objects you will start to debug the Controller classes and if you get database errors you debug the Connector classes.

4.3 Test suite and Test Driven Development

This is by far the biggest benefit with the whole system – the “free” test suite you get while you are developing the system, each class will have a test suite that can be run at anytime. The test suite has another benefit as well – it gives new developers of the backend service a very good example on the intended use of the classes.

Test Driven Development is tedious and slows down the development pace but allows/forces the developer to take a step back and contemplate on the question “What should this class really do?”. This will result in very robust and neat classes and by following the Single-Responsibility principle the classes can be easily tested. When you get used to TDD, the process is a very good way to develop your software after.


5 Discussion and conclusions

This chapter will consist of the personal experience of the development process, technologies and the frameworks that has been used.

5.1 .NET Framework

The system is developed in .NET 4.0 but in .NET 4.5 there is a new Wizard for something called “Web API” and this is more or less the same thing as this thesis implements but in a even more structured way and utilizes the Entity Model properties in a better way. This is one big downside with the .NET framework, it is so vast and big that it is hard to find the perfect solution to your problems – when you Google for tips and tricks you will get hits on solutions to old .NET versions which is not relevant to the current version and when you try to find tips for the current versions there is not much there yet as it is new – but this is also a downside to all frameworks that is developed with time.

The .NET Framework has quite a big learning curve as well – you have lots of configuration in XML files to begin with, sure when you get the hang of it is quite simple but it takes time to get there. The first week in development of the FIToolBackend service was just configuring and figuring out how the Visual Studio 2010 IDE works and to get the Test framework to work with the specific Test databases. To solve the connection issues I had to add a specific App.config XML-file in the Test project that had the unique connection strings to the databases so that the EntityFramework could locate them while running the automated test suite.

5.2 Other frameworks and platforms

There are a vast number of frameworks and platforms you can use to implement a Service Oriented application. For example for the Python language you have the Django and Twisted frameworks, for the Ruby language you have Ruby on Rails (RoR) and Sinatra. Both theses language is free to use and the frameworks are free to use and open source – this means there is a big community around the frameworks and lots of help online to be found.

All these four frameworks are only intended for developing Service Oriented application and are therefore less schizophrenic as the .NET Framework can be, the .NET Framework wants to solve many different issues and has a big standard library while Python and Ruby has relatively small standard libraries compared to .NET. Python and Ruby relies more on community developed packages and frameworks. Sinatra for Ruby is a nice and easy framework to get started with web application with minimal effort.

```
# myapp.rb
require 'sinatra'

get '/hello' do
  'Hello world!'
end
```

Code snippet 15: Simple Web app in Sinatra
In Code snippet 15 is an example of a simple “Hello world” application in the Sinatra framework for Ruby.

As both Ruby and Python is interpreted language you will get inferior performance compared to a precompiled language such as C#. You can solve this issue by using better and/or more hardware but it can cost quite a bit of money to upgrade your current servers.

5.3 Conclusions and future work

The .NET Framework is adapting the concepts of Service Oriented Architectures and REST more and more with every new release of the framework and it is getting to be a quite competent choice to implement these services. With the release .NET 4.5 there have been great improvement on just this frontier and with the introduction of the Web API project wizard it allows you to start the development of your application even quicker.

It would have been interesting to also investigate the security implementations in the WCF Framework and .NET and how these could have been used in these could have been used in this application but because of limited time this had to be dropped. It would also be interesting to try to implement a number of front-end applications on different applications to really show the power of the Service-Oriented Architecture.

Another aspect that would be fun to investigate is the concept of reflections, allows the program to examine and modify the structure and behaviour of the code at runtime, in C# and to implement a plugin system for the backend to allow the system to “plugin” a new database easily without changing any of the code of the backend, which will allow the system to be even more dynamic.
6 Bibliography


7 Appendices

7.1 File structure of the project in Visual Studio 2010

FIToolBackend
  /References
  /FIToolBackend.DataContracts
/App_Data
  /FIToolDatabase.mdf
  /RecordingDatabase.mdf
/Controllers
FIToolController.cs
  /RecordingController.cs
/DatabaseConnectors
FITool
  /ExpressionConnector.cs
  /LinkConnector.cs
  /NodeConnector.cs
  /SchemaConnector.cs
  /VariableConnector.cs
/Recording
  /TypeRecordingConnector.cs
  /VariableRecordingConnector.cs
FIToolDatabaseConnector.cs
  /RecordingDatabaseConnector.cs
  /IConnector.cs
/FIEntityModel.edmx
/FIEntityModel.edmx.sql
/FIToolService.svc
/IFIToolService.cs
/RecEntityModel.edmx
/RecEntityModel.edmx.sql
/Web.config

FIToolBackend.DataContracts
  /FITool
  /ExpressionContract.cs
  /LinkContract.cs
  /NodeContract.cs
  /SchemaContract.cs
  /VariableContract.cs
/Recording
  /TypeContract.cs
  /VariableRecordingContract.cs
FIToolBackend.Test

/Controllers
  /ControllerTests.cs
  /FIToolControllerTests.cs
  /RecordingControllerTests.cs

/DataContracts
  /ContractTests.cs
  /ExpressionContractTest.cs
  /LinkContractTest.cs
  /NodeContractTest.cs
  /SchemaContractTest.cs
  /VariableContractTest.cs

/FIToolConnectors
  /ConnectorTests.cs
  /ExpressionConnectorTests.cs
  /FIToolDBCTests.cs
  /IntegrationTests.cs
  /LinkConnectorTests.cs
  /NodeConnectorTests.cs
  /SchemaConnectorTests.cs
  /VariableConnectorTests.cs

/RecordingConnectors
  /RecDBConnectorTests.cs
  /RecordingTests.cs
  /TypeRecConnectorTests.cs
  /VariableRecordingConnectorTests.cs

App.config
FIToolsDatabaseTest.mdf
RecordingDatabaseTest.mdf