Network Design and Services

Project (Web Services)

Document History

Version	Date	Kind	Author
0.1	30.04.2015	Theory	strut1 & touwm1
0.2	21.05.2015	Implementation	strut1 & touwm1
0.3	23.05.2015	Review draft	strut1 & touwm1
1.0	25.05.2015	Prepare final document	strut1 & touwm1

Table 1: Document History

Content

1 INTRODUCTION AND PURPOSE ... 3
2 MATERIALS ... 3
3 THEORY ... 4
3.0 STATEMENT ... 4
3.1 WEB SERVICES ... 4
3.1.1 SOAP ... 5
3.1.2 REST ... 9
3.2 COMPARISON OF SOAP AND REST ... 11
3.3 FURTHER READING AND REFERENCES ... 11
4 REALISATION ... 12
4.1 INTRODUCTION ... 12
4.2 GRAPHICAL USER INTERFACE ... 13
4.2.1 Control Station ... 13
4.2.2 Terminal ... 15
4.3 CODE ... 18
4.3.1 Web Service Implementation ... 18
4.3.2 Client Implementation ... 18
4.3.3 Control Station Service Implementation ... 19
4.3.4 Control Station Client Implementation ... 21
4.3.5 Terminal Service Implementation ... 23
4.3.6 Terminal Client Implementation ... 24
5 DISCUSSION AND ANALYSIS ... 25
6 CONCLUSIONS ... 25

Figures

FIGURE 1: SOAP ENVELOPE 5
FIGURE 2: APPLICATION SCHEMA 12
FIGURE 3: CONTROL STATION APPLICATION – PRIMARY VIEW 13
FIGURE 4: CONTROL STATION APPLICATION –STATISTICS VIEW 14
FIGURE 5: TERMINAL APPLICATION – MAIN FORM 15
FIGURE 6: TERMINAL APPLICATION – ADD EMPLOYEE DIALOG 16
FIGURE 7: TERMINAL APPLICATION – EMPLOYEE DETAILS DIALOG 16
FIGURE 8: CONTROL STATION APPLICATION – NEW NEWS TEXT 17
FIGURE 9: TERMINAL APPLICATION – NEW NEWS TEXT 17
FIGURE 10: CONTROL STATION SERVICE CLASS DIAGRAM 19
FIGURE 11: CONTROL STATION CLIENT CLASS DIAGRAM 21
FIGURE 12: TERMINAL SERVICE CLASS DIAGRAM 23
FIGURE 13: TERMINAL CLIENT CLASS DIAGRAM 24

Code Fragments

CODE FRAGMENT 1: SOAP REQUEST 6
CODE FRAGMENT 2: SOAP RESPONSE 6
CODE FRAGMENT 3: WSDL V1.1 FILE – PART 1 7
CODE FRAGMENT 4: WSDL V1.1 FILE – PART 2 8
CODE FRAGMENT 5: REST REQUEST 10
CODE FRAGMENT 6: REST RESPONSE 10
CODE FRAGMENT 7: CONTROL STATION SERVICE IMPLEMENTATION 20
CODE FRAGMENT 8: CONTROL STATION CLIENT IMPLEMENTATION 22
CODE FRAGMENT 9: TERMINAL SERVICE IMPLEMENTATION 23

Tables

TABLE 1: DOCUMENT HISTORY 1

1 Introduction and Purpose

Instructor: Ueli Schrag
Date: 30.04 – 21.05.2015
Group: I2a - #4
Members: Marc Touw & Thomas Reto Strub

For the *Network Design and Services* group project, the instructor published a list of possible projects and we had to choose one of the projects on this list or to come up with our own idea.
We chose the project called "Web Services" because we are very interested in web technologies and Web Services in particular. We already heard a lot about web services, but we never had the chance to implement one by ourselves.

2 Materials

Unlike before, we decided to implement this project in (MS Visual) C# and therefore used Visual Studio. During the implementation phase we used our own notebooks to test the application on. But the application could also be used in the lab environment if needed to.

3 Theory

3.0 Statement

The following chapters have been written by the team members only.
No paragraphs or figures have been copied from other sources, although some paragraphs and figures may have been influenced by web pages or presentations listed in chapter 3.3 (Further Reading and References) on page 11.

3.1 Web Services

A Web Service is many ways similar to a Web Application. The main difference is the targeted audience. A Web Application is designed to present its data to a human user via a specific category of client software – a web browser. A Web Services is designed to deliver its data to a non-human client of any kind. A Web Service thus does not “present” but merely serialise the delivered data.
Web Services can be used to allow different applications to communicate over a network. Web Services are specifically useful when communicating over the internet. They make use of standard network protocols like HTTP or SMTP to bypass even strictly configured firewalls.
Web Services are mostly independent of the platform they run on or the language they are implemented in. To achieve the required level of interoperability, various standards have been defined on how to design the Web Service interfaces. Two of these have grown to become widely accepted and will be further explained in this document: SOAP and REST.

3.1.1 SOAP

3.1.1.1 Introduction

SOAP was originally an acronym for *Simple Object Access Protocol*, but this meaning has since been lost.
It is usually served over HTTP, but can use other protocols as well (e.g. SMTP). Using predefined and well-known protocols has the already mentioned advantage that even strictly configured firewalls will highly likely let the packets pass into and out of the local network. By using a proprietary, unknown protocol you risk being blocked by a firewall configured not to deliver packets through certain ports.
However, its messages always have to be encoded in XML. Its specification requires the messages to be contained in a so-called envelope. This envelope may contain additional message headers besides the already existing protocol headers. But it has to contain the mandatory message body. This message body may contain arbitrary XML data or optionally a SOAP fault indicating an error of any type. It may optionally be wrapped in a MIME container and contain non-XML attachments.

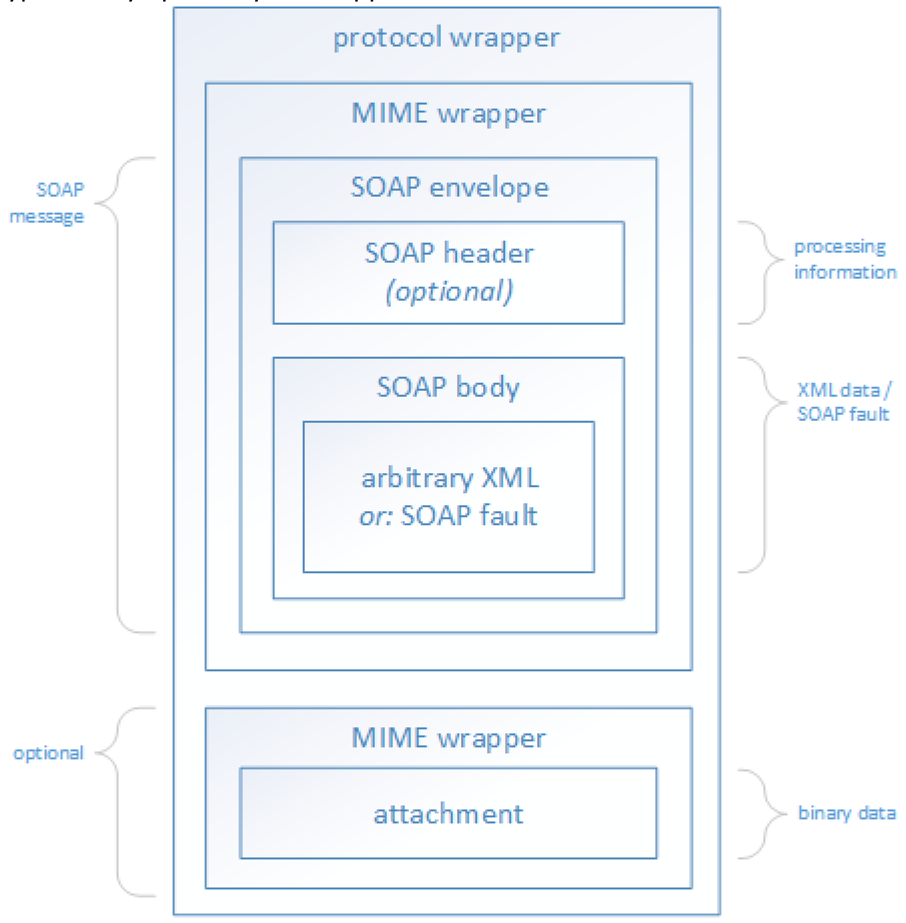


Figure 1: SOAP envelope

3.1.1.2 Example Request

```
POST http://localhost:5678/ HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "http://tempuri.org/TerminalService/GetStatistics"
Content-Length: 218
Host: localhost:5678
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
```

Code Fragment 1: SOAP request

3.1.1.3 Example Response

```
HTTP/1.1 200 OK
Content-Length: 439
Content-Type: text/xml; charset=utf-8
Server: Microsoft-HTTPAPI/2.0
Date: Sun, 24 May 2015 11:54:06 GMT

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <GetStatisticsResponse xmlns="http://tempuri.org/">
   <GetStatisticsResult
     xmlns:a="http://schemas.datacontract.org/.../BFH.NetDS.WebServices.Terminal"
     xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
    <a:numberOfTimeStamps>3</a:numberOfTimeStamps>
    <a:numberOfUniqueUsers>1</a:numberOfUniqueUsers>
   </GetStatisticsResult>
  </GetStatisticsResponse>
 </s:Body>
</s:Envelope>
```

Code Fragment 2: SOAP response

3.1.1.4 WSDL

3.1.1.4.1 Introduction

The WSDL or *Web Service Description Language* is a way to describe Web Services (most notably SOAP Web Services).
The WSDL is particularly useful for easily publishing information about the Web Service in a well-known and reliable way. WSDL files can be used in a contract to specify how two parties can invoke each other's Web Services.
A WSDL v1.1 file contains the following information:

- **XML Schemas** define the transferred arbitrary (user-defined) XML data.
- **Messages** define the transferred SOAP messages. *Messages have been removed in v2.0.*
- **Port types** *(**interfaces** in v2.0)* define the Web Service interface(s).
  - **Operations** define the methods the interface provides.
- **Bindings** define the protocol(s) to use to invoke the operations.
- **Services** define the actual Web Services available.
  - **Ports** *(**endpoints** in v2.0)* define the actually available Web Service interfaces.

3.1.1.4.2 Example Document (Version 1.1)

```
<wsdl:definitions name="TerminalService" targetNamespace="http://tempuri.org/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://tempuri.org/"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl>
 <wsdl:types>
  <xsd:schema targetNamespace="http://tempuri.org/Imports">
   <xsd:import schemaLocation="http://localhost:5678/?xsd=xsd0"
     namespace="http://tempuri.org/"/>
   <xsd:import schemaLocation="http://localhost:5678/?xsd=xsd1"
     namespace="http://schemas.microsoft.com/2003/10/Serialization/"/>
   <xsd:import schemaLocation="http://localhost:5678/?xsd=xsd2"
     namespace="http://schemas.datacontract.org/2004/07/BFH...Terminal"/>
  </xsd:schema>
 </wsdl:types>
```

Code Fragment 3: WSDL v1.1 file – part 1

Please continue reading on the next page.

```
 <wsdl:message name="TerminalService_SetNews_InputMessage">
  <wsdl:part name="parameters" element="tns:SetNews"/>
 </wsdl:message>
 <wsdl:message name="TerminalService_SetNews_OutputMessage">
  <wsdl:part name="parameters" element="tns:SetNewsResponse"/>
 </wsdl:message>
 <wsdl:message name="TerminalService_GetStatistics_InputMessage">
  <wsdl:part name="parameters" element="tns:GetStatistics"/>
 </wsdl:message>
 <wsdl:message name="TerminalService_GetStatistics_OutputMessage">
  <wsdl:part name="parameters" element="tns:GetStatisticsResponse"/>
 </wsdl:message>
 <wsdl:portType name="TerminalService">
  <wsdl:operation name="SetNews">
   <wsdl:input message="tns:TerminalService_SetNews_InputMessage"
     wsaw:Action="http://tempuri.org/TerminalService/SetNews"/>
   <wsdl:output message="tns:TerminalService_SetNews_OutputMessage"
     wsaw:Action="http://tempuri.org/TerminalService/SetNewsResponse"/>
  </wsdl:operation>
  <wsdl:operation name="GetStatistics">
   <wsdl:input message="tns:TerminalService_GetStatistics_InputMessage"
     wsaw:Action="http://tempuri.org/TerminalService/GetStatistics"/>
   <wsdl:output message="tns:TerminalService_GetStatistics_OutputMessage"
     wsaw:Action="http://tempuri.org/TerminalService/GetStatisticsResponse"/>
  </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="BasicHttpBinding_TerminalService" type="tns:TerminalService">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="SetNews">
   <soap:operation soapAction="http://tempuri.org/TerminalService/SetNews"
     style="document"/>
   <wsdl:input>
    <soap:body use="literal"/>
   </wsdl:input>
   <wsdl:output>
    <soap:body use="literal"/>
   </wsdl:output>
  </wsdl:operation>
  <wsdl:operation name="GetStatistics">
   <soap:operation soapAction="http://tempuri.org/TerminalService/GetStatistics"
     style="document"/>
   <wsdl:input>
    <soap:body use="literal"/>
   </wsdl:input>
   <wsdl:output>
    <soap:body use="literal"/>
   </wsdl:output>
  </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="TerminalService">
  <wsdl:port name="BasicHttpBinding_TerminalService"
    binding="tns:BasicHttpBinding_TerminalService">
   <soap:address location="http://localhost:5678/"/>
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Code Fragment 4: WSDL v1.1 file – part 2

3.1.2 REST

3.1.2.1 Introduction

REST stands for *Representational State Transfer*.
Unlike SOAP, REST is not a protocol but merely an architecture style defining six constraints a RESTful (Web) Service has to fulfil.
Typically RESTful Web Services are invoke over HTTP. The data transferred with REST can be in any format (machine-readable or not) but most RESTful Web Services use either XML, JSON(-LD), (X)HTML or plain text. To transfer binary data, you do not need attachments or similar workarounds, you can simply return the binary data together with the correct Content-Type (in case of HTTP).
The six constraints are:

1) **Client-Server**: Client and server have to be properly separated by a uniform interface. Clients and servers have different responsibilities: e.g. servers have to deal with data storage but not with the user interface. The servers and clients are independent of each other and may be updated independently as long as the interface is not altered.
2) **Stateless**: The communication between the client and the server has to be stateless. There must not be maintained any sessions, cookies or other storage of sorts. All information necessary (the *State*) for an operation has to be included in the request or response (to be *Transfer*red).
3) **Cacheable**: Just like in Web Applications, responses can be cached by any involved part (the server, the client or any intermediary).
4) **Layered System**: The Web Service must be able to support intermediary layers e.g. to improve scalability or to enforce security policies.
5) **Code on Demand**: *optional and not discussed here*
6) **Uniform Interface**
   a. **Identification of resources**: Every resource must have a unique identification (in Web Services, resources are identified by URIs). The identification of the resource does not depend on the representation they are returned in. If used with HTTP, the client may specify the desired representation using the Accept header.
   b. **Manipulation of resources through these representations**: Once the client obtained a representation of a resource, he holds all the information necessary to update or delete a resource. The action to perform on a resource is defined by the HTTP method (if used in conjunction with HTTP): GET returns the resource, PUT replaces the resource, POST creates a new resource and DELETE deletes a resource.
   c. **Self-descriptive messages**: A response must include all the information necessary to process it. For example, it has to indicate the status (success, error), the format of the representation.
   d. **Hypermedia as the engine of application state (HATEOAS)**: The only way to maintain an application state is by using hypermedia and the references within it. A client only knows where to enter the application, from there on it can use the hypermedia to dynamically identify the actions available to it.

This last sub-constraint (HATEOAS) is a bit of a catch. Many developers (including us) do not strictly follow it. By specifying *hypermedia* as the engine of application state it requires the requests and responses to use hypermedia. But: JSON is not hypermedia (although JSON-LD is). This means, if you use JSON, you strictly cannot call your Web Service RESTful!

3.1.2.2 Example Request

```
POST http://localhost:6789/employee HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: application/json
Content-Length: 37
Host: localhost:6789
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

{"login": "marc","name": "Marc Touw"}
```

Code Fragment 5: REST request

3.1.2.3 Example Response

```
HTTP/1.1 200 OK
Content-Length: 35
Content-Type: application/json; charset=utf-8
Server: Microsoft-HTTPAPI/2.0
Date: Sun, 24 May 2015 13:31:10 GMT

{"login": "marc","name": "Marc Touw"}
```

Code Fragment 6: REST response

3.2 Comparison of SOAP and REST

It is not really fair to compare SOAP to REST because they are two different things. However, as a rough guide: use SOAP for big, complex applications and REST for small, dynamic ones.

3.3 Further Reading and References

- Web Services in general
  - Wikipedia page on Web Services
    http://en.wikipedia.org/wiki/Web_service
  - W3Schools tutorial on Web Services
    http://www.w3schools.com/webservices/default.asp
  - Information on both SOAP and RESTful Web Services on SoapUI's Testing Dojo
    http://www.soapui.org/testing-dojo/world-of-api-testing/soap-vs--rest-challenges.html
  - Presentation on Web Services (especially in Java EE) by Stephan Fischli (2015)
    http://www.sws.bfh.ch/~fischli/courses/info/javaee/scripts/JavaWebServices.pdf
- SOAP
  - Wikipedia on SOAP Web Services
    http://en.wikipedia.org/wiki/SOAP
  - W3C SOAP recommendation
    http://www.w3.org/TR/soap/
  - Presentation on SOAP Web Services by Marine Yegoryan (2008)
    http://www.slideshare.net/guest0df6b0/web-service-presentation
- REST
  - Wikipedia page on REST
    http://en.wikipedia.org/wiki/Representational_state_transfer
  - Presentation on RESTful Web Services by David Zülke at the Dutch PHP Conference (2008)
    http://www.slideshare.net/Wombert/designing-http-interfaces-and-restful-web-services-dpc2012-20120608
  - Presentation on HTTP by Ross Tuck at ZendCon (2014)
    http://www.slideshare.net/rosstuck/http-and-your-angry-dog
  - Tutorial on RESTful Web Services by M. Vaqqas on Dr. Dobb's (2014)
    http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069

4 Realisation

The realisation can be browsed and is available for downloading on GitHub.
Link: https://github.com/StrubT/NetDSWebServices

4.1 Introduction

We implemented a system made up of two separate applications.
The control station application provides a simple graphical user interface to check the terminals' statuses as well as set their news texts. It also provides a RESTful Web Service to fetch and update employee information.
The terminal application provides a simple graphical user interface to add employees and time stamps. It also provides a SOAP Web Service to fetch terminal statistics and set the news text.
Multiple terminal applications may be started on multiple hosts and connected to the same control station.
The implementation is supposed to demonstrate SOAP as well as RESTful Web Services but is not elaborate enough to be used in a productive manner. If one of the applications break, the control station and all of the terminals should be restarted and re-connected.

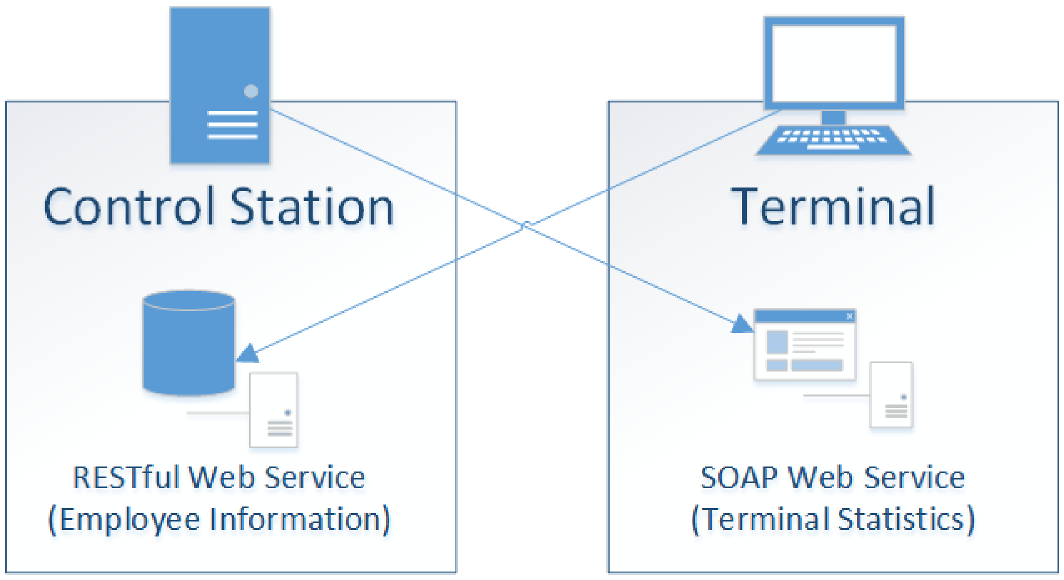
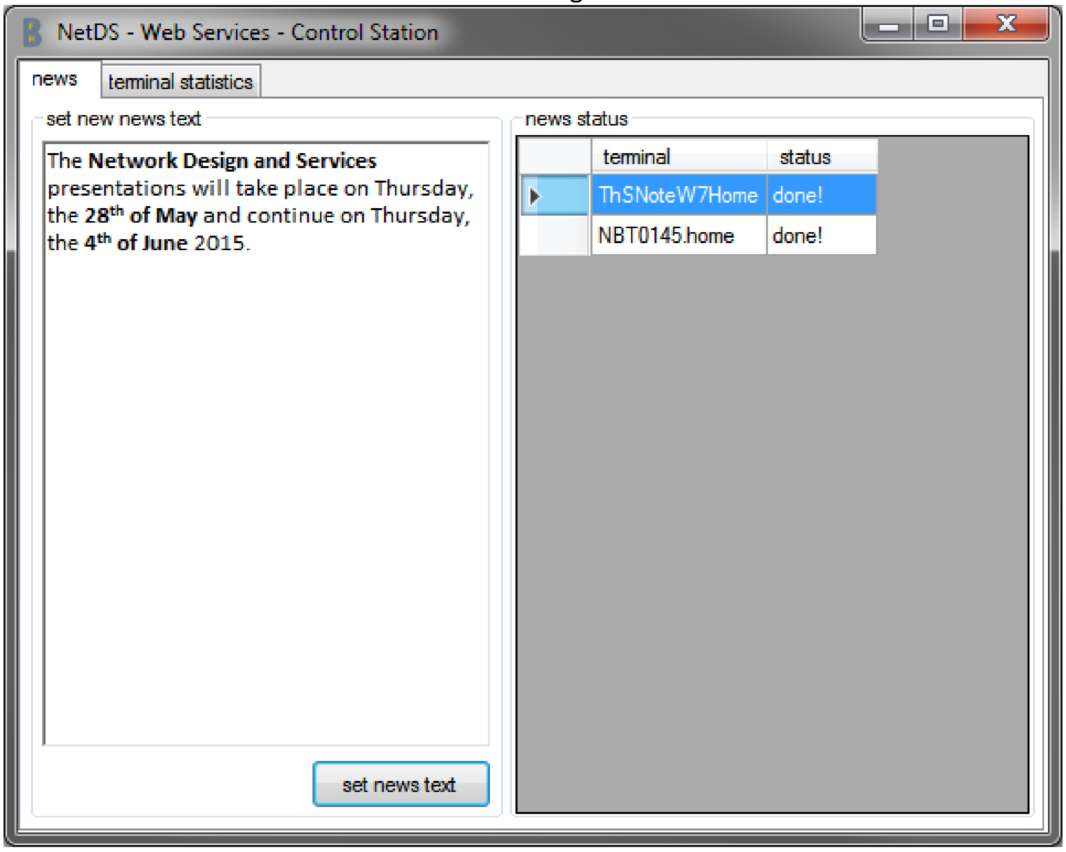


Figure 2: Application schema

4.2 Graphical User Interface

4.2.1 Control Station

The control station application provides to views: the primary view for checking the terminals' status and setting their news texts and the secondary view with the terminal statistics.
The primary view consists of the text area for entering the news texts on the left and the grid with the list of terminals on the right. If the user enters a new news text and clicks *set news text*, he can observe the different terminals' status on the right. *done!* Indicates the text has been set.



The **Network Design and Services** presentations will take place on Thursday, the **28th of May** and continue on Thursday, the **4th of June** 2015.

terminal	status
ThSNoteW7Home	done!
NBT0145.home	done!

Figure 3: Control station application – primary view

The statistics view only consists of a single chart with the possibly different terminals and the numbers of unique users and time stamps made on each of them.

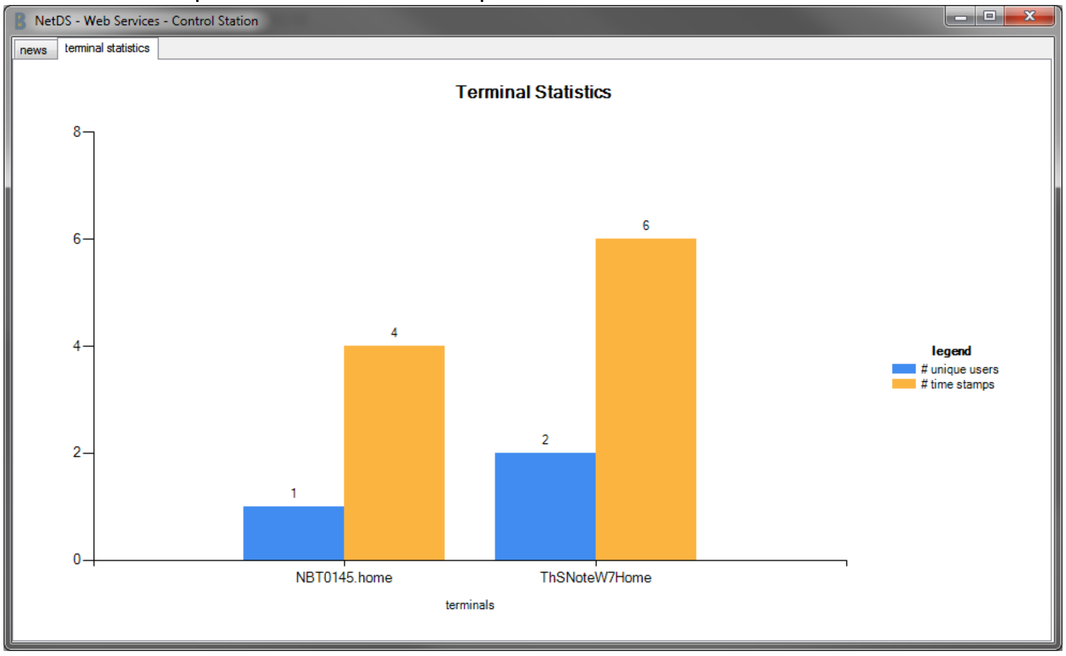
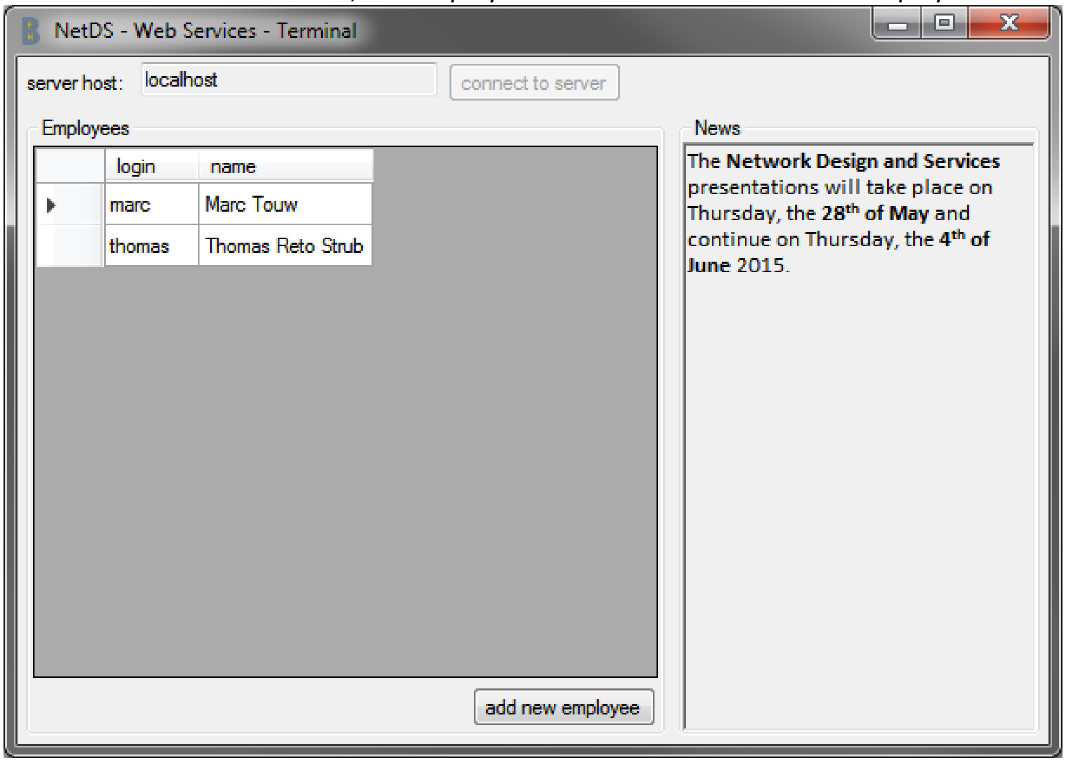


Figure 4: Control station application –statistics view

4.2.2 Terminal

The terminal application has one form with two popups. The form lists the different users and displays the news text. One popup is shown to create a new employee and the other to add new time stamps to a selected user.
The main form provides a text field to enter the host name or the IP address of the control station. Once entered and connected, the employees are listed and news text is displayed.



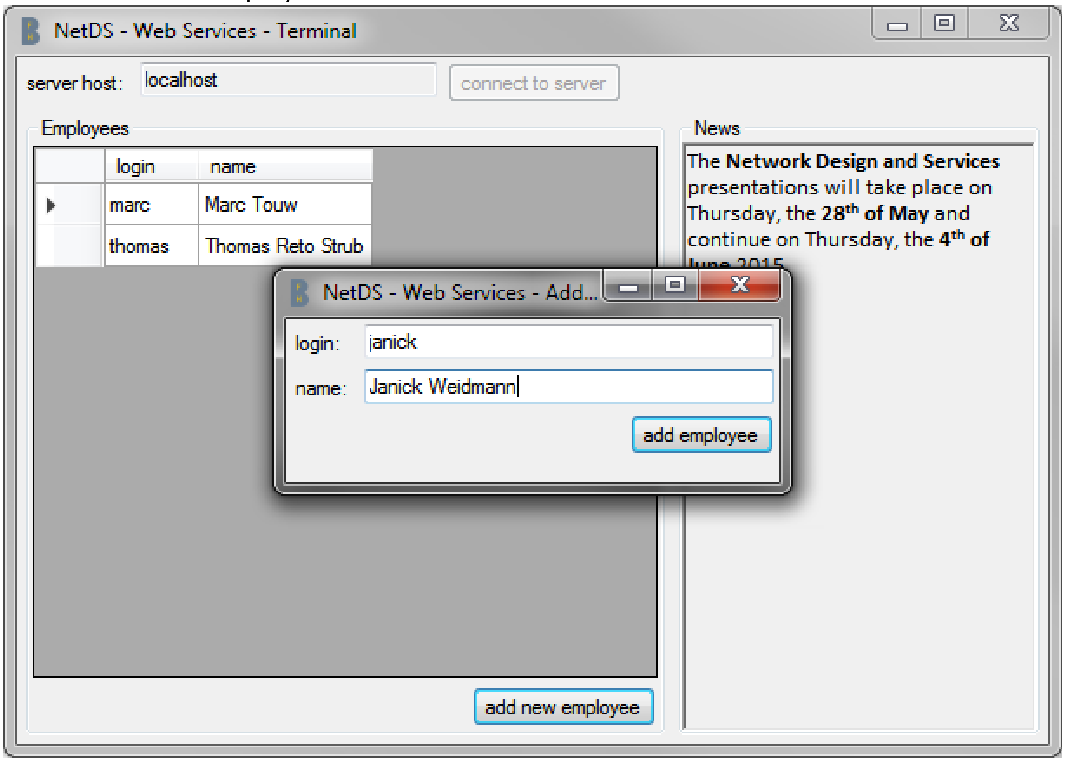
server host: localhost

login	name
marc	Marc Touw
thomas	Thomas Reto Strub

The **Network Design and Services** presentations will take place on Thursday, the **28th of May** and continue on Thursday, the **4th of June** 2015.

Figure 5: Terminal application – main form

If the user clicks the button *add new employee*, a popup opens and the user has to enter the login and name of the employee to add.



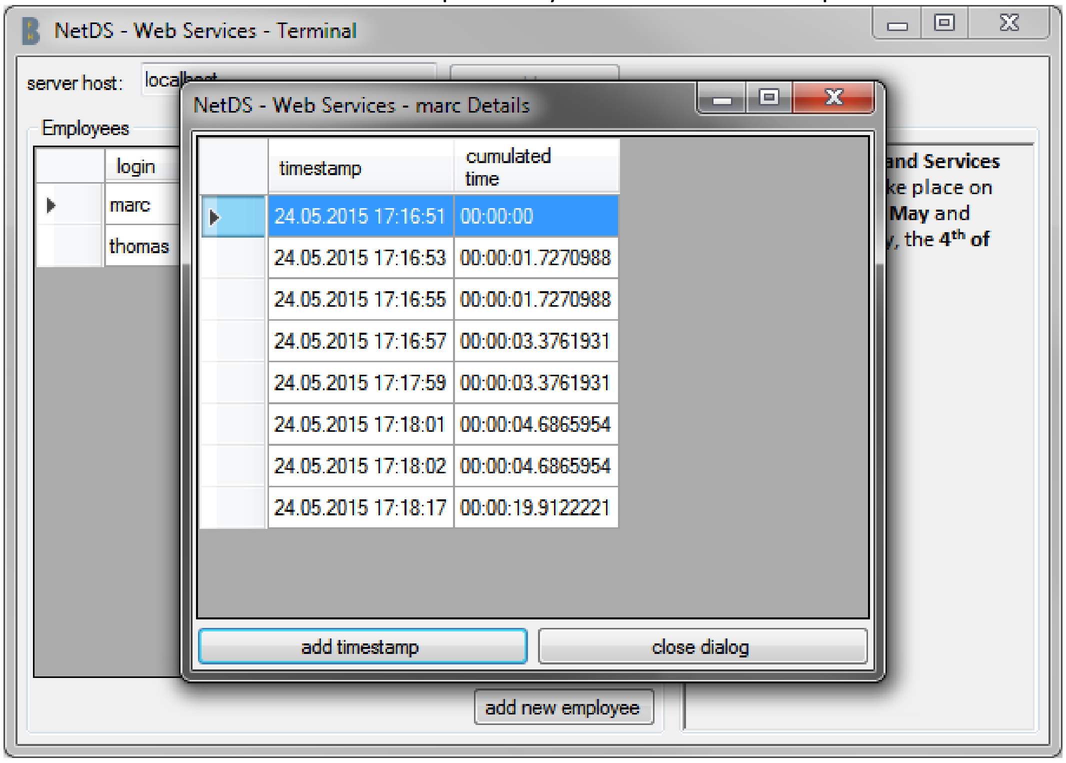
	login	name
▶	marc	Marc Touw
	thomas	Thomas Reto Strub

The **Network Design and Services** presentations will take place on Thursday, the **28th of May** and continue on Thursday, the **4th of June 2015**

login: janick
name: Janick Weidmann

Figure 6: Terminal application – add employee dialog

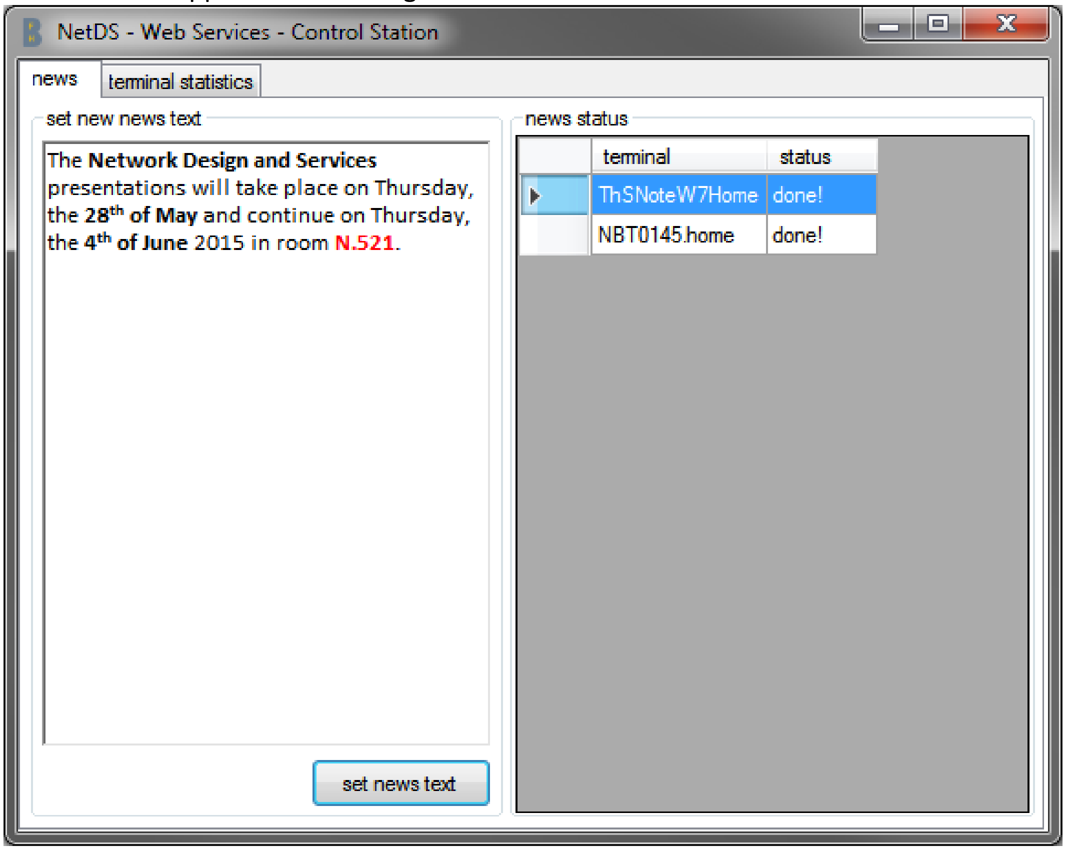
If the user selects an employee from the grid, a second popup opens and lists the time stamps of the selected user. The user then has the possibility to add new timestamps.


NetDS - Web Services - Terminal
server host:
Employees
NetDS - Web Services - marc Details
add timestamp
close dialog
add new employee

	timestamp	cumulated time
▶	24.05.2015 17:16:51	00:00:00
	24.05.2015 17:16:53	00:00:01.7270988
	24.05.2015 17:16:55	00:00:01.7270988
	24.05.2015 17:16:57	00:00:03.3761931
	24.05.2015 17:17:59	00:00:03.3761931
	24.05.2015 17:18:01	00:00:04.6865954
	24.05.2015 17:18:02	00:00:04.6865954
	24.05.2015 17:18:17	00:00:19.9122221

Figure 7: Terminal application – employee details dialog

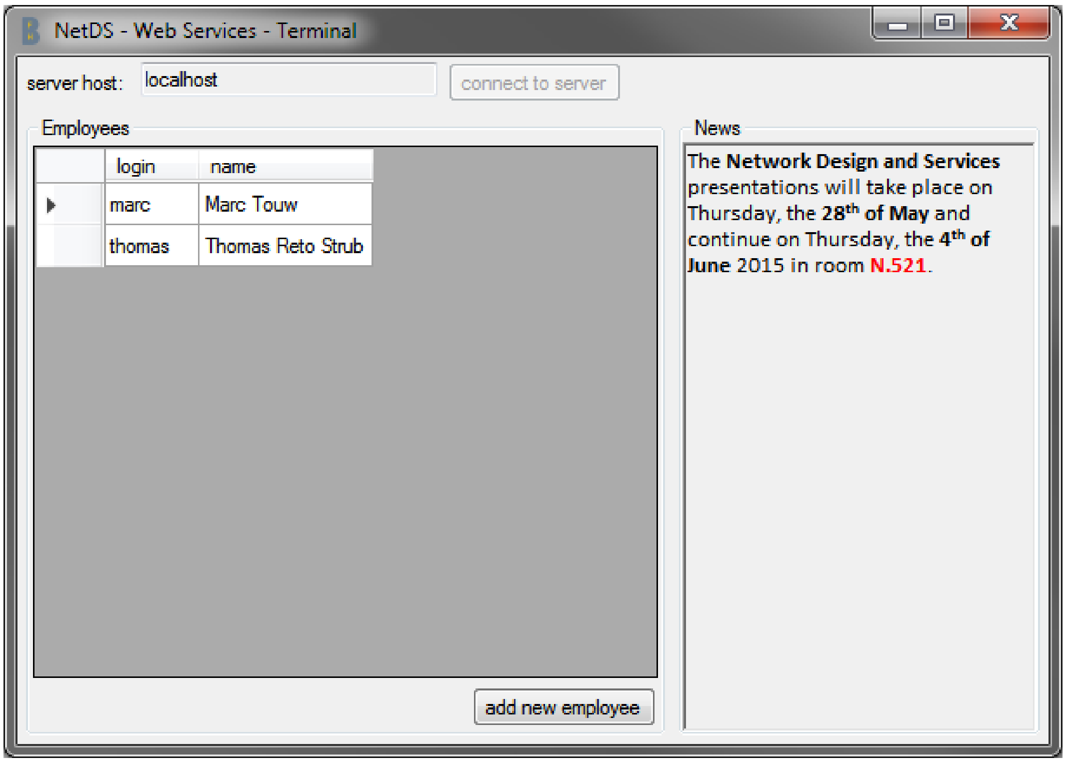
Once the user uses the control station application to set a new news text, it is automatically changed in all terminal applications running.



The **Network Design and Services** presentations will take place on Thursday, the **28th of May** and continue on Thursday, the **4th of June** 2015 in room **N.521**.

terminal	status
ThSNoteW7Home	done!
NBT0145.home	done!

Figure 8: Control station application – new news text



server host: localhost

login	name
marc	Marc Touw
thomas	Thomas Reto Strub

The **Network Design and Services** presentations will take place on Thursday, the **28th of May** and continue on Thursday, the **4th of June** 2015 in room **N.521**.

Figure 9: Terminal application – new news text

4.3 Code

We do not intend to document the whole code the application comprises of. However, we would like to explain some code fragments of the Web Service implementation.

4.3.1 Web Service Implementation

To provide a Web Service in the .NET Framework, the web service classes (or interfaces) need to be marked with the `[ServiceContract]` attribute and the invokable methods need to be marked with the `[OperationContract]` attribute.
They are called contracts because their signature (name, parameter and return types) represent a contract between the provider and the caller. The invocation can only succeed if both parties follow the contract.
If the Web Service is provided via a SOAP interface, these attributes suffice. But if the Web Service should be provided via a RESTful interface, the methods need to be marked with a second attribute as well: either the `[WebGet]` or the `[WebInvoke]` attribute. These attributes define the resource identifier (via their `UriTemplate` property). The `[WebInvoke]` attribute additionally defines the HTTP method to use (via the `Method` property); remember: with a RESTful Web Service, the methods to get, replace, update and delete all use the same identifier, but different methods.
If the Web Service should not only exchange standard classes (e.g. **`string`** or **`int`**), but your own classes, they need to be marked with the `[DataContract]` attribute and the individual properties need to be marked with the `[DataMember]` attribute.
Once again, the data classes and their marked properties represent a contract between the provider and the caller, only if both use the same names and data types, the invocation will properly work.
The implemented applications do host the Web Services by themselves and do not rely on a web server. Therefore, a ServiceHost (or WebServiceHost for RESTful Web Services) has to be created to host the Web Services. This is quite straightforward and not explained any further.

4.3.2 Client Implementation

A SOAP Web Service client can be implemented automatically by Visual Studio. The only information needed is the WSDL file and Visual Studio can create all the classes needed to call the Web Service.
A RESTful Web Service client cannot be implemented automatically because there is no widely accepted standard for describing them yet. However, it is much easier to manually write a Web Service client for a RESTful Web Service because they are designed much simpler.
The client only has to make an HTTP request with the correct method and URI and parse the response.

4.3.3 Control Station Service Implementation

The control station Web Service is the more complex of the two.
It consist of four classes:

1) ControlStationService: service contract, provide Web Service methods
2) DataSource: handle reading from / writing to data source XML file
3) Employee: data contract, hold an employee's basic information
4) EmployeeTimeStamps: data contract, hold the time stamps of an employee

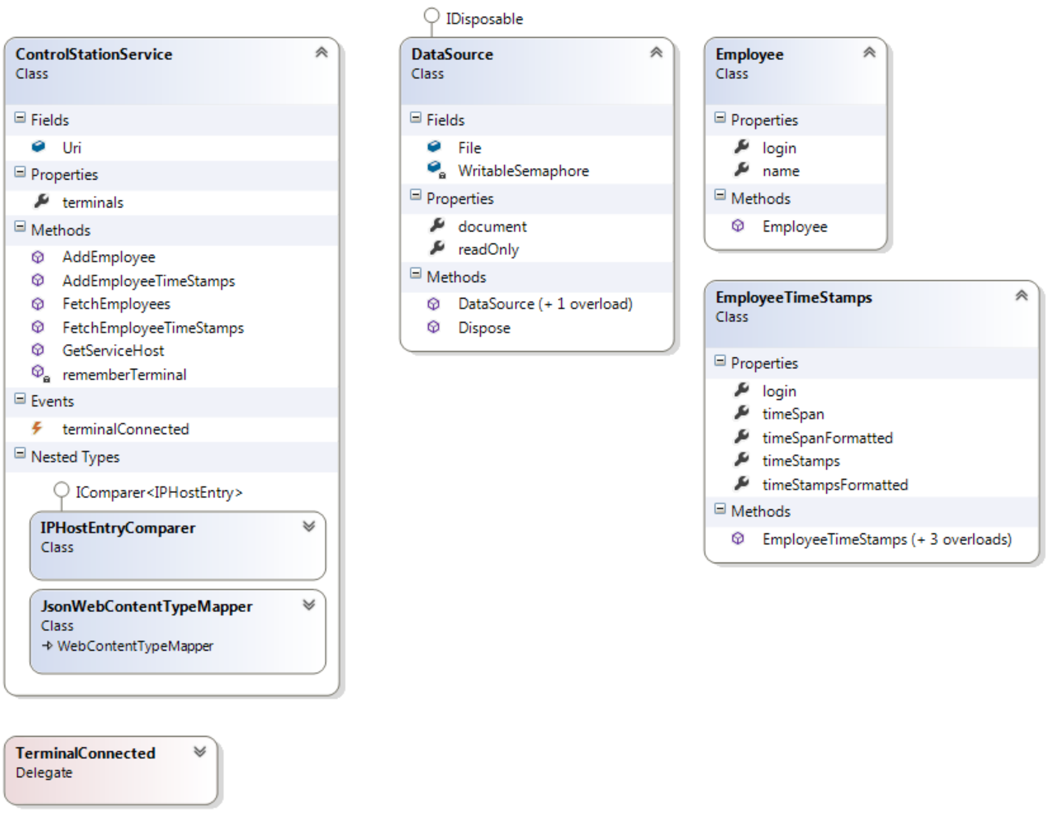


Figure 10: Control station service class diagram

```
[ServiceContract]
public class ControlStationService {

 public static readonly Uri Uri = new Uri("http://localhost:6789/");

 [OperationContract,
   WebGet(UriTemplate = "employee", ResponseFormat = WebMessageFormat.Json)]
 public List<Employee> FetchEmployees() { /*...*/ }

 [OperationContract,
   WebInvoke(Method = "POST", UriTemplate = "employee",
   RequestFormat = WebMessageFormat.Json,
   ResponseFormat = WebMessageFormat.Json)]
 public Employee AddEmployee(Employee employee) { /*...*/ }

 [OperationContract,
   WebGet(UriTemplate = "employee/{login}",
   ResponseFormat = WebMessageFormat.Json)]
 public EmployeeTimeStamps FetchEmployeeTimeStamps(string login) { /*...*/ }

 [OperationContract,
   WebInvoke(Method = "POST", UriTemplate = "employee/{login}",
   RequestFormat = WebMessageFormat.Json,
   ResponseFormat = WebMessageFormat.Json)]
 public EmployeeTimeStamps AddEmployeeTimeStamps(string login,
   EmployeeTimeStamps timeStamps) { /*...*/ }

 public static WebServiceHost GetServiceHost() {

  /*...*/
  var host = new WebServiceHost(typeof(ControlStationService), Uri);
  /*...*/
  host.Open();
  return host;
 }

 /*...*/
}

[DataContract]
public class Employee {

 [DataMember]
 public string login { get; private set; }

 [DataMember]
 public string name { get; private set; }

 public Employee(string login, string name) {

  this.login = login;
  this.name = name;
 }
}
```

Code Fragment 7: Control station service implementation

4.3.4 Control Station Client Implementation

The client also includes the Employee and EmployeeTimeStamps classes, and its own implementation of the service contract: the ControlStationServiceClient class.

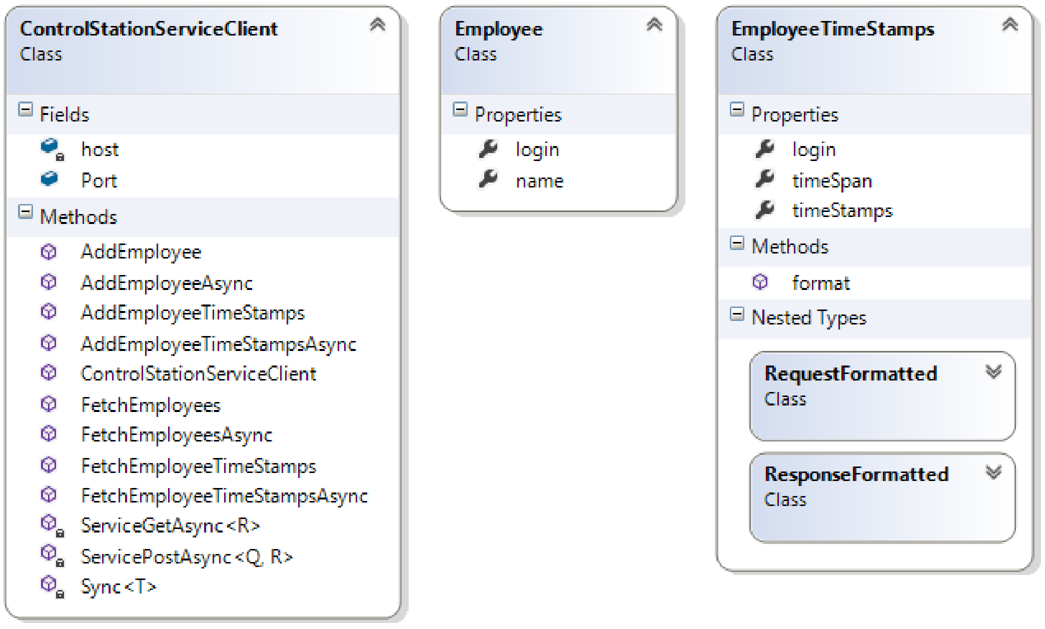


Figure 11: Control station client class diagram

```
public class ControlStationServiceClient {

 public static int Port = 6789;

 private string host;

 public ControlStationServiceClient(string host) { this.host = host; }

 public Task<List<Employee>> FetchEmployees() {
   return ServiceGet<List<Employee>>("employee"); }

 public Task<Employee> AddEmployee(Employee employee) {
   return ServicePost<Employee, Employee>("employee", employee); }

 public Task<EmployeeTimeStamps> FetchEmployeeTimeStamps(string login) {
   return ServiceGet<EmployeeTimeStamps>(string.Format("employee/{0}", login)); }

 public Task<EmployeeTimeStamps> AddEmployeeTimeStamps(
                                                  EmployeeTimeStamps timeStamps) {

  return (ServicePost<EmployeeTimeStamps.RequestFormatted,
    EmployeeTimeStamps.ResponseFormatted>(
    string.Format("employee/{0}", timeStamps.login), timeStamps.format())
    ).parse();
 }

 private Task<R> ServicePost<Q, R>(string path, Q body) {

  var req = WebRequest.CreateHttp(new UriBuilder("http", host, Port, path).Uri);
  req.Method = "POST";
  req.ContentType = "application/json";

  using (var stm = new StreamWriter(req.GetRequestStream()))
   stm.Write(JsonConvert.SerializeObject(body));

  var res = (HttpWebResponse)req.GetResponse();
  using (var stm = new StreamReader(res.GetResponseStream()))
   return JsonConvert.DeserializeObject<R>(stm.ReadToEnd());
 }

 private Task<R> ServiceGet<R>(string path) {

  var req = WebRequest.CreateHttp(new UriBuilder("http", host, Port, path).Uri);
  var res = (HttpWebResponse)req.GetResponse();
  using (var stm = new StreamReader(res.GetResponseStream()))
   return JsonConvert.DeserializeObject<R>(stm.ReadToEnd());
 }
}
```
Code Fragment 8: Control station client implementation

4.3.5 Terminal Service Implementation

The terminal service is much simpler than the control station service. The terminal service only consists of a TerminalService class providing the two Web Service methods GetStatistics and SetNews. The Statistics class is a data transfer class and contains the statistical information.

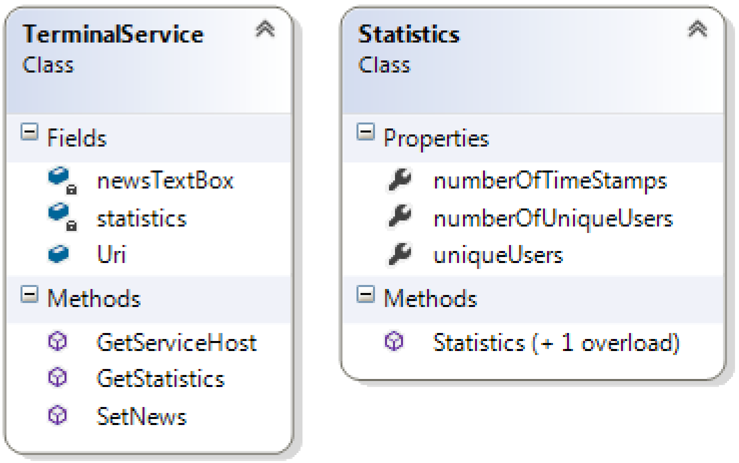


Figure 12: Terminal service class diagram

```
[ServiceContract]
public class TerminalService {

 public static readonly Uri Uri = new Uri("http://localhost:5678/");

 private static RichTextBox newsTextBox;
 private static Statistics statistics;

 [OperationContract]
 public void SetNews(string news) {

  newsTextBox.Rtf = news;
 }

 [OperationContract]
 public Statistics GetStatistics() {

 return statistics;
 }

 public static ServiceHost GetServiceHost(RichTextBox newsTextBox,
                                          Statistics statistics) {

  /*...*/
  var host = new ServiceHost(typeof(TerminalService), Uri);
  /*...*/
  host.Open();
  return host;
 }
}
```

Code Fragment 9: Terminal service implementation

4.3.6 Terminal Client Implementation

The terminal client has been generated automatically, therefore no code is included. However, as seen on the class diagram, Visual Studio did create an interface TerminalService based on the service contract in the WSDL file and created and interface implementation with the class TerminalServiceClient. The class Statistics is also included and event implements to interfaces we do not explicitly need in this project.

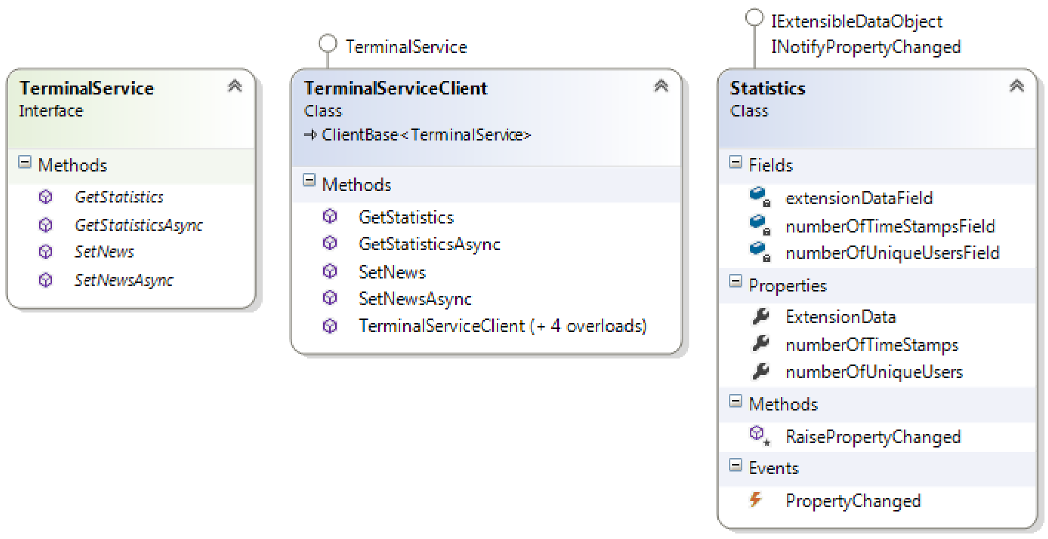


Figure 13: Terminal client class diagram

5 Discussion and Analysis

This project was a very interesting opportunity to become acquainted with SOAP and REST. As we mentioned before we had already heard about it but never had the chance to actually implement either a service or a client.
We were also given the opportunity to implement this application in C# using the .NET Framework, which we are very grateful for. Thomas already had experience in developing .NET applications, but Marc did not. Because will use C# in the second specialisation course (WBA2) it will surely be helpful to have already had worked with the .NET Framework.
To test our Web Services we used SoapUI, which is a very comfortable and useful tool.

6 Conclusions

From our point of view, this project was a success because we achieved the write a working system using both SOAP and RESTful Web Services and learned a lot about both technologies. We hope the theory we will present is both interesting and useful for the other students as well.
As to whether we prefer SOAP or RESTful Web Services, we cannot make up our minds. In our opinion, it always depends on the use case. SOAP Web Services are certainly better suited for complex business Web Services, whereas RESTful Web Services are great in a less formal, dynamic web environment.