

# NoSQL DBMS based on a new Data Model

---

## Ikarus DataBase Engine

### Content:

1	Introduction: .....	2
2	Operations:.....	3
2.1	JSON Objects:.....	4
2.1.1	store .....	4
2.1.2	get .....	4
2.1.3	delete .....	5
2.2	S-Collection Objects:.....	5
2.2.1	makecoll .....	5
2.2.2	deletecoll .....	6
2.2.3	insertcoll .....	6
2.2.4	removecoll .....	7
2.2.5	getcoll .....	7
2.3	Extra operations: .....	8
2.3.1	searchobj.....	8
2.3.2	reset .....	8
3	Ikarus DataBase Engine: .....	9
3.1	Rationale.....	9
3.2	Source Code.....	9
3.2.1	CoreWS.java.....	9
3.2.2	Main.java .....	10
3.3	Execution .....	11
4	Ikarus DataBase Engine Client: .....	12
4.1	Rationale.....	12
4.2	The Client.....	12
4.3	Example (From User input to WS invocation) .....	13
4.4	Web Service Reference.....	14
4.5	Additional example.....	15
5	Additional Information:.....	16
5.1	JSON.....	16
5.2	S-Collections .....	16

# 1 Introduction:

A DataBase Management System requires a defined Data Model. This Data Model consists of two major pieces. On the one hand Data Objects/Structures have to be classified, on the other hand a set of Operations have to be defined. This leads us to the following scheme.

Data Model:

*Data Model = <Data Objects, Operations>*

Data Objects:

*Data Object = <JSON file + unique ID>  
<SCollection + unique ID>*

Operations:

*operation name(parameter1, parameter2,) => return\_value*

The NoSQL DBMS will build up on the database engine, implemented as a web service, as its foundation. It is using persistent data objects encoded as JSON files. These data objects are identifiable by their unique ID and can be combined into S-Collections which themselves are also identifiable by their own unique CID.

For the web service the Java API for creating XML web services, JAX-WS will be used (available with Java EE 1.6+).

The Database Engine will support (at least) the following operations:

- Store, Modify and Delete data objects
- Create and Delete S-Collections.
- Insert and Remove members into/from S-collections.
- Search data objects and S-collections;
- Scan a list of data objects and get the JSON.

The engine is reachable under "http://coronet2.iicm.tugraz.at:8080/IkarusDBEngine/"

A test client is reachable under "http://coronet2.iicm.tugraz.at:8080/IkarusDBEngineClient/"

## 2 Operations:

There are three core web service operations regarding JSON objects: **store**, **get**, and **delete**. **store** creates a JSON object within the engine and automatically assigns a 6 digit ID to it. **get** retrieves a JSON object, referenced by a 6 digit ID, previously stored within the engine. **delete** erases a JSON object, referenced by a 6 digit ID, that is stored within the engine.

Five additional operations expand the engine's functionality by creating/manipulating/deleting sets of JSON objects, so called S-Collections(see 5.2):

**makecoll**, **deletecoll**, **insertcoll**, **removecoll** and **getcoll**.

**makecoll** creates a S-Collection by receiving a JSON object ID and a name for it. The object ID is automatically assigned as the head object of the recently created S-Collection.

**deletecoll** erases every information about the S-Collection, referenced by a 6 digit CID (with an additional pre-literal "s-") that is stored within the engine.

**insertcoll** adds a JSON object to an already existing S-Collection. The number of S-Collection members is limited by the maximum number of JSON objects existing at the same time (999.999).

**removecoll** eliminates a JSON object from a specific S-Collection. Note that the head object of a S-Collection cannot be removed, as S-Collections per definition require at least one member to exist.

**getcoll** behaves similar to **get**, but instead of returning the content of a JSON object it returns a String containing every member of an S-Collection (e.g. head\_id1,id2,id3).

All operations require at least one String as an input and also return a String as an output. The reason behind this is, to keep the engine lightweight, easy to use and consistent. The operations regarding S-Collections only manipulate references to JSON objects, not the objects themselves. As a result, the manipulation of S-Collections may never change, corrupt or erase any data stored within the engine.

Two additional operations, a hard reset of the engine and a search feature, also belong to the basic set of operations. **searchobj** implements the basic functionality of a search filter. By entering a search text, all available data was browsed, resulting in a list with all ID's containing the search text. **reset** resets the web service/engine to a point, that is according to the first start of the engine. This operation is designed to only be used by administrators and thus should never be presented to the user directly.

## 2.1 JSON Objects:

### 2.1.1 store

*store(String json\_content) => String json\_id*

The store operation takes a JSON file, parsed as a String, as an input and will return the unique ID of the JSON file to be stored within the DataBase. The ID's will be assigned automatically during a successful invocation of the store operation. ID's will always consist of a 6 digit number - ranging from 000001 - 999999 as the last valid object ID. A call of store with an empty String (= null) will fail.

e.g.:

```
store("{example content...}")    => 000001
store("")                       => null
```

### 2.1.2 get

*get(String json\_id) => String json\_content*

The get operation takes a unique ID as an input. The ID passed to this operation has to follow the requirement of a 6 digit number, corresponding to the return value of store. Meaning that, 000001 would be accepted, but neither 1 nor 001 would comply. The return value, in case of a call with a valid ID that is already stored within the DataBase, will return the JSON object parsed as a String. In case of an invalid call, either consisting of an invalid ID, or in case that nothing is stored within an maybe not yet existing object, referenced by the given ID, will fail.

e.g.:

```
get("000001")    => "{example content...}"
get("01")         => null // invalid ID
get("")          => null // empty ID
get("012345")    => null // nothing stored
```

### 2.1.3 delete

*delete(String json\_id) => String json\_id + " deleted"*

The delete operation erases all information a specific JSON object stored within the DataBase, referenced by the given ID. Despite that, delete behaves very similar to the get operation and returns either, in case of a successful call the specified ID followed by a "deleted" String literal, or will fail, in case of an invalid call, e.g. the JSON file is not stored within the DataBase.

e.g.:

<code>delete("000001")</code>	<code>=&gt; "000001 deleted"</code>
<code>delete("01")</code>	<code>=&gt; null // invalid ID</code>
<code>delete("")</code>	<code>=&gt; null // empty ID</code>
<code>delete("012345")</code>	<code>=&gt; null // nothing stored</code>

## 2.2 S-Collection Objects:

### 2.2.1 makecoll

*makecoll(String coll\_name, String head\_id)*  
*=> String coll\_id + "(" + String coll\_name + ")"*

The makecoll operation takes two parameters as an input: The first parameter is a freely choose able name, used to give a human readable identifier besides the CID(Collection ID). The second parameter is the ID of the JSON object to be marked as HEAD for the new S-Collection. The makecoll operation will return a unique CID, automatically assigned at runtime, for the newly created S-Collection. The collection ID will consist of a String literal "s-" plus a 6 digit number (same requirements as needed for the JSON object ID) followed by the specified name in brackets.

e.g.:

<code>makecoll("mycollection", "000001")</code>	<code>=&gt; "s-000001(mycollection)"</code>
<code>makecoll("", "000001")</code>	<code>=&gt; null // invalid name</code>
<code>makecoll("test", "015")</code>	<code>=&gt; null // invalid id</code>

### 2.2.2 deletecoll

*deletecoll(String coll\_id)*  
*=> String coll\_id + "(" + String coll\_name + ")" + " deleted"*

The deletecoll operation takes the unique CID as an input. The return value is the same from the makecoll operation with an additional " deleted" String literal appended. Its additional behaviour is identical to the delete(json\_id) operation.

e.g.:

```
deletecoll("s-000001")           => "s-000001(mycollection) deleted"
deletecoll("000001")             => null // invalid coll_id
```

### 2.2.3 insertcoll

*insertcoll(String coll\_id, String json\_id)*  
*=> String json\_id " successfully inserted into " String coll\_id + "(" + String coll\_name + ")"*

The insertcoll operation takes 2 parameters as an input: Firstly the CID of the S-Collection to be inserted into and secondly the ID of the JSON object to insert. It will either succeed, given that the ID's are correctly entered in addition to the S-Collection already existing. It will fail, if any of the above mentioned requirements aren't fulfilled.

e.g.:

```
insertcoll("s-000001", "000002")
=> "000002 successfully inserted into s-000001(mycollection)"
insertcoll("000001" "000002")    => null // invalid coll_id
insertcoll("s-000001", "002")    => null // invalid json_id
```

### 2.2.4 removecoll

*removecoll(String coll\_id, String json\_id)*

*=> String json\_id " successfully removed from " String coll\_id + "(" + String coll\_name + ")"*

The removecoll operation takes 2 parameters as an input: Firstly the ID of the S-Collection to be inserted into and secondly the ID of the JSON object to remove. It will either succeed, given that the ID's and the name are correctly entered in addition to the S-Collection already existing. It will fail, if any of the above mentioned requirements aren't fulfilled. Additionally, the removecoll operation will fail if the size of the S-Collection equals 1 (meaning that only the head object of the collection remains) - Existing S-Collections always require at least one element (= head), thus the head object can never be removed.

e.g.:

```
removecoll("s-000001", "000002")
=> "000002 successfully removed from s-000001(mycollection)"
removecoll("000001", "000002")           => null // invalid coll_id
removecoll("s-000001", "002")             => null // invalid json_id
```

### 2.2.5 getcoll

*getcoll(String coll\_id) => String head\_and\_members*

The getcoll operation takes the ID of the S-Collection to be searched as an input. The return value is a String starting with the head of the S-Collection, followed by the other members of the collections - the elements are separated by commas ','. It will fail, if the S-Collection doesn't exist or the CID is invalid.

e.g.:

```
getcoll("s-000001")           => "000001,000002,000004"
getcoll("000001")             => null // invalid coll_id
getcoll("s-000420")           => null // S-Coll. doesn't exist
```

## 2.3 Extra operations:

### 2.3.1 searchobj

*searchobj(String search\_text) => String all\_ids\_containing\_text*

The searchobj operation is a feature to browse the DataBase for a specific text or keyword. The user is able to enter a search text as a String, which can be of any size except null. As a result the searchobj operation returns a list with all the ID's containing the search text or keyword (same format as the getcoll operation). Note that it is also possible to enter an entire JSON object as a search text to get the specific ID of the JSON object.

e.g.:

searchobj("WhatDoesTheFoxSay")	=> "00001, 000031"
searchobj("ThisIsNotPresentInData")	=> null // no id contains the text
searchobj("")	=> null // empty passphrase

### 2.3.2 reset

*reset(String passphrase) => String success*

The reset operation is a feature to clear the DataBase without restarting the server completely. As a passphrase enter "IKnowWhatIamDoing" to clear all JSON objects and S-Collections stored within the DataBase as well as the automatically assigning ID counters.

e.g.:

reset("IKnowWhatIamDoing")	
=> " Database was successfully cleared!"	
reset("")	=> null // empty passphrase
reset("IDontKnowWhatIamDoing")	=> null // wrong passphrase



## 3 Ikarus DataBase Engine:

### 3.1 Rationale

The reason Ikarus was created was bachelor thesis embedded in an international Project (NoSQL DBMS based on a new Data Model) involving the University of Technology (Graz - Austria) as well as the Saint Petersburg University of Technology & Design.

As a consequence, Ikarus was implemented as a web service, using Java as a programming language, in order to make it as versatile and accessible as possible. JAX-WS, the Java API for creating XML based Web Services, was used as its core component. JAX-WS is available with Java EE 1.6+. The client and server communicate through SOAP messages, which makes it independent from protocols and transportation. JAXB, as a delegate of JAX-WS translates XML into Java conforming data objects. This makes using JAX-WS as convenient and lightweight as possible.

### 3.2 Source Code

Two main source files have to be mentioned as they form the core functionality of the engine, CoreWS.java and Main.java. Besides those two files, small Utility files (e.g. implementing S-Collections) replenish Ikarus.

#### 3.2.1 CoreWS.java

To start off with the basics, for every web service operation, a web method was created. These web methods/operations are combined in a single java file called CoreWS.java:

Method	Parameters	Parameter Name	Output	Faults	Parameter Type	Description
store		content			java.lang.String	
get		id			java.lang.String	
delete		id			java.lang.String	
makecoll		name id			java.lang.String java.lang.String	
deletecoll		sid			java.lang.String	

Figure 1)

Parameters	Output	Faults	Description
Parameter Name sid			Parameter Type java.lang.String
id			java.lang.String

Parameters	Output	Faults	Description
Parameter Name sid			Parameter Type java.lang.String
id			java.lang.String

Parameters	Output	Faults	Description
Parameter Name sid			Parameter Type java.lang.String

Parameters	Output	Faults	Description
Parameter Name doom			Parameter Type java.lang.String

Parameters	Output	Faults	Description
Parameter Name text			Parameter Type java.lang.String

Figure 2)

Figure 1) and 2) show a graphical illustration of the 10 web service operations with names and input parameters.

To give an example of what a web method looks like as code:

```
/**
 * Web service operation
 */
@WebMethod(operationName = "store")
public String store(@WebParam(name = "content") String content)
{
    if(content == null)
    {
        return "content NULL";
    }

    String new_id = Main.getInstance().getNewJsonId();
    Main.getInstance().addNewJson(new_id, content);

    // server log:
    System.out.println("STORE: " + new_id + " " + content);

    return new_id;
}
```

Besides checking the input parameters and returning the output string nothing special happens in the web service operation class directly. Its main functionality is outsourced to another file, **Main.java**.

### 3.2.2 Main.java

The **Main.java** class forms the heart piece of Ikarus as it acts as a manager behind the scenes. Its main attribute is defined in being a static instance of class, granting singleton like behaviour - simply put, there is exactly one instance of the main class that is omnipresent, visible to all operations and cannot be destroyed.

As mentioned above, the Main class implements the web service operations in detail. Besides taking care of storing, manipulating and freeing data by handling data structures, Main.java also ensures accurate behaviour of the web methods.

For storing available data, three maps are used:

```
// id (e.g. "000001") - content (e.g. "{yljasldamy.xc}")
public static Map<String, String> json_map = new HashMap<String, String>();
// id (e.g. "S-000001") - name (e.g. "examplecollection")
public static Map<String, String> coll_name_map = new HashMap<String, String>();
// id (e.g. "S-000001") - s-coll-objects (e.g. "examplecollection")
public static Map<String, SCollection> s_coll_map = new HashMap<String, SCollection>();
```

To conclude, the Main class is the most important component of Ikarus, not only does it ensure uniqueness of the engine and omnipresent behaviour through its instance-like implementation, but also avoid race conditions by having a static attribute, realizing a task-queueish purpose.

### 3.3 Execution

Ikarus is currently online at:

<http://coronet2.iicm.tugraz.at:8080/IkarusDBEngine/>

In order to invoke Ikarus' web service operations, SOAP messages have to be sent to the running engine. As this is not a trivial task, Ikarus also includes a basic test client extension that was created to illustrate Ikarus' correct behaviour.

## 4 Ikarus DataBase Engine Client:

### 4.1 Rationale

As explained at 3.3, in order to test Ikarus' behaviour and correct execution, SOAP messages have to be sent to it. The easiest way to achieve this is by creating a web application, consisting of html and jsp files combined with a reference to Ikarus' wsdl file. Thus, the test client is build out of these three components. The client is currently online at:

<http://coronet2.icm.tugraz.at:8080/IkarusDBEngine/>

### 4.2 The Client

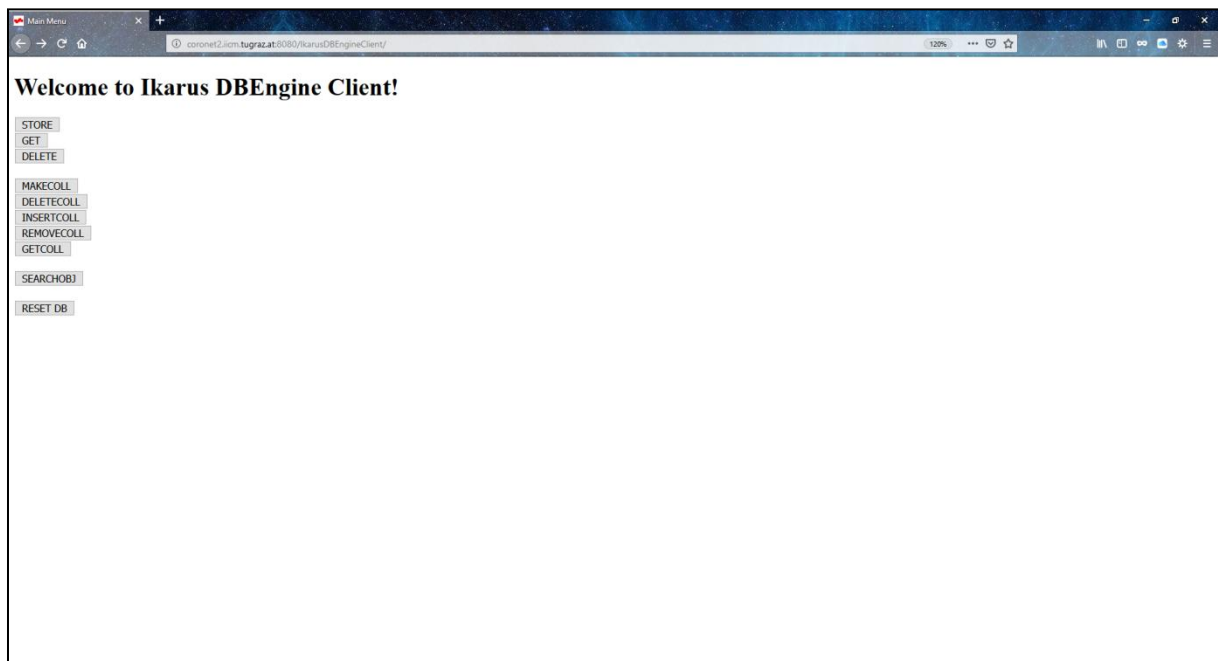


Figure 3)

Figure 3 illustrates the client's main menu. The index page contains buttons with links to all the web service operation available on Ikarus. Each operation has its own html file, enabling the user to specify required test inputs as well as a jsp file that is linked to it. The html files use <form>-attributes to send a post request to the jsp files, which then parse the input into Java conforming variables, open a port to Ikarus and open communication with Ikarus.

### 4.3 Example (From User input to WS invocation)

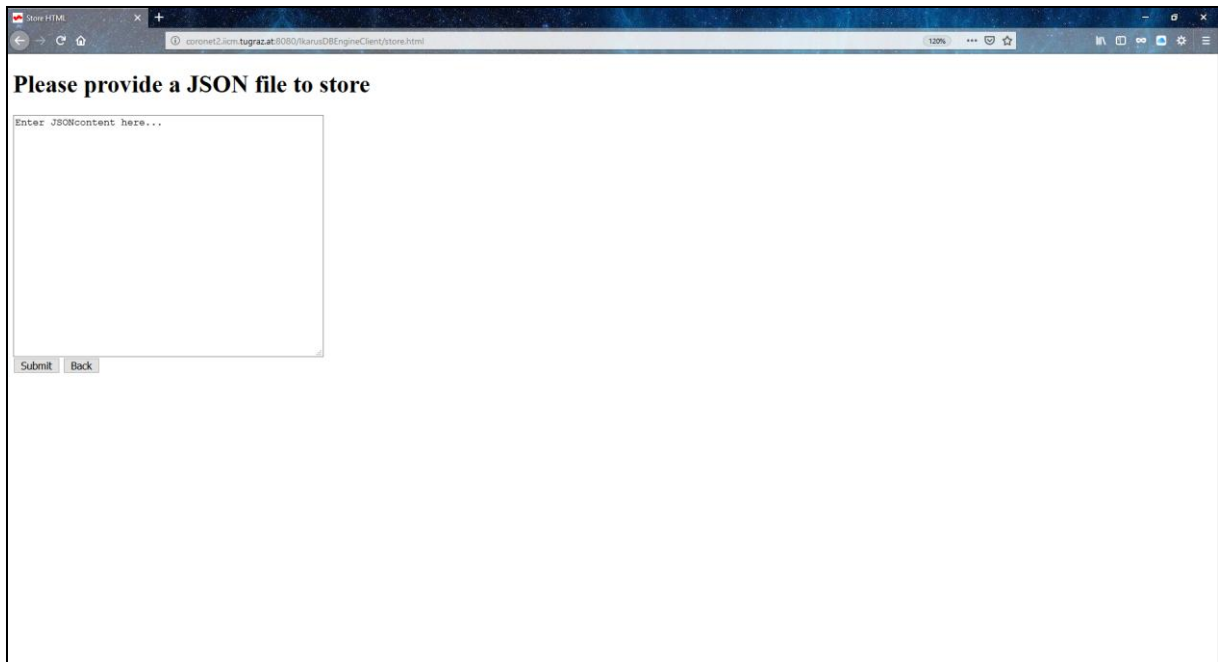


Figure 4)

Figure 4) shows the store operation, illustrated as store.html as an example. To make it even easier users can simply copy the content of a JSON file into the text field and press Submit. By pressing Submit, the content entered will be sent (via POST) to the store.jsp file.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Delete JSP</title>
  </head>
  <body>

    <!-- start web service invocation --%><hr/>
    <%
    try {
        String json_id = request.getParameter("json_id");
        org.me.ikarus.CoreWS_Service service = new org.me.ikarus.CoreWS_Service();
        org.me.ikarus.CoreWS port = service.getCoreWSPort();
        String id = json_id;
        java.lang.String result = port.delete(id);
        out.println("Result: "+result);
    }
    %>
    <!-- end web service invocation --%><hr/>

    <input type="button" onclick="location.href='index.html';" value="Main Menu" />
    <button onclick="goBack()">Back</button>
    <script>
        function goBack()
        {
            window.history.back();
        }
    </script>

  </body>
</html>
```

This is what the store.jsp source code looks like.

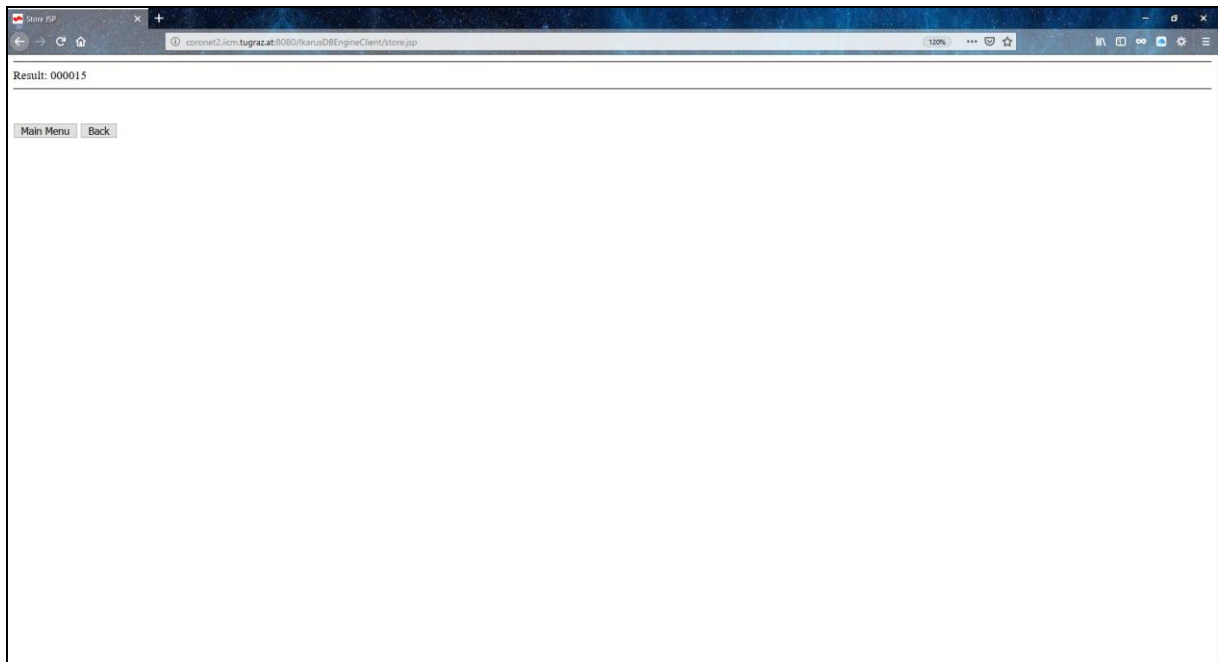


Figure 5)

Figure 5) illustrates the graphical output of the jsp file. The JSON content submitted through the store.html page is now stored on the server, and is accessible with the ID 000015. A call of get(000015) would result with the output of exactly this content.

## 4.4 Web Service Reference

The web service reference CoreWS.wsdl is the key component for the connection between Ikarus and the client. It is imported through the wsdl file stored within the engine.

Service Name: <a href="http://ikarus.me.org/">http://ikarus.me.org/</a> CoreWS	Address: <a href="http://coronet2.icm.tugraz.at:8080/IkarusDBEngine/CoreWS">http://coronet2.icm.tugraz.at:8080/IkarusDBEngine/CoreWS</a>
Port Name: <a href="http://ikarus.me.org/">http://ikarus.me.org/</a> CoreWSPort	WSDL: <a href="http://coronet2.icm.tugraz.at:8080/IkarusDBEngine/CoreWS?wsdl">http://coronet2.icm.tugraz.at:8080/IkarusDBEngine/CoreWS?wsdl</a>
	Implementation class: org.me.ikarus.CoreWS

Figure 6)

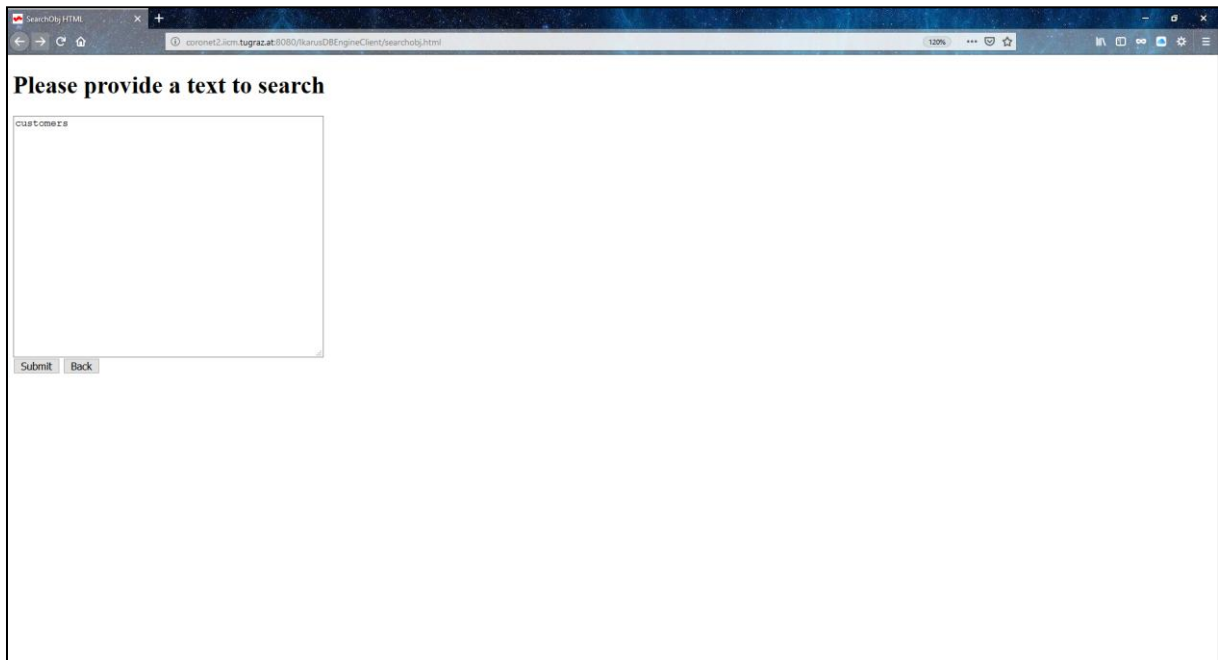
Figure 6) shows the location (found at: <http://coronet2.icm.tugraz.at:8080/IkarusDBEngine/CoreWS>) of the wsdl file to be imported.

This is the import statement within the client's reference wsdl file:

```
<xsd:schema>
<xsd:import namespace="http://ikarus.me.org/" schemaLocation="http://coronet2.icm.tugraz.at:8080/IkarusDBEngine/CoreWS?xsd=1" />
</xsd:schema>
```

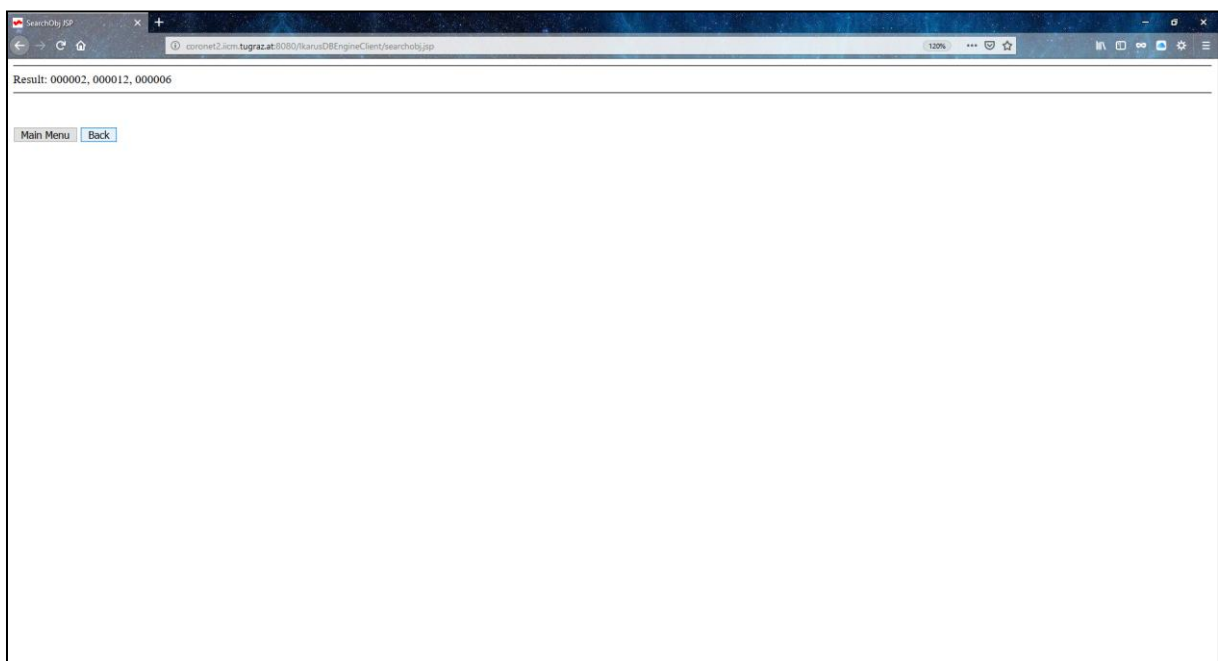
These two pieces, put together, enable the dispatch of SOAP messages needed in order to communicate with Ikarus.

## 4.5 Additional example



A screenshot of a web browser window titled "SearchObj HTML". The address bar shows the URL "http://localhost:8080/ikarusDBEngineClient/searchobj.html". The page content includes the heading "Please provide a text to search" in bold. Below the heading is a large, empty text input field. At the bottom left of the input field, the placeholder text "Object name" is visible. Below the input field are two buttons: "Submit" and "Back".

Figure 7) Search Object operation input



A screenshot of a web browser window titled "SearchObj RP". The address bar shows the URL "http://localhost:8080/ikarusDBEngineClient/searchobj.jsp". The page content displays the search result: "Result: 000002, 000012, 000006". Below the result text are two buttons: "Main Menu" and "Back".

Figure 8) Search Object operation output

## 5 Additional Information:

### 5.1 JSON

JavaScript Object Notation (JSON) is a compact, easy to read data format with the purpose of data transfer between Applications. Every JSON file itself is required to validate as a valid JavaScript file. As Ikarus is designed to operate on JSON objects (JSON object = ID + content), a sample JSON file (= content) is illustrated here:

```
{
  "glossary": {
    "title": "example glossary",
    "GlossDiv": {
      "title": "S",
      "GlossList": {
        "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
          "Acronym": "SGML",
          "Abbrev": "ISO 8879:1986",
          "GlossDef": {
            "para": "A meta-markup language, used to create markup languages such as DocBook.",
            "GlossSeeAlso": ["GML", "XML"]
          },
          "GlossSee": "markup"
        }
      }
    }
  }
}
```

*example1.json*

### 5.2 S-Collections

S-Collections are a collection of JSON objects stored within Ikarus' database. As a requirement S-Collections consist of at least one object - this object is also called the HEAD object of the collection. The number of possible members of a S-Collection is limited by the total possible number of JSON object stored on Ikarus. S-Collections can be created by defining a head object, inserted into and removed from by providing the new member's ID and deleted by pointing to the SID of the S-Collection. A theoretical example a S-Collection would look like this:

S-Collection: {[HEAD], [member1], [member2], [member3]}

s-000001: {[000001], [000015], [000315], [000007]}