Contact Database Assignment

CSCU9YW

2615649

*March 2021*

Computing Science and Mathematics

University of Stirling

Contents

[Contact Database Assignment i](#_Toc67316573)

[CSCU9YW i](#_Toc67316574)

[2615649 i](#_Toc67316575)

[**1** **The problem** 2](#_Toc67316576)

[**1.1** **Description** 2](#_Toc67316577)

[**1.2** **Assumptions** 2](#_Toc67316578)

[**2** **Solution** 2](#_Toc67316579)

[**2.1** **Implementation overview** 2](#_Toc67316580)

[**2.2** **Storage structure** 2](#_Toc67316581)

[**2.3** **Basic features** 3](#_Toc67316582)

[**2.3.1** **GUI** 3](#_Toc67316583)

[**2.3.2** **Search/retrieve** 3](#_Toc67316584)

[**2.3.3** **Add** 4](#_Toc67316585)

[**2.3.4** **Edit** 4](#_Toc67316586)

[**2.3.5** **Delete** 5](#_Toc67316587)

[**2.4** **Advanced features** 5](#_Toc67316588)

[**2.4.1** **Robust handling of error cases** 5](#_Toc67316589)

[**3** **Evaluation** 7](#_Toc67316590)

[**3.1** **Completeness and areas for improvement** 7](#_Toc67316591)

[**4** **Code listings** 7](#_Toc67316592)

[**4.1** **ContactService.java** 7](#_Toc67316593)

[**4.2** **ContactClient.java** 13](#_Toc67316594)

Figures

[Figure 1 XML storage structure 3](#_Toc67316642)

[Figure 2. Client graphical user interface 3](#_Toc67316643)

[Figure 3. Search output in GUI 4](#_Toc67316644)

[Figure 4. Wireshark request and response for retrieve non-existent contact. 5](#_Toc67316645)

[Figure 5. Retrieve non-existent contact warning. 6](#_Toc67316646)

[Figure 6. Wireshark request and response for delete non-existent contact. 6](#_Toc67316647)

[Figure 7. Delete non-existent contact warning. 6](#_Toc67316648)

[Figure 8. Wireshark request and response for add duplicate contact. 6](#_Toc67316649)

[Figure 9. Add duplicate contact warning. 6](#_Toc67316650)

[Figure 10.Wireshark request and response for editing non-existent contact. 6](#_Toc67316651)

[Figure 11. Edit non-existent contact warning. 7](#_Toc67316652)

# **1 The problem**

## **Description**

This project calls for a web service that will allow users to retrieve, add, edit, and delete contacts in a database structure such as a HashMap. The service will be implemented using REST and will be controlled using a simple GUI through the client. A record will consist of a contacts first name, last name, address including street, town and postcode, and phone number serving as the unique ID.

The web service should implement advanced features such as robust handling of potential error cases such as adding duplicate contacts and deleting non-existent contacts, as well as the ability to interact with groups of contacts like all contacts living in a specific town.

## **Assumptions**

One assumption that was made about the project was that a user cannot add a partial entry, they must fill out all fields when adding or updating a contact. Another was that phone number can only include digits 0-9 and other fields can contain characters.

# **Solution**

## **Implementation overview**

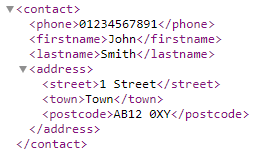
We begin by running the Contacts service which acts as a manager for the list of contacts. We then run the client which displays a simple Graphical User Interface (GUI). The GUI allows the user to interact with the list of contacts where they can search for, add, edit, and delete contacts. The implementation of these features will be covered in further detail in the following sections.

## **Storage structure**

As suggested in the brief, the storing of contact details was implemented using a HashMap. The phone number acts as the unique ID for a contact, and we can use it to reference a record in the HashMap. This unique phone number is linked to a string array storing the contact details of the person with that phone number.

1. HashMap<String,String[]> contactsMap = new HashMap<String,String[]> ();

An example of a contact record can be seen in Figure 1. Each contact has a node storing their phone number, first name, last name and address including street, town, and postcode.



There can be many contact nodes each with their own set of child nodes storing their details. The structure of these nodes are as follows:

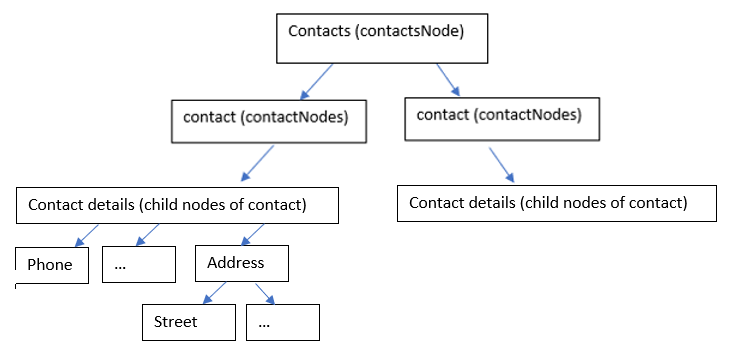


Figure 1 XML storage structure

## **Basic features**

## **GUI**

In the brief, one of the requirements was to create a basic UI for the client. This feature was implemented using a java swing layout. JLabels to label the JTextFields used to enter a persons contact details, a series of JButtons used to perform the actions required and lastly the TextArea to display the results of the actions. If there is many contacts added and they go off the window, the user can scroll through them.

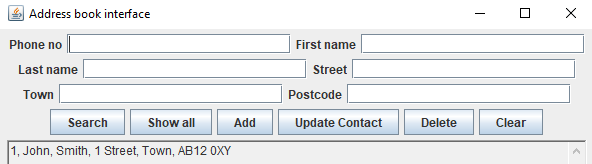


Figure 2. Client graphical user interface

## **Search/retrieve**

The search feature allows the user to search for a contact using their phone number. The user must enter a phone number and click the search button seen in Figure 2. This will display the contact details of the person with that phone number in the text area, if it exists.

This was implemented using an action listener which responds to the ‘Search’ button being pressed. If there is a number in the text field, the action listener calls the invoke method, passing the ‘GET’ method and the phone number from the phone text field as parameters (code snippet below). If the field is empty, they will be prompted to enter a number and try again.

1. client.invoke("GET", phonetf.getText());

As a result, an if statement will be triggered inside the invoked method which will communicate with the service to retrieve the contact details associated with that phone number, format them in a string and display them in the GUI text area.

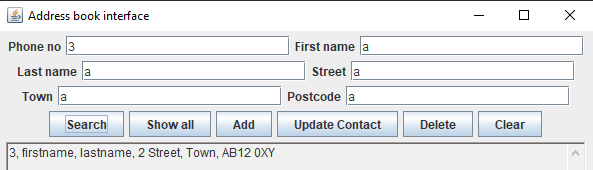


Figure 3. Search output in GUI

Another variation of the search feature is the ‘Show all’ button. This works similarly to the ‘Search’ feature but displays all contacts in the text area. Instead of using setText(output), append(output) is used to put each contact on its own line.

## **Add**

The add feature lets the user add a contact to the database. The user must fill out all text fields and click the add button for the new contact to be added. If there is a blank field, they will be prompted to fill in all fields before the record is added, then text fields are cleared, and a confirmation is given letting the user know the contact was added successfully.

In the action listener for the ‘Add’ button is an if statement checking that all fields are complete, then the invoke method is called with ‘POST’ and all text fields as parameters. ‘POST’ can be thought of as ‘create’ as it is the method responsible for creating a new contact. In the invoke method, the ‘POST’ if statement will be triggered which will create a string, formatted with XML tags for each contact detail (code snippet below).

1. contacts = "<contacts><contact>"
2. + "<phone>" + num + "</phone>"
3. + "<firstname>" + first + "</firstname>"
4. + "<lastname>" + last + "</lastname>"
5. + "<address>"
6. + "<street>" + street + "</street>"
7. + "<town>" + town + "</town>"
8. + "<postcode>" + post + "</postcode>"
9. + "</address>"
10. + "</contact></contacts>"

This will signal the service class to create a contact in the defined structure using the given parameters and it will be entered into the database. In the service the contact details are store in the contact array and input into the HashMap ‘contactsMap’.

1. contactsMap.put(phone, contact);

## **Edit**

The edit/update contact feature allows the user to edit the information associated with a given phone number. They can type in an existing phone number and fill in new details for a contact, then using the ‘Update contact’ button, the contact details relating to that phone number will be updated to the new values.

This is done similarly to the add feature, checking all data fields are complete, the invoke method is called by the ‘Update contact’ action listener, with the ‘PUT’ method and all text field values as parameters. ‘PUT’ works almost identically to ‘POST’ but instead of ensuring the record does not exist, this method checks that it does. This then trigger the put if statement signalling to the service to update the contact details of the contact with the given phone number.

1. contactsMap.put(phone, contact);

## **Delete**

Delete is a simple feature which removes a contact from the database. When the ‘Delete’ button is clicked, it takes in phone number from the text field, and calls the invoke method with the parameters ‘DELETE’ and the phone number. This triggers the delete if statement which will send a signal to the service to remove that contact from the database using the method remove.

1. contactsMap.remove(phone);

## **Advanced features**

## **Robust handling of error cases**

The program can handle error cases such as the adding of a contact that already exists, searching for, editing or deleting a contact that does not exist. This is done but checking the http response codes which as defined as follows:

|  |  |  |
| --- | --- | --- |
| http response code | meaning | Trigger |
| 200 | OK | GET/PUT/DELETE successful |
| 201 | Created | POST successful |
| 404 | Not Found | GET/PUT failed, contact doesn’t exist |
| 405 | Method Not Allowed | PUT failed, contact does not exist, POST failed, contact already exists |

**Search:**

An example of robust error handling can be seen below for the search feature. This works by checking the method used, in this case ‘GET’, with an if statement. Then within this if statement, it checks the http response code, setting the text area to ‘Not found’ for response code 404/not found. Otherwise, the retrieval was successful, and the contact details will be displayed in the text area.

1. if (method.equals("GET")) {
2. if (httpResponse2 == 404) {
3. textArea.setText("Not found.");
4. }
5. }



Figure 4. Wireshark request and response for retrieve non-existent contact.

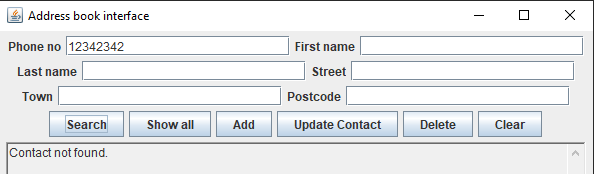


Figure 5. Retrieve non-existent contact warning.

The validation for add, edit, and delete works in the same way by checking different response codes and outputting appropriate warning messages.

**Delete:**



Figure 6. Wireshark request and response for delete non-existent contact.

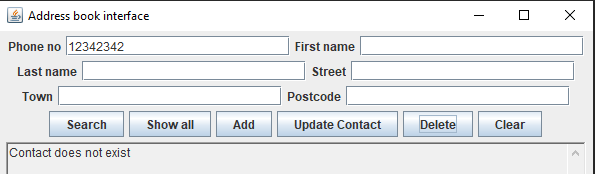


Figure 7. Delete non-existent contact warning.

**Add:**



Figure 8. Wireshark request and response for add duplicate contact.

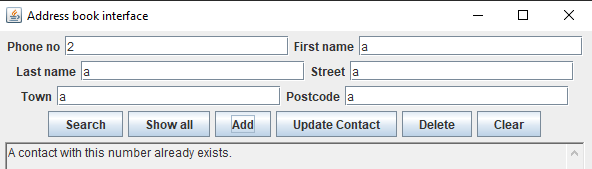


Figure 9. Add duplicate contact warning.

**Edit:**



Figure 10.Wireshark request and response for editing non-existent contact.

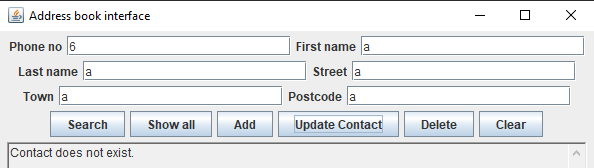


Figure 11. Edit non-existent contact warning.

Lastly, it would have been possible to validate the GUI text fields further e.g., using regex to control the format of the postcode and phone number etc. but considering not all countries use the same formats, the decision was made to not implement this.

# **Evaluation**

## **Completeness and areas for improvement**

The solution is mostly complete with only a few areas where I would consider to be lacking. The first being the inability to interact with a group of contacts. This would have been a useful feature I would liked to have included having had more time. The thoughts I had for implementing this was to get the common term from the text field and search through the details in the hash map and return any matching records. A similar feature to this could be a partial search where the user types in part of a name or number and it returns any contacts with the matching segment. Also, although not required from the brief, I would have liked to have create a more professional GUI to aid the user experience.

I chose to implement the web service using REST instead of SOAP as REST is considered easier to work with and faster. Given more time I would have liked to try implementing the web service using Spring as it seems many employers consider that an important skill to have.

It would have also been possible to implement the contact storage using an external database, but having worked a fair amount with external databases before, I chose to use a HashMap as this is a programming construct I was rather unfamiliar with until this point. As a result, I now feel more confident using HashMaps.

All implemented features are fully functional.

# **Code listings**

|  |  |
| --- | --- |
| 5.1 ContactService.java | RESTful web service to manage a list of contacts. |
| 5.2 ContactClient.java | RESTful web client to manage a list of contacts. |

## **ContactService.java**

1. import java.io.StringReader;
3. import javax.annotation.Resource;
5. import java.util.HashMap;
6. import java.util.Iterator;
8. import javax.xml.bind.JAXBContext;
9. import javax.xml.transform.\*;
10. import javax.xml.transform.dom.DOMResult;
11. import javax.xml.transform.stream.StreamSource;
13. import javax.xml.ws.\*;
14. import javax.xml.ws.handler.MessageContext;
15. import javax.xml.ws.http.HTTPBinding;
16. import javax.xml.ws.Service.Mode;
18. import org.w3c.dom.\*;
20. /\*\*
21. RESTful web service to manage a list of contacts.
22. \*/
24. @WebServiceProvider
25. @ServiceMode(Mode.MESSAGE)
26. public class ContactService implements Provider<Source>{
28. HashMap<String,String[]> contactsMap = new HashMap<String,String[]>(); // HashMap storing list of contacts mapping unique phone number to an array of contact details
29. private JAXBContext jaxbContext;
30. MessageContext requestContext;
31. Transformer messageTransformer;
33. @Resource(type=Object.class)
34. protected WebServiceContext serviceContext;
36. private static final String serviceURI = "http://localhost:8182/contacts/"; // initialise service uri
38. /\*\*
39. \* ContactService constructor
40. \* creates new
41. \*/
42. public ContactService() throws Exception {
43. jaxbContext = JAXBContext.newInstance(ContactService.class); // create new instance of the
44. messageTransformer = TransformerFactory.newInstance().newTransformer(); //initialise transformer to transform a source tree into a results tree
45. }
47. /\*\*
48. Return XML text for currently defined contacts in the form
49. "<contacts>
50. <contact>
51. <phone>phone num</phone>
52. <firstname>first</firstname>
53. <lastname>last</lastname>
54. <address>
55. <street>street</street>
56. <town>town</town>
57. <postcode>post</postcode>
58. </address>
59. </contact>
60. </contacts>",
62. @param phone - phone number of a contact
63. @return XML text for defined contacts
64. \*/
65. private String getContacts(String phone) {
67. String contacts; // declare string to store contact details
68. if (phone != null) { // if value given for phone
69. String[] contact = contactsMap.get(phone); // string array storing contact details retrieved from the hashmap
70. if (contact != null) { // if there are contact details in the contact array
71. String num = contact[0]; // assign individual contact details from array to variables
72. String first = contact[1];
73. String last = contact[2];
74. String street = contact[3];
75. String town = contact[4];
76. String post = contact[5];
78. contacts = "<contacts><contact>"
79. + "<phone>" + num + "</phone>"
80. + "<firstname>" + first + "</firstname>"
81. + "<lastname>" + last + "</lastname>"
82. + "<address>"
83. + "<street>" + street + "</street>"
84. + "<town>" + town + "</town>"
85. + "<postcode>" + post + "</postcode>"
86. + "</address>"
87. + "</contact></contacts>"; // format output string with the details of the contact surrounded int he appropriate xml tags
88. }else
89. contacts = null; // if there are no contacts details in the array set output string to null
91. } else { // if there is no phone number given in parameters, output all contacts using while loop
92. contacts = "<contacts>"; // root contacts xml tag
93. Iterator<String> contactIterator = contactsMap.keySet().iterator(); // set up hash map iterator
94. while (contactIterator.hasNext()) { // while iterator has another contact
95. phone = (String) contactIterator.next(); // move iterator to next
97. String[] contact = contactsMap.get(phone); // get contact details associated with given phone number
98. String num = contact[0]; // assign contact details to individual variables
99. String first = contact[1];
100. String last = contact[2];
101. String street = contact[3];
102. String town = contact[4];
103. String post = contact[5];
105. contacts = contacts + "<contact>" // add contact details to output string with xml tags
106. + "<phone>" + num.toString() + "</phone>"
107. + "<firstname>" + first + "</firstname>"
108. + "<lastname>" + last + "</lastname>"
109. + "<address>"
110. + "<street>" + street + "</street>"
111. + "<town>" + town + "</town>"
112. + "<postcode>" + post + "</postcode>"
113. + "</address>"
114. + "</contact>";
115. }
116. contacts += "</contacts>"; // contacts xml closing root tag
117. }
118. //System.out.println(contacts);
119. return (contacts); // return output string storing results of get formatted in xml
120. }
122. /\*\*
123. Respond to service invocation.
124. @param source service data source
125. \*/
126. public Source invoke(Source source) {
128. String contacts = "<contacts/>";
129. int httpResponse = 200; // assume success
130. MessageContext requestContext = serviceContext.getMessageContext(); // variable storing request message context
131. String method = (String) requestContext.get(MessageContext.HTTP\_REQUEST\_METHOD); // variable storing the method e.g. DELETE, GET etc
132. String phone = (String) requestContext.get(MessageContext.PATH\_INFO); // variable storing the phone number used for action
134. // analyse service call
135. try {
136. if (method.equals("DELETE")) { // delete/remove contact from the hash map of the given phone number
137. String[] deleted = contactsMap.remove(phone);
138. if (deleted == null) // if contact doesnt exist, return http response code 404 not found
139. httpResponse = 404;
140. } else if (method.equals("GET")) { // get contact from hash map for given phone number
141. String contactList = getContacts(phone);
142. if (contactList != null) // if contacts exists, set contacts to the retrieved contact details
143. contacts = contactList;
144. else
145. httpResponse = 404; // contact doesnt exist, return 404 not found http response
146. } else if (method.equals("POST")) { // create new contact
147. if (!contactsMap.containsKey(phone)) { // if phone number doesnt exist in hash map
148. DOMResult domXML = new DOMResult(); //initialise new DOMResult object to store transformed tree
149. messageTransformer.transform(source, domXML);
150. Element contactsElement = (Element) domXML.getNode().getFirstChild(); // root node all contacts <contacts>
151. Element contactElement = (Element) contactsElement.getFirstChild(); // a contact node <contact>
152. NodeList contactChildList = contactElement.getChildNodes(); // contact detail node list e.g. <firstname>
154. String[] contact = new String[6]; // declare string array to store contact details
155. contact[0] = contactChildList.item(0).getFirstChild().getNodeValue(); // set position 0 of contact detail array to phone number, retrieved from the node list storing contact details
156. contact[1] = contactChildList.item(1).getFirstChild().getNodeValue(); // same process repeated for the rest of entries
157. contact[2] = contactChildList.item(2).getFirstChild().getNodeValue();
158. NodeList addressChildList = contactChildList.item(3).getChildNodes();
159. contact[3] = addressChildList.item(0).getFirstChild().getNodeValue();
160. contact[4] = addressChildList.item(1).getFirstChild().getNodeValue();
161. contact[5] = addressChildList.item(2).getFirstChild().getNodeValue();


165. System.out.println("contact added: " + contact[0] + ", " + contact[1] + " " + contact[2] + ", " + contact[3] + ", " + contact[4] + ", " + contact[5]);
166. contactsMap.put(phone, contact); // enter contact into hash map
168. httpResponse = 201; // set http response code to created
169. } else {
170. httpResponse = 405; // set http response to method not allowed as contact already exist
171. }
172. } else if (method.equals("PUT")) { // same process as post but with subtle difference
173. if (contactsMap.containsKey(phone)) { // if the hash map DOES contain the contact then update to given details
174. DOMResult domXML = new DOMResult();
175. messageTransformer.transform(source, domXML);
176. Element contactsElement = (Element) domXML.getNode().getFirstChild();
177. Element contactElement = (Element) contactsElement.getFirstChild();
178. NodeList contactChildList = contactElement.getChildNodes();
180. String[] contact = new String[6];
181. contact[0] = contactChildList.item(0).getFirstChild().getNodeValue(); // set position 0 of contact detail array to phone number, retrieved from the node list storing contact details
182. contact[1] = contactChildList.item(1).getFirstChild().getNodeValue(); // same process repeated for the rest of entries
183. contact[2] = contactChildList.item(2).getFirstChild().getNodeValue();
184. NodeList addressChildList = contactChildList.item(3).getChildNodes();
185. contact[3] = addressChildList.item(0).getFirstChild().getNodeValue();
186. contact[4] = addressChildList.item(1).getFirstChild().getNodeValue();
187. contact[5] = addressChildList.item(2).getFirstChild().getNodeValue();

190. System.out.println("contact updated: " + contact[0] + ", " + contact[1] + " " + contact[2] + ", " + contact[3] + ", " + contact[4] + ", " + contact[5]);
191. contactsMap.put(phone, contact);
192. httpResponse = 200; // set http response OK
193. } else {
194. httpResponse = 404; // set http response to not found, contact not in hash map
195. }
196. } else
197. throw new WebServiceException("unsupported method: " + method); // unsupported method given
198. } catch (Exception exception) {
199. System.err.println("service invocation exception: " + exception);
200. }
201. requestContext.put(MessageContext.HTTP\_RESPONSE\_CODE, httpResponse);
202. Source responseSource = new StreamSource(new StringReader(contacts));
203. return (responseSource);
204. }
206. /\*\*
207. Main method to initialise service.
208. \*/
209. public static void main(String arguments[]) throws Exception {
210. ContactService service = new ContactService(); // create new contact service object
211. System.out.println("Starting service ...");
212. Endpoint endpoint = Endpoint.create(HTTPBinding.HTTP\_BINDING, service);
213. endpoint.publish(serviceURI);
214. }
216. }

## **ContactClient.java**

1. import java.awt.BorderLayout;
2. import java.awt.Component;
3. import java.awt.TextArea;
4. import java.awt.event.ActionEvent;
5. import java.awt.event.ActionListener;
6. import java.io.StringReader;
7. import java.util.Map;
9. import javax.swing.JButton;
10. import javax.swing.JFrame;
11. import javax.swing.JLabel;
12. import javax.swing.JOptionPane;
13. import javax.swing.JPanel;
14. import javax.swing.JTextArea;
15. import javax.swing.JTextField;
16. import javax.xml.namespace.QName;
17. import javax.xml.transform.\*;
18. import javax.xml.transform.dom.DOMResult;
19. import javax.xml.transform.stream.StreamSource;
21. import javax.xml.ws.\*;
22. import javax.xml.ws.handler.MessageContext;
23. import javax.xml.ws.http.HTTPBinding;
25. import org.w3c.dom.\*;
27. /\*\*
28. \* RESTful web client to manage a list of courses.
29. \*/
30. public class ContactClient {
32. private static final String SERVICE\_PATH = "/contacts/"; // initialise service path to match service
33. private static final QName SERVICE\_QNAME = new QName("urn:Contacts", ""); // set qualified name from xml specification
34. private static final String SERVICE\_URI = "http://localhost:8182" + SERVICE\_PATH; // initialise service uri to match service
36. static TextArea textArea = new TextArea(49,80); // set up text area to display results
38. static JTextField phonetf = new JTextField(20); // set up text fields
39. static JTextField firsttf = new JTextField(20);
40. static JTextField lasttf = new JTextField(20);
41. static JTextField streettf = new JTextField(20);
42. static JTextField towntf = new JTextField(20);
43. static JTextField posttf = new JTextField(20);
45. static JFrame frame = new JFrame("Address book interface"); // initialise new window to house the gui
47. // Creating the panel at bottom and adding components
48. static JPanel panel = new JPanel(); //initialise new j panel to house the gui components
50. static JLabel phone = new JLabel("Phone no"); // set up labels for text fields
51. static JLabel firstName = new JLabel("First name");
52. static JLabel lastName = new JLabel("Last name");
53. static JLabel street = new JLabel("Street");
54. static JLabel town = new JLabel("Town");
55. static JLabel post = new JLabel("Postcode");
57. static JButton search = new JButton("Search"); // set up buttons for actions
58. static JButton clear = new JButton("Clear");
59. static JButton showAll = new JButton("Show all");
60. static JButton add = new JButton("Add");
61. static JButton update = new JButton("Update Contact");
62. static JButton delete = new JButton("Delete");
64. Transformer messageTransformer;
65. private static Dispatch<Source> serviceDispatcher;
67. public ContactClient() throws Exception {
68. Service service = Service.create(SERVICE\_QNAME);
69. service.addPort(SERVICE\_QNAME, HTTPBinding.HTTP\_BINDING, SERVICE\_URI);
70. serviceDispatcher = service.createDispatch(SERVICE\_QNAME, Source.class, Service.Mode.MESSAGE);
71. messageTransformer = TransformerFactory.newInstance().newTransformer();
72. }
74. /\*\*
75. \* Call course service with given method for contact phone, first, last name, address including street, town, post code
76. \*
77. \* @param method HTTP method ("DELETE"/"GET"/"POST"/"PUT")
78. \* @param arguments contact phone number, first, last name, street, town, post code with parameters used as follows:
79. \* "DELETE",<phone> (delete contact)
80. \* "GET" (list all contacts)
81. \* "GET",<phone> (one contact)
82. \* "POST",<phone>,<firstname>,<lastname>,<street>,<town>,<postcode> (create contact)
83. \* "PUT",<phone>,<firstname>,<lastname>,<street>,<town>,<postcode> (update contact)
84. \*/
85. private void invoke(String method, String... arguments) {
86. try {
87. String phone = ""; // initialise empty string variables for details
88. String first = "";
89. String last = "";
90. String street = "";
91. String town = "";
92. String post = "";
94. switch (arguments.length) { // switch to assign method arguments to the variables
95. case 6:
96. post = arguments[5];
97. case 5:
98. town = arguments[4];
99. case 4:
100. street = arguments[3];
101. case 3:
102. last = arguments[2];
103. case 2:
104. first = arguments[1];
105. case 1:
106. phone = arguments[0];
107. }
109. String contacts = "<contacts/>"; // contacts closing tag
111. if (method.equals("DELETE")) { // if method is DELETE print contact deleted with phone number
112. System.out.println("delete contact: " + phone);
114. } else if (method.equals("GET")) { // if only GET print all or if phone number given print that too
115. System.out.println("retrieve contact: " + (phone.equals("") ? "All" : phone));
117. } else if (method.equals("POST")) { // if POST, set output contacts string to formatted contacts details with xml tags
118. System.out.println("create contact: " + phone + ", " + first + " " + last + ", " + street + ", " + town + ", " + post);
120. contacts = "<contacts><contact>" +
121. "<phone>" + phone + "</phone>" +
122. "<firstname>" + first + "</firstname>" +
123. "<lastname>" + last + "</lastname>" +
124. "<address>" +
125. "<street>" + street + "</street>" +
126. "<town>" + town + "</town>" +
127. "<postcode>" + post + "</postcode>" +
128. "</address>" +
129. "</contact></contacts>"; // contacts string to formatted contacts details with xml tags

132. } else if (method.equals("PUT")) { // if PUT, set output contacts string to formatted contacts details with xml tags
133. System.out.println("update contact: " + phone + ", " + first + " " + last + ", " + street + ", " + town + ", " + post);
135. contacts = "<contacts><contact>" +
136. "<phone>" + phone + "</phone>" +
137. "<firstname>" + first + "</firstname>" +
138. "<lastname>" + last + "</lastname>" +
139. "<address>" +
140. "<street>" + street + "</street>" +
141. "<town>" + town + "</town>" +
142. "<postcode>" + post + "</postcode>" +
143. "</address>" +
144. "</contact></contacts>"; // contacts string to formatted contacts details with xml tags
145. } else
146. throw (new Exception("unrecognised method: " + method)); // invalid method throw exception
148. Source requestSource = new StreamSource(new StringReader(contacts)); // initialise reader for XML source
149. Map<String, Object> requestContext = serviceDispatcher.getRequestContext();
150. requestContext.put(MessageContext.HTTP\_REQUEST\_METHOD, method);
151. requestContext.put(MessageContext.PATH\_INFO, SERVICE\_PATH + phone);
152. Source responseSource = serviceDispatcher.invoke(requestSource);
154. if (method.equals("GET")) { // if method = get
155. DOMResult domXML = new DOMResult();
156. messageTransformer.transform(responseSource, domXML);
157. Node topNode = domXML.getNode();
159. Element contactsNode = (Element) domXML.getNode().getFirstChild(); // get child of top node - <contacts>
160. NodeList contactNodeList = contactsNode.getChildNodes(); // node list storing all contact nodes <contact>
162. for (int i = 0; i < contactNodeList.getLength(); i++) { // for every contact do..
164. Element contact = (Element) contactNodeList.item(i); // variable storing contact element i
166. NodeList contactChildNodes = contact.getChildNodes(); // node list storing all contact details nodes e.g. <firstname>
168. if(contactChildNodes.item(0).getNodeType() != Node.TEXT\_NODE) { // if the first item in the node list is not a text node
169. NodeList addressChildList = contactChildNodes.item(3).getChildNodes();
170. String output =
171. contactChildNodes.item(0).getFirstChild().getNodeValue() +", "+ // set position 0 of contact detail array to phone number, retrieved from the node list storing contact details
172. contactChildNodes.item(1).getFirstChild().getNodeValue()+", "+ // same process repeated for the rest of entries
173. contactChildNodes.item(2).getFirstChild().getNodeValue() +", "+
174. addressChildList.item(0).getFirstChild().getNodeValue()+", "+
175. addressChildList.item(1).getFirstChild().getNodeValue()+", "+
176. addressChildList.item(2).getFirstChild().getNodeValue() + "\n";

179. if(i > 0) { // if there is more than one contact then append output string to text area
180. textArea.append(output);
181. System.out.println(output);
182. }
183. else { // if only one contact then set text
184. textArea.setText(output);
185. System.out.println(output);
186. }
187. }
188. }
189. }
190. } catch (Exception exception) {
191. if (!(exception instanceof WebServiceException))
192. exception.printStackTrace();
193. }
195. // analyse HTTP response
196. Map<String, Object> responseContext = serviceDispatcher.getResponseContext();
197. int httpResponse2 = (Integer) responseContext.get(MessageContext.HTTP\_RESPONSE\_CODE); // variable storing the http response of the current action, from the service

200. if (method.equals("GET")) { // if method is GET
201. if (httpResponse2 == 404) { // and if htttp response is 404
202. textArea.setText("Contact not found."); // output not found to text area
203. }
204. }
205. else if (method.equals("POST")) { //same as above but for post,
206. if (httpResponse2 == 405) {// checking 405, not allowed http response
207. textArea.setText("A contact with this number already exists.");
208. }
209. else if(httpResponse2 == 201) { // if created successfully
210. textArea.setText("Contact added.");
211. phonetf.setText(""); // clear text fields
212. firsttf.setText("");
213. lasttf.setText("");
214. streettf.setText("");
215. towntf.setText("");
216. posttf.setText("");
217. }
218. }
219. else if(method.equals("PUT")) { // if method PUT
220. if (httpResponse2 == 405 || httpResponse2 == 404) { // if contact not found
221. textArea.setText("Contact does not exist."); // error message to text area
222. }
223. else if (httpResponse2 == 200) { // if updated ok
224. textArea.setText("Updated successfully."); // success message to text area
225. phonetf.setText("");// clear text fields
226. firsttf.setText("");
227. lasttf.setText("");
228. streettf.setText("");
229. towntf.setText("");
230. posttf.setText("");
231. }
233. }
234. else if (method.equals("DELETE")) { // if method DELETE
235. if(httpResponse2 == 404) {// if contact not found
236. textArea.setText("Contact does not exist"); // error to text area
237. }
238. else if(httpResponse2 == 200) { // if removed ok
239. textArea.setText("Contact deleted."); // success to text area
240. phonetf.setText(""); // clear text fields
241. firsttf.setText("");
242. lasttf.setText("");
243. streettf.setText("");
244. towntf.setText("");
245. posttf.setText("");
246. }
247. }
248. }
250. public static void main(String arguments[]) throws Exception {

253. frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE); // close program on window close
254. frame.setSize(610, 800); // set frame size
255. frame.getContentPane().add(BorderLayout.NORTH, panel);
256. frame.setVisible(true); // set windows visible
258. // add all GUI components to the panel
259. panel.add(phone);
260. panel.add(phonetf);
262. panel.add(firstName);
263. panel.add(firsttf);
265. panel.add(lastName);
266. panel.add(lasttf);
268. panel.add(street);
269. panel.add(streettf);
271. panel.add(town);
272. panel.add(towntf);
274. panel.add(post);
275. panel.add(posttf);
277. panel.add(search);
279. panel.add(showAll);
280. panel.add(add);
281. panel.add(update);
282. panel.add(delete);
283. panel.add(clear);
285. textArea.setEditable(false); // set text area not editable
286. panel.add(textArea);
288. ContactClient client = new ContactClient(); // create new contact client object
289. client.invoke("POST","1", "John", "Smith", "1 Street", "Town", "AB12 0XY"); // add some example data to hash map using POST
290. client.invoke("POST","2", "first", "last", "st", "twn", "post");
291. client.invoke("POST","3", "firstname", "lastname", "2 Street", "Town", "AB12 0XY");
293. search.addActionListener(new ActionListener() { // action listener for search button
294. public void actionPerformed(ActionEvent e) {
295. if (!phonetf.getText().equals("") ) { // if phone test field is not empty
297. if (phonetf.getText().matches("[0-9]+")) { // if phone test field is only digits
299. client.invoke("GET", phonetf.getText()); // get contact details relating to given phone number
300. }
301. else {
302. textArea.setText("Phone number must use Digits 0-9 only."); // only digits warning message
303. }
304. }
305. else {
306. textArea.setText("Enter phone number."); // empty phone field warning message
307. }
308. }
309. });
311. showAll.addActionListener(new ActionListener() { // action listener for show all button
312. public void actionPerformed(ActionEvent e) {
314. client.invoke("GET"); // get all contacts
315. }
316. });
318. add.addActionListener(new ActionListener() { // action listener for add button
319. public void actionPerformed(ActionEvent e) {
320. // if all text fields are filled in
321. if (!phonetf.getText().equals("") && !firsttf.getText().equals("") && !lasttf.getText().equals("") && !streettf.getText().equals("") && !towntf.getText().equals("") && !posttf.getText().equals("")) {
323. if(phonetf.getText().matches("[0-9]+")) {// if only digits in phone text field
324. //post/create contact with given contact details gathered from texts fields
325. client.invoke("POST", phonetf.getText().toString(), firsttf.getText().toString(),lasttf.getText().toString(), streettf.getText().toString(), towntf.getText().toString(), posttf.getText().toString());
327. } else {
328. textArea.setText("Phone number must use Digits 0-9 only."); // only digits warning message
329. }
330. } else {
331. textArea.setText("Please fill in all fields"); // empty field warning message
333. }
334. }
335. });
337. update.addActionListener(new ActionListener() { // action listener for update button
338. public void actionPerformed(ActionEvent e) {
339. // if all text fields are filled in
340. if (!phonetf.getText().equals("") && !firsttf.getText().equals("") && !lasttf.getText().equals("") && !streettf.getText().equals("") && !towntf.getText().equals("") && !posttf.getText().equals("")) {
341. //put/update given contact details gathered from texts fields
342. client.invoke("PUT", phonetf.getText().toString(), firsttf.getText().toString(),lasttf.getText().toString(), streettf.getText().toString(), towntf.getText().toString(), posttf.getText().toString());
344. } else {
345. textArea.setText("Please fill in all fields");// empty field warning message
346. }
347. }
348. });
350. delete.addActionListener(new ActionListener() { // action listener for delete button
351. public void actionPerformed(ActionEvent e) {
353. if (!phonetf.getText().equals("")) { // if phone text field filled in
355. if(phonetf.getText().matches("[0-9]+")) { // if only digits used
357. client.invoke("DELETE", phonetf.getText().toString()); // delete contact with given phone number
359. }else {
360. textArea.setText("Phone number must use Digits 0-9 only.");// only digits warning message
361. }
362. } else {
363. textArea.setText("Enter phone number.");// empty phone field warning message
364. }
365. }
366. });
368. clear.addActionListener(new ActionListener() { // action listener for delete button
369. public void actionPerformed(ActionEvent e) {
371. phonetf.setText(""); // clear text fields
372. firsttf.setText("");
373. lasttf.setText("");
374. streettf.setText("");
375. towntf.setText("");
376. posttf.setText("");
377. }
378. });
379. }
380. }