IST PROJECT 2001-35399

A Governmental Knowledge-based Platform for Public Sector Online Services

Project Number:	IST-2001-35399
Project Title:	A Governmental Knowledge-based Platform for Public Sector Online Services
Deliverable Type:	Public

Deliverable Number:	-
Contractual Date of Delivery:	-
Actual Date of Delivery:	-
Title of Deliverable:	Legacy system database connectivity guide
WP contributing to the Deliverable:	WP8
Nature of the Deliverable:	Report
Editor(s):	Costas Vassilakis
Author(s):	Costas Vassilakis

Abstract: This document provides detailed information on how the services developed using the SmartGov platform may interact with databases to create documents with pre-populated fields and to store submitted documents to databases.

Project funded by the European Community under the "Information Society Technologies" Programme (1998-2002)

The SmartGov Consortium consists of:

Partner's Name	Acronym	Role	Country
University of Athens	UoA	Project Coordinator	Greece
T-Systems Nova	TNB	Partner	Germany
Indra Sistemas S.A.	Indra	Partner	Spain
Archetypon S.A.	ARC	Partner	Greece
Napier University	NU	Partner	UK
General Secretariat for Information Systems	GSIS	Partner	Greece
City of Edinburgh Council	CEC	Partner	UK

Table of Contents

1 Introduction

Electronic services developed using the SmartGov platform may need to interact with external databases, used by the organisation for information storage and retrieval. In most cases, this interaction will have the two following forms:

- retrieving values from registries to create documents with pre-populated fields, that will be displayed to the end-user.
- storing the elements of the submitted XML document into database table fields for further manipulation.

In this document, database connectivity for SmartGov services is exemplified through a simple service, in which a citizen declares the car (s)he owns. The rest of the document is organised as follows: in section 2 the car declaration service is described and the structure of the registry and document storage table is outlined; in section 3 the method for retrieving registry values to create pre-populated documents is detailed. Finally, in section 4 the method for storing submitted documents to a database is illustrated.

2 The car declaration service

The car declaration service is a simple electronic service allowing a citizen to declare the cars (s)he owns. The service's id is CAR_DCL and it comprises of a single form[1] that displays the citizen's id number, surname and name (distinct TSEs named "TSE_id_0001", "TSE_surname_0001" and "TSE_name_0001" respectively; note that the names of the *instantiated TSEs* are used, rather than the names of the generic ones). The form also provides facilities for entering each car's registration plate number and year of purchase. Since a citizen may own multiple cars, a TSE group named "cars_0001" has been created, which contains the elements "GRP_reg_plate_no_0001" and "GRP_purchase_year_0001"; the group is repeating with a minimum of zero rows (no cars owned) and a maximum of 50 rows. A possible rendering of the HTML form is illustrated in Figure 1.

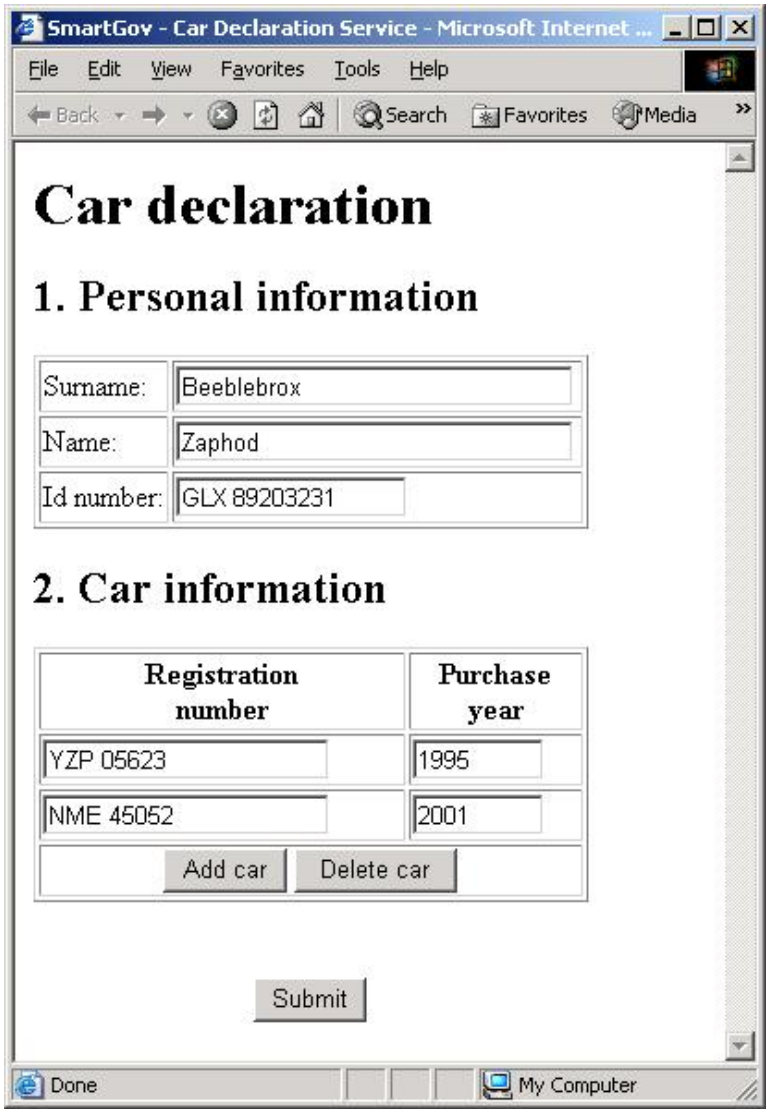

Figure 1 – The car declaration service

[1] The number of forms within a service is irrelevant from the techniques used for database connectivity. The same techniques work equally well, regardless from whether the TSEs and TSE groups are located in a single form or span across multiple forms.

The organisation delivering the service maintains a registry with the citizen's identification data and car ownership data. This registry consists of two database tables, the first one storing the citizen's personal information (surname, name and id number, possibly amongst others), and the second one storing the car ownership information (the registration number and the purchase year, together with the owner's identification number). The registry schema is depicted in Figure 2.

```
create table personal_info (
          username      varchar(32) not null unique,
          id_number     varchar(15) not null unique,
          surname       varchar(32) not null,
          name          varchar(32) not null
);

create table car_registration (
          id_number     varchar(15) not null,
          reg_number    varchar(15) not null,
          year_purchase integer not null,
          primary key(id_number, reg_number)
);
```

Figure 2 - Registry schema

Note that in the personal information table a column `username` appears. This column stores the SmartGov platform e-service user username, so as to provide the necessary association between the registry data and the SmartGov platform user information. Both the `username` and the `id_number` columns are declared as unique, since the e-service user name is unique across the SmartGov platform (system requirement), while the id number is unique by virtue of the business logic.

Besides the registry presented above, the organisation also wants to maintain information about the submitted documents. In order to store this information, two additional tables are created, as illustrated in Figure 3.

```
create table ecar_document_header (
          username      varchar(32) not null,
          timestamp     varchar(12) not null,
          id_number     varchar(15) not null,
          surname       varchar(32) not null,
          name          varchar(32) not null,
          primary key(username, timestamp)
);

create table ecar_document_detail (
          username      varchar(32) not null,
          timestamp     varchar(12) not null,
          reg_number    varchar(15) not null,
          year_purchase varchar(10) not null,
          primary key(id_number, timestamp, reg_number)
);
```

Figure 3 – Tables for storing the submitted documents

Note that in these tables the primary key is extended to include the timestamp column; this is necessary since the user may submit multiple documents, and exclusion of the document timestamp from the table's key would only allow storing the first document; storage of subsequent documents would be rejected, since they would violate the primary key constraint. Note also that all columns are declared as `varchar` type, while some of them are actually of other types (e.g. the `year_purchase` column is of type integer). This is done here for

simplicity; it is possible for the programmers to insert code to appropriately coerce the string-type representations of the TSE values to their "natural" type.

3 Creating documents pre-populated with registry values

When the e-service end-user requests to fill in a new document, the SmartGov platform invokes the document retrieval service running in the context of IIG to fetch a (possibly personalised) "initial document" for the user. This "initial document" may contain pre-assigned values for specific TSEs; TSEs for which no such value assignment is made will resort to their default values, as defined in the SmartGov development environment.

"Initial document" creation is performed by means of a Java class, whose name should be `IIGCreateserviceIdDocument`, where the portion `serviceId` should be replaced by the actual service id; in our example, where the service id is equal to `CAR_DCL`, the class should be named `IIGCreateCAR_DCLDocument`. Note that Java is a case-sensitive language, thus the proper usage of upper and lower case characters is important. The class in question should implement the method `createDocument` with the following prototype:

```
public String createDocument(String userName, String serviceName,
String timestamp) throws Exception;
```

If the class is found and it implements the aforementioned method, then the method is invoked to handle "initial document" creation. The method is supplied with three parameters, which provide the user name of the end-user requesting to create the new document (parameter `userName`), the id of the service (parameter `serviceName`) and the timestamp of user action in standard Java representation (a long integer, formatted as string – parameter `timestamp`). The method should return a string, designating the desired value assignments for TSEs. This string is actually an XML document, which however may be created through the facilities provided by the `gr.uoa.di.IIGXMLDocument` package, without any knowledge of XML, as will be illustrated below.

Writing the Java code for creating "initial document" for this service may start off with the generic template illustrated in Figure 4. This template declares a database connection (`theConn`), which is used to interact with the database. The link to the database is established through the first line in the `try` block, which connects to the `db` database within a `MySQL` instance running on host `dbhost`. The username used for this connection is `root`, while the password is the empty string. Naturally, these parameters should be edited to match the respective information in the organisation's installation.

Another important part in Figure 4 is the line

```
IIGXMLDocument reply = new IIGXMLDocument(serviceName, userName, timestamp);
```

This line creates an IIGXMLDocument convenience object, in which the method result will be built. The result is initialised to contain indications about the current service, user and timestamp. After the `try` block, in which TSE value assignment will be performed, the lines

```
reply.endDocument();
return reply.getContents();
```

are used to finalise the document and forward it to the SmartGov platform, to be used as "initial document" for the user.

```
import java.lang.String;
import java.sql.*;
import gr.uoa.di.IIGXMLDocument.*;

public class IIGCreateCAR_DCLDocument  {
   public String createDocument(String userName, String serviceName, String timestamp)
throws Exception{
       /*the database connection and result set*/
       Connection theConn = null;

       IIGXMLDocument reply = new IIGXMLDocument(serviceName, userName, timestamp);
       try {
         theConn = DriverManager.getConnection("jdbc:mysql://dbhost/db", "root", "");
         /* Code for TSE value assignment will be placed here */
         theConn.close();
       }
       catch (Exception e) {
          /* handle the exception */
       }
       reply.endDocument();

       return reply.getContents();
   }
}
```

Figure 4 – Initial template for document creation

3.1 Setting simple TSE values

In order to get values from the database, the first action to be performed is the formulation of the SQL query that will retrieve the data. In our case, the SQL query will be declared and executed via the piece of code depicted in Figure 5:

```
String SQLstr = "select id_number, surname, name " +
            "from personal_info where username = ?";
String idNumber = "", surname = "", name = "";
PreparedStatement SQLqry = theConn.prepareStatement(SQLstr);
SQLqry.setString(1, userName);
ResultSet res = SQLqry.executeQuery();
if (res.next()) {
            idNumber = res.getString(1);
            surname = res.getString(2);
            name = res.getString(3);
}
res.close();
```

Figure 5 – Creating and executing the SQL query

In this code fragment, the first statement declares the textual representation of the SQL query, using the question mark placeholder to designate that a value will be provided for this location later on. The second statement declares the variables into which the query results will be stored, while the third statement is a Java technicality, creating an internal structure appropriate for submitting the query to the database.

The fourth statement arranges for providing the required parameter to the query (the value to be used in the location of the question mark placeholder), and the fifth statement actually executes the query. Query results are stored in the `res` variable.

The following `if` block checks first if the query has been successful in finding matching data in the database (a row for the specific e-service end-user). If data have been found, they are assigned to the respective variables; if not, the variables retain their original values, which are empty strings. The last statement arranges for closing the result set, freeing system resources associated with this result set.

Having stored the desired values into Java variables, the next step is to include the relevant TSE assignments in the document to be returned. This can be achieved via the code fragment depicted in Figure 6:

```
reply.addTSE("TSE_id_0001", idNumber);
reply.addTSE("TSE_surname_0001", surname);
reply.addTSE("TSE_surname_0001", name);
```

Figure 6 – Including TSE value assignments in the reply

Each TSE value assignment is performed through an invocation of the `addTSE` method on the `reply` object; the first parameter to the invocation is the name of the TSE, while the second parameter is the desired value. Note that the order of these lines is irrelevant.

Combining the code from the figures 4, 5 and 6, the code fragment in Figure 7 results.

```
import java.lang.String;
import java.sql.*;
import gr.uoa.di.IIGXMLDocument.*;

public class IIGCreateCAR_DCLDocument  {
   public String createDocument(String userName, String serviceName, String timestamp)
throws Exception{
       /*the database connection and result set*/
       Connection theConn = null;

       IIGXMLDocument reply = new IIGXMLDocument(serviceName, userName, timestamp);
       try {
         theConn = DriverManager.getConnection("jdbc:mysql://dbhost/db", "root", "");
         String SQLstr = "select id_number, surname, name " +
           "from personal_info where username = ?";
         String idNumber = "", surname = "", name = "";
         PreparedStatement SQLqry = theConn.prepareStatement(SQLstr);
         SQLqry.setString(1, userName);
         ResultSet res = SQLqry.executeQuery();
         if (res.next()) {
           idNumber = res.getString(1);
           surname = res.getString(2);
           name = res.getString(3);
         }
         res.close();
         reply.addTSE("TSE_id_0001", idNumber);
         reply.addTSE("TSE_surname_0001", surname);
         reply.addTSE("TSE_surname_0001", name);
         theConn.close();
       }
       catch (Exception e) {
          /* handle the exception */
       }
       reply.endDocument();

       return reply.getContents();
   }
}
```

Figure 7 – Code for creating an initial document with simple TSEs

3.2 Setting values for repeating TSE groups

A repeating TSE group may have multiple matching rows in the corresponding database table. Each such database row must be fetched and mapped into a *group row* in the document that will be returned. All rows must be enclosed in a *group container* that identifies the group these rows belong to.

Again, the first action to be performed is the formulation of the SQL query that will retrieve the data. In our case, the SQL query will be declared and executed via the piece of code depicted in Figure 8:

```
String SQLstr1 = "select reg_number, year_purchase" +
         "from car_registration where id_number = ?";
String regNumber = "";
int yearPurchase = "";
PreparedStatement SQLqry1 = theConn.prepareStatement(SQLstr);
SQLqry.setString(1, idNumber);
ResultSet res1 = SQLqry1.executeQuery();
```

Figure 8 – Creating and executing the SQL query

In this code fragment, the first statement declares the textual representation of the SQL query, using the question mark placeholder to designate that a value will be provided for this location later on. The second and third statement declares the variables into which the query results will be stored; two distinct declarations are used, since columns are of different types. The fourth statement is a Java technicality, creating an internal structure appropriate for submitting the query to the database.

The fifth statement arranges for providing the required parameter to the query (the value to be used in the location of the question mark placeholder). Note that the `idNumber` variable -previously fetched from the database- is used here, since the id number is used in the `car_registration` table to identify the owner. The sixth statement actually executes the query. Query results are stored in the `res1` variable.

Once the desired rows have been fetched into the Java variable, the appropriate TSE assignments can be included in the result document. This may be achieved via the code illustrated in Figure 9.

```
reply.beginGroup("cars_0001");
String regPlateNo, strYearPurchase;
int yearPurchase;
while (res1.next()) {
          reply.beginGroupRow();
          regPlateNo = res1.getString(1);
          yearPurchase = res1.getInt(2);
          strYearPurchase = String.valueOf(yearPurchase);
          reply.addTSE("GRP_reg_plate_no_0001", res1.getString(1));
          reply.addTSE("GRP_purchase_year_0001", strYearPurchase);
          reply.endGroupRow();
}
res1.close();
reply.endGroup();
```

Figure 9 – Processing database rows and creating group rows

The first statement opens the group container, indicating that all TSE assignments created hereafter will pertain to the designated group. The second and third statements declare variables that will be used for fetching database values and creating group rows. Note that an extra variable named `strYearPurchase` is declared; this is needed because the database representation of the year of purchase is an integer, while TSE representations are always of type String. This variable is used to implement the necessary conversion between the two types.

The condition in the `while` statement advances to the next row in the result set and also arranges for terminating the loop once the rows in the result set have been exhausted. Within the `while` body, the first statement indicates that a new TSE group row begins. The two following statements fetch the database values into Java variables, while the subsequent statement converts the integer database representation of the year of purchase to the String type. Once both row values have been fetched into Java variables and coerced to the appropriate

type (String), the TSE assignments for this row are performed through the two following statements. The last statement in the `while` body indicates that the group row is complete.

The two statements immediately after the `while` loop, arrange for closing the result set (freeing system resources associated with it) and for closing the group container, indicating that TSE assignments for this group are complete.

Combining the fragments from figures 7, 8 and 9 we have the complete document creation code, illustrated in Figure 10.

```
import java.lang.String;
import java.sql.*;
import gr.uoa.di.IIGXMLDocument.*;

public class IIGCreateCAR_DCLDocument {
   public String createDocument(String userName, String serviceName, String timestamp)
throws Exception{
       /*the database connection and result set*/
       Connection theConn = null;

       IIGXMLDocument reply = new IIGXMLDocument(serviceName, userName, timestamp);
       try {
         theConn = DriverManager.getConnection("jdbc:mysql://dbhost/db", "root", "");
         String SQLstr = "select id_number, surname, name " +
           "from personal_info where username = ?";
         String idNumber = "", surname = "", name = "";
         PreparedStatement SQLqry = theConn.prepareStatement(SQLstr);
         SQLqry.setString(1, userName);
         ResultSet res = SQLqry.executeQuery();
         if (res.next()) {
           idNumber = res.getString(1);
           surname = res.getString(2);
           name = res.getString(3);
         }
         res.close();
         reply.addTSE("TSE_id_0001", idNumber);
         reply.addTSE("TSE_surname_0001", surname);
         reply.addTSE("TSE_surname_0001", name);

        String SQLstr1 = "select reg_number, year_purchase" +
            "from car_registration where id_number = ?";
       PreparedStatement SQLqry1 = theConn.prepareStatement(SQLstr);
       SQLqry.setString(1, idNumber);
       ResultSet res1 = SQLqry1.executeQuery();
       reply.beginGroup("cars_0001");
       String regPlateNo, strYearPurchase;
       int yearPurchase;
       while (res1.next()) {
            reply.beginGroupRow();
            regPlateNo = res1.getString(1);
            yearPurchase = res1.getInt(2);
            strYearPurchase = String.valueOf(yearPurchase);
            reply.addTSE("GRP_reg_plate_no_0001", res1.getString(1));
            reply.addTSE("GRP_purchase_year_0001", strYearPurchase);
            reply.endGroupRow();
         }
         res1.close();
         reply.endGroup();
         theConn.close();
       }
       catch (Exception e) {
          /* handle the exception */
       }
       reply.endDocument();

       return reply.getContents();
   }
}
```

Figure 10 – Complete code for document creation

3.3 Compiling and installing the code

The complete Java code can now be compiled and installed. Compilation can be performed by means of any standard Java compiler or development environment. Before initiating compilation it must be ascertained that the library `iigxmldoc.jar` is referenced in the `CLASSPATH` variable or is otherwise made available to the compiler/development environment. The result of the code compilation will be a `class` file named `IIGCreateCAR_DCLDocument.class`. In order to be made available to the SmartGov runtime environment, the following actions need to be performed:

1. The `class` file must be packed into a `jar` archive. This can be achieved using the standard `jar` utility or through any other facilities offered by the development environment. In order to create the jar file using the `jar` utility, the following command may be issued:

   ```
   jar cf ecars.jar IIGCreateCAR_DCLDocument.class
   ```
   This command will create the `ecars.jar` file.

2. The `jar` file created in the previous step should be copied into the `lib` subfolder of the IIG installation root folder (normally, `c:\iig\lib`).

3. The IIG servers must be stopped and re-started.

3.4 Tracing and debugging code

The document creation procedure is a standard Java piece of code that may be run and debugged normally, by supplying a `main` method that will invoke the document creation method, supplying the appropriate parameters. For instance, in the previous example the main method might be coded as illustrated in .

```
public static void main(String args[]) {
            System.out.println(createDocument("testuser", "CAR_DCL", "012345678"));
}
```

Figure 11 – Main method for debugging

This will display the XML document returned to the SmartGov platform by the document creation procedure.

Tracing the behaviour of the document creation procedure in the actual runtime of the SmartGov platform can be achieved by including appropriate diagnostic message emission statements within the code. Emitted statements will appear in the console window of the IIG server; if local administrators have arranged so that document storage is performed under SSL communications, diagnostic messages will appear on the console of the SSL IIG server.

4 Storing document elements into a database

When the end-user submits a document, this is passed to the document storage facility of the SmartGov platform. The document storage facility always stores the document into the XML repository, but also checks if there exists a service-specific method for handling document storage. Service-specific document storage isis performed by means of a Java class, whose name should be `IIGStoreserviceIdDocument`, where the portion `serviceId` should be replaced by the actual service id; in our example, where the service id is equal to `CAR_DCL`, the class should be named `IIGStoreCAR_DCLDocument`. Note that Java is a case-sensitive language, thus the proper usage of upper and lower case characters is important. The class in question should implement the method `createDocument` with the following prototype:

```
public void storeDocument(IIGServiceResults document) throws Exception;
```

If the class is found and it implements the aforementioned method, then the method is invoked to handle service-specific document storage. The method is supplied with a single parameter, which contains a "flattened" representation of the XML document, to facilitate storage in RDBMSs. The flattened representation is provided by the `gr.uoa.di.IIGServiceResults` package. The method returns nothing.

Writing the Java code for storing the document content into a relational database, may start off with the generic template illustrated in Figure 12. This template declares a database connection (`theConn`), which is used to interact with the database. The link to the database is established through the third line in the `try` block, which connects to the `db` database within a `MySQL` instance running on host `dbhost`. The username used for this connection is `root`, while the password is the empty string. Naturally, these parameters should be edited to match the respective information in the organisation's installation. The two first lines in the `try` block extract from the flattened representation the user name and the timestamp of the document; these values will be used later on.

```
import java.sql.*;
import java.lang.String;
import gr.uoa.di.IIGServiceResults.IIGServiceResults;

public class IIGStoreCAR_DCLDocument  {
  public void storeDocument(IIGServiceResults document) throws Exception{
    Connection theConn = null;
    try{
      String userName = document.getUserName();
      String tstamp = document.getTimestamp();
      theConn = DriverManager.getConnection("jdbc:mysql://dbhost/db", "root", "");

      /* Code for storing document contents is placed here */

      theConn.close();
    }
    catch (Exception e){
      System.err.println(e);
      e.printStackTrace();
      System.err.println("ecars: Error while storing document\n" + e.getMessage());
    }
    return;
  }
}
```

Figure 12 – Template for document storage

Once the connection has been established, the document contents may be inserted in the database; this procedure will be detailed in the following paragraphs. After content insertion, the link to the database is broken by applying

the `close` method on the connection, freeing any system resources associated with it.

4.1 Storing values of simple TSEs

In order to store values to the database, the first action to be performed is the formulation of the SQL query that will store the data. In our case, the SQL query will be declared and executed via the piece of code depicted in Figure 5:

```
String headerQRY = "insert into ecar_document_header " +
  " (username, timestamp, id_number, surname, name)\n" +
  "values(?, ?, ?, ?, ?)";
PreparedStatement headerQuery = theConn.prepareStatement (headerQRY);
headerQuery.setString(1, userName);
headerQuery.setString(2, tstamp);
headerQuery.setString(3, document.getElementValue("TSE_id_0001"));
headerQuery.setString(4, document.getElementValue("TSE_surname_0001"));
headerQuery.setString(5, document.getElementValue("TSE_name_0001"));
headerQuery.execute();
```

Figure 13 – Storing simple TSE values in the database

In this code fragment, the first statement declares the textual representation of the SQL query, using question mark placeholders to designate that a value will be provided for these locations later on. The second statement is a Java technicality, creating an internal structure appropriate for submitting the query to the database.

The five following statements (statement 3 to statement 7) arrange for providing the required parameters to the query (the values to be used in the location of the question mark placeholders). In the first two of them, the username and the timestamp of the document are used (extracted in the template depicted in Figure 12), while in the remaining three of them the document is queried to return the values of specific TSEs (via the `getElementValue` method which accepts as a parameter the name of the desired TSE). Finally, the last statement in this excerpt executes the SQL statement, inserting the values in the database.

4.2 Storing values of repeating groups

In order to store the values for a repeating group, all the TSEs in the "flattened" XML document representation need to be examined, in order to determine whether each of them belongs to the desired repeating group. Elements that do belong to the repeating group in question should then be broken down to individual rows, and each row will be inserted into the database. This may be performed through the code excerpt illustrated in Figure 14.

```
      String detailQRY = "insert into ecar_document_detail " +
        "(username, timestamp, reg_number, year_purchase)\n" +
        "values(?, ?, ?, ?)";
      PreparedStatement detailQuery = theConn.prepareStatement(detailQRY);
      for (int TSEcount = 0; TSEcount < document.size(); TSEcount++) {
        if (document.elementGroupNameAt(TSEcount).equals("cars_0001")) {
          int currentElementRow =
               new Integer(document.elementGroupRowAt(TSEcount)).intValue();
          int ctr1 = 0;
          String regPlateNo = "";
          String yearPurchase = "";
          while (
                  (TSEcount + ctr1 < document.size()) &&
                  (document.elementGroupNameAt(TSEcount  +  ctr1).equals("cars_0001"))
&&
                  ((new          Integer(document.elementGroupRowAt(TSEcount          +
ctr1))).intValue() == currentElementRow)
                ) {
            String theElementName = document.elementNameAt(TSEcount + ctr1);
            String theElementValue = document.elementValueAt(TSEcount + ctr1);
            if (theElementName.equals("GRP_reg_plate_no_0001"))
              regPlateNo = new String(theElementValue);
            else if (theElementName.equals("GRP_purchase_year_0001"))
              yearPurchase = new String(theElementValue);
            ctr1++;
          }
          detailQuery.clearParameters();
          detailQuery.setString(1, userName);
          detailQuery.setString(2, tstamp);
          detailQuery.setString(3, regPlateNo);
          detailQuery.setString(4, yearPurchase);
          detailQuery.execute();
          TSEcount = TSEcount + ctr1 - 1;
        }
      }
```

Figure 14 – Storing repeating groups in the database

In this code fragment, the first statement declares the textual representation of the SQL query, using question mark placeholders to designate that a value will be provided for these locations later on. The second statement is a Java technicality, creating an internal structure appropriate for submitting the query to the database. Immediately following is a loop, which will scan all the TSEs in the document and each TSE is checked whether it belongs to the desired group (`cars_0001`) via the `if` statement enclosed in the loop.

The `if` statement body will only be executed if a TSE belonging to the repeating group in question is found. Once the first TSE belonging to a group is found, the remaining TSEs belonging to the same group row must be located, in order to formulate a complete row to be inserted in the database. Thus the first statement in the `if` block marks the row number of the current element, to be checked against the row numbers of other elements. Then, an index variable (`ctr1`) is declared to assist in locating the remaining TSEs in the same row, and subsequently one String-typed variable is declared for every TSE within the repeating group. The `cars_0001` repeating group contains two TSEs, thus two variables are declared.

The immediately following `while` loop actually locates the TSEs in the same group row and extracts their values in the previously declared variables. The condition of the loop is standard, and only the group name needs to be edited to make it work for any group. The first two statements in the loop extract the TSE name and the TSE value, and the following `if` statement arranges so that the value is assigned to the proper variable. The `if` statement may be extended with as many `else if` alternatives as are required for all the TSEs included in the repeating group. The final statement in the loop advances the next TSE.

When the `while` loop is exited, the values of the TSEs of a group row will have been assigned to the proper variables. Now, the variable values need to be

associated with the proper query placeholders, and the query should be executed against the database. Before commencing these actions, however, the query needs to be *reset*, since parameter assignments may have been previously performed for another group row. Resetting the query is achieved through the

```
detailQuery.clearParameters();
```

statement, while the four subsequent statements assign the proper values (two from the document and two from the group row) to the related query placeholders. Finally, the query is executed to insert the row into the database, and the scan moves to the first TSE not yet considered by adjusting the `TSEcount` index variable.

By combining the code in Figures 12, 13 and 14 we get the complete document content storage code, depicted in Figure 15.

```
import java.sql.*;
import java.lang.String;
import gr.uoa.di.IIGServiceResults.IIGServiceResults;

public class IIGStoreCAR_DCLDocument  {
  public void storeDocument(IIGServiceResults document) throws Exception{
    Connection theConn = null;
    try{
      String userName = document.getUserName();
      String tstamp = document.getTimestamp();
      theConn = DriverManager.getConnection("jdbc:mysql://dbhost/db", "root", "");
      String headerQRY = "insert into ecar_document_header " +
        " (username, timestamp, id_number, surname, name)\n" +
        "values(?, ?, ?, ?, ?)";
      PreparedStatement headerQuery = theConn.prepareStatement (headerQRY);
      headerQuery.setString(1, userName);
      headerQuery.setString(2, tstamp);
      headerQuery.setString(3, document.getElementValue("TSE_id_0001"));
      headerQuery.setString(4, document.getElementValue("TSE_surname_0001"));
      headerQuery.setString(5, document.getElementValue("TSE_name_0001"));
      headerQuery.execute();

      String detailQRY = "insert into ecar_document_detail " +
        "(username, timestamp, reg_number, year_purchase)\n" +
        "values(?, ?, ?, ?)";
      PreparedStatement detailQuery = theConn.prepareStatement(detailQRY);
      for (int TSEcount = 0; TSEcount < document.size(); TSEcount++) {
        if (document.elementGroupNameAt(TSEcount).equals("TSEG_EVAT_DETAIL")) {
          int currentElementRow =
               new Integer(document.elementGroupRowAt(TSEcount)).intValue();
          int ctr1 = 0;
          String regPlateNo = "";
          String yearPurchase = "";
          while (
                  (TSEcount + ctr1 < document.size()) &&
                  (document.elementGroupNameAt(TSEcount + ctr1).equals("cars_0001")) &&
                  ((new Integer(document.elementGroupRowAt(TSEcount + ctr1))).intValue()
== currentElementRow)
                ) {
            String theElementName = document.elementNameAt(TSEcount + ctr1);
            String theElementValue = document.elementValueAt(TSEcount + ctr1);
            if (theElementName.equals("GRP_reg_plate_no_0001"))
              regPlateNo = new String(theElementValue);
            else if (theElementName.equals("GRP_purchase_year_0001"))
              yearPurchase = new String(theElementValue);
            ctr1++;
          }
          detailQuery.clearParameters();
          detailQuery.setString(1, userName);
          detailQuery.setString(2, tstamp);
          detailQuery.setString(3, regPlateNo);
          detailQuery.setString(4, yearPurchase);
          detailQuery.execute();
          TSEcount = TSEcount + ctr1 - 1;
       }
    }
    theConn.close();
    theConn = null;
    }
    catch (Exception e){
      System.err.println(e);
      e.printStackTrace();
      System.err.println("ecars: Error while storing document\n" + e.getMessage());
    }
    return;
  }
}
```

Figure 15 – Complete code for storing document contents

Note that the code presented above does not modify the registry contents in any sense; document contents are stored in different tables and it is subject to the organisational workflow procedures to move/copy the values submitted through electronic documents to the registry. However, the storage code may be crafted

to immediately modify the registry contents without any need for external procedure intervention. For this simple service, this may be achieved by

(a) deleting all rows in the `car_registration` table pertaining to the submitting user

(b) inserting the rows contained in the submitted document to the `car_registration` table, complementing each row with the citizen's id, as required by the database table schema.

For optimisation purposes, the code storing the repeating group in the `ecar_document_detail` table may be co-bundled with the code storing rows in the `car_registration` table, as shown in Figure 16, so as to populate both tables with a single TSE scan. The new lines appear in shaded background. The `personal_info` registry table is not modified in this service, since all elements used from this table are read-only.

```
      String detailQRY = "insert into ecar_document_detail " +
        "(username, timestamp, reg_number, year_purchase)\n" +
        "values(?, ?, ?, ?)";
      PreparedStatement detailQuery = theConn.prepareStatement(detailQRY);
      String idNo = document.getElementValue("TSE_id_0001")
      String delReg = "delete from car_registration where id_number = ?";
      PreparedStatement delPrepP = theConn.prepareStatement(delReg);
      delPrepP.setString(1, idNo);
      delPrepP.execute();
      String regIns = insert into car_registration " +
        " (id_number, reg_number, year_purchase) values (?, ?, ?);"
      PreparedStatement regInsP = theConn.prepareStatement(regIns);
      for (int TSEcount = 0; TSEcount < document.size(); TSEcount++) {
        if (document.elementGroupNameAt(TSEcount).equals("cars_0001")) {
          int currentElementRow =
               new Integer(document.elementGroupRowAt(TSEcount)).intValue();
          int ctr1 = 0;
          String regPlateNo = "";
          String yearPurchase = "";
          while (
                  (TSEcount + ctr1 < document.size()) &&
                  (document.elementGroupNameAt(TSEcount  +  ctr1).equals("cars_0001"))
&&
                  ((new          Integer(document.elementGroupRowAt(TSEcount          +
ctr1))).intValue() == currentElementRow)
                ) {
            String theElementName = document.elementNameAt(TSEcount + ctr1);
            String theElementValue = document.elementValueAt(TSEcount + ctr1);
            if (theElementName.equals("GRP_reg_plate_no_0001"))
              regPlateNo = new String(theElementValue);
            else if (theElementName.equals("GRP_purchase_year_0001"))
              yearPurchase = new String(theElementValue);
            ctr1++;
          }
          detailQuery.clearParameters();
          detailQuery.setString(1, userName);
          detailQuery.setString(2, tstamp);
          detailQuery.setString(3, regPlateNo);
          detailQuery.setString(4, yearPurchase);
          detailQuery.execute();
          regInsP.clearParameters();
          regInsP.setString(1, idNo);
          regInsP.setString(2, regPlateNo);
          Integer intYearP = new Integer(yearPurchase);
          regInsP.setInt(3, yearPurchase.intValue());
          regInsP.execute();
          TSEcount = TSEcount + ctr1 - 1;
        }
    }
```

Figure 16 – Modifying the registry contents

Statements one to five in the first shaded area arrange for deleting the current registry contents, while the two last statements in the same area declare the query that will re-populate the registry.

The code in the second shaded area arranges for re-populating the registry: the first statement resets the query, while the four next set the query parameters. Note that an intermediate Integer-typed variable is used, to provide for coercing the String-typed TSE representation to an integer, so as to match the database schema. The last statement in the second shaded area executes the query to insert the values in the registry.