University of New Orleans

ScholarWorks@UNO

University of New Orleans Theses and Dissertations

Dissertations and Theses

12-20-2009

Automation of the Client Side of Web Services Using PHP

Menad Medjkane
University of New Orleans

Follow this and additional works at: https://scholarworks.uno.edu/td

Recommended Citation

Medjkane, Menad, "Automation of the Client Side of Web Services Using PHP" (2009). *University of New Orleans Theses and Dissertations*. 1096.
https://scholarworks.uno.edu/td/1096

This Thesis is protected by copyright and/or related rights. It has been brought to you by ScholarWorks@UNO with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in University of New Orleans Theses and Dissertations by an authorized administrator of ScholarWorks@UNO. For more information, please contact scholarworks@uno.edu.

Automation of the Client Side of Web Services Using PHP

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
In partial fulfillment of the
Requirements for the degree of

Master of Science
in
Computer Science

by

Menad Medjkane

B.S. University of Sciences and Technologies of Oran (Algeria), 1995

December, 2009

Acknowledgment

I would like to express my deep gratitude toward everyone who has helped me during my graduate study at the University of New Orleans. Especially, I am truly grateful for my advisor, Dr. Shengru Tu, for his constant guidance and continued support that go far above and beyond the call of duty. I thank Dr. Mahdi Abdelguerfi and Dr. Adlai DePano for their encouragement and help during the course of my degree curriculum.

Finally, I wish to thank my parents for sacrificing so much for me to have a better life. Most importantly, I wish to thank my wife and my children for their patience, their understanding, and for always standing by me.

Table of Contents

List of Figures v

Abstract vii

Chapter 1 Introduction 1

Chapter 2 Background and Related Work 3
2.1 XML 3
2.2 Web Service 4
2.3 WSDL 5
2.4 SOAP 6
2.5 DOM 7
2.6 XPATH 9
2.7 AJAX 9

Chapter 3 Structure and System Design 10
3.1 Web Service 12
3.2 Front end 17
3.3 WS-PHP Middleware component 18
3.3.1 WSDL elements 19
3.3.2 Service operations 21
3.3.3 Automating the process that enables PHP to call Web Services 23
3.3.3.1 Existing PHP-WS interactions approaches 23
3.3.3 .2 The procedures for PHP to call Web Services operations 25
3.3.4 The structure of the WS-PHP middleware 26

Chapter 4 Implementation 31
4.1 Client Implementation 32
4.2 Class WSDL _Element 37
4.3 Implementation of WS-PHP Middleware component 47
4.3.1 Call Without Parameters 47
4.3.2 Call With Parameters 50
4.4 Web Service Operation Selector 58

Chapter 5 Experiments 63
5.1 The Car sales application 65
5.2 The House sales application 71

Chapter 6 Conclusion 74

References 75

Vita 76

List of Figures

Figure 2.1: XML 3
Figure 2.2: Web Service 4
Figure 2.3:WSDL Structure 6
Figure 2.4:System Architecture 7
Figure 2.5: book Store Dom Tree 8
Figure 3.1: System View 11
Figure 3.2: Class Store Diagram 12
Figure 3.3: Specification of operations in WSDL 13
Figure 3.4: Class Car Diagram 15
Figure 3.5: Class CarArg Diagram 16
Figure 3.6: Ajax Call 18
Figure 3.7: Operation element in WSDL 19
Figure 3.8: message element of WSDL 19
Figure 3.9: message response element of WSDL 19
Figure 3.10: element of type WSDL 20
Figure 3.11: operation that has all WSDL elements 21
Figure 3.12: operation with no parameter 22
Figure 3.13: operation without return value 23
Figure 3.14: System Details 27
Figure 3.15:Sequence Diagram HandleParameters 28
Figure 3.16: Sequence Diagram for HandleRequest_WP.php 29
Figure 3.17: Sequence Diagram HandleRequest_NP 30
Figure 4.1: System View 31
Figure 4.2: Function getHTTPObject() 32
Figure 4.3 : Event handler using GET 33
Figure 4.4: Client functions which process the php response results 34
Figure 4.5: Client functions which process the php response results 35
Figure 4.6: Event Handler for POST method 35
Figure 4.7: Client function returning element of a form 36
Figure 4.8: Static form showing the event handler 36
Figure 4.9: Constructor of the Wsdl_Element class 37
Figure 4.10: Class diagram 38
Figure 4.11: Method which parse the operation in wsdl document 39
Figure 4.12: Result of the getOperation method 40
Figure 4.13: element types in wsdl document 41
Figure 4.14: Method getType 42
Figure 4.15: types of getCar operation 43
Figure 4.16: Signature of the getCar Web Service method 43
Figure 4.17: operation with complex type 44

Figure 4.18: Signature of the getStore web service method 44
Figure 4.19: Definition of the complex object CarArg 45
Figure 4.20 :getAttributes method 46
Figure 4.21: MyClient Class 47
Figure 4.22:Function parse_result 49
Figure 4.23:Function HandleRequest_NP 50
Figure 4.24: Function HandleParameters 51
Figure 4.25:data posted transformed in simple case 52
Figure 4.26:data posted transformed in complex case 52
Figure 4.27:the second parameter used in SOAP call 53
Figure 4.28:Soap request in simple case 53
Figure 4.29:SOAP request in complex case 54
Figure 4.30:Soap response of getCar 55
Figure 4.31:Function Handle Request_WP 57
Figure 4.32:multidimensional array returned by Handle2 58
Figure 4.33:HTML form displayed at the end user 59
Figure 4.34: Function handle1 60
Figure 4.35:Sequence Diagram for Handle1 61
Figure 4.36: System Details 62
5.1 index page 64
5.2: Set of operations in the car Web Service 65
5.3: IT set the operations to be published to the user 66
5.4: user page interface 67
5.5: request and response for getCar operation 68
5.6: error message for invalid data 69
5.7:No result found for the user request 70
5.8: operation add_House , no return 71
5.9:Request for find_House operation 72
5.10:request for search_House operation 73

Abstract

Web Services have been the dominant technology in business integration and implementation of service oriented architectures. PHP is a server-side language popular for development of applications. A significant advantage of PHP is its light weight development for feature-rich web applications. Typically, PHP is used for making good-looking front end user interfaces; Java or other programming languages are used to develop the back end application. A secure and robust way for PHP programs to call back-end services is by Web Services. However, when the Web Service operations have complex interfaces, writing PHP client code can be difficult and error-prone.

This thesis research seeks to develop a Web service-PHP program middleware that automatically handles the client-side Web Service calls. Two Web Services are developed, as well as two Web applications that consume the two Web Services, and experiments that demonstrate the usage of the WS-PHP middleware component are conducted.

Keywords: Web Services, PHP, DOM, XPATH, WSDL, SOAP, Ajax, Client Program, Automation.

Chapter 1. Introduction

Web Services have been the dominant technology in business integration and implementation of service-oriented architectures. Web Services do not dictate any restrictions in language or platforms; it is based on the widely accepted data exchange language XML and the W3C protocol SOAP. Typically, SOAP-based web services are for heavy duty server-side computation or business transactions. It is a common choice for enterprise applications. The client can directly process the SOAP message in any programming language such as PHP.

PHP is a server-side language, particularly popular in the development of Web applications. PHP is a member of the most significant open source LAMP collection (Linux, Apache, MySQL, PHP). An important advantage of PHP is its light-weight development for feature-rich Web applications. PHP has received attention from all the leading software vendors such as IBM and ORACLE. They have provided a number of ways to support PHP. Using PHP as a client of Web Services has become the mainstream practice for integrating complex server-side computations into feature-rich PHP Web applications. A native support for consuming Web Services has been added to PHP version 5.

However, writing PHP client code can be complex and error-prone, when the Web Service operations require multiple arguments and/or return objects of composite types. Developing a PHP client program for a given service requires the programmer to convert any type of objects that is delivered from or needed by the Web service operation to various nested array structures. Even though this goes through a repeated routine, the complexity of the objects can add significant burden to the programmer and often cause a high possibility of mistakes.

The work that culminates in this thesis seeks to develop a WS-PHP middleware component that facilitates automation for the client-side PHP programs that call Web Services. The development of the WS-PHP middleware was based on the experience obtained by studying numerous Web Services and implementing a number of the client PHP programs for various Web Service. In doing so, the common processes were singled out in the PHP client programming. The solution is not just to handle a specific given cases but also to figure out a means to handle different, possible, unknown cases.

The usage of the WS-PHP middleware component is illustrated in two Web applications that consume two Web Services – a car service and a house service. It has been illustrated that the WS-PHP middleware component is capable of automatically handling different types of operations with complex parameter objects and return values.

Chapter 2. Background and Related Work

In this chapter, a number of key concepts are reviewed: Web Service technology including the SOAP and the WSDL standards, as well as other XML technologies such as the DOM model, the XPATH language, and the Ajax technique.

2.1 XML

XML, which stands for e**X**tensible **M**arkup **L**anguage, is a specification, defined by the W3C [W3C], which provides syntax to structurally represent data using markups. XML, however, usually refers to the entire family of related-technologies. The W3C recommends not abbreviating XML elements and attribute names. For example, the following listing shows how a student record might be marked up using XML (Fig. 2.1):

```
<book>
        <Author>Giada De Laurentis</Author>
        <Title>Every Day Italian</Title>
        <Price>35.00</Price>
        <Year>2005</Year>
</book>
```

Figure 2.1: XML

A well-formed XML definition must also adhere to certain requirements such as the start tag and end tag having to be the same, no overlapping tags, and element and attribute names being surrounded by quotes.

In other words, XML documents have a structure format that allows data to be sent across networks in an inter-operable manner. Naturally, XML has widespread support and has become the messaging standard.

2.2 Web Services

According to IBM, Web Services are a technology that allows applications to communicate with each other in a way that is independent of which platform or which programming language is being used. A Web Service is a software interface that describes a collection of operations that can be accessed over the network through standardized XML messaging. It uses protocols based on the XML language to describe an operation to execute or data to exchange.

The Web Services model follows the publish, find and bind paradigm: a service provider publishes a Web Service in a registry called UDDI; the client searches in a registry the service which meet the requirements; and, after the result of the search, the client downloads the service description and binds with that to invoke and use the service.

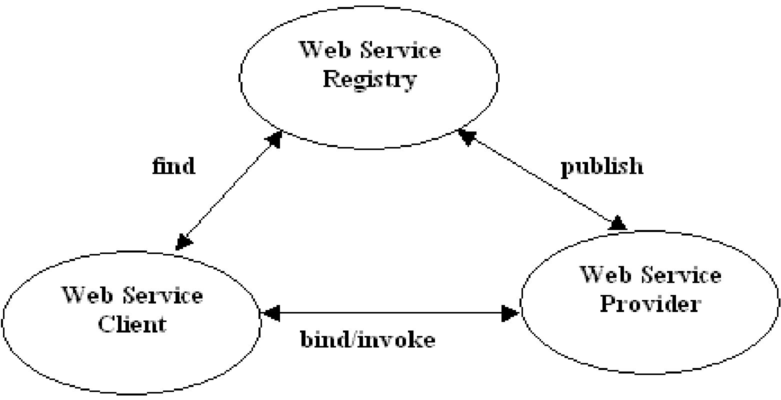


Figure 2.2: Web Service

2.3 WSDL

The Web Services Description Language (WSDL) is an XML language for describing Web Services as a set of network endpoints that operate on messages.

A WSDL service description contains an abstract definition for a set of operations and messages, a concrete protocol binding for these operations and messages, and a network endpoint specification for the binding. A WSDL document describes a web service using the following major elements:

- **type:** data type definition used to describe the messages exchanged
- **message:** represents an abstract definition of the data being transmitted; a message consists of logical parts, each of which is associated with the definition within some type system
- **portType:** a set of operations, each of which refers to an input message and an output message
- **binding:** specifies a concrete protocol and data format specifications for the operations and messages defined by a particular portType.
- **port:** specifies an address for a binding, thus defining a single communication endpoint
- **service:** used to aggregate a set of related ports

```
<definitions>
        <types>
        ……
        </types>
        <message>
        ……
        </message>
        <portType>
        ……
        </portType>
        <binding>
        ……
        </binding>
        <service>
                ……
        </service>
</definitions>
```

Figure 2.3:WSDL Structure

2.4 SOAP

Originally defined as *Simple Object Access Protocol*, SOAP is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on Extensible Markup Language (XML) as its message format, and usually relies on other Application Layer protocols (most notably Remote Procedure Call (RPC and HTTP).

A SOAP message could be sent to a Web Service enabled web site (for example, a house price database) with the parameters needed for a search. The site would then return an XML-formatted document with the resulting data (prices, location, features, etc). Because the data is

returned in a standardized machine-parse-able format, it could then be integrated directly into a third-party site.

A SOAP message consists of an envelope containing an optional header and a required body. The header contains blocks of information relevant to how the message is to be processed. This includes delivery settings, authentication or authorization assertions, and transactions contexts. The body contains the actual message to be delivered and processed. Anything that can be expressed in XML syntax can go in the body of a message.

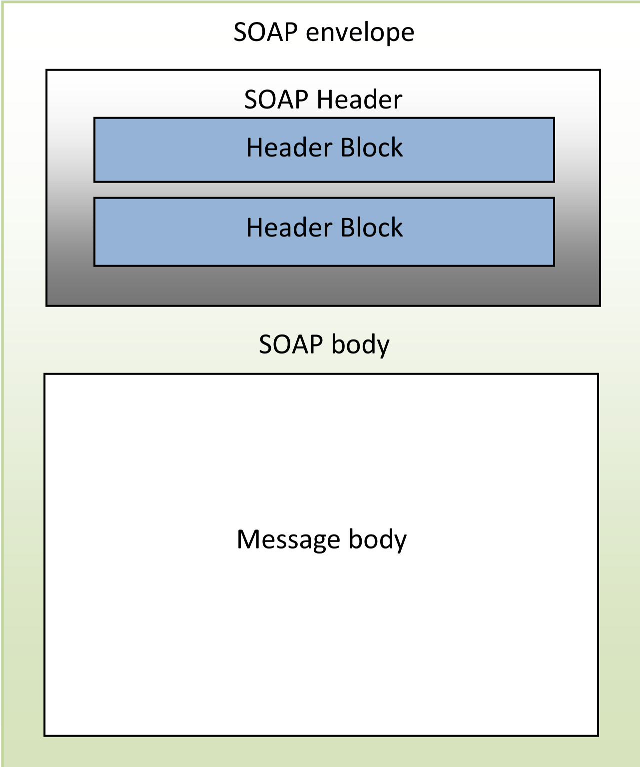


Figure 2.4:System Architecture

2.5 Document Object Model (DOM)

The Document Object Model (DOM) is an object model for XML documents that can be used to access parts of an XML document directly. In DOM, the document is modeled as a tree,

where each component of the XML syntax (such as an element or text content) is represented by a node. DOM is an API that allows you to navigate this tree, moving from parent to child node, to siblings, and more, taking advantage of special properties of certain types of nodes (for instance, elements can have attributes, while text nodes have text data). DOM is designed to be language-neutral.

DOM is generally much easier to master than Simple API for XML (SAX) because it does not involve callbacks and sophisticated state management. However, DOM implementations generally keep all XML nodes in memory, which can be inefficient for larger documents.

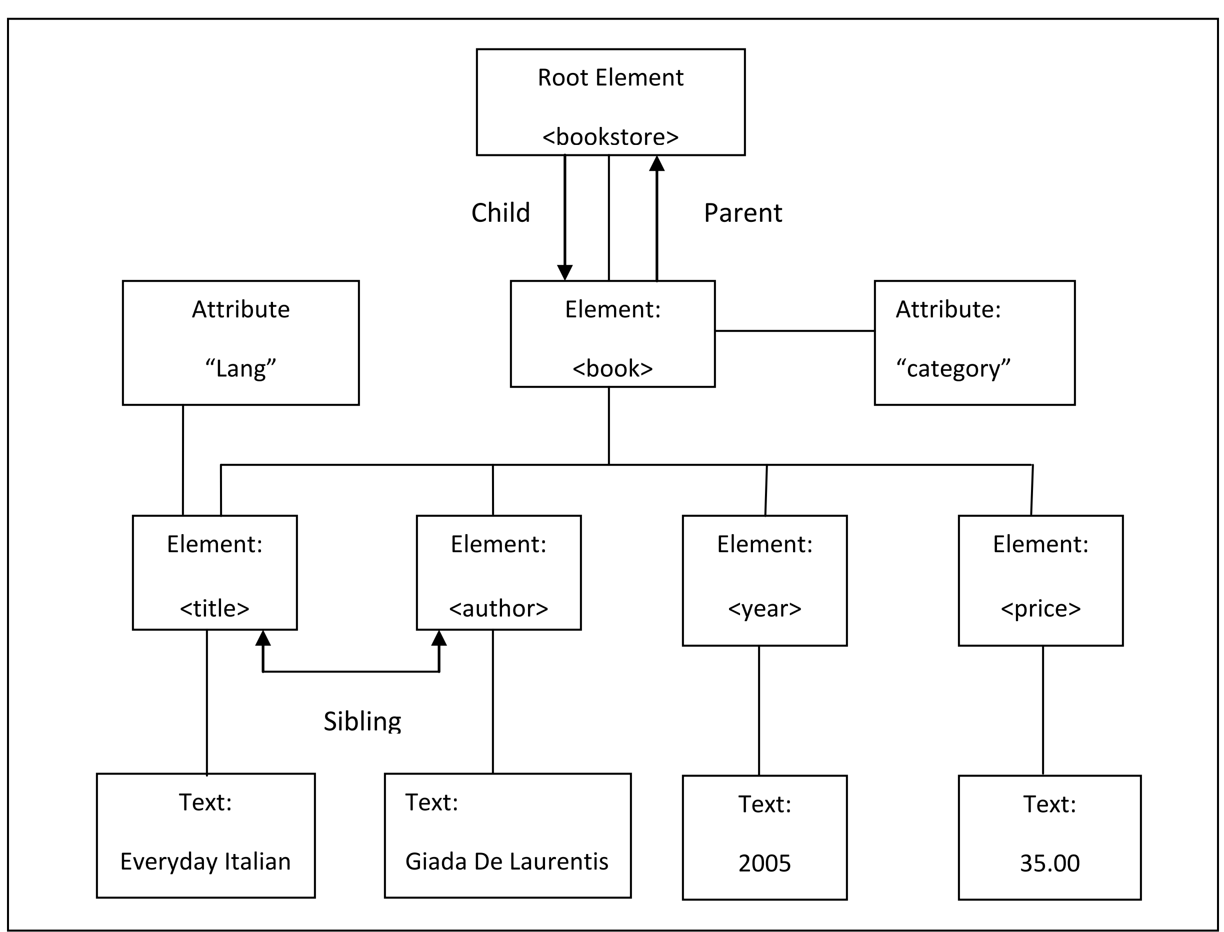


Figure 2.5: book Store Dom Tree

2.6 XPath

XPath is a query language for selecting nodes from an XML document. In addition, XPath may be used to compute values (*e.g.*, strings, numbers, or Boolean values) from the content of an XML document. XPath was defined by the World Wide Web Consortium (W3C).

The XPath language is based on a tree representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by a variety of criteria. In popular use (though not in the official specification), an XPath expression is often referred to simply as *an XPath*.

2.7 Ajax

Asynchronous JavaScript and XML (or Ajax for short) is a method of building interactive applications for the Web that processes user requests immediately. Ajax combines several programming tools including JavaScript, dynamic HTML (DHTML), Extensible Markup Language (XML), cascading style sheets (CSS), the Document Object Model (DOM), and the Microsoft object, XMLHttp Request. Ajax allows content on Web pages to update immediately when a user performs an action, unlike an HTTP request, during which users must wait for a whole new page to load. For example, a weather forecasting site could display local conditions on one side of the page without delay after a user types in a zip code.

Chapter 3. Structure and System Design

The technical goal of this research project is to realize business Web applications, in a multi-tier architecture, which includes the presentation tier (the Web browsers), the Web tier (Web server), the middle application tier and the persistent tier (the database). System view of such architecture is given on the left-hand side of Figure 3.1. It is worth emphasizing that the PHP Web Server performs both the tasks of the presentation tier and the application; not only do the PHP programs produce high quality Web pages, they can implement business rules. Typically, these implementations of business rules are encapsulated in PHP functions. Therefore, the "PHP Web Server" is divided into "presentation" and application "service" as shown in Figure 3.1. In complex systems, some business logic or information resource is given as a Web Service and the PHP programs need to interact with Web Services. The Web Service could be chosen due to many reasons such as interoperability and security among others. As the Web Service has become the de facto technology integration software. "our special situation" will become more and more common.

A number of methods exist for PHP programs to interact with Web Services, which will be discussed in Section 3. What we will have achieved is not only to choose the best method for PHP-WS interaction, but also to automate the process for PHP program to call Web Services. The capabilities of doing so are encapsulated in WS-PHP middleware components shown at the upper right corner of "PHP Web server" in Figure 3.1.

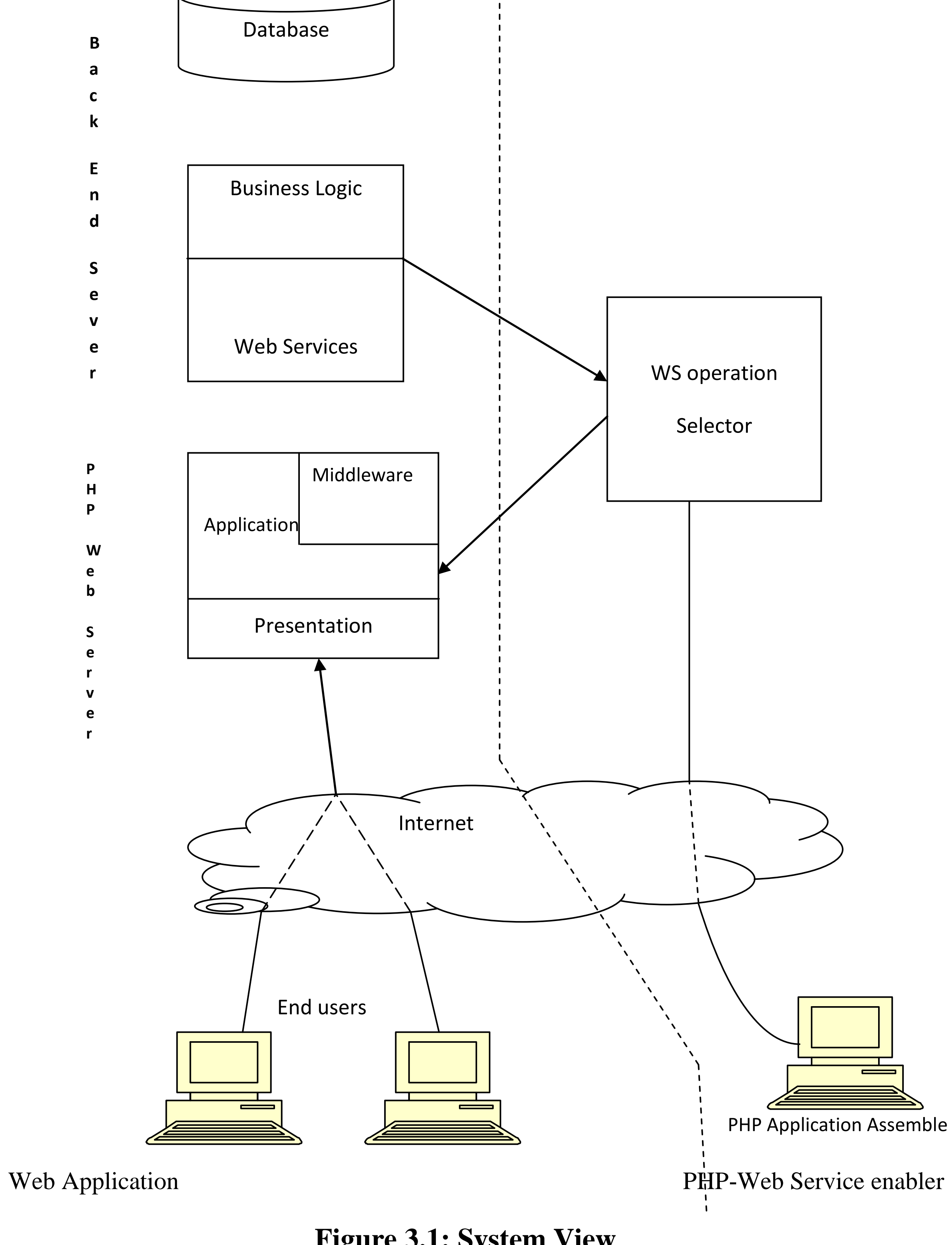


Figure 3.1: System View

3.1 Web Services

A Web Service usually presents a number of operations. These operations are the contact points between the back-end logic (often complex and proprietary) and the external clients. The service and the client exchange SOAP messages. The strength of Web Service is that the service's operations are formally specified in an industrial standard.

The Web Service Description Language (WSDL) can be processed programmatically. Thus along with each Web Service, there is always a WSDL document.

For example, in one of the experimental implementations, the Car service has four operations:

- Car [] getDefault().
- Car [] getCar(String make, String model, int year).
- Store [] getStore(CarArg car).
- Car [] getSelectedCar(String make, String model, int year, String year_from, String year_to, String price_from, String price_to).

These four operations are specified in the WSDL document as four operation elements as shown in Figure 3.2.

Store
+name: String +address: String +city: String +State: String +Zip: String +store_id: int
+getName():String +setName():void etc..

Figure 3.2: Class Store Diagram

```
wsdl:portType name="MyCarServicePortType">

- <wsdl:operation name="getDefault">

<wsdl:input xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
message="axis2:getDefaultMessage" wsaw:Action="urn:getDefault" />

<wsdl:output message="axis2:getDefaultResponse" />

</wsdl:operation>

- <wsdl:operation name="getCar">

<wsdl:input xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
message="axis2:getCarMessage" wsaw:Action="urn:getCar" />

<wsdl:output message="axis2:getCarResponse" />

</wsdl:operation>

- <wsdl:operation name="getStore">

<wsdl:input xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
message="axis2:getStoreMessage" wsaw:Action="urn:getStore" />

<wsdl:output message="axis2:getStoreResponse" />

</wsdl:operation>

- <wsdl:operation name="getSelectedCar">

<wsdl:input xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
message="axis2:getSelectedCarMessage" wsaw:Action="urn:getSelectedCar" />

<wsdl:output message="axis2:getSelectedCarResponse" />

</wsdl:operation>

</wsdl:portType>
```

Figure 3.3: Specification of operations in WSDL

It should be noted that the operation getDefault has no parameters. The operations getCar, getStore and getSelectedCar have parameters.

The operation getCar has three parameters: make, model as String and year as **int**. It will fetch the database for the tuples satisfying the condition of the three parameters and return an array of object Car .Operation getDefault queries the database for the six cheapest cars and returns an array of Car data type. Operation getStore retrieves the stores satisfying the characteristics of the car entered in the parameters. It takes as a parameter an object of type CarArg. Operation getSelected also fetches the cars satisfying certain parameters, with more options, returning an array of Car objects, taking in seven parameters, including year_from, year_to, price_from and price_to, which are defined as String because they are used in the SQL statement.

Some of the parameters and return values have complex object structures. For example, the Car data type is an object, and one of its instance variables is a Shape object, which in turn has, as one of its instance variables, another object of the Dimension class. On the other hand, the Store object is a simple one with all its instances variables being simple types. For the arguments, a very complex type that is three levels deep as the Car data type are used, the structure of this object is illustrated in class diagram.

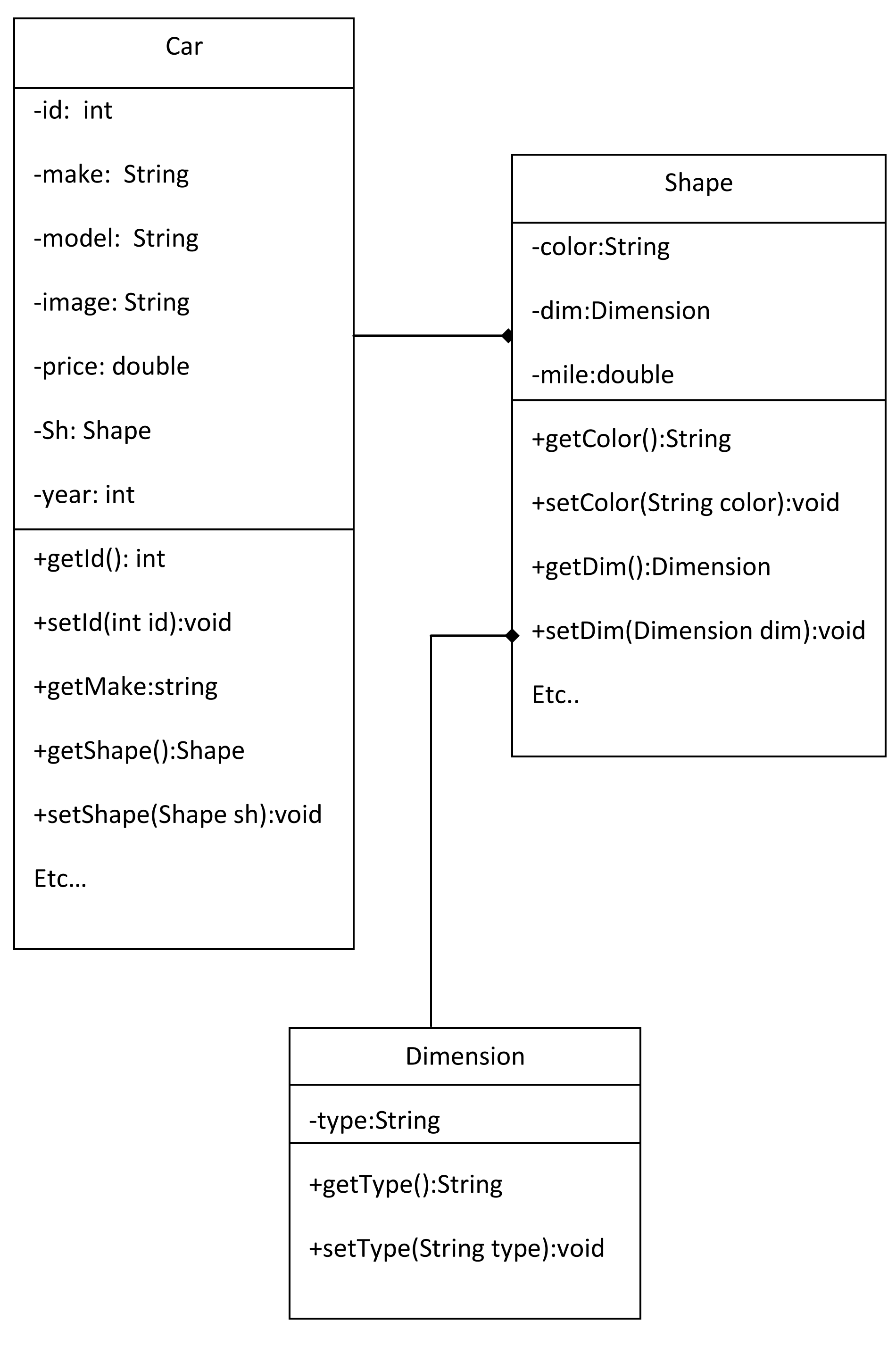


Figure 3.4: Class Car Diagram

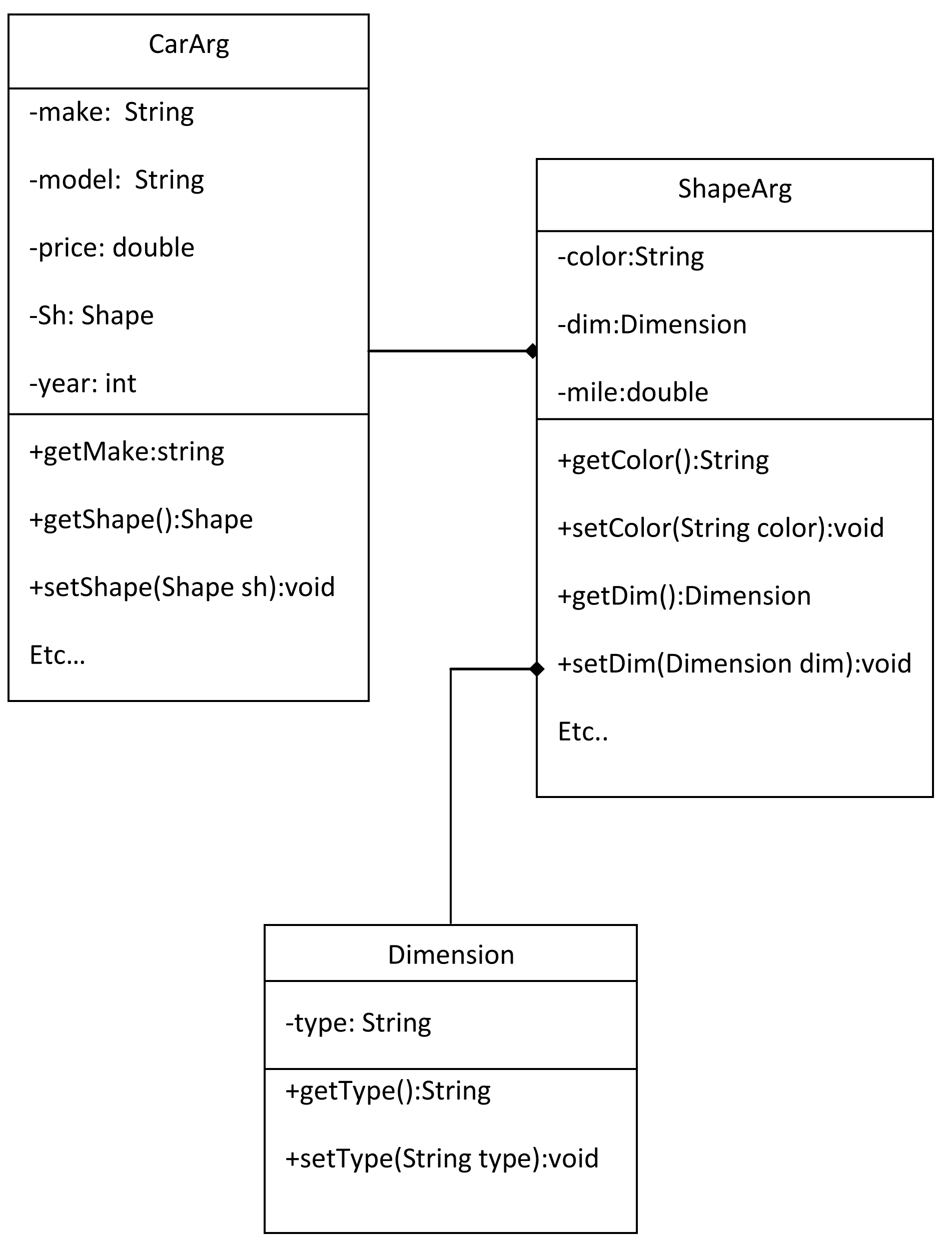


Figure 3.5: Class CarArg Diagram

3.2 Front End

The front end for the system consists of Web pages interacting with the users. For the end users, one static HTML page only is used. All the following displays and data entry forms are dynamically created by JavaScript programming. The communications (requests and responses) between the front end and the PHP engine on the Web server are handled by Ajax technology, which allows for the user's inputs and server's responding data to not cause Web pages to be reloaded. Figure 3.6 illustrates the calling relationship between the front end and the Web server.

Additionally, there is one Web page for the company management to select which services should be exposed to the end users (customers). Upon receipt of this choice, a set of front end templates are used to form suitable user interfaces.

When the end user's requests require triggering the Web Services at the back end, a bridge component in the PHP server is needed. This is the WS-PHP middleware component described in the next section.

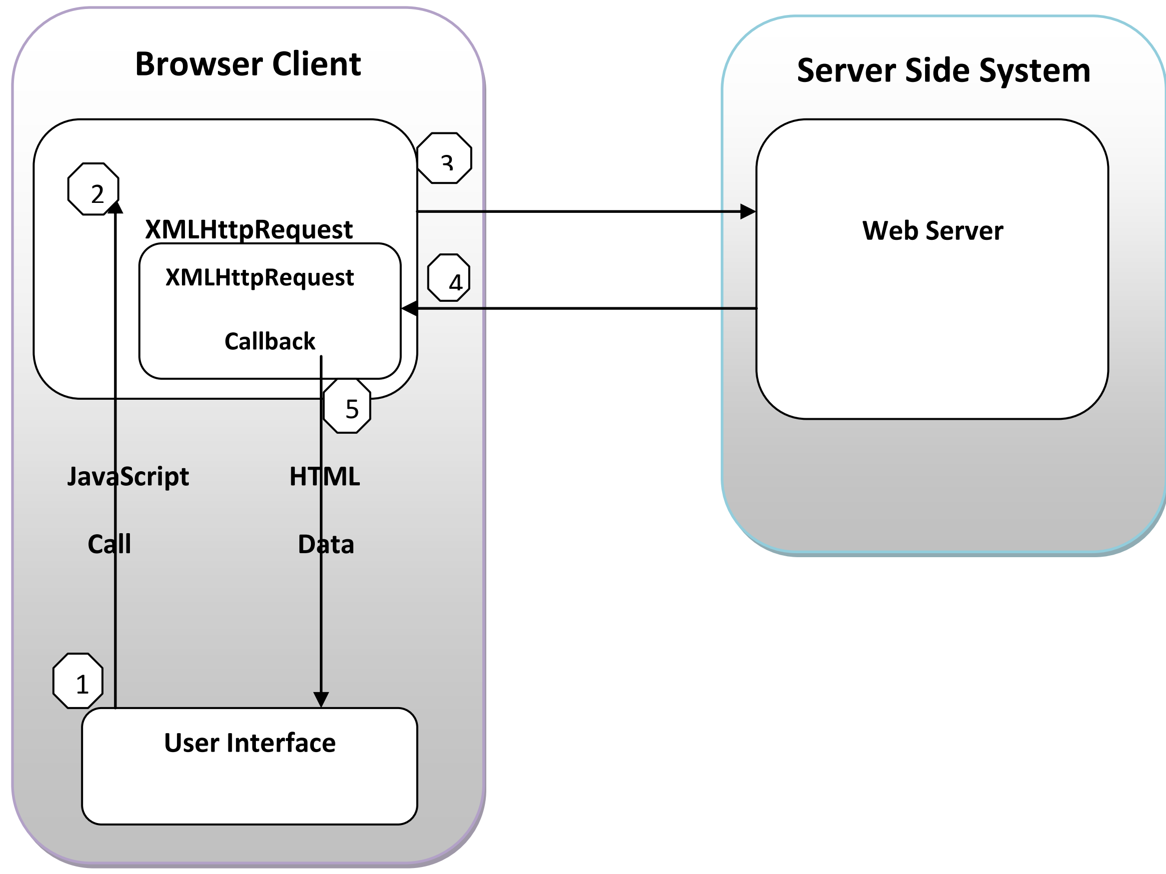


Figure 3.6: Ajax Call

3.3 The WS-PHP Middleware component

The WS-PHP middleware component has many responsibilities such as formulation of request, validation of data, calling the Web Service, and formatting the results from the Web Service for the client. The middleware component is to form the call to the operations and parse the result of the call solely based on the WSDL.

Before any further actions, recognizing the important elements in WSDL document, namely the portTypes, messages and the types, need to be done.

3.3.1 WSDL elements

- **Element Operations**

These are the operations in the Web Service, located in the portType element, of the WSDL belonging to the namespace "wsdl". This element has attribute name of value 'getCar'. This operation has two Child elements, input and output, which has attribute message as shown in Figure 3.7.

```
<wsdl:operation name="getCar">
  <wsdl:input xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
message="axis2:getCarMessage" wsaw:Action="urn:getCar" />
  <wsdl:output message="axis2:getCarResponse" />
</wsdl:operation>
```

Figure 3.7: Operation element in WSDL

- **Element message**

This constitutes the request and the response and belongs to "wsdl" namespace and is an element of definitions element. Each message element might have as attributes name and child part, which has attribute element. In Figure 3.8, element has a value ns0:getCar.

```
<wsdl:message name="getCarMessage">
<wsdl:part name="part1" element="ns0:getCar" />
</wsdl:message>
```

Figure 3.8: message element of WSDL

```
<wsdl:message name="getCarResponse">
  <wsdl:part name="part1" element="ns0:getCarResponse" />
</wsdl:message>
```

Figure 3.9: message response element of WSDL

- **Element types**

This concerns the description part of the data type used in the messages, the value of the attribute element in the element part of message must be one of the descendants of element types.

```
<xs:element name="getCar">
- <xs:complexType>
- <xs:sequence>
  <xs:element name="make" nillable="true" type="xs:string" />
  <xs:element name="model" nillable="true" type="xs:string" />
  <xs:element name="year" nillable="true" type="xs:int" />
  </xs:sequence>
  </xs:complexType>
  </xs:element>
```

Figure 3.10: element of type WSDL

3.3.2 Services operations

In this study, three kinds of operations appear in WSDL documents:

1. **Operations with parameters and return value**

In this case, all the attributes and children nodes will be present as shown in Figure 3.11:

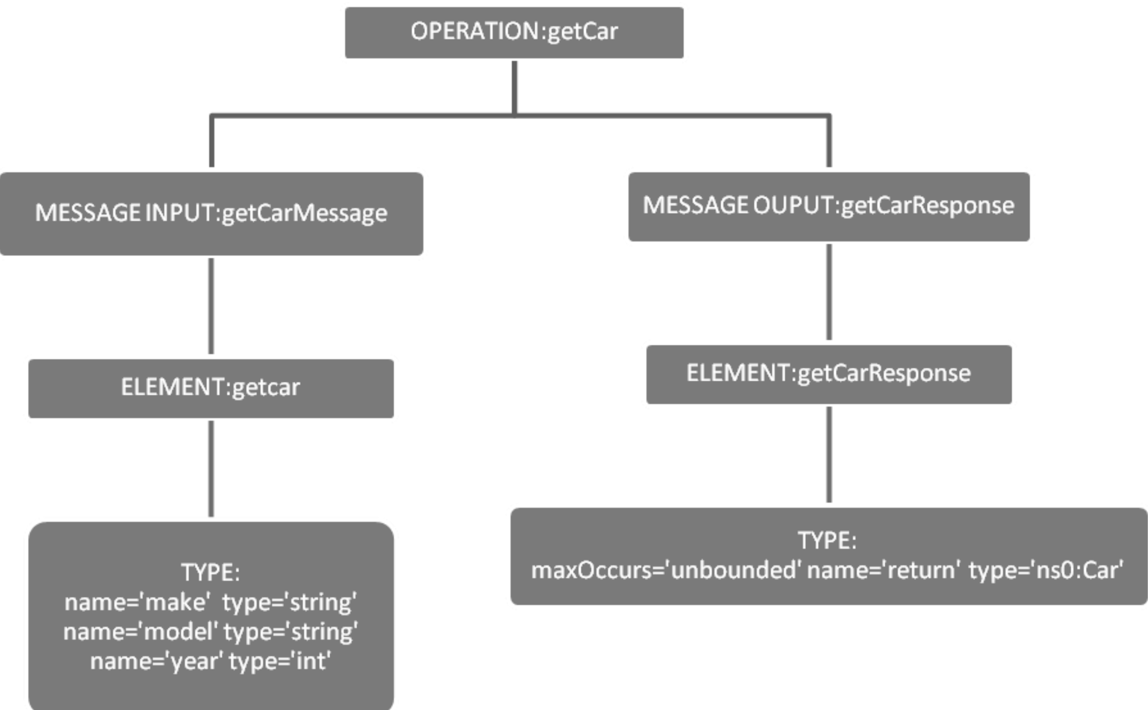


Figure 3.11: operation that has all WSDL elements

2.**The operations that have no parameters but have a return value**

In this case, the message input has no child node:

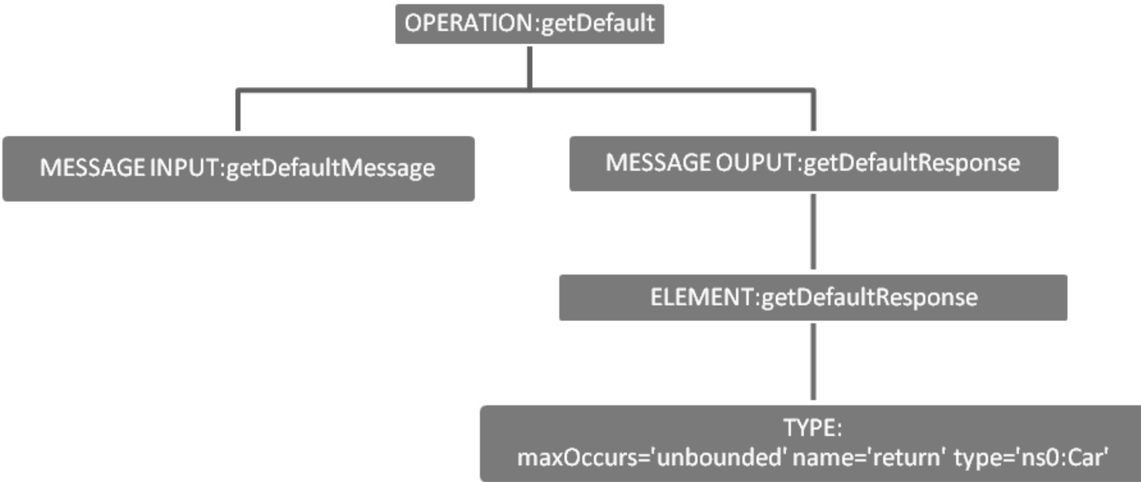


Figure 3.12: operation with no parameter

3.**The operations that have parameters but have no return value**

In this case the output message does not exist:

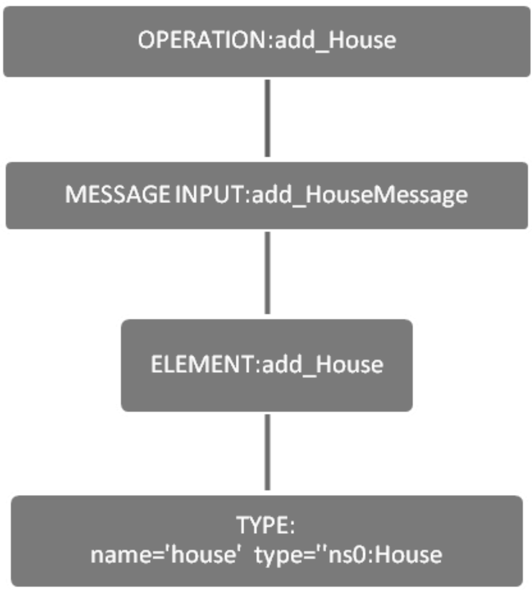


Figure 3.13: operation without return value

3.3.3 Automating the Process that enables PHP to call Web Services

3.3.3.1 Existing PHP-WS interactions approaches

There are a number of approaches to PHP-WS interactions, namely:

1. **Pear:SOAP:** The Pear package has a number of modules. SOAP is one of them. To use Pear:SOAP, the programmer must include this module in his application, and needs to be installed, separately.
2. **NuSOAP:** This is a library used to convert PHP data types to proper XML Schema types. It had enjoyed huge attention in the previous PHP version (version 4) when there

was not many options for PHP-WS interaction. However, its latest version (0.7.3) supports neither SOAP 1.2 nor any WS-* specification [NuSOAP].

3. **WSO2's WSF/PHP**: This is an external extension of PHP that could be used to provide and consume Web Services [WSF/PHP]. This extension could be used in providing and consuming Web Services in PHP. WSO2 WSF/PHP is a complete solution for building and deploying Web Services and is the only PHP extension with most extensive implementations for the widest range of WS-* specification. Its key features include secure services and clients with WS-Security support, binary attachments with MTOM, automatic WSDL generation (code first model), WSDL mode for both services and clients (contract first model) and interoperability with .NET and J2EE (WSO2.org). However, this external extension is often not feasible to apply because an external shared hosting environment will not allow recompile and reconfigure the Web server.
4. **PHP's native SOAP extension (PHP SOAP):** This is available if the SOAP option is turned on. That is, the PHP installation is configured with " **--enable-soap**" (for example, **./configure --with-xml --with-mysqli --enable-soap**). This PHP native solution can be used to write SOAP server and client. Fortunately, most shared hosting environments that support PHP5 conventionally have this SOAP feature enabled. After the evaluation of the existing PHP-WS interaction approaches, the PHP's native SOAP extension approach is chosen.

3.3.3.2 The Procedures for PHP to call Web Service operations

The task of the PHP-WS middleware component is to make the call to the Web Services operations based on three kinds of operations. The procedures for making calls to these three kinds of operations are specified:

List1: Calling operation that has no parameters but has return value

This is the simplest case in which only two steps are needed.

Step1: Make Request to the Web Service

Step2: Parse the results, format them to be sent to the placeholder "content" in the user interface.

List: 2. calling operation with parameters but without return value (the void operations)

Step 1: Parse the type of element of message input.

Step 2: Format the result of the parsing and display an HTML form with elements the parameters of the operation in the placeholder "menu" in the user interface.

Step3: In the client browser, the user enters his data.

Step4: If the data are valid

Step5: Make Request to the Web Service.

Step6: Display a message that the data been updated in the content in the user interface.

Step7: If not, display the errors, in the content div, repeat 3.

List: 3. calling operation with parameters and return value

Step1: Parse the type of element of message input.

Step2: Format the result of the parsing and display an HTML form with elements the parameters of the operation in the placeholder "menu" in the user interface.

Step 3: In the client browser, the user enters his data.

Step4: If the data are valid

Step5: Make Request to the Web Service.

Step6: Parse the result, format them to be sent to the placeholder “content “in the user interface.

Step 7: If not, display the errors, in the content div, repeat 3.

3.3.4 The Structure of the WS-PHP middleware

Corresponding to the three kinds of Web Service calls, three PHP functions are implemented:

- HandleRequest_WP: This functions handles the request to a Web Service operation that has parameters in the call. This also handles the void operation.
- HandleRequest_NP: This function handles the request to Web Service operation that does not have parameters.
- HandleParameters: This intermediate function seeks to complete the request to the Web Service operation that has parameters.

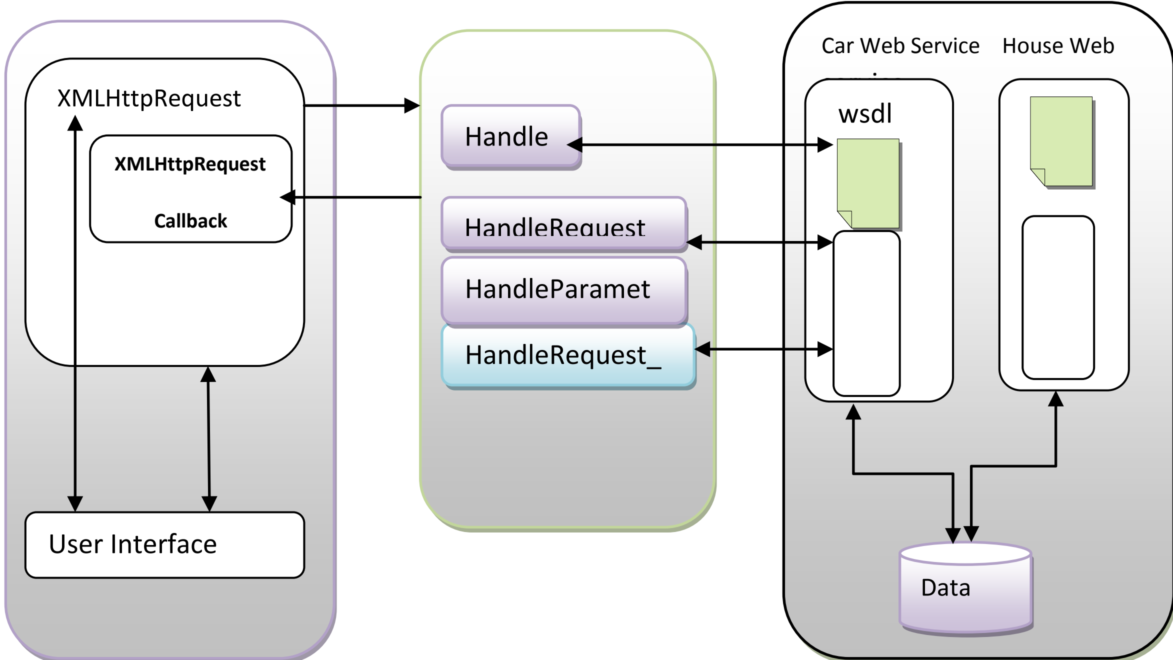


Figure 3.14: System Details

The role of PHP functions in the system is depicted in Figure 3.14. When the front end makes an Ajax request to the PHP server and this request eventually leads to a call to the Web Service, one of the above PHP functions will be called depending on the type of the Web Service operation. In Figure 3.14 "Handle1" is another PHP function that can fetch the WSDL document of the Web Service from the back end server and help the management to select an operation. The use of this will be shown later.

The tasks of HandleRequest_WP are to fulfill the steps listed in List 3 of Section 3.3.3.2. Its sequence diagram is shown in Figure 3.16.

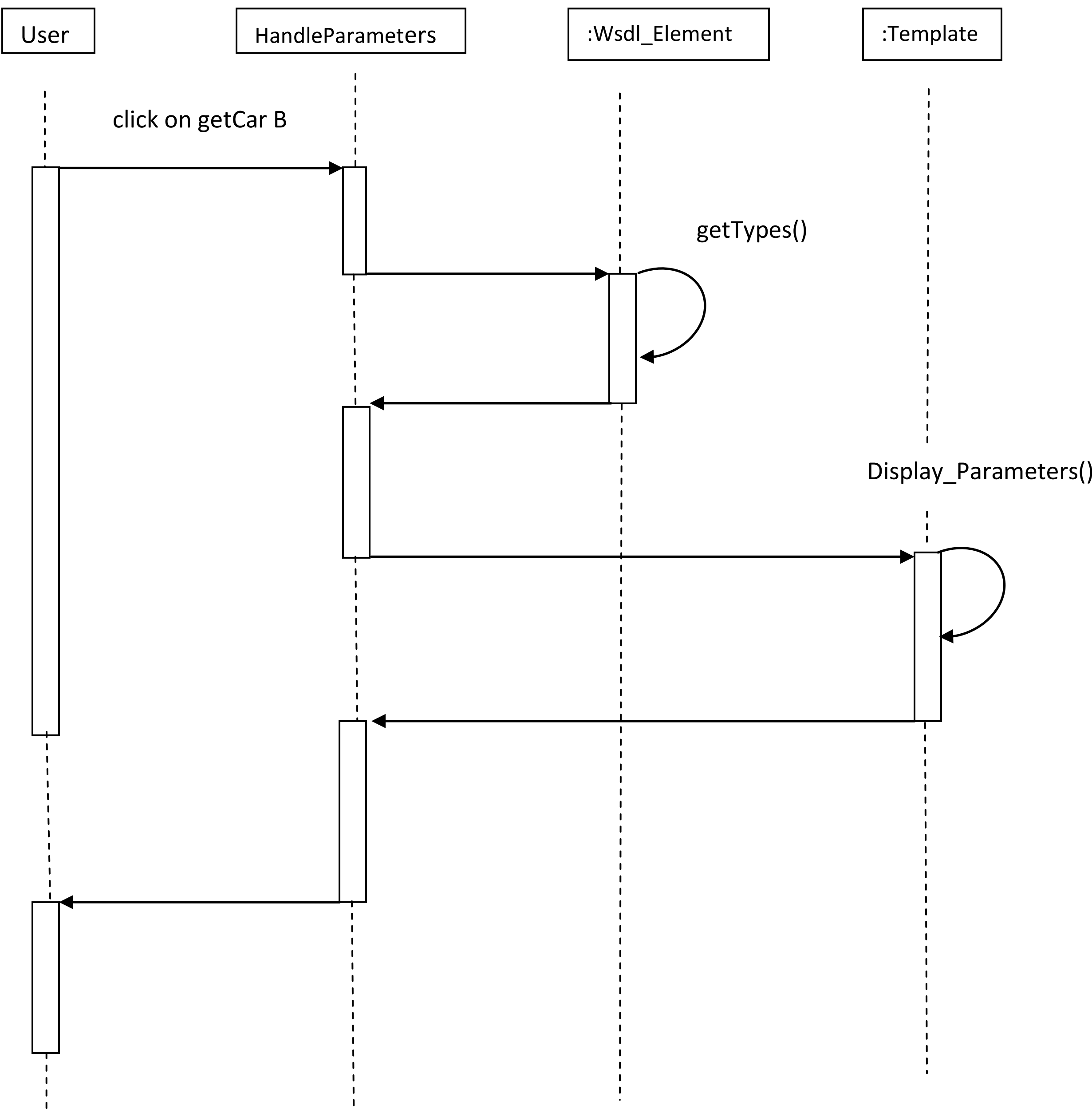


Figure 3.15:Sequence Diagram HandleParameters

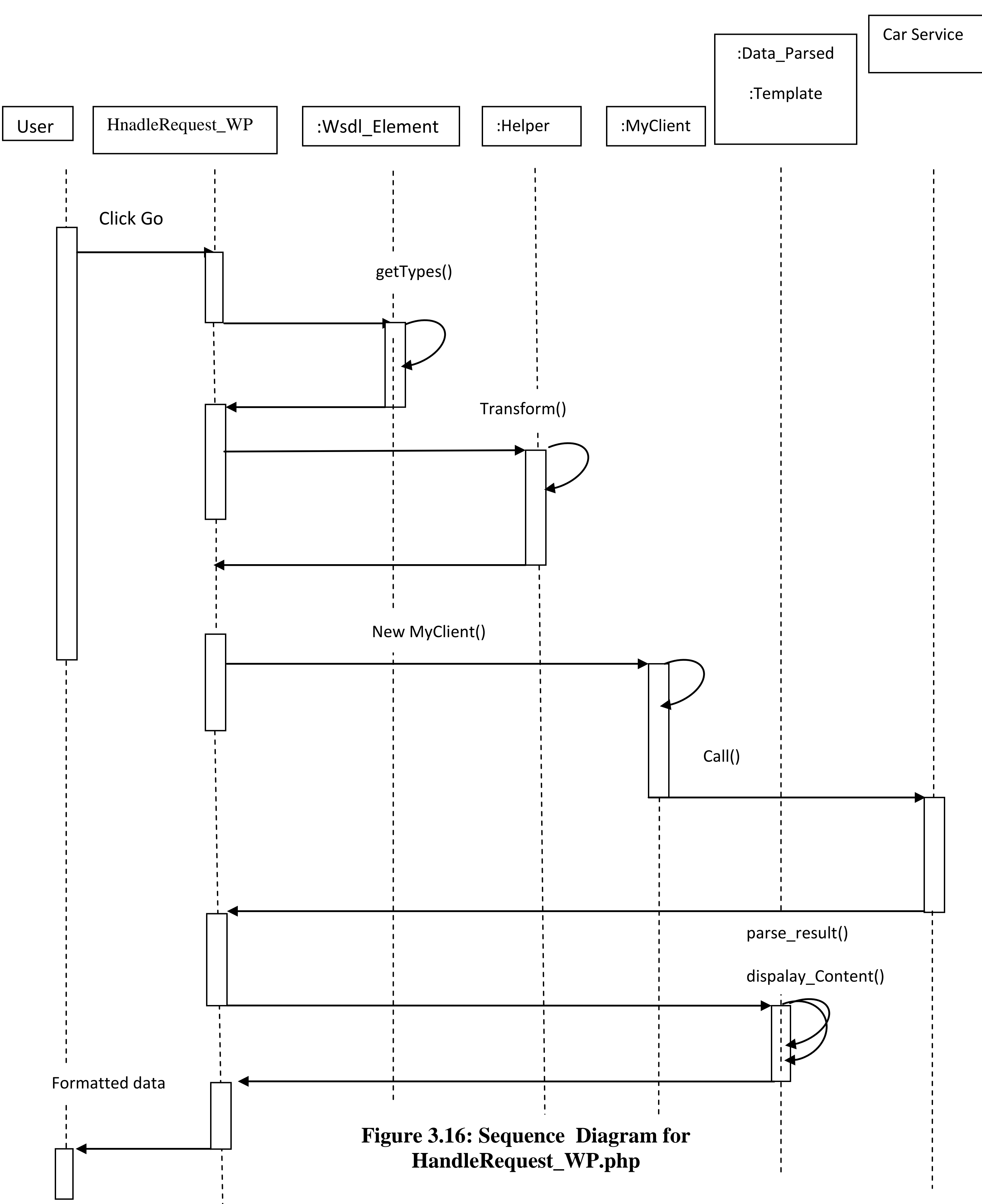


Figure 3.16: Sequence Diagram for HandleRequest_WP.php

The tasks of HandleRequest_NP is to fulfill the steps listed in List 1 of Section 3.3.3.2. Its sequence diagram is given in Figure 3.17:

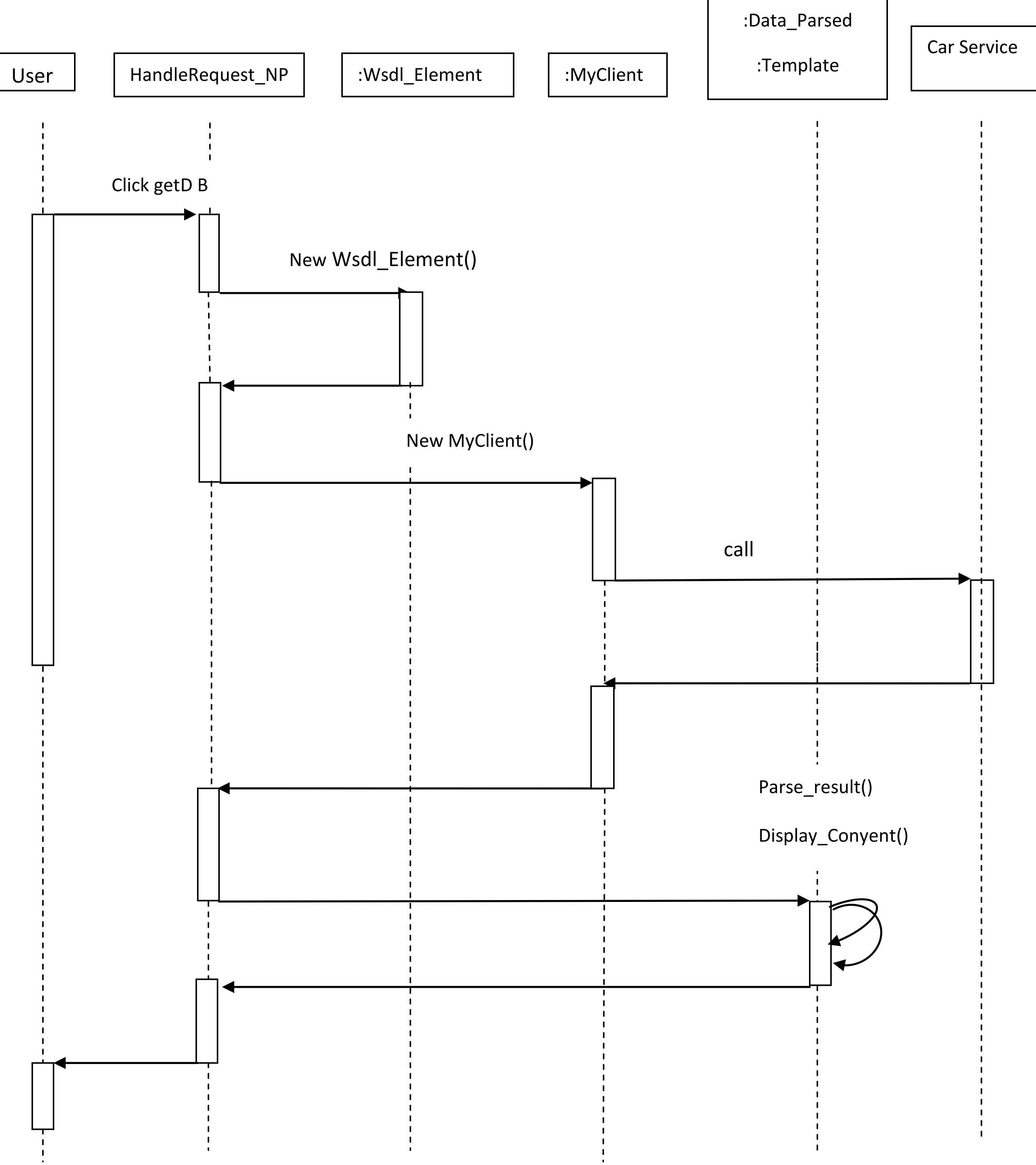


Figure 3.17: Sequence Diagram HandleRequest_NP

Chapter 4. Implementation

According to the system design described in Chapter 3, the WS-PHP middleware is the component that automates the process for a PHP program to call a Web Service. Using this component, two Web applications, Car Sales and House Sales are implemented. Each program relies on back end Web Services *Car Service* and *House Service* respectively. This is illustrated in Figure 4.1.

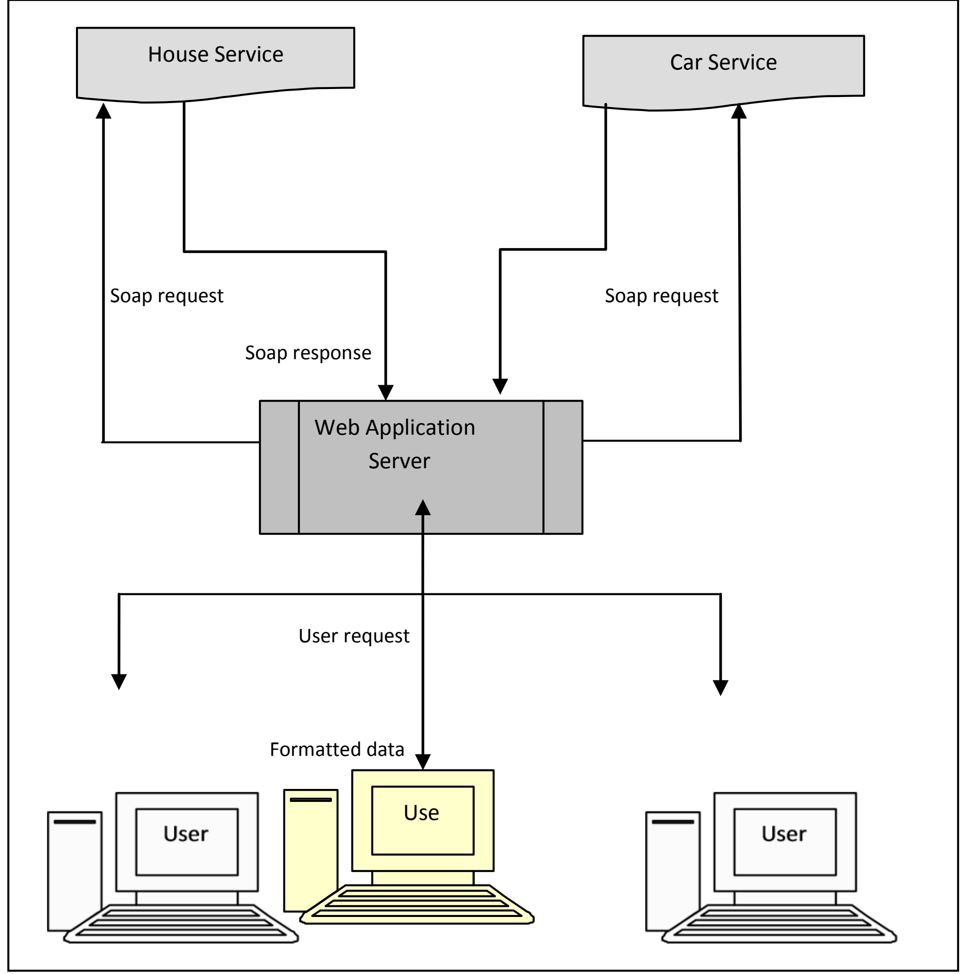


Figure 4.1: System View

4.1 Client implementation

In the first page of the Web application, a set of JavaScript functions handle the events. The Ajax technology uses the XMLHttpObject which has to be instantiated. It communicates directly with the PHP server. All modern browsers such as "chrome, IE7, Firefox, Opera, Safari" have XMLHttpObject built-in. Too, JavaScript IE5 and IE6 use ActiveXObject.

```
function getHTTPObject(){
if(window.ActiveXObject) return new ActiveXObject("Microsoft.XMLHTTP");
else if(window.XMLHttpRequest) return new XMLHttpRequest();
else{
    alert("the Browser doesn't support AJAX");
            return null;
   }
}
```

Figure 4.2: Function getHTTPObject()

Two kinds of requests are made by the client, namely methods GET and POST. The open() and the send() are the methods of the XMLHttpObject to send off the request to the server. onreadystatechange() is a property that stores the function that will process the response from the server. The event handler makeRequestGet(url,val,x), shown in Figure 4.3, uses GET, sends the value x and sets the Placeholder or div of id='val' for the output of the results. This function calls three other functions, respectively: setOutputRes,

setOutputRes1 and setOutputRes2 depending on the value of a parameter val: “res , menu, content”.

```
function makeRequestGet(url,val,x){
s1=document.getElementById("res");
 s2=document.getElementById("menu");
s3=document.getElementById("content");
s2.innerHTML='';
 s3.innerHTML='';
var str5=x;
httpObject=getHTTPObject();
if(httpObject !=null){
str=url+'?data='+str5;
httpObject.open("GET",str,true);
httpObject.send(null);
if(val=='res'){
        httpObject.onreadystatechange=setOutputRes;
                }
if(val=='menu')
httpObject.onreadystatechange=setOutputRes1;
if(val=='content')
httpObject.onreadystatechange=setOutputRes2;
}
}
```

Figure 4.3 : Event handler using GET

The functions setOutputRes, setOutputRes1, setOutputRes2 and alertContents have the same structure. They differ only in where to place the output coming from the PHP server. alertContents is used in the makeRequestPost function. The code for function setOutputRes is shown in Figure 4.4.

```
function setOutputRes() {
 if (httpObject.readyState == 4) {
        if (httpObject.status == 200) {
             result = httpObject.responseText;
             document.getElementById('res').innerHTML = result;
        }
else {
          alert('There was a problem with the request.');
        }
     }
  }
 function alertContents() {
 if ( httpObject.readyState == 4) {
        if ( httpObject.status == 200) {
             result =  httpObject.responseText;
             document.getElementById('content').innerHTML = result;
        }
else {
          alert('There was a problem with the request.');
        }
     }
  }
```

Figure 4.4: Client functions which process the php response results

The function makeRequestPost(url, parameters) uses the POST method, and sends the input values of the HTML form. The difference between makeRequestPost and makeRequestGet is that the former is a statement in the event handler show(formid, filep)

but the latter is itself an event handler. In addition, POST request has to set some parameters in the HTTP header. This function is shown in Figure 4.5.

```
function makeRequestPost(url,parameters){
httpObject=getHTTPObject();
if(httpObject!=null){
    httpObject.onreadystatechange = alertContents;
    httpObject.open('POST', url, true);
    httpObject.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
    httpObject.setRequestHeader("Content-length", parameters.length);
    httpObject.setRequestHeader("Connection", "close");
    httpObject.send(parameters);
}
}
```

Figure 4.5: Client functions which process the php response results

The function show is a handler that has two arguments: the id of an HTML form, and the processing program formid, filep as shown in Figure 4.6. The function createQuery shown in Figure 4.7 returns an array an HTML form elements.

```
function show(formid,filep) {
    var poststr = createQuery(document.getElementById(formid));
    makeRequestPost(filep, poststr);
  }
```

Figure 4.6: Event Handler for POST method

```
function createQuery(form){
    var elements = form.elements;
    var pairs = new Array();
    for (var i = 0; i < elements.length; i++) {
      if ((name = elements[i].name) && (value = elements[i].value))
        pairs.push(name + "=" + encodeURIComponent(value));
    }
    return pairs.join("&");
  }
```

Figure 4.7: Client function returning element of a form

The client event handlers appear in the HTML forms. First, a call for handler is made at the static form as shown in Figure 4.8. Then the rest of the calls appear in the dynamic form results.

```
<form id="services1">

<table id="table1" width=90%  align="center">

<tr>

<td>Car services :<input type="radio"  name="service" value='0'
onClick="makeRequestGet('Handle1.php','res',this.value);"></td>

<td><strong>House Service:</strong><input type="radio" name="service" value='1'
onClick="makeRequestGet('handle1.php','res',this.value);"></td>

</tr>

</table>

</form>
```

Figure 4.8: Static form showing the event handler

4.2 Class Wsdl_Element

The core of this application resides in the class Wsdl_Element, which extracts the different elements from the WSDL document. This class has an instance variable dom, a DOMDocument object, for use by the API DOM, and an instance xpath of an XPATH object for querying the document. The constructor function of Wsdl_Element is shown in Figure 4.9.

```
public function __construct($wsdl){

        self::$dom=new DOMDocument();

        self::$dom->load($wsdl);

        self::$xpath=new DOMXPath(self::$dom);

        self::$xpath->registerNamespace("wsdl","http://schemas.xmlsoap.org/wsdl/");

        self::$xpath->registerNamespace("xs","http://www.w3.org/2001/XMLSchema");

}
```

Figure 4.9: Constructor of the Wsdl_Element class

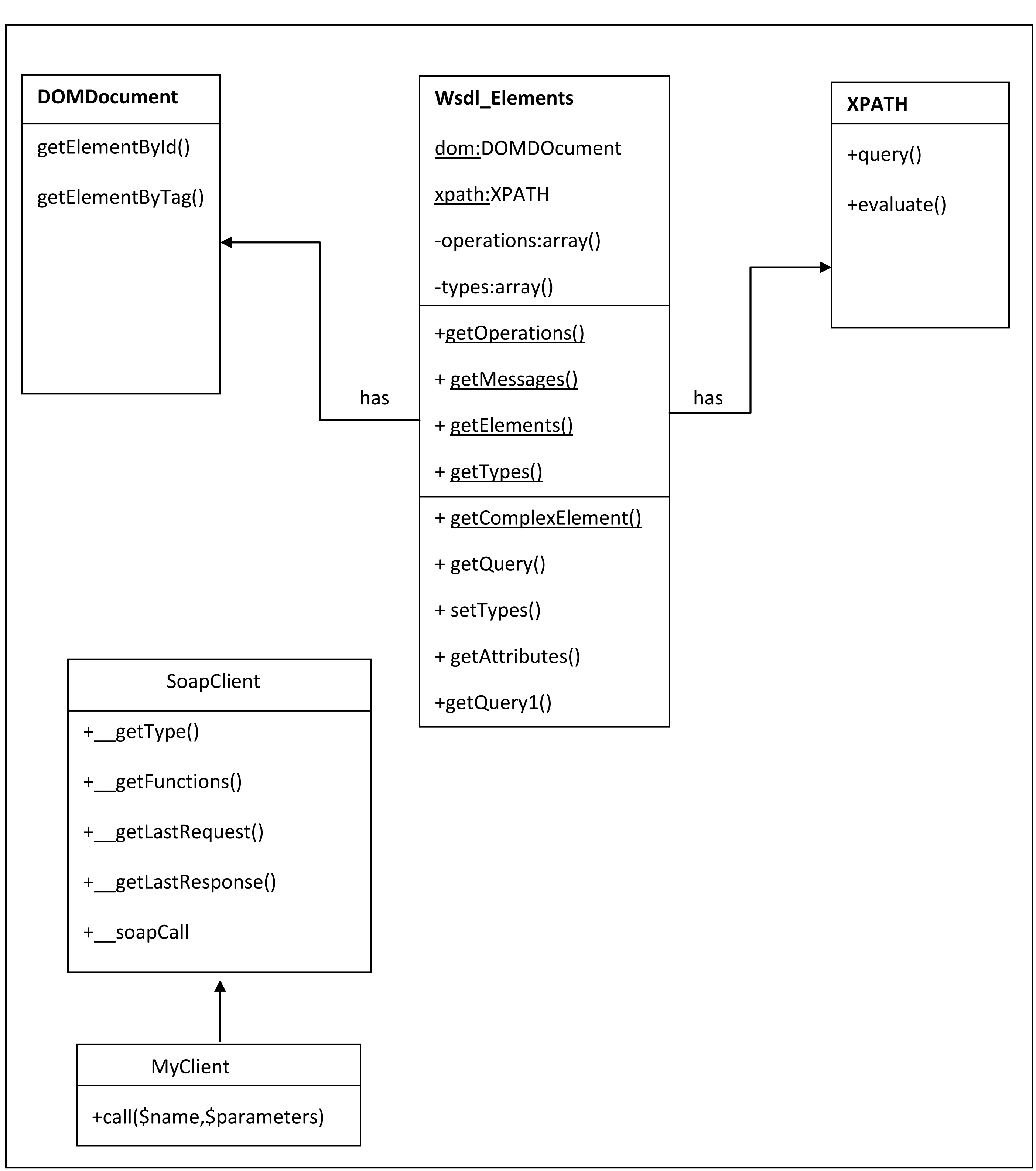


Figure 4.10: Class diagram

The function getMessages returns an array of the messages of an operation. The function getElements returns an array of elements for a certain message of the WSDL document. The function getOperations retrieves all the operations "Web Service operations" of the document as shown in Figure 4.11. The parsing of the portType element of WSDL produces an array of all the operations provided by the Web Service. For example, from the Car Service application, the array shown in Figure 4.12 is returned .

```
public static function  getOperations(){
        $tmp=array();
        $nodes=self::$xpath->query("wsdl:portType/wsdl:operation");
        if($nodes){
                foreach($nodes as $node){
                        if($node->hasAttributes()){
                                $att=self::$xpath->query("@name",$node);
                                $tmp[]= $att->item(0)->nodeValue;
                        }
                }
        }
return $tmp;
}
```

Figure 4.11: Method which parse the operation in wsdl document

```
Array (

[0] => getDefault

 [1] => getCar

[2] => getStore

[3] => getSelectedCar

)
```

Figure 4.12: Result of the getOperation method

```
<wsdl:types>
<xs:schema xmlns:ns="http://pack/xsd" attributeFormDefault="qualified"
           elementFormDefault="qualified" targetNamespace="http://pack/xsd">
<xs:element name="getCar">
<xs:complexType>
<xs:sequence>
<xs:element name="make" nillable="true" type="xs:string"/>
<xs:element name="model" nillable="true" type="xs:string"/>
<xs:element name="year" nillable="true" type="xs:int"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="getCarResponse">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" name="return" nillable="true" type="ns0:Car"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Car" type="ns0:Car"/>
<xs:complexType name="Car">
<xs:sequence>
<xs:element name="id" type="xs:int"/>
<xs:element name="image" nillable="true" type="xs:string"/>
<xs:element name="make" nillable="true" type="xs:string"/>
<xs:element name="model" nillable="true" type="xs:string"/>
<xs:element name="price" type="xs:double"/>
<xs:element name="sh" nillable="true" type="ns0:Shape"/>
<xs:element name="year" type="xs:int"/>
</xs:sequence>
</xs:complexType>

<xs:element name="getSelectedCar">
<xs:complexType>
<xs:sequence>
<xs:element name="make" nillable="true" type="xs:string"/>
<xs:element name="model" nillable="true" type="xs:string"/>
<xs:element name="year_from" nillable="true" type="xs:string"/>
<xs:element name="year_to" nillable="true" type="xs:string"/>
<xs:element name="price_fom" nillable="true" type="xs:string"/>
<xs:element name="price_to" nillable="true" type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="getSelectedCarResponse">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" name="return" nillable="true" type="ns0:Car"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

Figure 4.13: element types in wsdl document

The types used in the WSDL might be simple or complex such as the fragment of WSDL in Figure 4.13. The method getTypes has one argument called elements which is the elements of the message, return a simple associative array in case of simple type, the key is the name of the parameter and the value is the type of the parameter. In case of complex types, recursive calls are used, an associative array of associative arrays is returned which reflects in the types used in the operations. The code of function getTypes is shown in Figure 4.14.

```
public  function getTypes($elements){
$res=array();$val  ="";
foreach($elements as $elt){
          $attributes=$elt->attributes;
          foreach($attributes as $atr){
                    switch($atr->name){
                              case "name" :
                                        $val=$atr->value;
                                        break;
                              case "type" :
                                        $num=strpos($atr->value,':');
                                        if ($num !=0){
                                                  $str=substr($atr->value,0,$num);
                                                  if($str=="xs"){
                                                            $res[$val]=substr($atr->value,$num+1);
                                                  }
                                                  else{
                                                            $t=self::getComplexElement(substr($atr->value,$num+1));
                                                            $res[$val]=$this->getTypes($t);
                                                  }
                                         break;
                                        }
               }
  }
return $res;
}
```

Figure 4.14: Method getType

For example, in the *Car Service* application, if the management user clicks on the element of list getCar, the types carried by this operation are shown in Figure 4.15. They are the same as the types of the Web Service operation.

```
Array (
 [make] => string
[model] => string
 [year] => int
 )
```

Figure 4.15: types of getCar operation

In the back-end server, the types corresponding to the WSDL, shown in the Java interface of the method getCar (cf. Figure 4.16). So the element of the getTypes is the parameters in the getCar method running in the back-end service.

```
public Car[] getCar(String make,String model, int year)
```

Figure 4.16: Signature of the getCar Web Service method

If it is a complex type, such as those in operation getStore, PHP has to use an array of arrays to represent object graphs.

```
Array (
 [car] => Array ( [make] => string [model] => string [price] => double
[sh] => Array ( [color] => string
[dim] => Array ( [type] => string ) ) [year] => int ) )
```

Figure 4.17: operation with complex type

For example, the back-end Java method getStore carries an object of CarArg as shown in Figure 4.18. Type CarArg contains ShapeArg type which in turn contains class Dimension as shown in Figure 4.19. In PHP, such a composite object will be represented as three arrays shown in Figure 4.17. The conversion from the operation's composite types to PHP array representations is not from Java to the PHP, but from the type element in the WSDL document to PHP because the back-end program has been wrapped as Web Services.

Figure 4.18: Signature of the getStore web service method

```
public class CarArg {

      private String make;
      private String model;
      private int year;
      private ShapeArg sh;
      private double price;

}

public class ShapeArg {
      private String color;
      private Dimension  dim;

}

public class Dimension {
private String type;

}
```

Figure 4.19: Definition of the complex object CarArg

So, in the elements of the types of WSDL, each time the program finds an object, a recursive call is made and another array is created inside the precedent.

The getAttrributes method returns an associative array with key the name of attribute and its value the value of the attribute as shown in Figure 4.20.

```
public  function getAttributes($op){
$tmp=array();if($tmp1=$this->getQuery($op)){
        if($tmp1->item(0)->hasAttributes()){
                $values=$tmp1->item(0)->attributes;
                foreach($values as $val)
                        $tmp[$val->nodeName]=$val->nodeValue;
        }
}
return $tmp;
}
```

Figure 4.20 :getAttributes method

All the functions of class Wsdl_Element listed above serve to prepare the proper parameters for making a SOAP call. PHP5 has its built-in class SOAP Client, as shown in Fig. 4.10. The function __soapCall($name, array($parameters)), is a built-in function for PHP applications to call. The class MyClient extends the class SoapClient, and is a class PHP applications can instantiate as a SOAPClient. The sole function of MyClient, namely call($name, $parameters) delegates the call to __soapCall($name, array($parameters)) as shown in Figure 4.21. The precondition for this function is to have the structure (array structure) of "$parameters" match with the types obtained from getTypes of the operation with "$name".

```
class MyClient extends SoapClient{

        public function __construct($wsdl){

                parent::__construct($wsdl);

        }

        public  function  call($name,$parameters){


                return $this->__soapCall($name,array($parameters));

        }

}
```

Figure 4.21: MyClient Class

4.3 Implementation of WS-PHP Middleware component

The possible complexity for PHP programs to call Web Services has been explained in Section 3.3.4. This section presents the actual code. In Section 3.3.3 , the steps for each kind of call that the WS-PHP has to carry out are enumerated.

4.3.1 Call Without Parameters

The HandleRequest_NP.php is a parameter in the event handler in the form generated dynamically. It handles only the operation with no parameters; the call is made with null parameters to the Web Service call($name, null). This call returns the SOAP message response as object, since the types of the results are unknown, referring to the WSDL to determine the types returned. The function parse_result returns an array of objects; it will organize the data in a structure that is easy to be displayed, and has three parameters: the object returned by the SOAP call, the instance of object Wsdl_Element and the name of the operation as shown in Figure 4.22. For example, the getDefault function returns an array of

Car type. This object will have two properties: image and an array called data. The image property carries the image if there is image to return and the array data carries the other data of primitive type. Therefore, if the array of objects “the SOAP response parsed” has images, then each object is displayed in a div. If there are no images then the result is an array of simple data types and they are displayed in an HTML table like for the case of the operation getStore. The HandleRequest PHP function is shown in Fig. 4.23.

```
public static function parse_result($vem1,$a,$str2){
$final=array(); $data=array();
$a1=null;
$mes=$a->getMessages($str2);
$element=$a->getElements($mes[1]);
$attributes=$a->getAttributes($element[0]);
if(count(Helper::getArray_FromObject($vem1))==0)
        print "<h2> No Results match this Request </h2>";
else
        if(array_key_exists('maxOccurs',$attributes)){
                 $results=$vem1->$attributes['name'];
                if(is_array($results)){
                        $res1=Helper::getArray_FromObject($results);
                        $res=$res1;
                        }
                        else{
                                $res1=Helper::getArray_FromObject($results);
                                $res[0]=$res1;
                                }
```

(continued)

(continued)

```
if(count($results)!=0){
            if(substr($attributes['type'],0,strpos($attributes['type'],':'))!='xs'){
                  $str=substr($attributes['type'],strpos($attributes['type'],':')+1);
                  $typeElement=$a->getTypes($a->getQuery1($str));
                  for($i=0;$i< count($res);$i++){
                        $obj=new stdClass();
                        foreach($t=Helper::change_ToSimpleArray($res[$i],&$a1) as
$key=>$value){
                              if(self::getType_Value($key,$typeElement)=='string'){
                                                      if(((substr($t[$key],strlen($t[$key])-3) )=='jpg')
                                                      ||((substr($t[$key],strlen($t[$key])-3) )=='gif')){
                                       $obj->image=$value;
                                       }
                                       else{     $data[$key]=$value;
                                       }
                              }
                                      else  {  $data[$key]=$value;    }
                              }
                              $obj->data=$data;
                              $final[]=$obj;
                              $obj=null;
                        }
                  }
            else
      return " No results found for this request";
}
else
{
}
 }
return $final;
```

Figure 4.22:Function parse_result

```
<?php
include 'Parse_Wsdl.php';
include 'soapclient.php';
include 'Templates.php';
include 'Data.php';
if($val=$_GET[data]){
        $data_get=explode(':',$val);
        $wsdl = $data_get[1];
        $object_wsdl=new Wsdl_Elements($wsdl);
        $para1=null;
        $cliente=new MyClient($wsdl);
        $vem1=$cliente->call($data_get[0],  $para1);
   $x= Data_Parsed::parse_result($vem1,$object_wsdl,$data_get[0]);
   Template::display_Content($x);
}
?>
```

Figure 4.23:Function HandleRequest_NP

4.3.2 Call with Parameters

As shown in Section 3.3.3, calling a Web Service operation with parameter is much more complicated because the values of the arguments of the call have to be collected and packed properly. In the implementation, the middleware component automatically displays a form according to the parameters specified in the WSDL document and prompts the end user to give the input. This is done by the function HandleParameters as shown in Fig. 4.24. Each time, the operation name and the corresponding WSDL file name are needed, they can be extracted from the $_GET variable. The data sent by the client browser is a string of structure "operation_name:WSDL_file name". For instance, "getCar:car.wsdl" has the operation name getCar and the holding WSDL file car.wsdl. To reduce traffic between the PHP server and

the back end server hosting the Web Service, WSDL document is saved on the PHP server as a file.

```
<?php
include 'Parse_Wsdl.php';
include 'Templates.php';
if($val=$_GET[data]){
 $data_get=explode(':',$val);
 $object_wsdl=new Wsdl_Elements($data_get[1]);
 $mes=$object_wsdl->getMessages($data_get[0]);
 $myTypes=$object_wsdl->getTypes($object_wsdl->getQuery($data_get[0]));
 Template::head_Form();
 Template::display_Parameters($myTypes);
 $x=strpos($data_get[1],'.');
 $y=substr($data_get[1],0,$x);
 $k=$data_get[0].':'.$y;
 Template::end_Form($k);
}
?>
```

Figure 4.24: Function HandleParameters

When the end user fills in values into the form created by the function HandleParameter and clicks on the "Go" button, a call to a Web Service's operation is processed by PHP function Handle_Request_WP. First, it checks the validity of the inputs posted by the user. It then makes sure that all the texts are not empty and the types of their values are correct. After the validation, an object is created having the structure as specified by the type element of the WSDL and populated with the user inputs. If the type of every parameter

is simple (simple), such as in operation getCar, an array as shown in Figure 4.25 will be given to the PHP built-in function __soapCall as the second parameter.

```
Array (
 [make] => Toyota
 [model] => camry
[year] => 2003
)
```

Figure 4.25:data posted transformed in simple case

If some composite types are involved in a parameter such as operation getStore an array similar to the one shown in Figure 4.26 will be fed to __soapCall as the second parameter.

```
Array (
 [car] => Array (
 [make] => toyota [model] => camry [price] => 10000 [sh] => Array (
 [color] => grey [dim] => Array (
 [type] => car ) ) [year] => 2003 ) )
```

Figure 4.26:data posted transformed in complex case

The SOAP call can be made either by using the array returned by the function Transform_toWSDLarray or by using an object which can be obtained by calling the function

getObject_FromArray, will be the second parameter of the method

call($name, $parameters) the Fig. 4.27 shown the object used to call a Web Service.

```
stdClass Object ( [car] => stdClass Object ( [make] => toyota [model]
=> camry [price] => 10000 [sh] => stdClass Object ( [color] => grey
[dim] => stdClass Object ( [type] => car ) ) [year] => 2003 ) )
```

Figure 4.27:the second parameter used in SOAP call

Upon receiving the proper array argument and the WSDL document name, PHP5 built-in function soapCall will generate the a complete SOAP request and sends it to the Web Service. The SOAP request message calling operation getCar corresponding to the array shown in Fig. 4.25 is illustrated in Fig. 4.28.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns1="http://pack/xsd"><SOAP-ENV:Body>
<ns1:getCar>
<ns1:make>toyota</ns1:make>
<ns1:model>camry
</ns1:model><ns1:year>2003
</ns1:year>
</ns1:getCar>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4.28:Soap request in simple case

The SOAP message request calling operation getStore produced based on the data array shown in Fig. 4.26 is listed in Fig. 4.29.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns1="http://pack/xsd">
<SOAP-ENV:Body>
<ns1:getStore>
    <ns1:car>
      <ns1:make>toyota</ns1:make>
      <ns1:model>camry</ns1:model>
      <ns1:price>60000</ns1:price>
      <ns1:sh>
             <ns1:color>red</ns1:color>
             <ns1:dim>
                    <ns1:type>car</ns1:type>
                    </ns1:dim>
                            </ns1:sh>
                                <ns1:year>2003</ns1:year>
     </ns1:car>
</ns1:getStore>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 4.29:SOAP request in complex case

The result received in the SOAP response is the return of the SOAP call already cited. The Web Service returns a response message to each call. The result of calling getCar is a SOAP Response message as listed in Fig. 4.30.

```
<?xml version='1.0' encoding='utf-8'?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body><ns:getCarResponse xmlns:ns="http://pack/xsd"><ns:return><id
xmlns="http://pack/xsd">1</id><image xmlns="http://pack/xsd">http://localhost/toyota-
camry2003.jpg</image><make xmlns="http://pack/xsd">toyota</make><model
xmlns="http://pack/xsd">camry</model><price xmlns="http://pack/xsd">10000.0</price><sh
xmlns="http://pack/xsd"><color>grey</color><dim><type>car</type></dim><mile>2000.0</mile></sh><year
xmlns="http://pack/xsd">2003</year></ns:return><ns:return><id xmlns="http://pack/xsd">5</id><image
xmlns="http://pack/xsd">http://localhost/image_projet/toyota/2003_toyota_camry_side.jpg</image><make
xmlns="http://pack/xsd">toyota</make><model xmlns="http://pack/xsd">camry</model><price
xmlns="http://pack/xsd">11345.87</price><sh
xmlns="http://pack/xsd"><color>blue</color><dim><type>car</type></dim><mile>67234.0</mile></sh><year
xmlns="http://pack/xsd">2003</year></ns:return><ns:return><id xmlns="http://pack/xsd">6</id><image
xmlns="http://pack/xsd">http://localhost/image_projet/toyota/toyota-2camry2003.jpg</image><make
xmlns="http://pack/xsd">toyota</make><model xmlns="http://pack/xsd">camry</model><price
xmlns="http://pack/xsd">12345.75</price><sh
xmlns="http://pack/xsd"><color>silver</color><dim><type>car</type></dim><mile>54008.45</mile></sh><year
xmlns="http://pack/xsd">2003</year></ns:return><ns:return><id xmlns="http://pack/xsd">7</id><image
xmlns="http://pack/xsd">http://localhost/image_projet/toyota/toyota-3camry2003.jpg</image><make
xmlns="http://pack/xsd">toyota</make><model xmlns="http://pack/xsd">camry</model><price
xmlns="http://pack/xsd">10345.5</price><sh
xmlns="http://pack/xsd"><color>beige</color><dim><type>car</type></dim><mile>87123.6</mile></sh><year
xmlns="http://pack/xsd">2003</year></ns:return><ns:return><id xmlns="http://pack/xsd">8</id><image
xmlns="http://pack/xsd">http://localhost/image_projet/toyota/toyota-4camry2003.jpg</image><make
xmlns="http://pack/xsd">toyota</make><model xmlns="http://pack/xsd">camry</model><price
xmlns="http://pack/xsd">8764.51</price><sh
xmlns="http://pack/xsd"><color>silver</color><dim><type>car</type></dim><mile>88765.76</mile></sh><year
xmlns="http://pack/xsd">2003</year></ns:return><ns:return><id xmlns="http://pack/xsd">9</id><image
xmlns="http://pack/xsd">http://localhost/image_projet/toyota/toyota-6camry2003.jpg</image><make
xmlns="http://pack/xsd">toyota</make><model xmlns="http://pack/xsd">camry</model><price
xmlns="http://pack/xsd">13456.87</price><sh
xmlns="http://pack/xsd"><color>red</color><dim><type>car</type></dim><mile>34213.33</mile></sh><year
xmlns="http://pack/xsd">2003</year></ns:return><ns:return><id xmlns="http://pack/xsd">10</id><image
xmlns="http://pack/xsd">http://localhost/image_projet/toyota/toyota-5camry2003.jpg</image><make
xmlns="http://pack/xsd">toyota</make><model xmlns="http://pack/xsd">camry</model><price
xmlns="http://pack/xsd">14577.99</price><sh
xmlns="http://pack/xsd"><color>black</color><dim><type>car</type></dim><mile>45321.66</mile></sh><year
xmlns="http://pack/xsd">2003</year></ns:return></ns:getCarResponse></soapenv:Body></soapenv:Envelope>
```


Figure 4.30:Soap response of getCar

Finally, the entire process in function HandleRequest_WP as shown in Fig. 4.31.

```
<?php
include 'Parse_Wsdl.php';
include 'soapclient.php';
 include 'Help.php';
include 'Templates.php';
include 'Data.php';
$v=$_POST;  $val=''; $n=count($v); $i=0;
foreach($v as $key=>$value){
	if($n-1==$i){
		$val=$key;
		break;
	}
	$i=$i+1;
}
$pop=array_pop($v);
$data_get=explode(':',$val);
$wsdl = $data_get[1].'.wsdl';
$object_wsdl=new Wsdl_Elements($wsdl);
$mes=$object_wsdl->getMessages($data_get[0]);
$myTypes=$object_wsdl->getTypes($object_wsdl->getQuery($data_get[0]));
if(!is_null($p=Data_Parsed::valid(&$v,&$myTypes))){
	Template::dispaly_errors($p);
}
```

(continued)

(continued)

```
else{
        if(count($mes)<=1){
        Helper::transforme_ToWSDLArray($v,$myTypes);
        $cliente = new MyClient($wsdl);
        $para1=Helper::getObject_FromArray($myTypes);
        $vem1=$cliente->1($data_get[0],$para1);
        print "<h2>Data are updated</h2>";
}
else{
        Helper::transforme_ToWSDLArray($v,$myTypes);
        $cliente = new MyClient($wsdl);
        $para1=Helper::getObject_FromArray($myTypes);
        $vem1=$cliente->call($data_get[0],$para1);
        echo ($cliente->__getLastResponse());
        $vem2=Helper::getArray_FromObject($vem1);
        $t=Data_Parsed::parse_result($vem1,$object_wsdl,$data_get[0]);
        Template:: display_Content($t);
        }
}
?>
```

Figure 4.31:Function Handle Request_WP

4.4 Web Service Operation Selector

In the automation of WS-PHP interactions, the WS-PHP needs to know specifically which operation(s) the PHP application wants to call among the operations offered by the service. This is the duty of the WS operation selector as shown in Fig. 3.1 and implemented in the PHP function Handle1. An HTML form is generated based on the WSDL of the Web Service. In the experimental project, Handle1 is the first to take the choice between the *Car Service* and *House Service* applications, and then a Wsdl_Element object is created. By calling the function getFormElement, all the operations of the Web Service are retrieved and are output in an HTML form of check boxes to let the management, select the ones that will be published to the customers.

The function Handle2 basically parses the operations sent by the management. This function calls getFormElement1 and then determines the nature of operation. If the operation has parameters, these will be handled first by 'HandleParameters.php', and the output of the parameters will be in the div of id='menu'. If the operation doesn't have any parameters, it will be handled by 'HandleRequset_NP.php' and the output of the soap messages formatted displayed in the div of id='content'. The function getFormElement1 returns a multidimensional array as shown in Fig. 4.32 if the selected operations are getDefault and getStore

```
array[$op][0]='HandleRequset_NP.php' array[$op][1]='content'
array[$op][0]='HandleParameters.php' array[$op][1]='menu'
```

Figure 4.32:multidimensional array returned by Handle2

The method display_Operations1 displays those results in an HTML form embedded in the event handler. Fig. 4.33 illustrates this form.

```
<ul>
<li onClick=”MakeRequestGet(’HandleRequset_NP.php’,’content’,’getDefault:car.wsdl’)”>
<li onClick=”MakeRequestGet(’HandleRequset_NP.php’,’menu’,’getStore:car.wsdl’)”>>
</ul>
```

Figure 4.33:HTML form displayed at the end user

The function of Handle1 is given in Fig. 4.34. Its work flow is illustrated in the sequence diagram in Fig. 4.35. The relationship between the five main PHP functions described in the current Chapter is summarized in Fig. 4.36.

```
<?php
include 'Parse_Wsdl.php';
include 'Templates1.php';
include 'Data.php';
session_start();
$v=$_GET[data];
$b=null;
$vals=array();
$file_array=array("http://localhost:8080/CarWebTest2/services/MyCarService?wsdl",
"http://localhost:8080/HouseWebTest1/services/MyHouseService?wsdl",);
$file_name=array("car.wsdl","house.wsdl");
if(isset($v)){
        if(!$_SESSION['vals'][$v]){
                if(file_exists($file_name[$v])){
                        unlink($file_name[$v]);
                }
                $object_wsdl=new Wsdl_Elements($file_array[$v]);
                Wsdl_Elements::$dom->save($file_name[$v]);
                $_SESSION['vals'][$v]=true;
                $b=$file_name[$v];
                $x=$object_wsdl;
        }
        else{

          $b=$file_name[$v];
                $x=new Wsdl_Elements($file_name[$v]);
        }
        Template::display_Operations(Data_Parsed::getFormElement($x),$b);
         $object_wsdl=null;
        }
else
echo "no value is avilable ";
?>
```

Figure 4.34: Function handle1

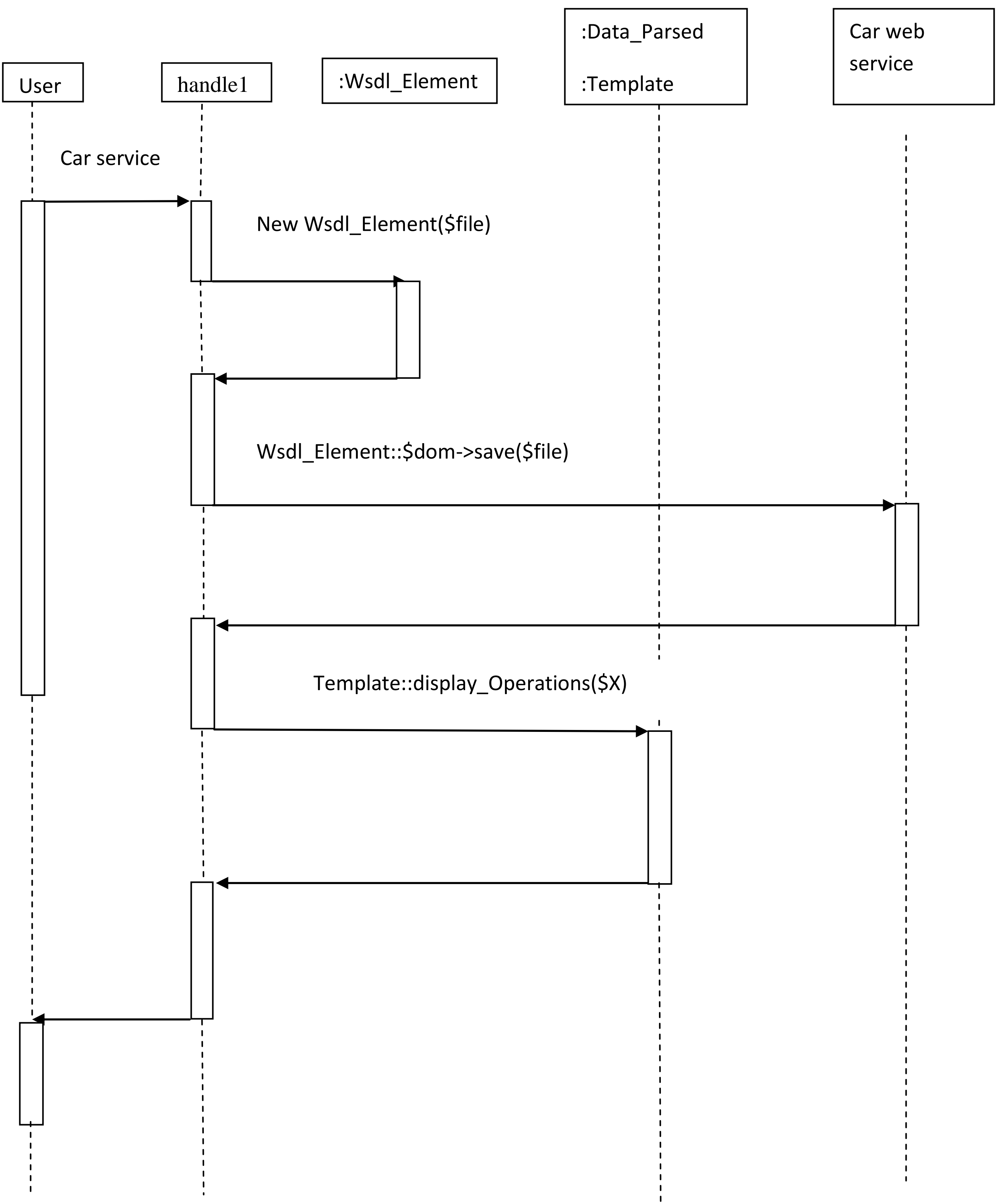


Figure 4.35:Sequence Diagram for Handle1

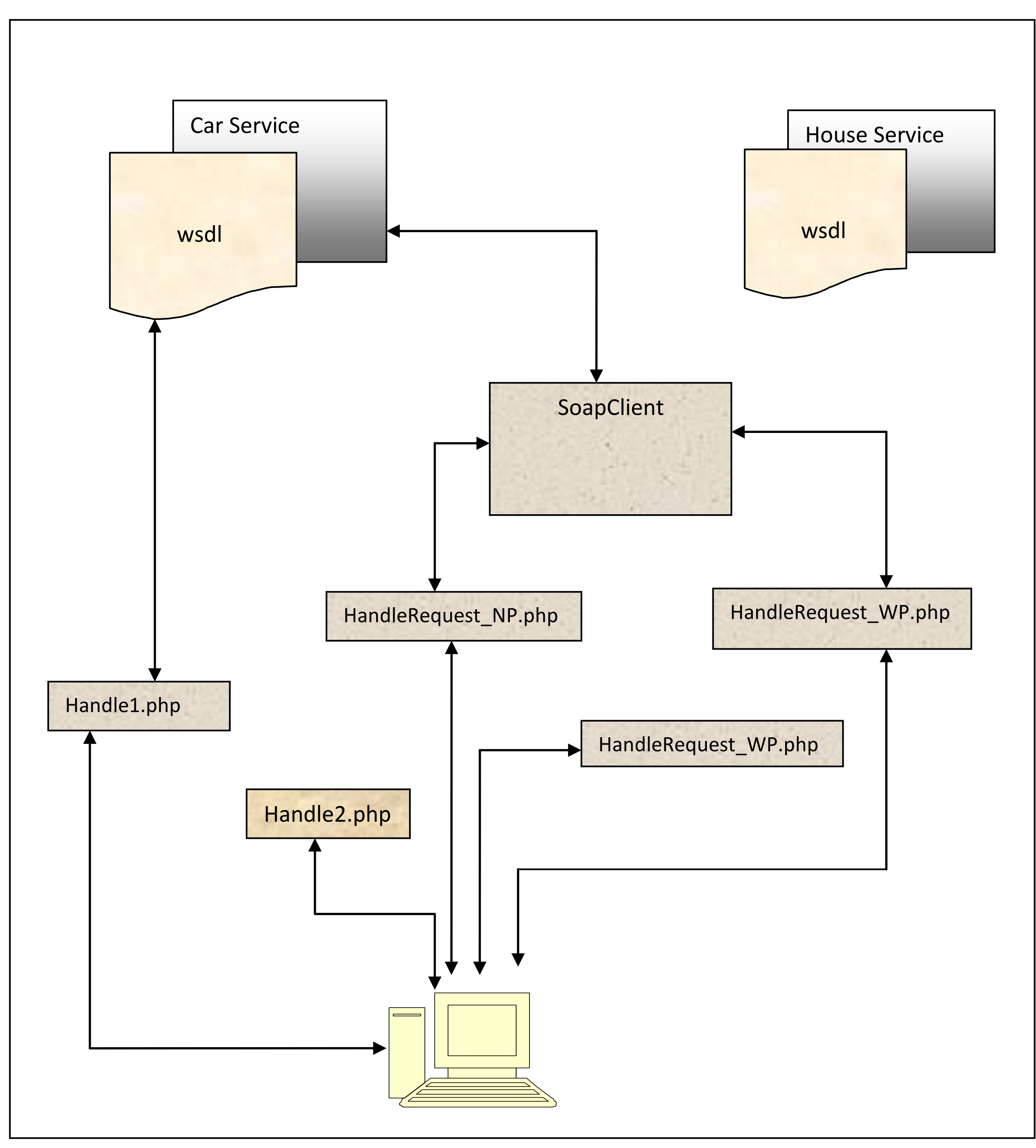


Figure 4.36: System Details

Chapter 5. Experiments

The *Car Service* and the *House Service* applications are developed using the Eclipse Web tool (WTP). Both are created in bottom-up manner. That is, the back-end Java program of each service is written first, then the Eclipse WTP tool is employed to generate all the Web Service components and the WSDL specification document of each service.

In the back-end program, the Java program queries a MySql database. Each Web Service has its own database tables and connections. Tables store_table and car_table are for the *Car Service*. Table house is for the *House Service*.

STORE_TABLE

STORE_ID	NAME	ADRESS	CITY	ZIP	STATE	CAR_ID
INT	VARCHAR(15)	VARCHAR(60)	VARCHAR(20)	CHAR(5)	CHAR(2)	INT

HOUSE

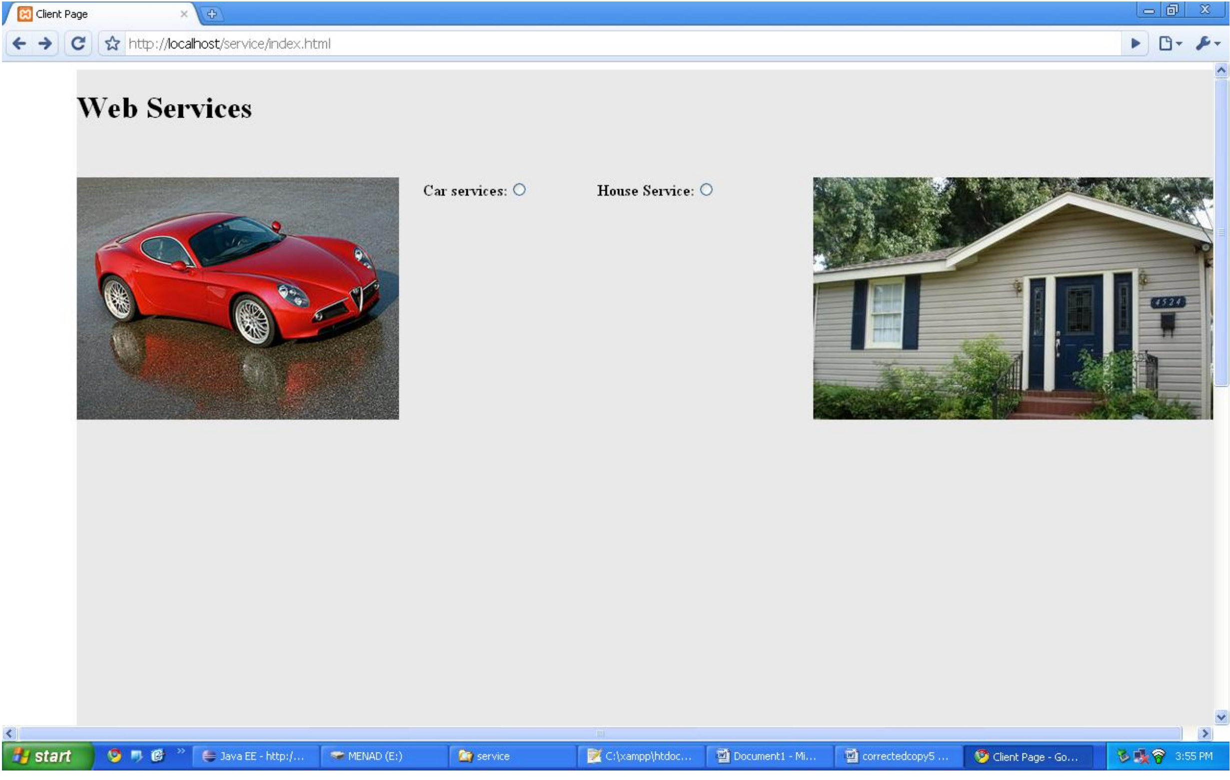
NUM_BEDROOM	PRICE	ADRESSE	CITY	ZIP	STATE	IMAGE
INT	DOUBLE	VARCHAR(100)	VARCHAR(30)	CHAR(5)	CHAR(2)	VARCHAR(100)

CAR_TABLE_2

CAR_ID	MAKE	MODEL	YEAR	TYPE	COLOR	PRICE	MILE	IMAGE
INT	VARCHAR(15)	VARCHAR(15)	INT	VARCHAR(5)	VARCHAR(15)	DOUBLE	DOUBLE	VARCHAR(10)

Since we have used the XAMPP (Apache Web server, MySql Database, interpreters PHP and Perl) open source package, the root directory of the server is under the httdocs of the xampp

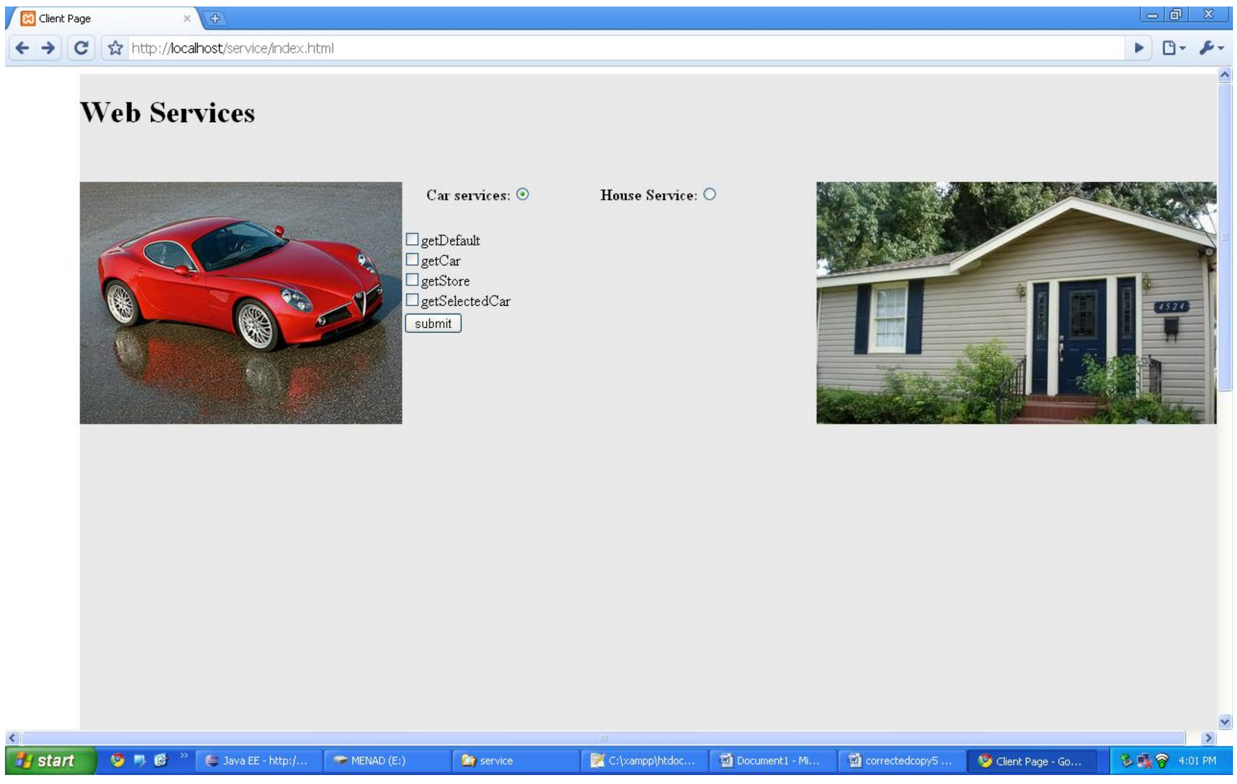
directory. The PHP server runs under Apache Web server. All the files of the application are in the service directory. The back-end services runs in a TOMCAT Web Server. The WS-PHP middleware components are in files HandleRequest_NP.php, HandleRequest_WP.php, Handle1.php, Handle2.php and HandleParameters.php which processes the call to the Web Services applications. Files help.php, data.php, Templates.php, soapclient.php and Wsdl_Element.php carry the supporting PHP classes. Fig. 5.1 shows the initial Web page promoting the IT worker to choose.



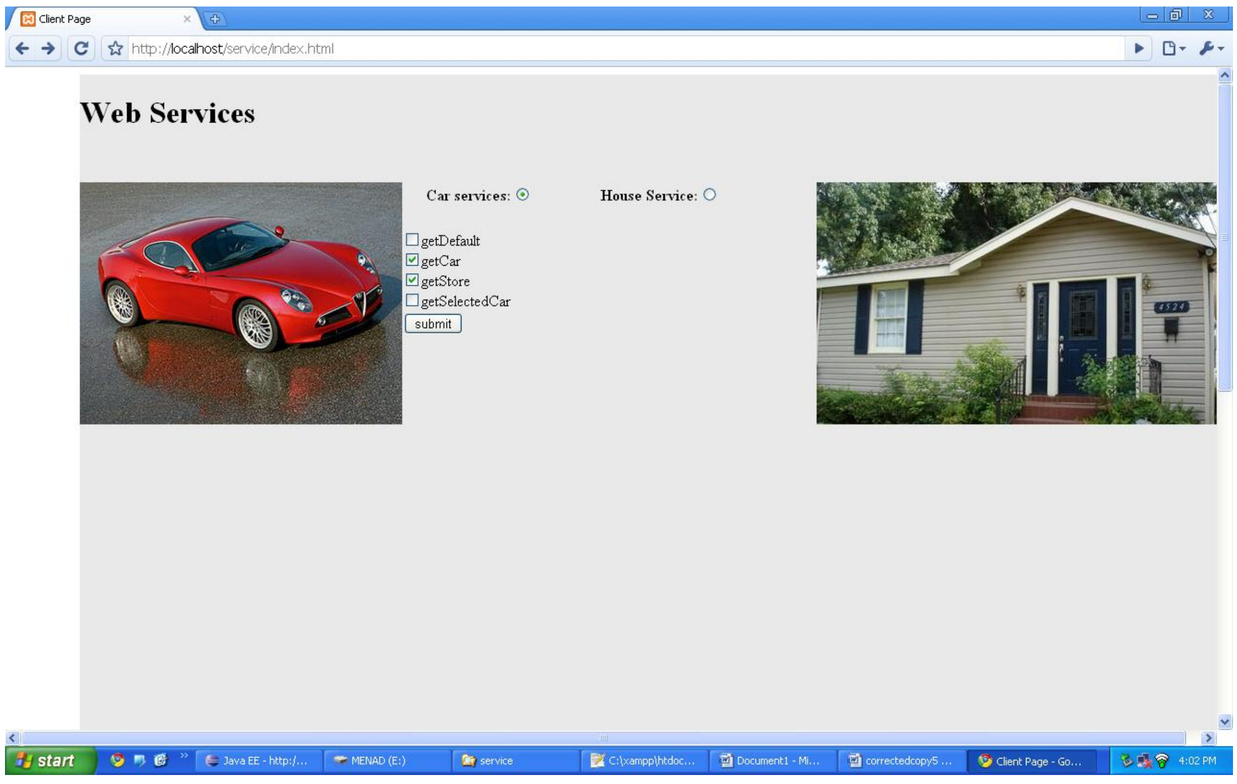
5.1 index page

5.1 The Car Sales application

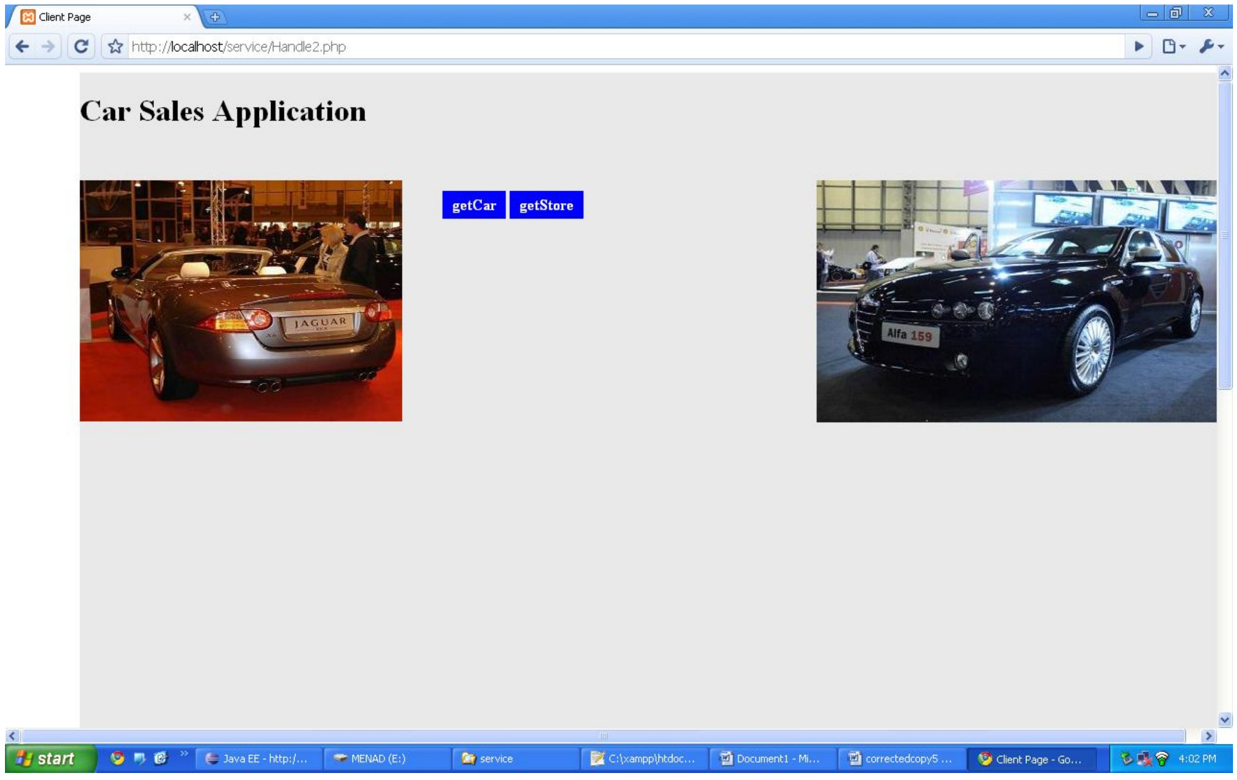
Upon the user's selection, the page produced by `Handle1.php` is shown in Fig. 5.2. If the user chooses the *Car Service*, in the page shown in Fig. 5.3, all the operations of the Car Service are listed. Suppose that the user selected the two operations `getCar` and `getStore`; the WS-PHP middleware will produce a page as shown in Fig. 5.4. This page is for the end user, the potential car buyer.



5.2: Set of operations in the car Web Service

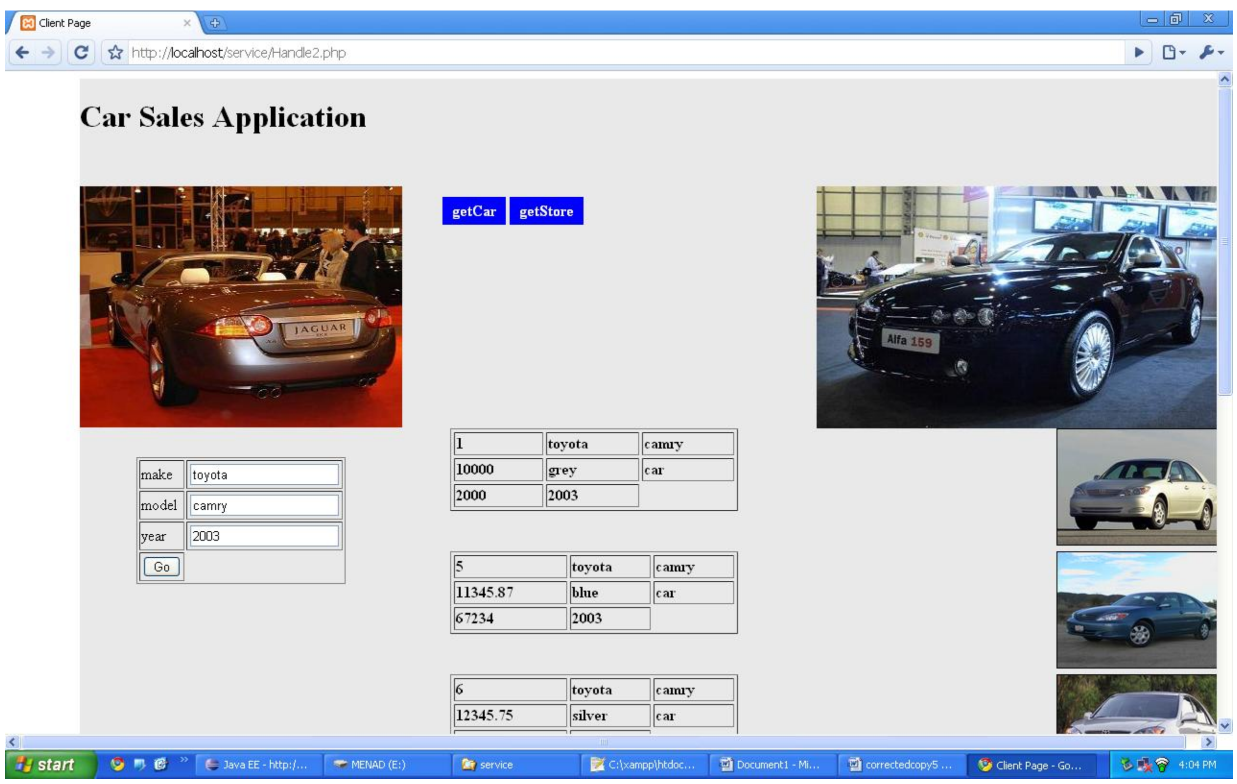


5.3: IT set the operations to be published to the user



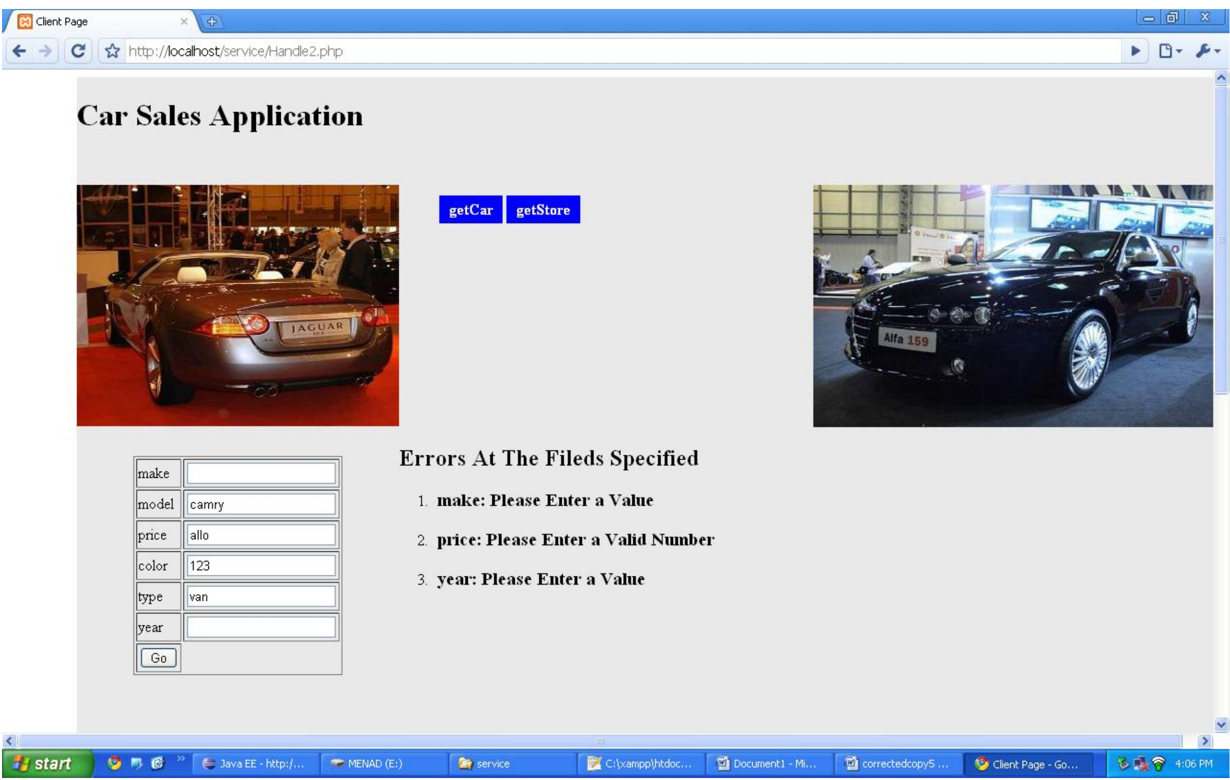
5.4: user page interface

When a user clicks on an operation `getCar`, a form appears in the left side in the div id='menu', then enters the inputs in text inputs, then the button `Go` is hit, and the formatted results are displayed in the div id='content'.



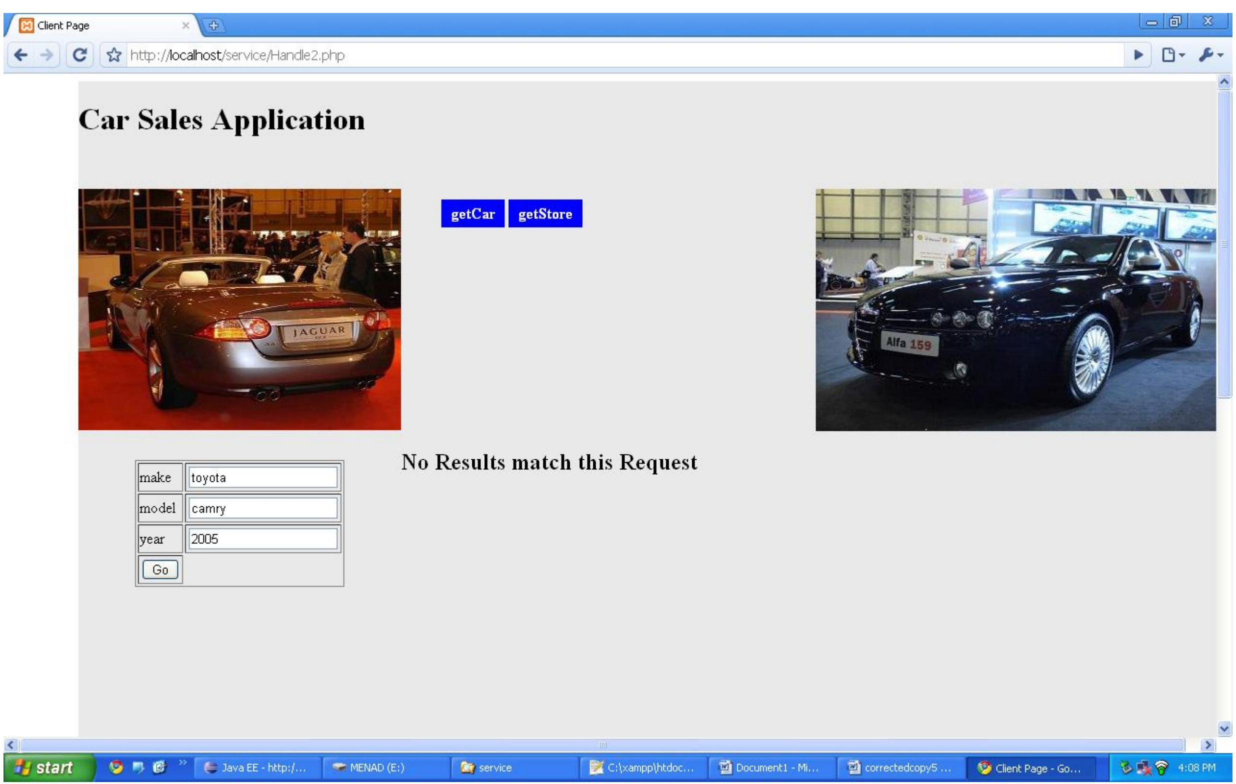
5.5: request and response for getCar operation

In case the user leaves some blanks or enters invalid inputs, the system warns him and lists the errors registered.



5.6: error message for invalid data

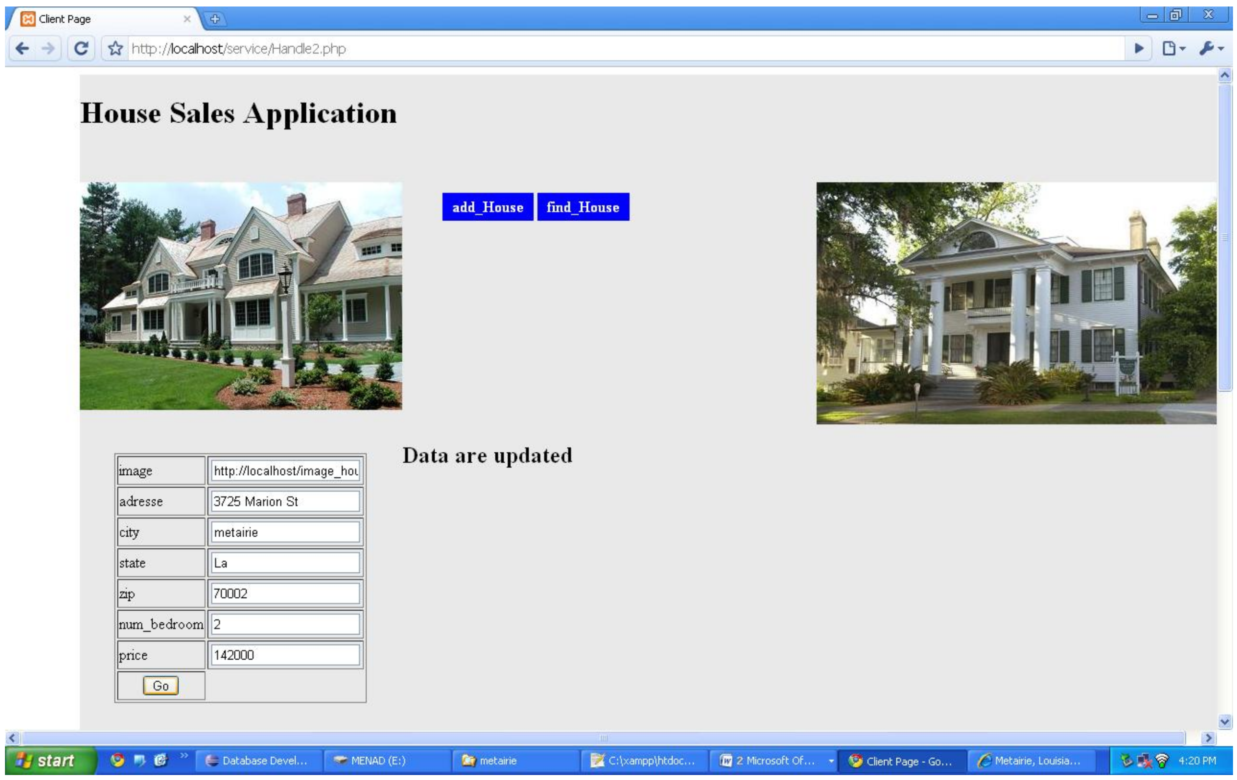
If there is no match to the request of the user, a message to that effect is displayed in the next view.



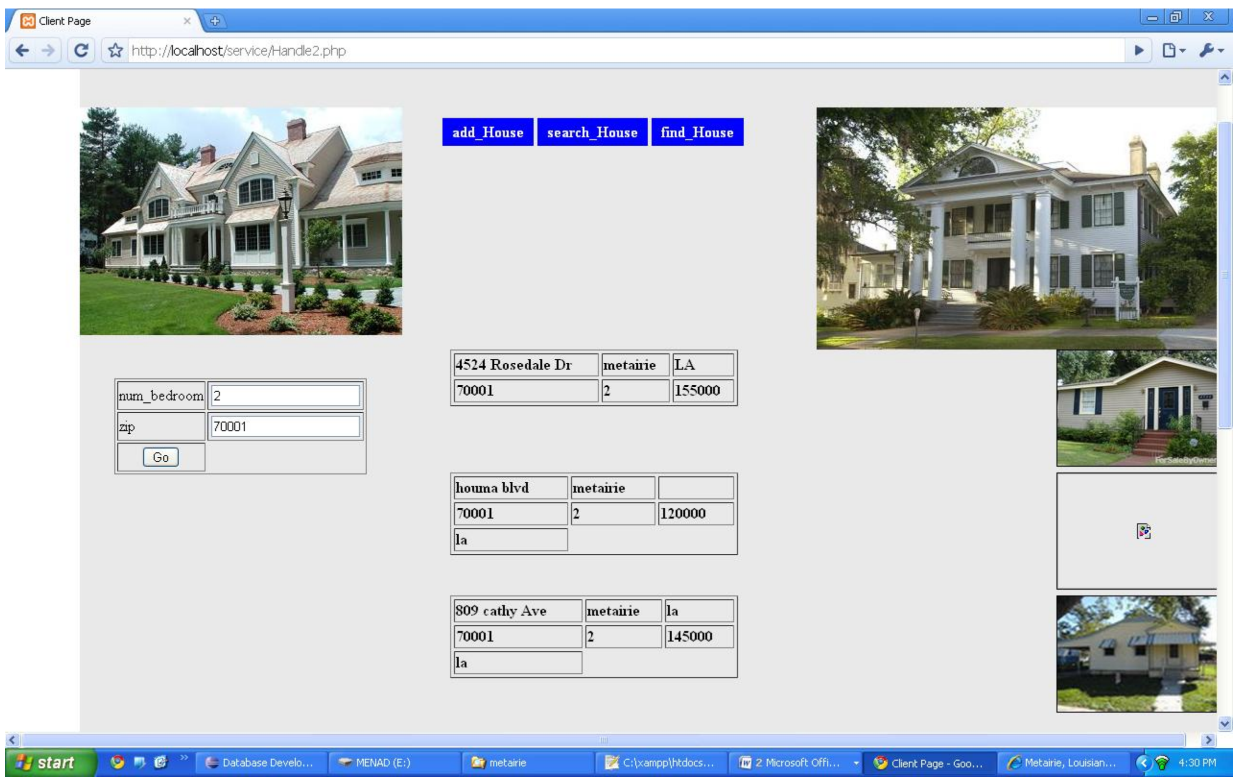
5.7:No result found for the user request

5.2 The House sales application

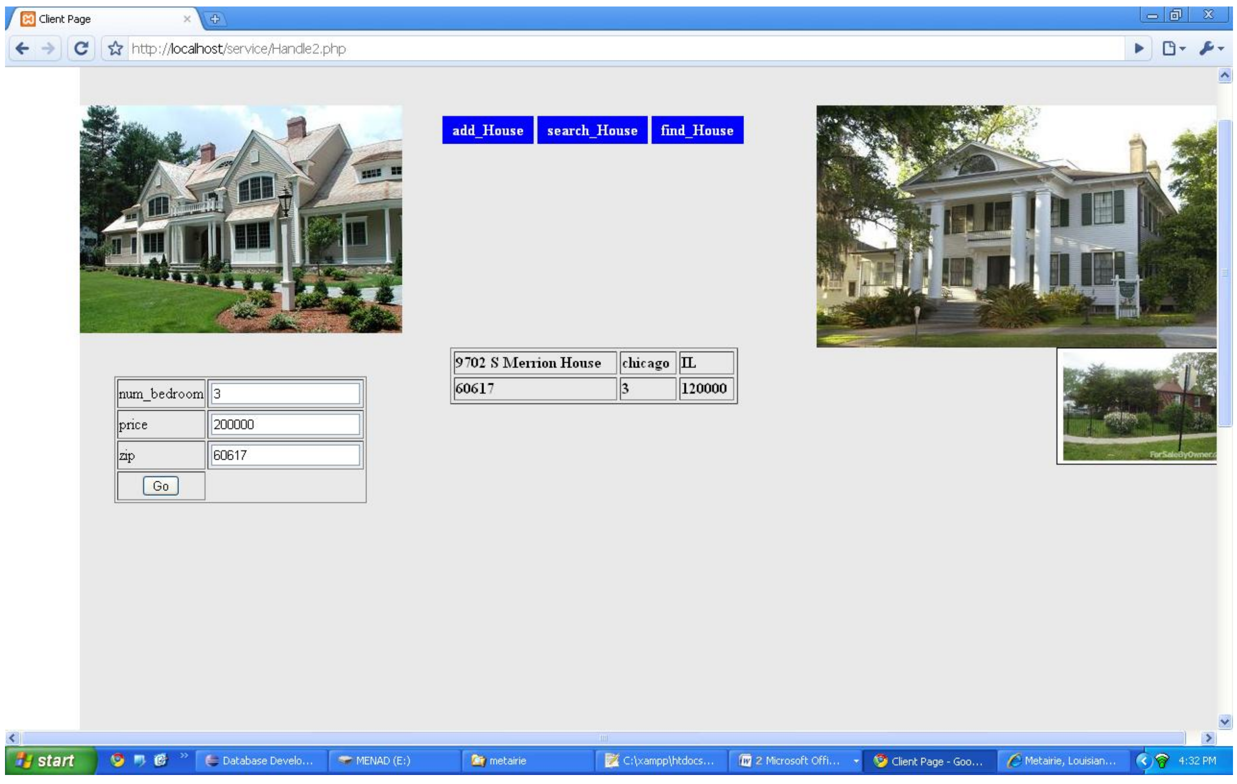
If the operation selected by the user is something like `add_House` which has parameters but returns no value (*i.e.*, void), then a message shows that the data are updated.



5.8: operation add_House , no return



5.9:Request for find_House operation



5.10:request for search_House operation

Chapter 6. Conclusion

Web Service is the mainstream technology in server-side development. PHP is a very popular programming language for applications. The WS-PHP middleware component facilitates PHP programs to call Web Services even when complicated object structures are involved. The WS-PHP middleware takes all the information from one single source – the WSDL document – and then generates the elements necessary to consume the Web Services,

In this project, it's been shown the practicality of creating the client side of the application that utilizes a Web service according to the WSDL document of the Web service only. Typically, a service includes multiple operations. A user interface is generated automatically, where the user makes the choice of the operations. The results of the SOAP call might be a collection of complex data. Still, they can be displayed. Using the WS-PHP middleware component, two Web applications were generated. They are the Car Sale and House Sale based on two Web services, Car Service or House Service respectively.

The significant different styles of the back-end programming language and the front-end programming may potentially cause some problems. For example, the modern Java version (Java 6) has a number of classes serving collections such as `ArrayList` as well as Java's old-style `Vector` class. However, these kinds of collection types would be specified as being of "Any" type. This ambiguity would make conversion to PHP impossible. PHP is a weak typed language; it does not have any matching classes for `ArrayList` and `Vector`. Thus, avoiding using the ambiguous types in the interface of Web Services is required.

References

[DOM]	Pro PHP XML and Web Services
[AJAX]	AJAX and PHP
[NuSOAP]	NuSOAP http://sourceforge.net/projects/nusoap/
[PHP]	PHP: Hypertext Preprocessor http://php.net/
[PHP]	Learning PHP 5
[Pear:SOAP]	Pear:SOAP http://pear.php.net/package/SOAP
[PARSING]	XML for PHP developers http://www.ibm.com/developerworks/library/x-xmlphp2.html
[SOAP ClIENT]	PHP SOAP Extension http://www.herongyang.com/PHP/php_soap_server.html
[SOAP]	W3C Recommendation, “SOAP version 1.2”, Apr 27, 2007. http://www.w3.org/TR/soap12-part1/
[PHP ClIENT]	PHP Client for Java-based Web Service http://oleksiy.wordpress.com/2007/08/22/php-client-for-java-based-webservice
[WSDL]	Which Style of WSDL Should I use http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/
[W3C]	The World Wide Web Consortium http://www.w3.org/
[WEB SERVICE]	Generate Web Service With Eclipse to Access a Database http://www.farmbio.uu.se/upload/avd2/eclipse-usecase/usecase_ws.html
[WSDL]	W3C Note, “Web Service Description Language”, Mar 15, 2001. http://www.w3.org/TR/wsdl
[RECURSION]	Resursion in PHP Recursion In PHP: Tapping Unharnessed Power
[WSF/PHP]	WS02 OxygenTank’s Web Service Framework for PHP, http://wso2.org/projects/wsf/php
[AJAX]	Ajax PHP Tutorial Ajax PHP tutorial - PHP code and the complete AJAX example
[Xpath]	W3C Recommendation, “XML Path Language”, Nov 16, 1999. http://www.w3.org/TR/xpath

VITA

Menad Medjkane was born in Algeria. In 1995, he received a Bachelor degree in Computer Science from the University of Sciences and Technologies of Oran (Algeria) and started the Master program in Computer Science at The University of New Orleans in The fall of 2004.