

Marcel ANTAL

Teodor PETRICAN

Tudor CIOARA

Claudia POP

Ciprian STAN

Ionut ANGHEL

Dorin MOLDOVAN

Ioan SALOMIE

Distributed Systems

Laboratory Guide



Editura UTPRESS
Cluj-Napoca, 2018
ISBN 978-606-737-329-5

Marcel ANTAL

Teodor PETRICAN

Tudor CIOARA

Claudia POP

Ciprian STAN

Ionut ANGHEL

Dorin MOLDOVAN

Ioan SALOMIE

Distributed Systems

Laboratory Guide



Editura UTPRESS
Cluj-Napoca, 2018
ISBN 978-606-737-329-5



Editura U.T. PRESS

Str. Observatorului nr. 34

C.P. 42, O.P. 2, 400775 Cluj-Napoca

Tel.:0264-401.999

e-mail: utpress@biblio.utcluj.ro

<http://biblioteca.utcluj.ro/editura>

Director: Ing. Călin D. Câmpăan

Recenzia: Prof.dr.ing. Mihaela Dînșoreanu

Conf.dr.ing. Viorica Chifu

Copyright © 2018 Editura U.T.PRESS

Reproducerea integrală sau parțială a textului sau ilustrațiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

ISBN 978-606-737-329-5

Contents

Preface	10
1. Prerequisites Installation and Configuration.....	12
1.1. Programming environment: Java.....	12
1.2. Integrated Development Environment (IDE): Eclipse	13
1.3. Relational database management system: MySQL	14
1.4. Web server: Apache Tomcat	15
1.5. Version control system: Git.....	16
1.6. Version control repository: Bitbucket	17
1.7. Configuring the lab working environment	18
2. Request-Reply and Sockets.....	21
2.1. Problem statement	21
2.2. Application analysis and design	21
2.2.1. Defining the message structure	22
2.2.2. Client request	22
2.2.3. Server response	23
2.3. Application structure and implementation	24
2.3.1. The client application.....	25
2.3.2. The server application.....	30
2.4. Building and running the example.....	36
2.5. Laboratory work: web application using request – reply	36
2.5.1. Requirements	36
2.5.2. Deliverables	37
2.5.3. Evaluation	37
2.6. Bibliography	38
3. Remote Procedure Call and Distributed Objects	39
3.1. Problem statement	39
3.2. Application analysis and design	39
3.2.1. General architecture	40
3.2.2. Communication mechanism.....	40
3.3. Application structure and implementation	42
3.3.1. Client application	43

3.3.2. RPC library	45
3.3.3. Common classes.....	49
3.3.4. Server application	50
3.3.5. Application sequence diagram	53
3.4. Building and running the example.....	55
3.5. Laboratory work: RPC application using distributed objects.....	55
3.5.1. Requirements	55
3.5.2. Deliverables	56
3.5.3. Evaluation	56
3.6. Bibliography	57
4. Indirect Communication and Queues.....	58
4.1. Problem statement	58
4.2. Application analysis and design	58
4.3. Application structure and implementation	60
4.3.1. Producer client application	61
4.3.2. Consumer client application	63
4.3.3. Queue server application.....	66
4.3.4. Application sequence diagram	71
4.4. Building and running the example.....	73
4.5. Laboratory work: asynchronous distributed system application	73
4.5.1. Requirements	73
4.5.2. Deliverables	74
4.5.3. Evaluation	74
4.6. Bibliography	75
5. XML based Communication and Web Services	76
5.1. Introduction	76
5.2. WSDL	79
5.3. SOAP	80
5.3. UDDI	81
5.4. Laboratory work: SOA web services.....	82
5.4.1. Requirements	82
5.4.2. Deliverables	83
5.4.3. Evaluation	84

5.5. Bibliography	84
6. Server-side Frameworks: Spring for Developing REST Web Services	85
6.1. Introduction	85
6.2. Hands-on application	86
6.2.1. Application installation and configuration.....	86
6.2.2. Application conceptual architecture	89
6.2.3. Application implementation details	90
6.2.4. Testing the application	100
6.3. Laboratory work: Spring REST backend for a distributed application	101
6.3.1. Requirements	101
6.3.2. Deliverables	101
6.3.3. Evaluation	102
6.4. References	102
7. Client-side Frameworks: Angular for Developing Single-page Web GUIs	103
7.1. Introduction	103
7.2. Hands-on application	103
7.2.1. Application installation and configuration.....	103
7.2.2. Application conceptual architecture	105
7.2.3. Application implementation details	106
7.2.4. Testing the application	110
7.3. Laboratory work: Angular GUI for Chapter 6 Spring backend.....	111
7.3.1. Requirements	111
7.3.2. Deliverables	111
7.3.3. Evaluation	112
7.4. References	112

List of Figures

Figure 1.1. JDK download options.....	12
Figure 1.2. JRE installation	12
Figure 1.3. Eclipse IDE installation	13
Figure 1.4. MySQL install options.....	14
Figure 1.5. MySQL setup types	14
Figure 1.1.6. MySQL server and workbench tools setup.....	15
Figure 1.7. Apache Tomcat distributions	15
Figure 1.8. Setting the Apache Tomcat environment variable.....	16
Figure 1.9. Git download options.....	16
Figure 1.10. Git terminal emulator selection.....	17
Figure 1.11. Creating Bitbucket account.....	17
Figure 1.12. Eclipse menu configuration	18
Figure 1.13. Example of files ignored in commit operation	18
Figure 1.14. Maven settings file.....	20
Figure 2.1. Client-server software architecture	21
Figure 2.2. Application conceptual architecture	24
Figure 2.3. Client package diagram	24
Figure 2.4. Server package diagram.....	25
Figure 2.5. Communication protocol package diagram	25
Figure 2.6. Sequence diagram for client POST operation.....	26
Figure 2.7. Post action listener code snippet.....	27
Figure 2.8. Encode method code snippet	27
Figure 2.9. Serialize method code snippet	28
Figure 2.10. SendRequest method class code snippet.....	29
Figure 2.11. Decode method code snippet	30
Figure 2.12. Sequence diagram for server POST operation.....	30
Figure 2.13. Run method from Server class code snippet.....	31
Figure 2.14. Run method from Session class code snippet.....	32
Figure 2.15. AbstractServlet class code snippet.....	33
Figure 2.16. Method createServlet code snippet	33
Figure 2.17. StudentServlet class code snippet	34
Figure 2.18. StudentDAO class code snippet.....	35
Figure 3.1. RPC system communication flow.....	41
Figure 3.2. Application conceptual architecture	42
Figure 3.3. ClientStart class code snippet	44
Figure 3.4. ServerConnection class code snippet.....	44
Figure 3.5. Registry class code snippet	45

Figure 3.6. Message class code snippet.....	46
Figure 3.7. Dispatcher class Code Snippet.....	47
Figure 3.8. Naming class code snippet.....	49
Figure 3.9. ITaxService interface code snippet.....	49
Figure 3.10. TaxService class code snippet	50
Figure 3.11. ServerStart class code snippet.....	51
Figure 3.12. Server class Code Snippet.....	51
Figure 3.13. Session class code snippet	53
Figure 3.14. Sequence diagram of a RPC call.....	54
Figure 4.1. MOM software architecture.....	59
Figure 4.2. Conceptual architecture	60
Figure 4.3. ClientStart class code snippet	61
Figure 4.4. QueueServerConnection class code snippet	63
Figure 4.5. ClientStart class code snippet	63
Figure 4.6. MailService class code snippet	65
Figure 4.7. QueueServerConnection class code snippet	66
Figure 4.8. ServerStart class code snippet.....	67
Figure 4.9. Queue class code snippet	67
Figure 4.10. Message class code snippet.....	68
Figure 4.11. Sever class Code Snippet.....	69
Figure 4.12. Session class code snippet	71
Figure 4.13. Sequence diagram for message insertion in queue (producer client)	72
Figure 4.14. Sequence diagram for message retrieval from queue (consumer client)	72
Figure 5.1. From distributed objects to services	76
Figure 5.2. SOA architecture overview.....	77
Figure 5.3. WSDL as a service Interface Description Language (IDL).....	79
Figure 6.1. Maven install in Eclipse.....	87
Figure 6.2. Application execution in Eclipse	87
Figure 6.3. User table from the example project.....	88
Figure 6.4. Project structure in Eclipse	88
Figure 6.5. Project conceptual architecture	89
Figure 6.6. Sequence diagram for GET operation	90
Figure 6.7. Spring controller mapping	91
Figure 6.8. RequestMapping for UserController	91
Figure 6.9. RequestMapping for getting all the users	92
Figure 6.10. PathVariable annotation example	92
Figure 6.11. Model object annotation example.....	92
Figure 6.12. ModelAttribute annotation example	92
Figure 6.13. UserService example	93

Figure 6.14. UserRepository definition.....	93
Figure 6.15. UserRepository interface	94
Figure 6.16. PersistenceConfig class.....	95
Figure 6.17. JpaRepository snippet	96
Figure 6.18. UserRepository interface	96
Figure 6.19. Custom defined queries	97
Figure 6.20. User entity.....	98
Figure 6.21. Creating a DTO.....	99
Figure 6.22. JSON with data from the DTO object	99
Figure 6.23. Response in the browser for a REST request	100
Figure 6.24. Example of using the Postman tool	100
Figure 7.1. Angular project structure	104
Figure 7.2. Hands-on example welcome page	105
Figure 7.3. Project conceptual architecture	105
Figure 7.4. Sequence diagram for retrieving all users.....	106
Figure 7.5. Snippet from index.html	107
Figure 7.6. Calling the method getUsers()	107
Figure 7.7. Snippet from header.component.ts	107
Figure 7.8. Snippet from HeaderComponent	108
Figure 7.9. Snippet from app.module.ts	108
Figure 7.10. Snippet from UserService	109
Figure 7.11. Snippet from user.service.ts.....	109
Figure 7.12. Snippet from UsersComponent.....	110
Figure 7.13. Snippet from app.module.ts	110
Figure 7.14. Angular GUI to display all the users	111

List of Tables

Table 2.1. Web application laboratory work grading details	38
Table 3.1. Relation between engine size and specific sum	39
Table 3.2. RPC laboratory work grading details	56
Table 4.1. Message types	68
Table 4.2. Asynchronous communication laboratory work grading details	74
Table 5.1. The main features of SOA architecture.....	78
Table 5.2. WSDL XML elements	79
Table 5.3. XML elements of a SOAP message.....	80
Table 5.4. UDDI pages.....	82
Table 5.5. SOA web services laboratory work grading details	84
Table 6.1. Project Component Description	89
Table 6.2. REST backend laboratory work grading details	102
Table 7.1. Project component description	106
Table 7.2. Angular GUI laboratory work grading details	112

Preface

A distributed system is composed from multiple processing components deployed and executed onto various computing nodes which communicate and coordinate their operation by exchanging messages. From a user perspective, the distribution of resources is transparent, and the functionalities should be provided as if the system is a centralized one. Thus, the exchange of messages is fundamental for distributed systems operation and provides the underlying base of each system level functionality implementation.

In this book we will provide an overview of the main types of inter-process communication used in the development of distributed systems aiming to provide insights on how they are implemented in lower architectural levels. These insights are usually transparent even for software developers which are used to implement distributed systems leveraging on high level frameworks or middleware. Therefore, we have opted to let our students to first test and extend some lower level communication software we have developed and then to ask them to implement similar functionality using some higher-level frameworks. Also, in this book we will show how service oriented distributed systems can be built using modern technologies by allowing students to work with the two major architectural styles in this area, Service Oriented Architecture and REST (Representational State Transfer).

Section 1 introduces the students into the distributed systems laboratory thematic providing guidelines for the installation of the technological infrastructure stack they will use during a semester.

Section 2 presents an implementation overview of the synchronous and direct communication protocol focusing on the Request–Reply paradigm. We start by explaining the fundamentals behind implementing such a communication protocol providing relevant examples from a hands-on application we had developed which features web server functionalities. Then we ask our students to implement a similar application using server-side technologies such as Java Servlets.

Section 3 addresses various aspects of the Remote Procedure Call (RPC) protocol used to request a service/operation from a program/component located in a different processing node. We provide a hands-on application in which this type of communication protocol implementation is detailed up to the level of socket-based communication allowing the students to gain more in-depth details of technical aspects such as remote references, serialization, the use of Java reflection, message encoding, etc. At the same time, students are asked to implement an application with similar functionality using a distributed objects framework such as Java RMI or .Net Remoting, allowing them to observe and evaluate how much of the lower level implementation details are hidden for a software developer.

Section 4 presents the indirect communication paradigm for distributed systems components, which provides a higher decoupling leading to an improved performance, but a decrease in

reliability in message delivery. We provide a hands-on implementation of such approach leveraging on queues as intermediary resources between the components that are communicating. In the second part of this section the students are asked to implement a similar communication approach leveraging on existing state of the art open source frameworks and message brokers.

Section 5 provides insights on the XML based communication in distributed systems using Service Oriented Architecture (SOA) web services. We present the main SOA components and the technologies that are used to build these kind of web services: Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP) and Universal Description, Discovery, and Integration (UDDI). Students are asked to implement an application using both JAVA and .NET web services to highlight the most important aspect of SOA, platform interoperability.

Section 6 deals with the alternative technology to SOA, RESTfull web services for which data and functionalities are considered resources which can be accessed by Uniform Resource Information (URI). These resources can be accessed by a set of simple and well-defined operations. We provide a hands-on implementation of a simple REST web service using the Spring framework. The students are asked to implement a complex application that exposes multiple REST services leveraging the hands-on example.

Section 7 addresses the development of dynamic single page web applications Graphical User Interfaces (GUIs). We choose Angular framework for the development of the single-page applications due to its popularity and versatility. We provide a hands-on implementation of a simple Angular GUI detailing the main components, their interaction and the GUI connection with business logic web services. The students are asked to use Angular to implement a modern GUI for the previous chapter developed REST services.

The Authors,

November 2018

1. Prerequisites Installation and Configuration

1.1. Programming environment: Java

- 1) Access <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- 2) Click on the Java Development Kit (JDK) icon and you will be redirected to Java downloads.

Java SE Development Kit 8u101		
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.		
<input type="radio"/> Accept License Agreement <input checked="" type="radio"/> Decline License Agreement		
Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.77 MB	jdk-8u101-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.72 MB	jdk-8u101-linux-arm64-vfp-hflt.tar.gz
Linux x86	160.28 MB	jdk-8u101-linux-i586.rpm
Linux x86	174.96 MB	jdk-8u101-linux-i586.tar.gz
Linux x64	158.27 MB	jdk-8u101-linux-x64.rpm
Linux x64	172.95 MB	jdk-8u101-linux-x64.tar.gz
Mac OS X	227.36 MB	jdk-8u101-macosx-x64.dmg
Solaris SPARC 64-bit	139.66 MB	jdk-8u101-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	98.96 MB	jdk-8u101-solaris-sparcv9.tar.gz
Solaris x64	140.33 MB	jdk-8u101-solaris-x64.tar.Z
Solaris x64	96.78 MB	jdk-8u101-solaris-x64.tar.gz
Windows x86	188.32 MB	jdk-8u101-windows-i586.exe
Windows x64	193.68 MB	jdk-8u101-windows-x64.exe

Figure 1.1. JDK download options

- 3) Click on the *Accept License Agreement* link.
- 4) Click on the link which corresponds to your version of the Operating System. For example, if the needed version corresponds to Windows x64 then the file is *jdk-8u101-windows-x64.exe*.
- 5) After the version is selected, a file with the same name will be downloaded.
- 6) Start the downloaded version of the installer.
- 7) You will be asked the next question: Do you want to allow the following program to make changes to this computer? Click *Yes*.
- 8) Click *Next*. You will be asked where you want to install Java. Use the default location.
- 9) After the JDK is installed, you will be asked where you want to install the Java Runtime Environment (JRE). Use the default location and click *Next*.

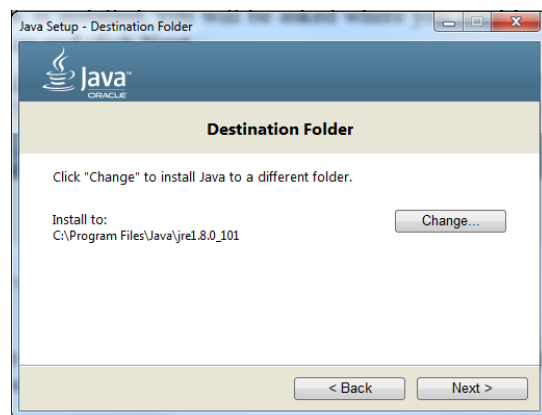


Figure 1.2. JRE installation

10) After the installation is completed click *Close*.

11) Set JAVA_HOME and JAVA_JRE variables by following the below steps:

- Click *Start*.
- Right-Click on *Computer*.
- Select *Properties*.
- Click on *Advanced System Settings*.
- Click on *Environment Variables*.
- Under *System Variables* click *New*.
- In the text field associated with the name of the variable insert JAVA_HOME and in the field associated with the value of the variable insert *C:\Program Files\Java\java_version;*
- Click *OK*.
- Under *System Variables* click *New* again.
- In the text field associated with the name of the variable insert JRE_HOME and in the field associated with the value of the variable insert *C:\Program Files\Java\java_version;*
- Click *OK*.

1.2. Integrated Development Environment (IDE): Eclipse

1) Access <http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/mars2>.

RELEASES

- Neon Packages
- Oxygen Packages
- Mars Packages
- Luna Packages
- Kepler Packages
- Juno Packages
- Indigo Packages
- Helios Packages
- Galileo Packages
- Ganymede Packages
- All Releases

 **Eclipse IDE for Java EE Developers**

Package Description

Tools for Java developers creating Java EE and Web applications, including a Java IDE, tools for Java EE, JPA, JSF, Mylyn, EGIt and others.

This package includes:

- Data Tools Platform
- Eclipse Git Team Provider
- Eclipse Java Development Tools
- Eclipse Java EE Developer Tools
- JavaScript Development Tools
- Maven Integration for Eclipse
- Mylyn Task List
- Eclipse Plug-in Development Environment
- Remote System Explorer
- Code Recommenders Tools for Java Developers
- Eclipse XML Editors and Tools

► Detailed features list

Download Links

- Windows 32-bit**
- Windows 64-bit**
- Mac OS X (Cocoa) 64-bit**
- Linux 32-bit**
- Linux 64-bit**

Downloaded 2,630,891 Times

► Checksums...

Bugzilla

► Open Bugs: 58

► Resolved Bugs: 140

File a Bug on this Package

Maintained by: WTP and the Eclipse Packaging Project

Figure 1.3. Eclipse IDE installation

- 2) In *Package Solutions* search for *Eclipse IDE for Java EE Developers* and click on the version which is appropriate for your computer: 32-bit or 64-bit.
- 3) You will be redirected to a page where you will be asked to select a mirror. Click on *Download*.
- 4) You will obtain a file named *eclipse-jee-mars-2-win32-x86_64.zip*.
- 5) Open the archive *eclipse-jee-mars-2-win32-x86_64.zip* and extract it to *C:*.

- 6) You can open Eclipse by clicking on the file *eclipse.exe* which should be at the location *C:\eclipse\eclipse.exe*.

1.3. Relational database management system: MySQL

- 1) Access the link: <https://dev.mysql.com/downloads/windows/installer/>.

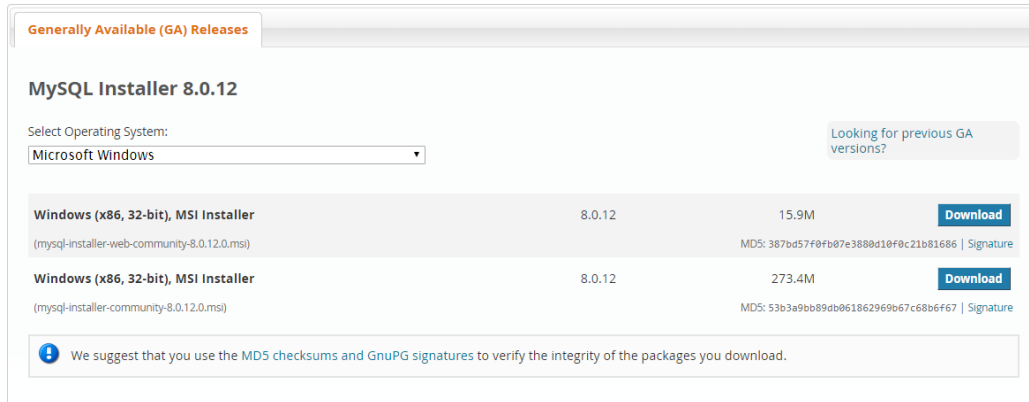


Figure 1.4. MySQL install options

- 2) Click on the second *Download* button.
3) Click on *No thanks, just start my download*.
4) Click on the downloaded file *mysql-installer-web-community8.0.12.0.msi*.
5) Click *Yes*.
6) Click *I accept the license terms* and then *Next*.
7) You will be asked to select the *Setup Type* that suits your use case. Select *Custom*.

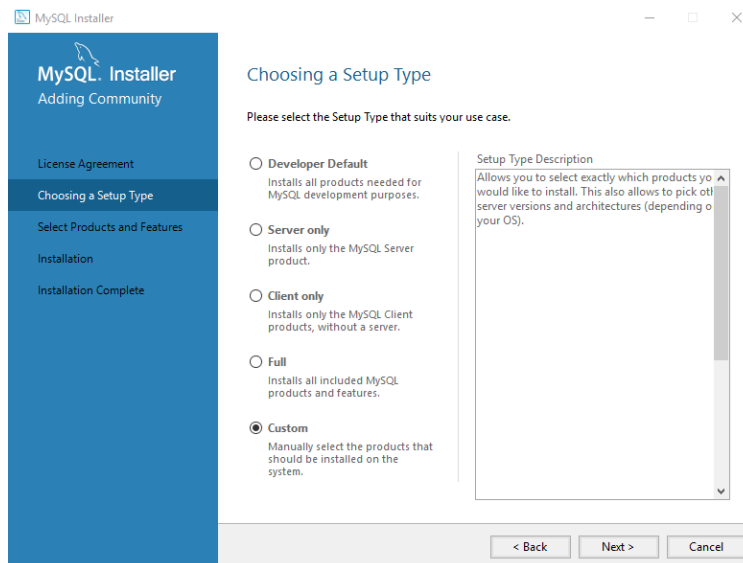


Figure 1.5. MySQL setup types

- 8) You will be redirected to *Select Products and Features*. Select *MySQL Server 8.0.12 – X64* and *MySQL Workbench 8.0.12 – X64* and click *Next*.

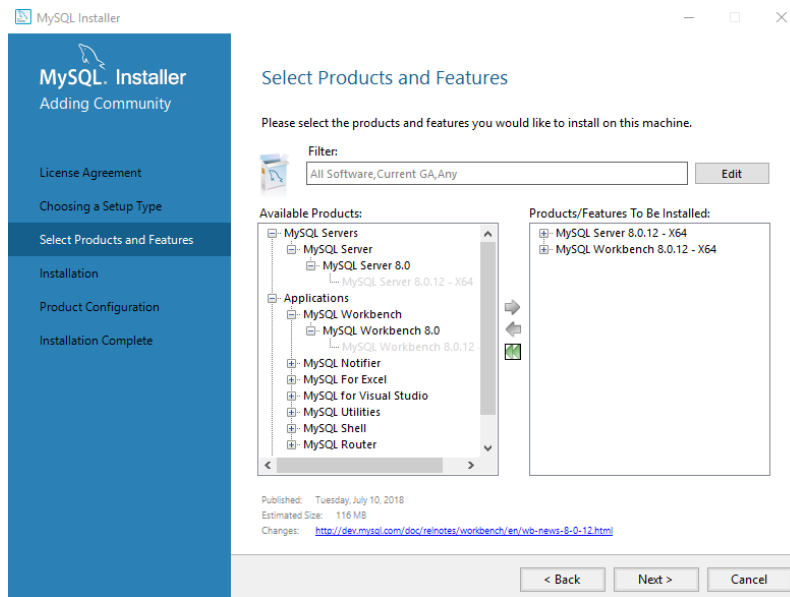


Figure 1.1.6. MySQL server and workbench tools setup

- 9) Click *Next*.
10) Click *Execute*.
11) Click *Next* and follow the steps for the configuration of the MySQL Server.

1.4. Web server: Apache Tomcat

- 1) Access the next link: <https://tomcat.apache.org/download-80.cgi>.
2) Under *Binary Distributions* look for *Core* and click on *zip*.

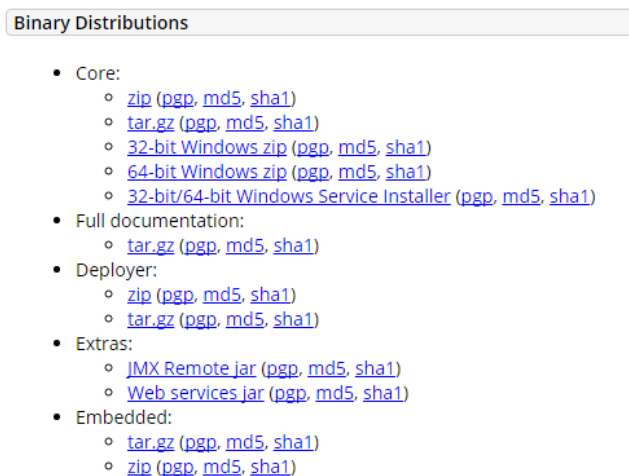


Figure 1.7. Apache Tomcat distributions

- 3) A file called *apache-tomcat-version.zip* is downloaded.
- 4) Extract the content of this file on C:\. The file *startup.bat* should be at the location *C:\apache-tomcat-version\bin*.
- 5) Set the CATALINA_HOME variable
 - Click *Start*.
 - Right-Click on *Computer*.
 - Select *Properties*.
 - Click on *Advanced System Settings*.
 - Click on *Environment Variables*.
 - Under *System Variables* click *New*.
 - In the text field associated with the name of the variable insert CATALINA_HOME and in the field associated with the value of the variable insert *C:\apache-tomcat-version;*

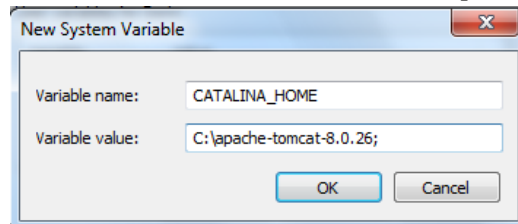


Figure 1.8. Setting the Apache Tomcat environment variable

- Click *OK*.

1.5. Version control system: Git

- 1) Access <https://git-scm.com/downloads>.
- 2) Select your operating system.



Figure 1.9. Git download options

- 3) If you select Windows, a file called *Git-2.10.0-64-bit.exe* should be downloaded. In the case you select another operating system or if your system is on 32 bits then a file with a similar name should be downloaded.
- 4) Click on this file and follow the default installation guidelines, *except for the step where you are asked which terminal emulator you want to use*. Select the second option as illustrated in the picture below.

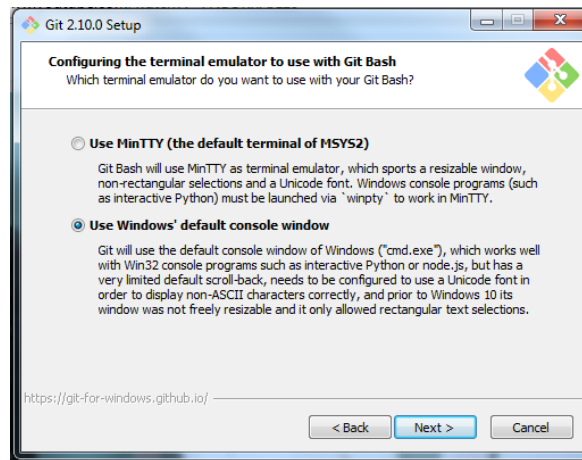


Figure 1.10. Git terminal emulator selection

1.6. Version control repository: Bitbucket

- 1) Access <https://bitbucket.org/>.
- 2) Click on *Get Started*. You will be asked to fill your personal information.

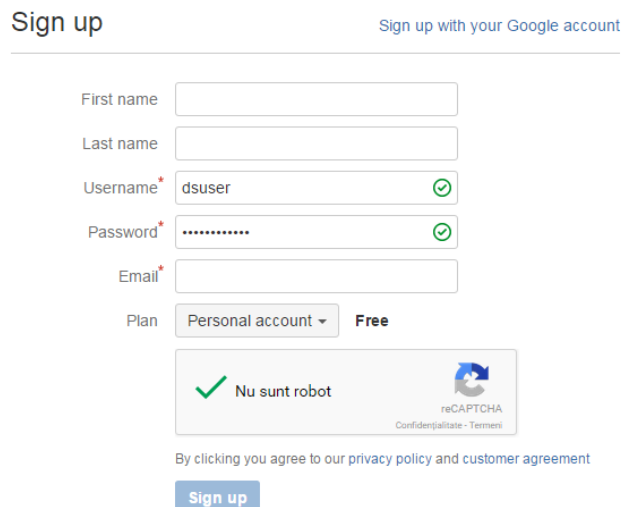


Figure 1.11. Creating Bitbucket account

- 3) You will be asked to create a new repository. Choose *Empty* and give the name *DS_Group_LastName_FirstName* to your new repository.
- 4) Click *Done*.

1.7. Configuring the lab working environment

The first step to create the laboratory working environment is to create a project from scratch by following the below instructions:

- 1) Create the folder *DS_Group_LastName_FirstName* on *D:*.
- 2) Right click on this folder and click *Git Bash Here*.
- 3) Execute the next commands to connect with the Bitbucket account:
 - a) *git init*
 - b) *git remote add origin https://dsuser@bitbucket.org/dsuser/ds_group_lastname_firstname.git*
- 4) Open Eclipse, select *File -> New -> Project... -> Maven -> Maven Project* and click *Next*.
- 5) Instead of using the default Workspace location use this one:
D:\DS_Group_LastName_FirstName
- 6) Click *Next*.
- 7) Introduce the next parameters:
 - a) Group id: *ds.demo*
 - b) Artifact id: *DemoProject*
- 8) Click *Finish*.
- 9) In order to see the files of the form *.filename* click on *View Menu -> Filters...* and unselect the option *.* resources*.



Figure 1.12. Eclipse menu configuration

- 10) Right click on the files *.settings*, *target*, *.classpath*, *.project* and select *Team -> Ignore*.
- 11) The file *.gitignore* will contain the files which will not be committed to the repository. You can also edit this file manually.

```
1 /target/  
2 /.settings/  
3 /.classpath  
4 /.project  
5
```

Figure 1.13. Example of files ignored in commit operation

12) Right click on the folder *DS_Group_LastName_FirstName* and introduce the next commands:

- a) `git add .`
- b) `git commit -a -m "initial commit"`
- c) `git push -u origin master`

To update the project contents please follow the steps below:

- 1) Create a new class named *Main* in the same package as the class *App*.
- 2) Right click on *DS_Group_LastName_FirstName* and select *Git Bash*
- 3) Insert the next commands:
 - a) `git add .`
 - b) `git commit -a -m "add new class"`
 - c) `git pull origin master`
 - d) `git push -u origin master`
- 4) You can always see the modification that were not committed yet by using: `git status`

In specific cases the Internet connection requires a proxy server (e.g. in the UTCN laboratories).

To make Git to work with a proxy server follow the steps:

- 1) Open *Git Bash*
- 2) Insert the following commands:
 - a) `git config --global http.proxy http://proxy.utcluj.ro:3128`
 - b) `git config --global --get http.proxy`
- 3) In order to unset the proxy, use the following command:
`git config --global --unset http.proxy`

Similarly, getting Maven¹ to work with a proxy server is detailed below:

- 1) Go to *Windows Explorer*-> *Drive C*-> *Users* -> *Your User* -> *.m2*
- 2) Create the folder *conf*
- 3) Go to *conf* folder and create the file *settings.xml* with the Figure 1.14 content
- 4) Go back to folder *.m2*
- 5) Delete the folder *repository*
- 6) Open *Eclipse*
- 7) Go to *Window*->*Preferences*->*Maven*->*User Settings*
- 8) At the *User Settings* tab browse for the *settings.xml* file created at step 3
- 9) Click *Apply* and *OK*
- 10) Right click on your project, and go to *Maven*->*Update Project*

¹ <https://maven.apache.org/>

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository/>
  <interactiveMode/>
  <usePluginRegistry/>
  <offline/>
  <pluginGroups/>
  <servers/>
  <mirrors/>
  <proxies>
    <proxy>
      <id>myproxy</id>
      <active>true</active>
      <protocol>http</protocol>
      <host>proxy.utcluj.ro</host>
      <port>3128</port>
      <username></username>
      <password></password>
      <nonProxyHosts>localhost,127.0.0.1</nonProxyHosts>
    </proxy>
  </proxies>
  <profiles/>
  <activeProfiles/>
</settings>
```

Figure 1.14. Maven settings file

2. Request-Reply and Sockets

2.1. Problem statement

Suppose we are requested to create a distributed application with the following requirements:

- A central database is located on a server.
- The database stores a table with students.
- The teachers (Remote Clients) must access the database to:
 - add student information
 - retrieve students by their Identifiers (IDs)
- The information retrieved from the central database is displayed for the users of the client application in a user-friendly Graphical User Interface (GUI).

2.2. Application analysis and design

The problem can be decomposed into the following subsystems:

- Communication protocol
- The server application:
 - Database
 - Data access layer over the database
 - Communication layer over the network
- The client application:
 - Communication layer over the network
 - GUI

We need to create a distributed application over the network. We choose a *client-server* software architecture as depicted in Figure 2.1.

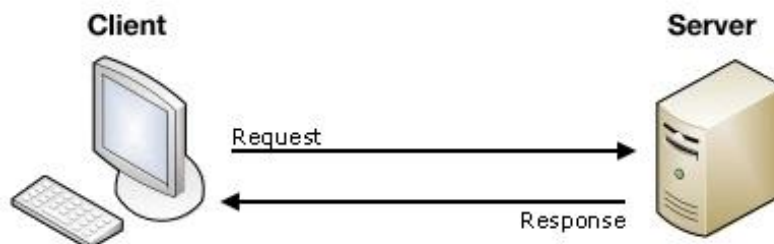


Figure 2.1. Client-server software architecture

For the transport layer of the application we use *sockets*² that assure a two-way communication between the client and the server, thus allowing us to implement a synchronous request-reply communication method.

2.2.1. Defining the message structure

Because we use sockets for communication, the messages are sent as a stream of bytes. We intend sending strings as a stream of bytes through the sockets. Thus, all our messages will be encoded as string.

We analyse the operations that we need to perform:

- OP1: Add a new student
 - Parameters: *Student.lastname*, *Student.firstname*, *Student.mail*
 - Return: *success* followed by the new student's ID or *fail*
- OP2: Return a student by his/her ID
 - Parameters: *Student.ID*
 - Return: *Student*

Each operation sends several pieces of information to the server, as string. We choose to concatenate these strings and separate the information from the message using “_” and “#” tokens.

For executing each operation, two steps are involved: the *client request* and the *server response*.

2.2.2. Client request

We determine that the client needs to send to the server the following information:

- OP1: send data to store an entity with all its fields in the database (e.g. *student*)
- OP2: request a resource from the server by specifying a resource ID

To be able to perform these operations, the message that is passed from the client to the server needs to contain the following information:

- Performed operation (send data to the server – **POST method**; request resource from the server – **GET method**)
- The entity upon which the operation will be executed - the **Uniform Resource Identifier (URI)** or **Uniform Resource Locator (URL)**, in our case the Data Access Object - DAO class that handles students
- Encode the data that needs to be passed to the server – method parameters already described for OP1 and OP2. These parameters will be converted to string and concatenated with “#” separator

² <https://docs.oracle.com/javase/tutorial/networking/sockets/>

By putting all this information in a string, we obtain a message of the following form:

Method_URL_messageBody

e.g. *OP1: POST_student_1#0George#Popescu#mail@mail.com*

OP2: GET_student_1

NOTES:

- Even if both operations can be performed using the same message structure, for the GET operation we will adopt another encoding, because this operation has another semantic meaning. When requesting a resource from the server, the client knows that resource location (in this case the student located in the database with a given ID), thus we will encode this information in the URL - *OP2: GET_student?id=1_*
- For a GET method the body of the sent message will be empty. That is because the URL (the name of the entity) together with the integer (the database ID of the entity) forms the resource identifier (i.e. it contains all the information needed by the server to identify the resource and return it to the client).
- For a POST method, the URL will specify the entity that will be sent to the server. The actual data will be encoded in the body of the message. These encoding will contain all the fields of the entity in the order that they appear in the class.

2.2.3. Server response

We determine that the server needs to answer to the client request by specifying the following:

- OP1: return a code that represents the status of the operation. (e.g. 200 – the operation was successful).
- OP2: return the resource requested by the client as a string encoded with the same rule as the request. In case of an error, return the code corresponding to the encountered error (e.g. 404 if the resource was not found).

Based on this information we determine the following response message structure:

StatusCode_messageBody

The *message_body* contains the returned values as string separated by #.

e.g. *OP1: 200_*

OP2: 200_1#George#20#Cluj#Romania

We define the following status codes for our operations:

- 200 – the operation was successful
- 400 – bad request

- 404 – if the resource was not found
- 405 – operation not allowed

2.3. Application structure and implementation

The solution is implemented in 3 different modules (see Figure 2.2). Each architectural module is detailed below.

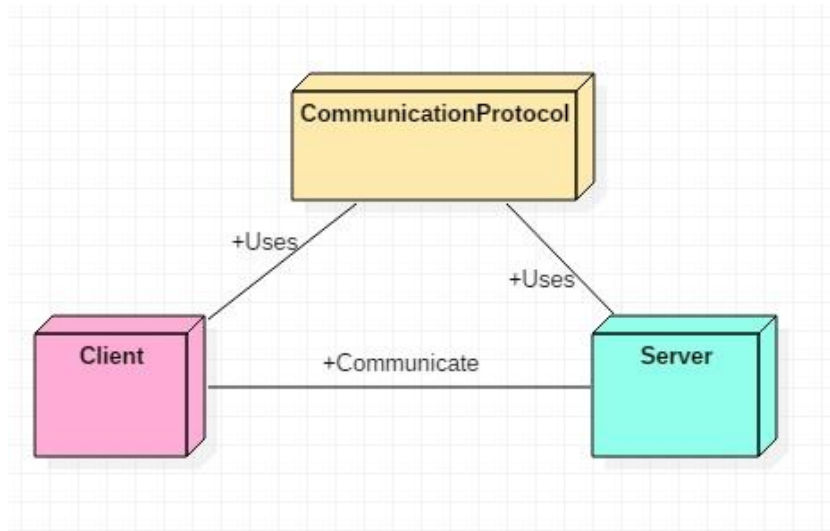


Figure 2.2. Application conceptual architecture

➤ Client

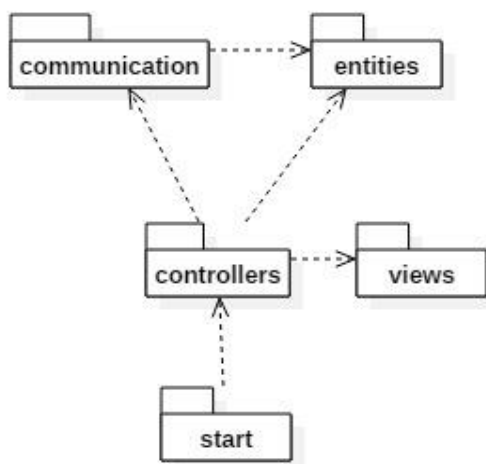


Figure 2.3. Client package diagram

- Communication - package that contains the classes responsible for the communication
- Controllers - package that contains the controller classes
- Entities - package that contains the entity classes
- Start - package that contains the class that starts the application
- Views - package that contains the GUI class

➤ Server

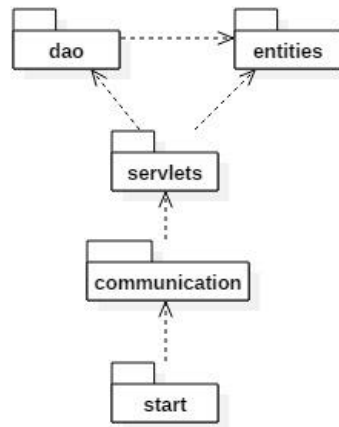


Figure 2.4. Server package diagram

- Communication - package that contains the classes responsible to the communication
- DAO - package that contains the classes responsible to the database access
- Entities - package that contains the entity classes
- Servlets - package that contains the classes that extend an abstract servlet class
- Start - package that contains the class that starts the application

➤ Communication Protocol – library that contains the protocol definition

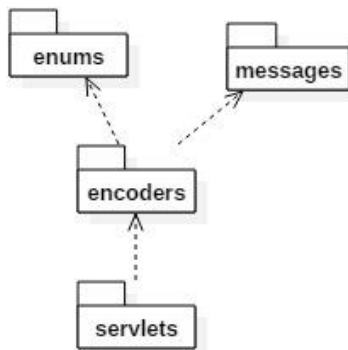


Figure 2.5. Communication protocol package diagram

- Encoders - package that contains the classes responsible with the serialization and deserialization of objects
- Enums – package that contains enumerations
- Messages – package that contains the request and response message classes
- Servlets - package that contains an abstract class with the definition of the servlet

In the next sub-sections, we will present the functionality of the client and server application by means of sequence diagrams and code examples.

2.3.1. The client application

The client makes a request by pressing a button on the GUI, the *CatalogView* class. When pressing the button, the action listener from the controller class, *CatalogController*, is called. There are two

buttons on the GUI, each of them corresponding to one operation specified and has the corresponding listeners (OP1- *PostActionListener* and OP2- *GetActionListener*).

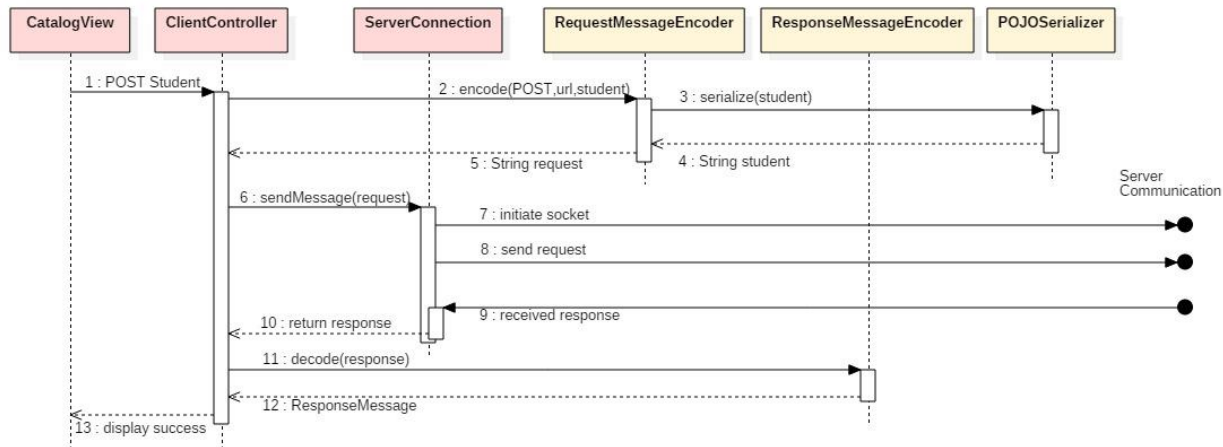


Figure 2.6. Sequence diagram for client POST operation

We present the steps involved in performing the OP1 operation, POST a student (see Figure 2.6). The action is similar for the OP2 operation (GET operation). To make a request and display the response, the client application performs the following steps:

1. When the post button is pressed on the *ClientView*, it triggers the *actionPerformed* method from the corresponding *PostActionListener* class located in the *ClientController* class. Lines 8-11 show how it takes the information from the *ClientView* and creates a *Student* object. Furthermore, line 14 shows how a string containing the request for the server is created using the encode method of the *RequestMessageEncoder* class.

```

1. class PostActionListener implements ActionListener {
2.     @Override
3.     public void actionPerformed(ActionEvent e) {
4.         String firstname = catalogView.getFirstname();
5.         String lastname = catalogView.getLastname();
6.         String mail = catalogView.getMail();
7.         if (!("".equals(firstname) || "".equals(lastname) || "".equals(mail))) {
8.             Student student = new Student();
9.             student.setFirstname(firstname);
10.            student.setLastname(lastname);
11.            student.setMail(mail);
12.            try {
13.                //encode request: POST request, URL "student", sending student object
14.                String encodedRequest =
15.                    RequestMessageEncoder.encode(ProtocolMethod.POST, "student", student);
16.                String response = serverConnection.sendRequest(encodedRequest);
17.                //decode the response from server
18.                ResponseMessage decodedResponse =

```

```

        ResponseMessageEncoder.decode(response);
18.    //if server responded OK, operation was successful, else display error
19.    if (decodedResponse.getStatusCode() == StatusCode.OK.getCode()) {
20.        displayInfoMessage("Successfully inserted; id=" +
                            decodedResponse.getSerializedObject());
21.    } else {
22.        displayErrorMessage("Status code " + decodedResponse.getStatusCode());
23.    }
24.    } catch (IOException ex) {
25.        LOGGER.info(ex.getMessage());
26.        displayErrorMessage(ex.getMessage());
27.    }
28.    }
29.    else {
30.        displayErrorMessage("Please fill all textboxes before submitting!");
31.    }
32. }
33. }

```

Figure 2.7. Post action listener code snippet

- The student is encoded by calling the *encode* method from the class *RequestMessageEncode* (see Figure 2.8). A message is created by concatenating the request method (GET or POST), the URL and the serialization of the object sent as parameter.

```

1. public static String encode(ProtocolMethod method, String url, Object o) {
2.     String messageString = method + "_" + url + "_";
3.     if (o != null) {
4.         if (o instanceof String) {
5.             messageString += o;
6.         } else {
7.             messageString += POJOSerializer.serialize(o);
8.         }
9.     }
10.    return messageString;
11. }

```

Figure 2.8. Encode method code snippet

- The *POJOSerializer* class implements the *serialize* method that receives as parameter a generic *Object* (see Figure 2.9). Using reflection techniques, it gets the class of the object (line 5), and the fields (line 7). Then, it iterates through the fields (line 8), sets the accessibility of the fields to true so they can be read even if private and appends the values of each field delimiting them by the special character “#” (line 12). Finally, it changes back the accessibility of the field to private.

```

1 public static String serialize(Object o) {
2     String result = "";
3     try {

```

```

4      //get the class type of the object
5      Class c = o.getClass();
6      //get the fields of that class
7      Field[] fields = c.getDeclaredFields();
8      for (Field f : fields) {
9          //set accessibility to true (can be read even if private)
10         f.setAccessible(true);
11         //append the field to the result
12         result += f.get(o) + "#";
13         f.setAccessible(false);
14     }
15     } catch (IllegalAccessException e) {
16         LOGGER.error("", e);
17     }
18     return result;
19 }

```

Figure 2.9. Serialize method code snippet

4. Step 4 from the sequence diagram returns the serialized student as a string to the *encode* method of the *RequestMessageEncoder* object while step 5 returns the encoded request from the *RequestMessageEncoder* to the *ClientController*.
5. Having the message encoded as a stream of bytes, the client application will send it to the server. Because it is a synchronous communication, the client waits until the server sends back the response. The method *sendRequest* from the *ServerConnection* class is called.
6. A socket connection to the server is opened (line 2). A set of output streams and input streams are opened on the socket to communicate with the server (lines 3,4).
7. The serialized request created at step 4 is written as a stream on the socket and sent to the server (line 5). Then, the client waits for a response (lines 6-12).
8. During this time, the client application is blocked waiting for the server response. After the response is received from the server, the socket and the connections are closed (lines 13-15) and the response is returned. The implemented behavior, synchronous (by waiting for the server response) and stateless (by closing each connection after every request) mimics the HTTP behavior.
9. The response is returned to the client application and stored in the *response* String.

```

1  public String sendRequest(String messageToSend) throws IOException {
2      Socket clientSocket = new Socket(host, port);
3      ObjectOutputStream outToServer =
4          new ObjectOutputStream(clientSocket.getOutputStream());
5      ObjectInputStream inFromServer =
6          new ObjectInputStream(clientSocket.getInputStream());
7      outToServer.writeObject(messageToSend);
8      String response;

```

```

7      try {
8          response = (String)inFromServer.readObject();
9      } catch (ClassNotFoundException e) {
10         response = null;
11         LOGGER.error("", e);
12     }
13     outToServer.close();
14     inFromServer.close();
15     clientSocket.close();
16     return response;
17 }

```

Figure 2.10. SendRequest method class code snippet

10. The response message is de-serialized by *decode* method of the *ResponseMessageEncoder* class and a *ResponseMessage* object is created (see Figure 2.11). The message received is split by the separator character “_” (line 5). Each of the resulting substrings will represent a part of the response message: method (GET or POST), URL (that can contain parameters, checked in lines 9-17) and message body, that will be de-serialized.

```

1. public static RequestMessage decode(String m) {
2.     RequestMessage requestMessage = null;
3.     //split the encoded message by the separator _
4.     //the splitMessage array should now contain at least 2 elements (METHOD and
5.     //URL) + optionally a third, BODY
6.     String[] splitMessage = m.split("_");
7.
8.     if (splitMessage.length >= REQUEST_MIN_COMPONENTS_NUM) {
9.         requestMessage = new RequestMessage();
10.        //set the method field of the requestMessage to the value of
11.        //splitMessage[0] (METHOD)
12.        requestMessage.setMethod(ProtocolMethod.valueOf(splitMessage[0]));
13.
14.        //split the url by separator ?, to check if there are parameters
15.        String[] splitUrl = splitMessage[1].split("\\?");
16.        //splitUrl[0] is the url, assign it to url field of requestMessage
17.        requestMessage.setUrl(splitUrl[0]);
18.
19.        //check if there was ? in the url (2 strings generated by previous split)
20.        if(splitUrl.length == METHOD_WITH_PARAMS_COMPONENTS_NUM) {
21.            //further split by & keyword
22.            String[] splitQuery = splitUrl[1].split("&");
23.
24.            //for each key=value pair, split in key and value
25.            for (String s : splitQuery) {
26.                String[] splitKeyValue = s.split("=");
27.                requestMessage.getQueryValues().put(splitKeyValue[0],

```

```

22.         splitKeyValue[1]);
23.     }
24.     //if there is a 3rd component to the request message (i.e. BODY), assign
25.     //it to serializedObject of requestMessage
26.     if (splitMessage.length == REQUEST_ALL_COMPONENTS_NUM) {
27.         requestMessage.setSerializedObject(
28.             splitMessage[REQUEST_ALL_COMPONENTS_NUM - 1]);
29.     }
30.     return requestMessage;

```

Figure 2.11. Decode method code snippet

11. Because it is a POST action, it will return only the status code of the operation. However, we also return the ID of the student because it is auto-generated and would not be known otherwise.

12. The information about the POST operation is displayed in the GUI.

2.3.2. The server application

The server responds to each client request. It has a thread that runs and listens for incoming connections from clients, in the *Server* class. Each time a new client sends a request, it establishes a connection to the client, creates a separate thread for that client (*Session* class), receives the message, processes it and returns a reply to the client.

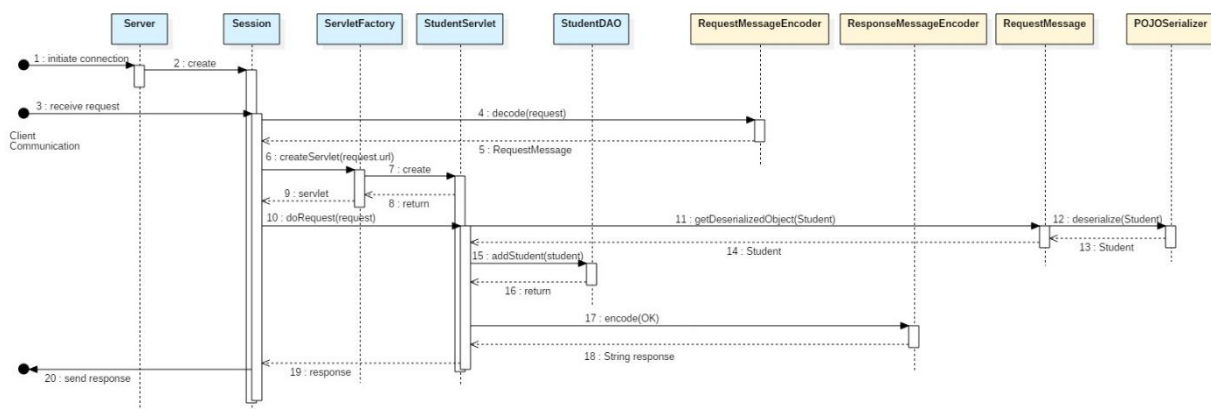


Figure 2.12. Sequence diagram for server POST operation

Below the processing on the server side when a POST message is received from a client is detailed:

1. The client initiates a connection. The thread from the *Server* class contains an infinite loop that listens and accepts the incoming connections (see Figure 2.13).

- When a new connection appears (Line 6) a new thread contains a session for that connection (line 7) is created and then started (line 8)

```

1. public void run() {
2.     while (true) {
3.         try {
4.             synchronized (this) {
5.                 Socket clientSocket;
6.                 clientSocket = serverSocket.accept();
7.                 Session cThread = new Session(clientSocket);
8.                 cThread.start();
9.             }
10.        } catch (IOException e) {
11.            LOGGER.error("", e);
12.        }
13.    }
14.}

```

Figure 2.13. Run method from Server class code snippet

- The *Session* class extends the *Thread* class and overrides the run method implementing the response behavior of a request-reply message. Using multiple instances of this class, the server can handle multiple client connections simultaneously (see Figure 2.14). Line 6 saves the incoming message as a string read from the socket input stream connection. The message is then decoded using the *RequesMessageDecoder decode* method already explained in the upper section. Then, the request message is checked for validity (line 11). If the message is not null, an *AbstractServlet* is instantiated based in the URL of the request (line 16). If the URL was valid and a servlet has been created, the request is processed by the *doRequest* method of the *AbstractServlet*. Finally, the request is sent back to the clients (line 26), this thread closes connections (line 30) ends its execution.
- The session receives a request from the client application, and the run method of the session thread is executed.
- The message is decoded (Figure 2.14 – line 8). This is performed by the *RequestMessageEncoder* class already described above.
- The *RequestMessage* is returned in line 8 from Figure 2.14, containing the method (POST or GET), the URL of the resource, the URL parameters sent in a <key,value> map, and the body in string format.
- Based on the request URL the *Session* calls the *ServletFactory* to create the *Servlet* asked by the client (line 13 from Figure 2.14).

```

1. @Override
2. public void run() {
3.     String messageReceived;
4.     try {
5.         // Wait for message from client

```



```

6.    messageReceived = (String) inFromClient.readObject();
7.    // Decode the request from the received message
8.    RequestMessage decodedRequest =
        RequestMessageEncoder.decode(messageReceived);
9.    String response;
10.   // Prepare response
11.   if (decodedRequest != null) {
12.       // Attempt creating the servlet that handles the request for the URL
13.       AbstractServlet abstractServlet =
           ServletFactory.createServlet(decodedRequest.getUrl());
14.       // Servlet successfully created, actually process the request
15.       if (abstractServlet != null) {
16.           response = abstractServlet.doRequest(decodedRequest);
17.       }
18.       // Servlet could not be created for the URL, mapping not found
19.       else {
20.           response = ResponseMessageEncoder.encode(StatusCode.NOT_FOUND, null);
21.       }
22.   } else {
23.       response = ResponseMessageEncoder.encode(StatusCode.BAD_REQUEST, null);
24.   }
25.   // Send encoded response
26.   sendMessageToClient(response);
27.   } catch (ClassNotFoundException | IOException e) {
28.       LOGGER.error("", e);
29.   }
30.   closeAll();
31. }
32. }

```

Figure 2.14. Run method from Session class code snippet

7. In our implementation we consider that each URL is mapped to a resource named Servlet derived from the base class AbstractServlet shown in Figure 2.15 that has implemented methods for each request type (GET –line 19 or POST –line 18). If there is no Servlet for an URL or the method is not implemented, then an exception is thrown. Furthermore, the class also implements the *doRequest* method (lines 3-17) that contains a switch statement which chooses the method to be executed by the *RequestMessage* method filed (line 5).

```

1. public abstract class AbstractServlet {
2.     private static final Log LOGGER = LogFactory.getLog(AbstractServlet.class);
3.     public String doRequest(RequestMessage message) {
4.         try {
5.             switch (message.getMethod()) {
6.                 case GET:
7.                     return doGet(message);
8.                 case POST:
9.                     return doPost(message);
10.                default:
11.                    return
                        ResponseMessageEncoder.encode(StatusCode.BAD_REQUEST);

```

```

12.         }
13.     } catch (UnsupportedOperationException e) {
14.         LOGGER.error("", e);
15.         return ResponseMessageEncoder.encode(StatusCode.NOT_ALLOWED);
16.     }
17. }
18. public abstract String doPost(RequestMessage message);
19. public abstract String doGet(RequestMessage message);
20. }

```

Figure 2.15. AbstractServlet class code snippet

7. The *ServletFactory* creates the Servlet mapped to the URL or throws an exception if no servlet is found matching the given URL. It uses reflection to create an instance of an object given its class name. The naming convention assumes that the servlets are located in the package *servlets* and each servlet has the name in the format *UrlServlet* (e.g. for the url = "Student" the class name will be *servlets.StudentServlet*). In Figure 2.16 lines 2-4 create a string containing the servlet class name based on a given URL. An *AbstractServlet* is declared to be instantiated (line 5). A class with the servlet name is searched in line 8, and if not found null is returned (lines 10-11). Else, a constructor is returned for the given class (line 13) and the *AbstractServlet* is instantiated with the given constructor (line 14).

```

1. public static AbstractServlet createServlet(String url) throws
    ClassNotFoundException {
2.     String className = "ro.tuc.dsrl.ds.handson.assig.one.server.servlets.";
3.     className +=
        url.replace(url.charAt(0), Character.toUpperCase(url.charAt(0)));
4.     className += "Servlet";
5.     AbstractServlet abstractServlet = null;
6.     Class<?> clazz;
7.     try {
8.         clazz = Class.forName(className);
9.
10.        if (clazz == null) {
11.            return null;
12.        }
13.        Constructor<?> ctor = clazz.getConstructor();
14.        abstractServlet = (AbstractServlet)ctor.newInstance();
15.    } catch (InvocationTargetException | NoSuchMethodException |
        IllegalAccessException | InstantiationException |
        ClassNotFoundException e) {
16.        LOGGER.error("", e);
17.    }
18.    return abstractServlet;
19. }

```

Figure 2.16. Method createServlet code snippet

8. A Servlet class must extend the *AbstractServlet* class and override the two abstract methods *doGet* and *doPost*. The *StudentServlet* class extends the *AbstractServlet* class and implements the desired functionality for handling the request. The *doPost* method (lines 7-17 from Figure 2.17) contains the code for inserting a student in the database. For this, a *StudentDAO* object was declared in line 2.

```

1. public class StudentServlet extends AbstractServlet {
2.     private StudentDAO studentDao;
3.     public StudentServlet() {
4.         studentDao = new StudentDAO(new
5.             Configuration().configure().buildSessionFactory());
6.     }
7.     @Override
8.     public String doPost(RequestMessage message) {
9.         String response;
10.        Student student = message.getDeserializedObject(Student.class);
11.        if (student != null) {
12.            studentDao.addStudent(student);
13.            response = ResponseMessageEncoder.encode(StatusCode.OK,
14.                String.valueOf(student.getId()));
15.        } else {
16.            response = ResponseMessageEncoder.encode(StatusCode.BAD_REQUEST);
17.        }
18.        return response;
19.    }
20.    @Override
21.    public String doGet(RequestMessage message) {
22.        String response;
23.        String id = message.getQueryValues().get("id");
24.        if (id != null) {
25.            try {
26.                Student student = studentDao.findStudent(Integer.parseInt(id));
27.                if (student == null) {
28.                    response =
29.                        ResponseMessageEncoder.encode(StatusCode.NOT_FOUND);
30.                } else {
31.                    response = ResponseMessageEncoder.encode(StatusCode.OK,
32.                        student);
33.                }
34.            } catch (NumberFormatException e) {
35.                response = ResponseMessageEncoder.encode(StatusCode.BAD_REQUEST);
36.            }
37.        } else {
38.            response = ResponseMessageEncoder.encode(StatusCode.BAD_REQUEST);
39.        }
40.        return response;
41.    }
42. }

```

Figure 2.17. StudentServlet class code snippet

9. A *Student* object is created from the *RequestMessage* object by calling the *getDeserializedObject* of the *RequestMessage* class (line 9 from Figure 2.17)
10. The actual message is decoded by reflection techniques by the *POJOSerializer* class.
11. A *Student* object is returned
12. The *StudentServlet* class calls the corresponding *addStudent* method from the *StudentDAO* class (line 11 from Figure 2.17). This class is implemented using Hibernate³. A part of the class is shown in Figure 2.18. The *add* method defined in lines 10-28 uses Hibernate specific methods to access the database. Initially, a session is opened (line 12) and a Transaction is defined (line 13) The Transaction is opened (line 15) and the *save* method is called with the *student* parameter (line 16). Finally, the transaction commits (line 18) and the session is closed (line 25).

```
1. public class StudentDAO {
2.     private static final Log LOGGER = LogFactory.getLog(StudentDAO.class);
3.
4.     private SessionFactory factory;
5.
6.     public StudentDAO(SessionFactory factory) {
7.         this.factory = factory;
8.     }
9.
10.    public Student addStudent(Student student) {
11.        int studentId = -1;
12.        Session session = factory.openSession();
13.        Transaction tx = null;
14.        try {
15.            tx = session.beginTransaction();
16.            studentId = (Integer) session.save(student);
17.            student.setId(studentId);
18.            tx.commit();
19.        } catch (HibernateException e) {
20.            if (tx != null) {
21.                tx.rollback();
22.            }
23.            LOGGER.error("", e);
24.        } finally {
25.            session.close();
26.        }
27.        return student;
28.    }
```

Figure 2.18. StudentDAO class code snippet

13. The inserted entity is returned

³ <http://hibernate.org/orm/>

14. If the operation succeeded, a return message with the status code OK and the ID of the student is created by the *ResponseMessageEncoder* class
15. The string containing the encoded message is returned
16. The response is returned to the servlet
17. The response is send back to the client through the sockets. The session closes.

2.4. Building and running the example

1. Setup GIT and download the project from
https://bitbucket.org/utcn_dsrl/ds.handson.addignment-1/
 - Create an empty local folder in the workspace on your computer
 - Right-click in the folder and select *Git Bash*
 - Execute commands:
 - *git init*
 - *git remote add origin https://bitbucket.org/utcn_dsrl/ds.handson.addignment-1.git*
 - *git pull origin master*
2. Import the DB script *assignment-one-db.sql* in MySQL. You can use MySQL WorkBench, go to *Server -> Data Import -> Import*.
3. Import the project into Eclipse: *FILE-> Import->Maven-> Existing Maven Projects-> Browse* for project in the folder created at step 1
4. Modify the *hibernate.cfg.xml* file from the *server* module (*main/src/resources/*):
 - Change the hibernate connection URL to localhost (the IP from line 16 to *localhost*)
 - Set the username and password from your local MySQL server (line 19 and 22)
5. Run the project:
 - Run the *ServerStart* class from the server module, package start
 - Run the *ClientStart* class from the client module, package start

2.5. Laboratory work: web application using request – reply

2.5.1. Requirements

Design, implement and test a three-tiered distributed system to view and post flights for an airport. The system consists of the following tiers: presentation, business layer and data access.

Functional requirements:

- Users log-in. Users are redirected to the page corresponding to their role (Client or Administrator).
- Client role
 - A client can view on his/her page all the flights in a list or in a table.
- Administrator role

- The administrator can perform CRUD (Create, Read, Update and Delete) operations on flights
- Each flight consists of the following information: flight number, airplane type, departure city, departure date and hour, arrival city, arrival date and hour (the local time of the flight arrival at the destination cities is computed based on cities geographical coordinates).
- Each city has associated its geographical coordinates: latitude and longitude.
- To display the local time, the geographical coordinates of the city are passed to an external web service⁴ which will return the actual time values.

Implementation technologies: HTML, Java Servlets and Hibernate.

Non-functional requirements: Security - use authentication to restrict users access (cookies, session, etc.). The client users will not be able to enter the administrator page (e.g. by log-in and then copy-paste the admin URL to the browser).

2.5.2. Deliverables

- A solution description document (about 4 pages, Times New Roman, 10pt, single spacing) containing:
 - Conceptual architecture of the distributed system.
 - DB design.
 - UML Deployment diagram.
 - Readme file containing build and execution considerations.
- Source files. The source files and the database dump will be uploaded on the personal *bitbucket* account, following the steps:
 - Create a repository on *bitbucket* with the exact name: *DS_Group_Name_Assignment_1*
 - Push the source code and the documentation (push the code not an archive with the code or war files)
 - Share the repository with the user *utcn_dsrl*

2.5.3. Evaluation

Table 2.1. shows how grading is performed for this assignment.

⁴ <http://new.earthtools.org/webservices.htm>

Table 2.1. Web application laboratory work grading details

Points	Requirements
5 p	<ul style="list-style-type: none"> • HTML page for presentation, Servlets for business logic and Hibernate for data access • DB • Documentation
1 p	Log-in with redirect (admin/clients)
1 p	Call external web service
1p	Minimum Security: the simple users will not be able to enter the administrator page
2p	Correct answers to assignment related topics: <ul style="list-style-type: none"> • URI and URL • Web Clients and Web Servers • HTTP protocol • GET and POST HTTP methods • HTML web forms • Query strings • Cookies • Session • Java Servlet • Object-Relational Mapping (ORM)

2.6. Bibliography

- [1] http://www.coned.utcluj.ro/~salomie/DS_Lic/
- [2] Lab Book: I. Salomie, T. Cioara, I. Anghel, T. Salomie, *Distributed Computing and Systems: A practical approach*, Albastra, Publish House, 2008, ISBN 978-973-650-234-7
- [3] Hibernate:
- <http://www.tutorialspoint.com/hibernate/>
 - <http://www.javatpoint.com/hibernate-tutorial>
 - <http://www.javacodegeeks.com/2015/03/hibernate-tutorial.html>
 - <http://www.mkymong.com/tutorials/hibernate-tutorials/>
- [4] Maven: <https://maven.apache.org/>
- [5] Servlets:
- <http://docs.oracle.com/javaee/6/tutorial/doc/bnafd.html>
 - <http://www.tutorialspoint.com/servlets/>
 - <http://www.javatpoint.com/servlet-tutorial>
 - <http://www.javacodegeeks.com/2014/12/java-servlet-tutorial.html>
- [6] HTML web forms – Servlets interaction: <http://www.tutorialspoint.com/servlets/servlets-form-data.htm>
- [7] JSP: <http://www.tutorialspoint.com/jsp/>
- [8] JSF: <http://www.tutorialspoint.com/jsf>

3. Remote Procedure Call and Distributed Objects

3.1. Problem statement

Suppose we are requested to create a distributed application for computing a car pollution tax using a computational expensive algorithm that cannot be run on any client machine. Thus, the algorithm is run on a remote physical machine having more resources (the server). The customers (remote clients) want to use the algorithm to compute the tax for their cars by sending data to the server and receiving the computation results to be displayed.

The client application sends the data regarding the car to the server. The car contains the following fields:

- *int year* – fabrication year
- *int engineCapacity* – engine size in cmc

Based on this data, the server will compute the tax value using the following formula:

$$tax = \left(\frac{engineCapacity}{200} \right) * sum \quad (1)$$

where *sum* depends on the engine size from Table 3.1.

Table 3.1. Relation between engine size and specific sum

Engine Size	Sum
<1600	8
1601-2000	18
2001-2600	72
2601-3000	144
>3001	290

NOTE: The formula is a simple one for this tutorial purpose only. Usually, the method from the server is a computational intensive calculus that requires more physical resources than are available on the client.

3.2. Application analysis and design

From the problem requirements we notice an important aspect: the algorithm used to compute the tax for the cars is computational intensive, thus being unsuited for the clients to run it locally on

their physical machines. Consequently, the chosen solution will be a distributed application having client-server architecture. The server, having more physical resources, will run the computational intensive algorithm. The server will expose a method that must be executed remotely by the client, leading to a remote procedure call technique.

The solution can be decomposed into the following subsystems:

- Communication protocol – remote method invocation between client and server
- The server application
 - Algorithm
 - Remote invocation
 - Communication layer over the network
- The client application
 - Communication layer over the network
 - Remote invocation

3.2.1. General architecture

We need to create a distributed application over the network. We choose *client-server* software architecture and a request-reply communication paradigm identical to the one from previous chapter (Figure 2.1).

For the transport layer of the application we use *sockets*⁵ that assure a two-way communication between the client and the server, thus allowing us to implement a synchronous request-reply communication method.

3.2.2. Communication mechanism

This section defines the message structure that will allow a remote method invocation, or Remote Procedure Call (RPC).

During the invocation of a method, the parameters are stored on the stack and the control is passed to the code section located at the address mapped to the procedure name. What is important to notice is that a procedure is defined by its name (that maps to an address in the memory where the actual code is located) and its parameters.

In an Object-Oriented Programming (OOP) environment, we have a Remote Method Invocation (RMI) technique that allows invoking a method from a remote object. In this case, we must know the object address (or name), the method name and its parameters. Furthermore, in a distributed environment, to identify a remote object, besides knowing the object name (and implicitly its memory address) we must also know the address of the server where it is located.

⁵ <https://docs.oracle.com/javase/tutorial/networking/sockets/>

Basically, this RPC/RMI technique introduces an intermediate layer between the method call and its actual execution, mainly because the method call happens on the client and the execution on the server.

For the client to make the call, it must know the signature of the method (name, parameters and return type). The signature of a method is defined in OOP languages in an interface. Thus, we might assume that the methods from the server are defined in an interface. This client has also a reference to this interface, thus knowing the method signature that will be called. Considering the above aspects, the system communication flow is shown in Figure 3.1.

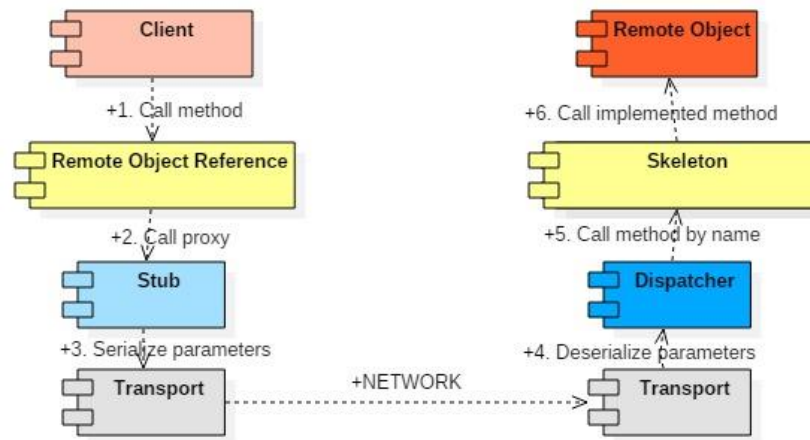


Figure 3.1. RPC system communication flow

The steps involved in calling a remote method are described below.

Client calls the method: The client application makes a call to a special proxy object that implements the remote interface. The client handles this object as it was a local object implementing the interface. The client calls the desired method.

Call forwarded to the proxy: The method call is forwarded to a proxy that has a special implementation of the interface. Instead of implementing the functionality of the methods, this proxy creates a communication mechanism that takes the method's name and parameters and serializes them to be sent over the network.

Data sent over the network: The data is packed and sent over the network. The following information is serialized: remote object name (address space), remote object method and method parameters.

Server receives data: The server receives the data, de-serializes it and sends it to the Dispatcher.

Server calls method: The Dispatcher is responsible for calling the method from the Skeleton that is the interface exposed by the remote object.

Server executes method: The server executes the method with the parameters send from the client. It computes the return value of the method and serializes the result for the client.

Result returned to the client: The result is returned to the client, which de-serializes it and returns it to the Stub as it has been computed locally.

3.3. Application structure and implementation

The solution is implemented in 4 different modules: Client application, Server application, RPC package that contains the classes for remote communication and the Common Classes for both client and server application. The relation between the modules is presented in Figure 3.2.

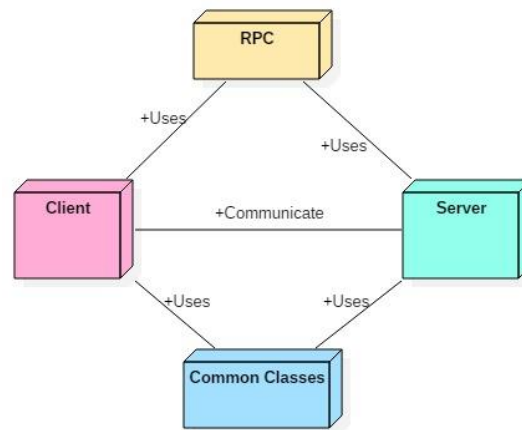


Figure 3.2. Application conceptual architecture

Each module has the following components:

- **Client application** - contains one package (*Communication*) with two classes:
 - *ClientStart* - Class which contains the main method. Here, the remote object is invoked after a reference is created.
 - *ServerConnection* – class that contains the sockets connecting the client with the server
- **Server Application** – contains two packages:
 - *Communication* - contains the server-side communication
 - *Services* – contains the implementation of the remote object
- **Common Classes** – contain two packages:
 - *Entities* - contains the entities (Car)
 - *ServiceInterface* - contains the definition of the interface exposed by the remote object (*Skeleton*)
- **RPC** – library that contains the protocol definition. Contains one package with five classes:

- *Connection* - interface specifying the connection of a client to the server. Such a connection must provide a method to send a message to the server and retrieve the message response.
- *Dispatcher* - dispatches the call received from the client. It interprets the given *Message*, gets the correct object from the registry, calls the required method of that object and then bundles and returns a response *Message*.
- *Message* - represents the object of communication between the client and the server. It contains all the necessary fields for communication. For example, when the client sends the message to the server, the message contains:
 - the endpoint from the *Registry*, which is associated to the remote object
 - the name of the method to be called
 - the arguments of the method, in order
 - when the server replies, it adds the result (return value of the method, or a status message, or an exception) in the arguments array, on the first position.
- *Naming* - provides a static method to look up for a remote object on the server.
- *Registry* - provides a mapping of endpoint-object. It is used by the server to specify which object can be remotely used by a client. The client must identify the object at the endpoint.

3.3.1. Client application

The *Client* application consists of two components, the application domain component, represented by the *ClientStart* class and remote method invocation mechanism represented by the *ServerConnection* class.

The *ClientStart* class has the role of starting the application and has the code presented in Figure 3.3. The main method (lines 5-16) declares an object reference of the remote object, on line 6. Then, it calls the lookup method of the *Naming* service from the remote procedure call package to instantiate the remote object reference (line 8). This call also uses as parameter an instance of the *ServerConnection* class that contains the transport layer access methods. Using the remote object reference newly instantiated, two calls for the remote methods are performed in lines 9 and 10, and the results are printed in the console. Finally, the connection to the server is closed (line 14).

```
1. public class ClientStart {
2.     private static final Log LOGGER = LoggerFactory.getLog(ClientStart.class);
3.     private ClientStart() {
4.     }
5.     public static void main(String[] args) throws IOException {
6.         ITaxService taxService = null;
7.         try {
8.             taxService = Naming.lookup(ITaxService.class,
                                     ServerConnection.getInstance());
9.             System.out.println("Tax value: " + taxService.computeTax(new Car(2009,
                                     2000)));
```

```

10.      System.out.println(taxService.computeTax(new Car(2009, -100)));
11.      ServerConnection.getInstance().closeAll();
12.  } catch (Exception e) {
13.      LOGGER.error("",e);
14.      ServerConnection.getInstance().closeAll();
15.  }
16.  }
17. }

```

Figure 3.3. ClientStart class code snippet

The class *ServerConnection* assures the connection to a server and the communication with it via messages (see Figure 3.4). It implements the *Connection* interface, specified in the *RPC* package, which requires the implementation of the *sendMessage()* method.

```

1.  public class ServerConnection implements Connection {
2.      private static final Log LOGGER = LogFactory.getLog(ServerConnection.class);
3.      private Socket clientSocket;
4.      private ObjectOutputStream outToServer;
5.      private ObjectInputStream inFromServer;
6.      private static ServerConnection instance;
7.      private static final String TERMINATE = "terminate";

8.      private ServerConnection() throws IOException {
9.          clientSocket = new Socket("localhost", 8889);
10.         outToServer = new ObjectOutputStream(clientSocket.getOutputStream());
11.         inFromServer = new ObjectInputStream(clientSocket.getInputStream());
12.     }

13.     public static ServerConnection getInstance() throws IOException {
14.         if (instance == null) {
15.             instance = new ServerConnection();
16.         }
17.         return instance;
18.     }

19.     public Message sendMessage(Message messageToSend) {
20.         Message messageReceived = null;
21.         try {
22.             outToServer.writeObject(messageToSend);
23.             messageReceived = (Message) inFromServer.readObject();
24.         } catch (IOException | ClassNotFoundException e) {
25.             LOGGER.error("", e);
26.         }
27.         return messageReceived;
28.     }

29.     public void closeAll() {...}
30. }

```

Figure 3.4. ServerConnection class code snippet

The host and port of the server with which the socket communication is established is default set here (*localhost*, 8887) in line 9. To establish connection to another location, these values should be changed. A pair of input=output sockets is declared and opened in lines 10-11.

The *ServerConnection* uses the Singleton design pattern⁶ to assure that only one such instance is used. The *getInstance* method defined in lines 13-18 checks if the class has already been instantiated. If this already happened, it returns the previously instantiated object. Otherwise it calls the constructor, which is defined to be private (lines 8-12). The *sendMessage* method (lines 19-27) defines a synchronous communication with the server using sockets and data streams. A message is sent to the socket, through the opened socket and receives a reply. The method waits for the server reply and saves the received message in a variable. The final method of the class is the *closeAll* method that closes the socket connection to the server and releases the resources.

3.3.2. RPC library

The RPC module contains the classes that allow the communication between the client application and the server application, managing the distributed objects and dispatching the remote calls.

The *Registry* class defines a mapping between object names declared as strings and remote object endpoints (see Figure 3.5. line 3).

```
1. public class Registry {
2.     private static Registry ourInstance = new Registry();
3.     private Map<String, Object> endPoints;
4.
5.     private Registry() {
6.         endPoints = new HashMap<String, Object>();
7.     }
8.
9.     public static Registry getInstance() {
10.        return ourInstance;
11.    }
12.
13.    public void registerEndpoint(String endpointName, Object endpoint) {
14.        endPoints.put(endpointName, endpoint);
15.    }
16.
17.    public Object getEndpoint(String endpointName) {
18.        return endPoints.get(endpointName);
19.    }
20.    public void unregisterEndpoint(String endpointName) {
21.        endPoints.remove(endpointName);
22.    }
23. }
```

Figure 3.5. Registry class code snippet

⁶ https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm

Also, the *Registry* implements the Singleton design pattern, having declared a *Registry* field (line 2), a private constructor (line 5) and a *getInstance* method (line 9). It also contains two methods for inserting new pairs of object names and endpoints (lines 13-15), for retrieving the endpoint for an object name (lines 17-19) and for removing an endpoint (lines 20-22).

A *Message* object is defined for communicating with the server. It contains all the necessary fields for communication. For example, when the client sends the message to the server, the message contains: the endpoint (in the *Registry*, which is associated to the remote object), the name of the method to be called and the arguments of the method, in their specific order. When the server replies, it adds the result (return value of the method, or a status message, or an exception) in the arguments array, on the first position.

```
1. public class Message implements Serializable {
2.
3.     private static final long serialVersionUID = 1L;
4.     private String endPoint;
5.     private String methodName;
6.     private Object[] arguments;
7.     ...
8. }
```

Figure 3.6. Message class code snippet

The *Dispatcher* class dispatches the call received from the client (see Figure 3.7.). It interprets the given *Message*, gets the correct object from the registry, calls the required method of that object and then bundles and returns a response *Message*.

The *Dispatcher* class also implements the Singleton design pattern. An instance on the *Dispatcher* type is declared in line 5, a private constructor is defined (lines 7-9) and a *getInstance* method that checks if the object was instantiated and creates the new object is defined in lines 10-12.

The main responsibility of the *Dispatcher* class is the *execute* method implemented in lines 13-34. This method executes the operation requested in the *Message* argument, by interpreting the message object received as parameter. At first, it gets from the registry the object from the required endpoint by the name saved in the message (line 16). Then, it gets the arguments of the method to be called from the argument array of the message (lines 17 - 21). The name of the method to be called is taken from the *method* field of the message and the method is called by invoking the *invoke* method of the reflection package on the method object, with parameters the object reference taken from the registry and its parameters taken from the message argument list (line 25). The result of the execution is saved in an array named *responseArgs*. Then, it is checked if an exception occurred (lines 25-27) or was propagated through execution (line 28) and add the root exception (the cause of *InvocationTargetException*) to response.

Finally, the return value of the method is added to the response *Message* or an exception is added if the method threw an exception (lines 31-32).

```

1. public class Dispatcher {
2.     private static final Log LOGGER = LoggerFactory.getLog(Dispatcher.class);
3.
4.     private Registry registry;
5.     private static Dispatcher ourInstance = new Dispatcher();
6.
7.     private Dispatcher() {
8.         registry = Registry.getInstance();
9.     }
10.
11.     public static Dispatcher getInstance() {
12.         return ourInstance;
13.     }
14.
15.     public synchronized Message execute(Message m) {
16.         Message response = new Message();
17.         Object[] responseArgs = new Object[1];
18.         Object objectEndpoint = registry.getEndpoint(m.getEndPoint());
19.         Class[] argTypes = new Class[m.getArguments().length];
20.         int length = m.getArguments().length;
21.         for (int i=0;i<length;i++) {
22.             argTypes[i] = m.getArguments()[i].getClass();
23.         }
24.         try {
25.             Method method =
26.                 objectEndpoint.getClass().getMethod(m.getMethodName(),argTypes);
27.             responseArgs[0] = method.invoke(objectEndpoint,m.getArguments());
28.         } catch (NoSuchMethodException | IllegalAccessException e) {
29.             LOGGER.error("",e);
30.         } catch (InvocationTargetException e) {
31.             responseArgs[0] = e.getCause();
32.         }
33.         response.setArguments(responseArgs);
34.         response.setEndPoint(m.getEndPoint());
35.         return response;
36.     }
37. }

```

Figure 3.7. Dispatcher class Code Snippet

The *Naming* class, with the code presented in Figure 3.8, has the role of resolving the naming issues: what happens when an object is called by its name. This is done by the *lookup* method, that has the role of looking up on the server registry if there is any class published by the server which implements the given interface. The method receives as parameters an interface of which implementation is to be found on the server, the connection through which to communicate with the server and if there is a remote object published with the interface given as parameter, it returns a Proxy to the object else returns null.

Initially, a special *Message* is composed (lines 6-8) to check if the endpoint is valid. The message is sent in line 9, and a response message is saved. The first position in the arguments field of the response *Message* contains the status of the endpoint verification.

```

1. public class Naming {
2.     private Naming() {
3.     }
4.     public static <T> T lookup(Class<T> clazz, Connection connection) {
5.         Message messageToSend;
6.         messageToSend = new Message();
7.         messageToSend.setEndPoint(clazz.getSimpleName());
8.         messageToSend.setMethodName("checkendpoint");
9.         Message messageReceived = connection.sendMessage(messageToSend);
10.        Object object = messageReceived.getArguments()[0];
11.        if (object != null) {
12.            if (object instanceof String && "OK".equals(object))
13.            {
14.                return (T) Proxy.newProxyInstance(clazz.getClassLoader(), new
15.                    Class[] { clazz }, new ProxyCallHandler(connection));
16.            }
17.            else if (object instanceof String && "ERROR".equals(object)) {
18.                System.out.println("There's no object on the provided endpoint: " +
19.                    clazz.getSimpleName() + " !");
20.            }
21.            return null;
22.        }
23.        return null;
24.    }
25.
26.    private static class ProxyCallHandler implements InvocationHandler {
27.        private Connection connection;
28.        public ProxyCallHandler(Connection connection) {
29.            this.connection = connection;
30.        }
31.
32.        public Object invoke(Object proxy, Method method, Object[] args) throws
33.            Throwable {
34.            Message messageToSend = new Message();
35.            messageToSend.setEndPoint(proxy.getClass().getInterfaces()[0]
36.                .getSimpleName());
37.            messageToSend.setMethodName(method.getName());
38.            messageToSend.setArguments(args);
39.            Message messageReceived = connection.sendMessage(messageToSend);
40.            Object result = messageReceived.getArguments()[0];
41.            if (result instanceof Throwable) {
42.                throw (Throwable) result;
43.            }
44.            return result;
45.        }
46.    }
47. }

```

Figure 3.8. Naming class code snippet

If status is OK, i.e. endpoint is valid, a new *Proxy*, which implements the interface given as parameter, is created in line 14. The *Proxy* has as parameter a *ProxyCallHandler* object. This is a special object that implements the *InvocationHandler* interface from the Java reflection package. It is similar to the *ActionListener* interface and is used to delegate the method calls on the *Proxy* object to the interface's *invoke* method, similar to the mechanism used in Swing when an action to a button triggers the execution of the *actionPerformed* method of the corresponding *ActionListener*.

If the status of the message is "ERROR" it means that no object with the given name has been found and the lookup function will return null.

The *ProxyCallHandler* class is defined in lines 23-40 and implements a special interface of the *java.lang.reflect* package, the *InvocationHandler* interface that defines the *invoke* method. The defined class has a reference to a *Connection* object that defines the *sendMessage* method for sending message between clients and servers.

The *invoke* method defined in lines 28-39 receives as parameters the *proxy* of the remote object, the method to be executed remotely, and an array of objects representing the arguments of the method. The body of the *invoke* method creates a message to be sent to the server (line 29), sets the message endpoint the interface name of the distributed object (line 30) and the remote method name will be set as the name of the method received as parameter (line 31). Next, the arguments of the method are set in the message to be sent (line 32) and the message is sent to the server (line 33). The result is saved in a *Message* object, and the result of the operation is saved in the first element of the argument array. If the result is an exception, it is thrown (line 35) otherwise the method execution result is returned (line 38).

3.3.3. Common classes

The common class package contains the classes and interfaces that must belong to both server and client applications. Basically, it contains the interfaces of the distributed objects, such as the *ITaxService* presented in Figure 3.9 and the classes of the objects from the methods headers, such as the *Car* class in this case. It is crucial that these classes appear in both applications, and any modification of a class should be made on both client and server applications.

```
1. public interface ITaxService {
2.     /**
3.      * Computes the tax to be payed for a Car.
4.      *
5.      * @param c Car for which to compute the tax
6.      * @return tax for the car
7.      */
8.     double computeTax(Car c);
9. }
```

Figure 3.9. ITaxService interface code snippet

3.3.4. Server application

The Server application contains two components: the application domain related component, that contains the distributed objects which implement the interfaces defined in the common class package and the remote method invocation mechanism component that acts as a transparent broker for the remote invocation.

Application Domain Component

The application domain of this example is the computation of the taxes for a car, performed by the *computeTax* method defined in the *ITaxService* interface. The *computeTax* method defined in lines 3-15 implements the rules for computing the tax of a car.

This is only a didactic example. Usually, the computations performed by distributed objects are either computationally intensive or require resources that are unavailable on clients (such as large files, databases, etc).

```

1. public class TaxService implements ITaxService {
2.
3.     public double computeTax(Car c) {
4.         // Dummy formula
5.         if (c.getEngineCapacity() <= 0) {
6.             throw new IllegalArgumentException("Engine capacity must be
                                                    positive.");
7.         }
8.         int sum = 8;
9.         if(c.getEngineCapacity() > 1601) sum = 18;
10.        if(c.getEngineCapacity() > 2001) sum = 72;
11.        if(c.getEngineCapacity() > 2601) sum = 144;
12.        if(c.getEngineCapacity() > 3001) sum = 290;
13.        return c.getEngineCapacity() / 200.0 * sum;
14.    }
15. }

```

Figure 3.10. TaxService class code snippet

Remote Method Invocation Mechanism

The remote method invocation mechanism on the server is composed of three classes: *Server* class responsible for publishing the distributed objects in the *Registry* and waiting for client connections, the *ServerStart* class responsible for starting the application and the *Session* class that creates a *Thread* for each client remote method invocation.

The *ServerStart* class defined in Figure 3.11 defines the port onto which the server will listen for incoming connections from clients, in this case 8889 (line 3). The main method, lines 9-14, instantiates a *Server* object with the given port.

```

1. public class ServerStart {
2.     private static final Log LOGGER = LogFactory.getLog(ServerStart.class);
3.     private static final int PORT = 8889;
4.
5.     private ServerStart() {
6.     }
7.
8.     public static void main(String[] args) {
9.         try {
10.            new Server(PORT);
11.            System.out.println("The server started.");
12.        } catch (IOException e) {
13.            LOGGER.error("",e);
14.        }
15.    }
16. }

```

Figure 3.11. ServerStart class code snippet

The *Server* object (Figure 3.12) implements the *Runnable* interface. It contains a *ServerSocket* for the client connections. An important step is the registration of the distributed object with the name *ITaxService* (line 6) and then the thread is started (line 7). The *run* method defined in lines 9-22, contains an infinite loop that listens for client connections, and when a connection is accepted (line 14), a new session thread for handling the request is created (line 15) and started (line 16).

```

1. public class Server implements Runnable {
2.     private static final Log LOGGER = LogFactory.getLog(Server.class);
3.     private ServerSocket serverSocket;
4.
5.     public Server(int port) throws IOException {
6.         serverSocket = new ServerSocket(port);
7.         Registry.getInstance().registerEndpoint("ITaxService", new TaxService());
8.         new Thread(this).start();
9.     }
10.
11.     public void run() {
12.         while (true) {
13.             try {
14.                 synchronized (this) {
15.                     Socket clientSocket;
16.                     clientSocket = serverSocket.accept();
17.                     Session cThread = new Session(clientSocket);
18.                     cThread.start();
19.                 }
20.             } catch (IOException e) {
21.                 LOGGER.error("",e);
22.             }
23.         }
24.     }
25. }

```

Figure 3.12. Server class Code Snippet

The *Session* class with the code shown in Figure 3.13 handles individually each remote method invocation from the clients. Being a *Thread*, the *Session* executes its *run* method that contains a while loop (lines 14-45). For each client request, it performs three possible operations: if the message received from the client contains the TERMINATE message, it sets the loop condition flag to false (lines 19-22), leading to finishing the thread execution and ending the session. If the message received from the client contains the string “checkpoint”, then the *checkEndpoint* method of the session is executed to determine if the endpoint is valid and there exists an object in the registry which has associated the given endpoint (lines 24-28). Otherwise, the client request is a normal remote method execution invocation that can be handled accordingly by calling the *Dispatcher* in line 27.

The *checkEndpoint* method of the *Session* defined in lines 47-60 checks if there exists an object in the registry mapped to the given endpoint. It creates a message (lines 48-50) and queries the registry to determine if there is an object registered with the endpoint name (line 51). If the endpoint exists it returns an “OK” message otherwise it returns an “ERROR” message (lines 52-59).

The *Session* class also contains the *sendMessageToClient* method for sending messages to the clients through sockets and a method to close the connections when the thread terminates.

```

1. public class Session extends Thread {
2.     private static final Log LOGGER = LogFactory.getLog(Session.class);

3.     private Socket clientSocket;
4.     private ObjectInputStream inFromClient;
5.     private ObjectOutputStream outToClient;

6.     private static final String TERMINATE = "terminate";

7.     public Session(Socket cSocket) {
8.         ...
9.     }

10.
11.     @Override
12.     public void run() {
13.         boolean run = true;
14.         while (run) {
15.             Message messageReceived = null;
16.             Message messageToSend;
17.             try {
18.                 messageReceived = (Message) inFromClient.readObject();
19.                 if (TERMINATE.equals(messageReceived.getMethodName())) {
20.                     run = false;
21.                     messageToSend = new Message();
22.                     messageToSend.setMethodName(TERMINATE);
23.                 } else {
24.                     if ("checkpoint".equals(messageReceived.getMethodName())) {
25.                         messageToSend = checkEndpoint(messageReceived);
26.                     } else {

```

```

27.         messageToSend = Dispatcher.getInstance().execute(messageReceived);
28.     }
29. }
30.     sendMessageToClient(messageToSend);
31. } catch (EOFException e) {
32.     run = false;
33. } catch (SocketException e) {
34.     run = false;
35.     e.printStackTrace();
36. } catch (IOException e) {
37.     LOGGER.error("", e);
38.     closeAll();
39.     break;
40. } catch (ClassNotFoundException e) {
41.     LOGGER.error("", e);
42. }
43. }
44. closeAll();
45. }
46.
47. private Message checkEndpoint(Message messageReceived) {
48.     Message messageToSend = new Message();
49.     Object[] arguments = new Object[1];
50.     String endpointName = messageReceived.getEndPoint();
51.     if (Registry.getInstance().getEndpoint(endpointName) != null) {
52.         arguments[0] = "OK";
53.     } else {
54.         arguments[0] = "ERROR";
55.     }
56.     messageToSend.setArguments(arguments);
57.     messageToSend.setEndPoint(endpointName);
58.     messageToSend.setMethodName(messageReceived.getMethodName());
59.     return messageToSend;
60. }
61.
62. public void sendMessageToClient(Message messageToSend) {
63.     ...
64. }
65.
66. public void closeAll() {
67.     ...
68. }
69. }

```

Figure 3.13. Session class code snippet

3.3.5. Application sequence diagram

In this we present the functionality of the client and server applications by means of sequence diagrams (see Figure 3.14).

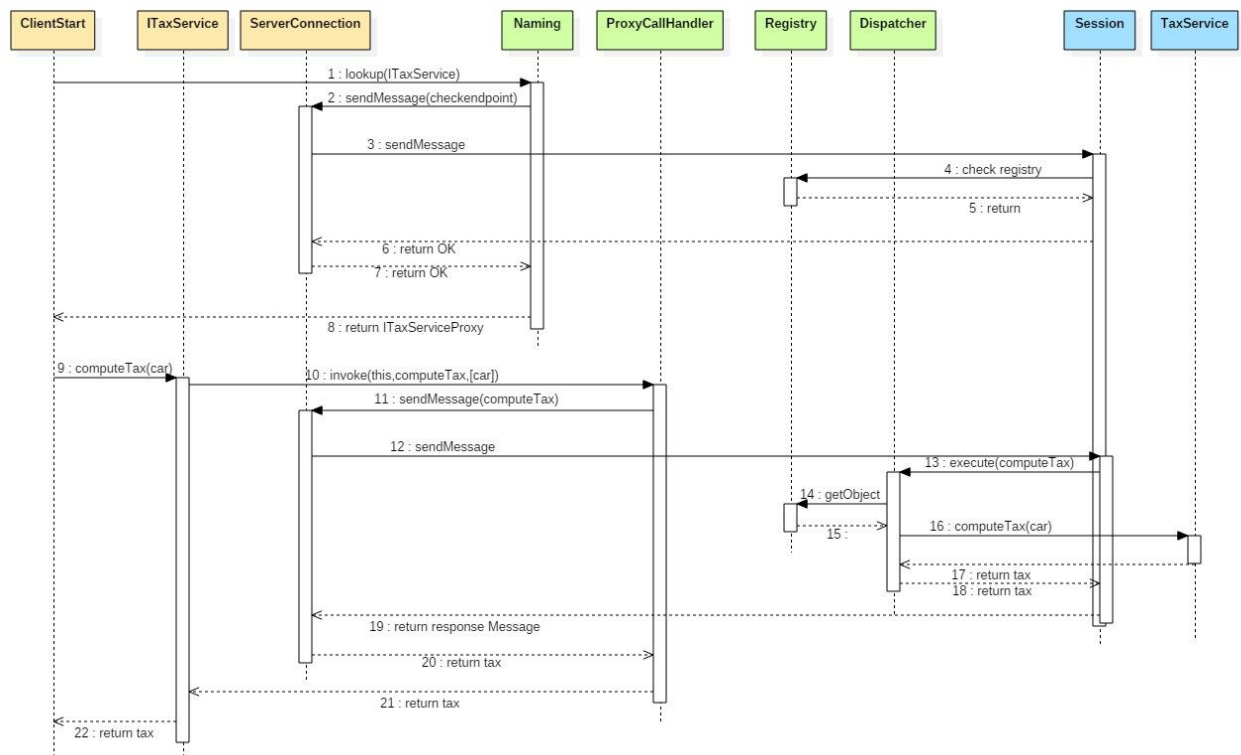


Figure 3.14. Sequence diagram of a RPC call

A RPC call has the following steps:

- The *Server* registers the new object in the *Registry*
- Client looks up for remote object by name
- The RPC *Naming* creates a special message that will check if there exists such an object on the server
- The RPC *Naming* sends the message to the server using a *ServerConnection*
- The *Server* receives the message, and checks in the *Registry* if there exists such an object
- If an object exists, OK is returned, otherwise an exception is thrown
- The *Naming* creates dynamically a reference of the Remote object using only the interface of the remote object, using the *java.lang.reflect.Proxy* class
- The client calls the remote method
- When the method is called the call is redirected automatically to the invoke method from the static *ProxyCallHandler* class
- This method uses reflection to create a message containing the name and parameters of the RPC (in this case the *computeTax* method with parameter *Car*)
- The *ServerConnection* sends the message to the server
- The *Session* receives the message and uses the *Dispatcher* to execute the call

- The *Dispatcher* uses the *Registry* to return a reference of the remote object
- The *Dispatcher* returns a local reference of the remote object.
- The method is called using reflection and is executed in the local *TaxService* object
- The tax is returned, and the *Dispatcher* creates a response *Message* with the results
- The result is returned to the *Session*
- The message is returned to the client
- The message is returned to the *ProxyHandler*. The return value is stored in the arguments array of the message, on the first position. It is extracted as an object and is converted automatically to double by the *Proxy*.
- The tax is returned to the client

3.4. Building and running the example

1. Setup GIT and download the project from
https://bitbucket.org/utcn_dsrl/ds.handson.assignment2.git
 - Create an empty local folder in the workspace on your computer
 - Right-click in the folder and select *Git Bash*
 - Execute commands:
 - `git init`
 - `git remote add origin https://bitbucket.org/utcn_dsrl/ds.handson.assignment2.git`
 - `git pull origin master`
2. Import the project into Eclipse: *FILE-> Import->Maven-> Existing Maven Projects->*
Browse for project in the folder created at step 1
3. Run the project:
 - Run the *ServerStart* class from the server module, package communication
 - Run the *ClientStart* class from the client module, package communication

3.5. Laboratory work: RPC application using distributed objects

3.5.1. Requirements

Design, implement and test a client-server distributed system that uses RPC to compute taxes and selling prices for cars.

Functional requirements:

- Users can fill the information of their cars using a simple GUI (web or desktop):
 - *int year* – fabrication year
 - *int engineSize* – engine size
 - *double price* - purchasing price

- The application uses RPC to send the car information to the distributed object from the server that computes two operations depending on the client request:
 - Tax for a car using formula (1) and Table 3.1 (see Section 3.1)
 - Selling price for a car using formula (2)

$$price_{selling} = \begin{cases} price_{purchasing} - \frac{price_{purchasing}}{7} * (2018 - year) & \text{if } 2018 - year < 7 \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

- The result of the invoked operation, tax, respectively selling price, is displayed on the client GUI.

Implementation technologies: Choose between JAVA RMI or .NET Remoting.

3.5.2. Deliverables

- A solution description document (about 4 pages, Times New Roman, 10pt, single spacing) containing:
 - Conceptual architecture of the distributed system.
 - UML Deployment diagram.
 - Readme file containing build and execution considerations.
- Source files. The source files will be uploaded on the personal *bitbucket* account following the steps:
 - Create a repository on *bitbucket* with the exact name:
DS_Group_FirstName_LastName_Assignment_2
 - Push the source code and the documentation (push the code not an archive with the code or war files)
 - Share the repository with the user *utcn_dsrl*

3.5.3. Evaluation

Table 3.2. shows how grading is performed for this assignment.

Table 3.2. RPC laboratory work grading details

Points	Requirements
5 p	<ul style="list-style-type: none"> • Client – Server application using Java RMI or .NET Remoting with one distributed object and at least one method implemented (tax or price) • Documentation
2 p	Simple GUI (Desktop or Web)

1 p	Both methods (tax and price) implemented in a distributed object
2p	<p>Correct answers to assignment related topics:</p> <ul style="list-style-type: none">• Distributed objects middleware components: Stub, Skeleton, Dispatcher, etc.• JAVA RMI architecture or .NET Remoting architecture• Distributed objects vs local objects• Distributed objects problems: security, latency, life-cycle, etc.

3.6. Bibliography

- [1] http://www.coned.utcluj.ro/~salomie/DS_Lic/
- [2] Lab Book: I. Salomie, T. Cioara, I. Anghel, T.Salomie, *Distributed Computing and Systems: A practical approach*, Albastra, Publish House, 2008, ISBN 978-973-650-234-7
- [3] Java RMI: <https://docs.oracle.com/javase/tutorial/rmi/>
- [4] .NET Remoting: <http://www.codeproject.com/Articles/14791/NET-Remoting-with-an-easy-example>

4. Indirect Communication and Queues

4.1. Problem statement

Suppose we are requested to create a distributed application with the following requirements:

- A client sends a message to a server
- The server may be offline or online when the message is being sent
- After the message was sent, the client must resume its processing without waiting for a response from the server
- The message is processed by the server when it becomes available

Imagine a real-world system with the following requirements:

- The administrator of a web site that sells DVDs must access remotely the application that manages the central database where he/she keeps the information about available stocks
- The administrator adds a new DVD to the database
- Each time a new DVD is added, the application must send automatically notification e-mails to all the subscriber customers to notify them about the new item.

4.2. Application analysis and design

From the problem requirements we notice the following aspects:

- The information is stored in a central node that must be accessed remotely by clients. Consequently, the chosen solution will be a distributed application having client-server architecture.
- The client must continue its execution as soon as it sends the message, without waiting for the server to respond. The standard client-server architecture with the response-reply mechanism is not enough for fulfilling this requirement.

Consequently, the chosen solution will be a distributed application based on the message passing inter-process communication paradigm. The application will be based on a Message Oriented Middleware (MOM) which has 3 major components:

- Message Sender
- Message Repository
- Message Receiver

Each component communicates with the other using the basic synchronous request-reply mechanism. However, by introducing the message repository component between the client and the server, an asynchronous communication is created from two synchronous communication mechanisms.

We need to create a distributed application over the network based on the message passing inter-process communication paradigm.

We start from the client-server software architecture and a request-reply communication paradigm and add an intermediate component: the message repository. Thus, we have two client-server request-reply communications:

- Sender – Message repository
- Message repository – Receiver

An asynchronous communication mechanism is developed:

1. The Sender uses a synchronous request-reply mechanism to send the message to the Message Repository.
2. The Message Repository keeps the messages in the queue
3. The Receiver connects to the Message Repository using a synchronous request-reply mechanism to ask for messages.

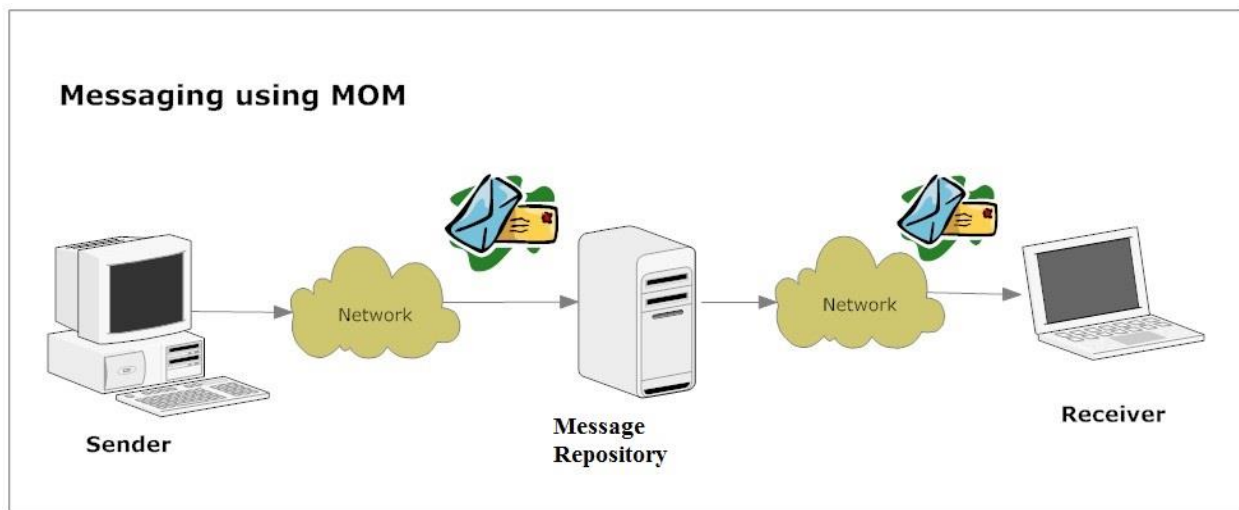


Figure 4.1. MOM software architecture

For the transport layer of the application we again use sockets that assure a two-way communication between the client. This architecture can be mapped on the requirements from subsection 4.1 as follows:

- The sender is the client application used by the administrator to introduce data regarding the DVDs
- The Message Repository is a special application where the sender connects to send the message (the characteristic of the new DVD)
- The receiver is a server that connects to the Message Repository, takes the message (the DVD) and starts sending e-mails to the subscribed customers

4.3. Application structure and implementation

The solution is implemented in 3 different modules, as seen in the conceptual architecture from Figure 4.2. There is a Producer Client, which has some data to process (in this case the message). It sends the data to a Queue Server component which holds a queue with messages and can push to/pop from the queue as required. The Consumer Client will request messages from the Queue Server, to process them.

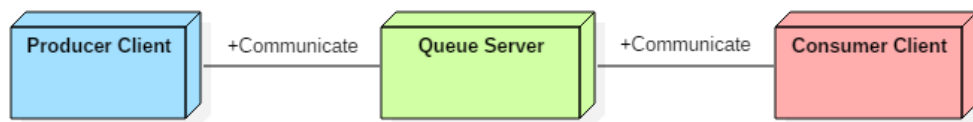


Figure 4.2. Conceptual architecture

Each module components are detailed below:

- **Producer Client**
 - *ClientStart* – class which contains the main method. Here, a *QueueServerConnection* instance is created and a loop sends some messages through it.
 - *QueueServerConnection* – class that contains the sockets connecting the client with the server and methods for communicating with it.
- **Consumer Client**
 - *ClientStart* – class which contains the main method. Here, a *QueueServerConnection* instance is created; an infinite loop will run asking for messages from the queue server and processing them when available.
 - *MailService* – service class which provides a method for sending an email to a specified address.
 - *QueueServerConnection* – class that contains the sockets connecting the client with the server and methods for communicating with it.
- **Queue Server**
 - *Communication* – contains the classes dealing with the communication of the queue server with clients: *Server* (waiting for incoming connections), *Session* (dealing with one request) and *Message* (the class representing the exchange mechanism of the communication).
 - *Queue* – contains the *Queue* class, which is the underlying mechanism; it is a *BlockingQueue* which allows for insertion and removal of elements to/from the structure.

- *Start* – contains the *ServerStart* class, which is the starting point of the component.

4.3.1. Producer client application

Producer client application has two components: *ClientStart* and *QueueServerConnection*.

The *ClientStart* class contains code for creating a connection with the queue server (line 9) using the specified HOST and PORT values and it also contains code for sending messages to the server (lines 12-14). The method *main* sends five messages to be inserted in the queue server.

```

1. public class ClientStart {
2.     private static final String HOST = "localhost";
3.     private static final int PORT = 8888;
4.
5.     private ClientStart() {
6.     }
7.
8.     public static void main(String[] args) {
9.         QueueServerConnection queue = new QueueServerConnection(HOST, PORT);
10.
11.         try {
12.             for (int i=0;i<5;i++) {
13.                 queue.sendMessage("this is email number "+i);
14.             }
15.         } catch (IOException e) {
16.             e.printStackTrace();
17.         }
18.     }
19. }

```

Figure 4.3. ClientStart class code snippet

The *QueueServerConnection* component is responsible for serving the connection between the client and the queue server. It contains two methods: (1) the first method (lines 17-36) sends requests to the server to insert a message in the queue and (2) the second method (lines 46-67) retrieves a message from the queue of the server.

```

1. public class QueueServerConnection {
2.     private String host;
3.     private int port;
4.
5.     public QueueServerConnection(String host, int port) {
6.         this.host = host;
7.         this.port = port;
8.     }
9.
10.    /**
11.     * Sends a request to the server to insert a message into the queue.
12.     *
13.     * @param messageToSend the message to be inserted into the queue
14.     * @return the status of the operation (true == successful)

```

```

15.  * @throws IOException thrown if there is a problem with the connection
16.  */
17.  public boolean writeMessage(String messageToSend) throws IOException {
18.      Socket clientSocket = new Socket(host, port);
19.      ObjectOutputStream outToServer = new
20.          ObjectOutputStream(clientSocket.getOutputStream());
21.      ObjectInputStream inFromServer = new
22.          ObjectInputStream(clientSocket.getInputStream());
23.      outToServer.writeObject(new Message("SEND",messageToSend));
24.
25.      Message response;
26.      try {
27.          response = (Message)inFromServer.readObject();
28.      } catch (ClassNotFoundException e) {
29.          response = null;
30.          e.printStackTrace();
31.      }
32.
33.      outToServer.close();
34.      inFromServer.close();
35.      clientSocket.close();
36.
37.      return (response!=null && response.getType().equals("ACK"));
38.  }
39.
40.  /**
41.   * Retrieves a message from the queue of the server, by sending a "READ"
42.   * request to it.
43.   * If the queue is empty, this method will hang until the queue will get a
44.   * message. In
45.   * other words, this method will wait for the server to provide a message.
46.   *
47.   * @return the message from the queue, sent by the server
48.   * @throws IOException thrown if there is a problem with the connection
49.   */
50.  public String readMessage() throws IOException {
51.      Socket clientSocket = new Socket(host, port);
52.      ObjectOutputStream outToServer = new
53.          ObjectOutputStream(clientSocket.getOutputStream());
54.      ObjectInputStream inFromServer = new
55.          ObjectInputStream(clientSocket.getInputStream());
56.      outToServer.writeObject(new Message("READ",null));
57.
58.      Message response;
59.      try {
60.          response = (Message)inFromServer.readObject();
61.      } catch (ClassNotFoundException e) {
62.          response = null;
63.          e.printStackTrace();
64.      }
65.
66.      outToServer.close();

```

```

61.     inFromServer.close();
62.     clientSocket.close();
63.
64.     if (response==null || !response.getType().equals("ACK")) return null;
65.     return response.getContent();
66. }
67. }

```

Figure 4.4. QueueServerConnection class code snippet

4.3.2. Consumer client application

Consumer client application has three components: *ClientStart*, *MailService* and *QueueServerConnection*.

ClientStart class contains the *main* (line 6) method which starts the application. The application contains an infinite loop that retrieves messages from the queue server and then sends the e-mails as they arrive. The line 16 should be uncommented and completed with the email address, the title of the email and a string message that represents the body of the message.

```

1. public class ClientStart {
2.
3.     private ClientStart() {
4.     }
5.
6.     public static void main(String[] args) {
7.         QueueServerConnection queue = new
8.                                     QueueServerConnection("localhost",8888);
9.         MailService mailService = new
10.                                     MailService("your_account_here","your_password_here");
11.         String message;
12.         while(true) {
13.             try {
14.                 message = queue.readMessage();
15.                 System.out.println("Sending mail "+message);
16.                 //mailService.sendMail("to_mail_address","Dummy Mail
17.                                     //Title",message);
18.             } catch (IOException e) {
19.                 e.printStackTrace();
20.             }
21.         }
22.     }

```

Figure 4.5. ClientStart class code snippet

MailService class uses Gmail SMTP by default for sending emails but the properties can be changed in the constructor if desired. The credentials must be the ones used for the connection to the SMTP server. The constructor (line 14) takes as arguments the *username* and the *password* of

the user and the method *sendMail* sends the actual email with the destination, subject and content that are specified as parameters.

```
1. public class MailService {
2.     final String username;
3.     final String password;
4.     final Properties props;
5.
6.     /**
7.      * Builds a mail service class, used for sending e-mails.
8.      * The credentials provided should be the ones needed to
9.      * authenticate to the SMTP server (GMail by default).
10.     *
11.     * @param username username to log in to the smtp server
12.     * @param password password to log in to the smtp server
13.     */
14.     public MailService(String username, String password) {
15.         this.username = username;
16.         this.password = password;
17.
18.         props = new Properties();
19.         props.put("mail.smtp.auth", "true");
20.         props.put("mail.smtp.starttls.enable", "true");
21.         props.put("mail.smtp.host", "smtp.gmail.com");
22.         props.put("mail.smtp.port", "587");
23.     }
24.
25.
26.     /**
27.      * Sends an email with the subject and content specified, to
28.      * the address specified.
29.      *
30.      * @param to address to send email to
31.      * @param subject subject of the email
32.      * @param content content of the email
33.      */
34.     public void sendMail(String to, String subject, String content) {
35.         Session session = Session.getInstance(props,
36.             new javax.mail.Authenticator() {
37.                 protected PasswordAuthentication getPasswordAuthentication() {
38.                     return new PasswordAuthentication(username, password);
39.                 }
40.             });
41.
42.         try {
43.
44.             Message message = new MimeMessage(session);
45.             message.setFrom(new InternetAddress(username));
46.             message.setRecipients(Message.RecipientType.TO,
47.                                     InternetAddress.parse(to));
48.             message.setSubject(subject);
49.             message.setText(content);
```

```

50.
51.         Transport.send(message);
52.
53.         System.out.println("Mail sent.");
54.     } catch (MessagingException e) {
55.         e.printStackTrace();
56.     }
57. }
58. }

```

Figure 4.6. MailService class code snippet

QueueServerConnection class serves as the connection between the client and the queue server. It contains two methods: (1) the first method (lines 17-36) allows the requests to be sent to the server and (2) the second method (lines 46-67) retrieves a message from the queue of the server. The constructor (lines 5-8) has two parameters: the host and the port.

```

1.  public class QueueServerConnection {
2.      private String host;
3.      private int port;
4.
5.      public QueueServerConnection(String host, int port) {
6.          this.host = host;
7.          this.port = port;
8.      }
9.
10.     /**
11.      * Sends a request to the server to insert a message into the queue.
12.      *
13.      * @param messageToSend the message to be inserted into the queue
14.      * @return the status of the operation (true == successful)
15.      * @throws IOException thrown if there is a problem with the connection
16.      */
17.     public boolean writeMessage(String messageToSend) throws IOException {
18.         Socket clientSocket = new Socket(host, port);
19.         ObjectOutputStream outToServer = new
20.             ObjectOutputStream(clientSocket.getOutputStream());
21.         ObjectInputStream inFromServer = new
22.             ObjectInputStream(clientSocket.getInputStream());
23.         outToServer.writeObject(new Message("SEND",messageToSend));
24.
25.         Message response;
26.         try {
27.             response = (Message)inFromServer.readObject();
28.         } catch (ClassNotFoundException e) {
29.             response = null;
30.             e.printStackTrace();
31.         }
32.
33.         outToServer.close();
34.         inFromServer.close();
35.         clientSocket.close();

```

```

34.
35.         return (response!=null && response.getType().equals("ACK"));
36.     }
37.
38.     /**
39.      * Retrieves a message from the queue of the server, by sending a "READ"
40.      * request to it.
41.      * If the queue is empty, this method will hang until the queue will get a
42.      * message. In
43.      * other words, this method will wait for the server to provide a message.
44.      * @return the message from the queue, sent by the server
45.      * @throws IOException thrown if there is a problem with the connection
46.      */
47.     public String readMessage() throws IOException {
48.         Socket clientSocket = new Socket(host, port);
49.         ObjectOutputStream outToServer = new
50.             ObjectOutputStream(clientSocket.getOutputStream());
51.         ObjectInputStream inFromServer = new
52.             ObjectInputStream(clientSocket.getInputStream());
53.         outToServer.writeObject(new Message("READ",null));
54.
55.         Message response;
56.         try {
57.             response = (Message)inFromServer.readObject();
58.         } catch (ClassNotFoundException e) {
59.             response = null;
60.             e.printStackTrace();
61.         }
62.
63.         outToServer.close();
64.         inFromServer.close();
65.         clientSocket.close();
66.
67.         if (response==null || !response.getType().equals("ACK")) return null;
68.         return response.getContent();
69.     }
70. }

```

Figure 4.7. QueueServerConnection class code snippet

4.3.3. Queue server application

Queue Server Application has three components: *ServerStart*, *Queue* and *Queue Communication*.

The *ServerStart* class contains a *main* method that creates a new *Server* object at the port that is specified as a parameter. By default, the port is 8888.

```
1. public class ServerStart {
2.
3.     private static final int PORT = 8888;
4.
5.     private ServerStart() {
6.     }
7.
8.     public static void main(String[] args) {
9.         try {
10.            new Server(PORT);
11.            System.out.println("Queue server started.");
12.        } catch (IOException e) {
13.            e.printStackTrace();
14.        }
15.    }
16. }
```

Figure 4.8. ServerStart class code snippet

The *Queue* class is a wrapper for a queue and the underlying queue is a *BlockingQueue*. If there are no elements in the queue, then this type of queue will block and will wait for elements to retrieve. The first method (lines 14-16) inserts elements in the queue and the second method (lines 18-20) retrieves elements from the queue. The underlying mechanism is FIFO and it works in a push and pop manner.

```
1. public class Queue {
2.     private static Queue queueInstance;
3.     private BlockingQueue<String> queue;
4.
5.     private Queue() {
6.         queue = new LinkedBlockingDeque<String>();
7.     }
8.
9.     public static Queue getInstance() {
10.        if (queueInstance==null) queueInstance=new Queue();
11.        return queueInstance;
12.    }
13.
14.    public void put(String message) throws InterruptedException {
15.        queue.put(message);
16.    }
17.
18.    public String get() throws InterruptedException {
19.        return queue.take();
20.    }
21. }
```

Figure 4.9. Queue class code snippet

The queue communication part contains three classes: *Message*, *Server* and *Session* and is responsible for the communication with the queue.

The *Message* class contains a constructor (lines 5-8) that takes as parameters the type and the content of the message and it also contains getters and setters. This class is used for communication between the components and it represents an exchange mechanism between the queue server and the clients.

```

1. public class Message implements Serializable {
2.     private String type;
3.     private String content;
4.
5.     public Message(String type, String content) {
6.         this.type = type;
7.         this.content = content;
8.     }
9.
10.    public String getType() {
11.        return type;
12.    }
13.
14.    public void setType(String type) {
15.        this.type = type;
16.    }
17.
18.    public String getContent() {
19.        return content;
20.    }
21.
22.    public void setContent(String content) {
23.        this.content = content;
24.    }
25. }

```

Figure 4.10. Message class code snippet

There are four types of messages which are illustrated in the table below and the message also has an associated content.

Table 4.1. Message types

Message Type	Description
SEND	Inserts content into the queue.
READ	Retrieves content from the queue.
ACK	The operation is successful on the server side.
ERR	The operation fails on the server side.

The *Server* class has the following properties: (1) it creates socket that accepts connections and (2) it creates a thread which deals with the communication with the client. The constructor (lines 9-12) creates a socket object to listen to and accept connections. The *run* method (lines 17-30) contains a while loop that runs continuously, accepts connections from the clients and it creates and starts a new thread for dealing with the client messages.

```
1. public class Server implements Runnable {
2.     private ServerSocket serverSocket;
3.
4.     /**
5.      * Create a socket object from the ServerSocket to listen to and accept
        connections
6.      * @param port the port on which the ServerSocket will be bound to
7.      * @throws IOException
8.      */
9.     public Server(int port) throws IOException {
10.         serverSocket = new ServerSocket(port);
11.         new Thread(this).start();
12.     }
13.
14.     /**
15.      * Accepts connections from clients and assigns a thread to deal with the
        messages from and to the respective client.
16.      */
17.     public void run() {
18.         while (true) {
19.             try {
20.                 synchronized (this) {
21.                     Socket clientSocket;
22.                     clientSocket = serverSocket.accept();
23.                     Session cThread = new Session(clientSocket);
24.                     cThread.start();
25.                 }
26.             } catch (IOException e) {
27.                 e.printStackTrace();
28.             }
29.         }
30.     }
31.
32. }
```

Figure 4.11. Sever class Code Snippet

Session class deals with client connection processes: (1) receiving messages, (2) decoding messages and (3) sending a response. The *run* method (lines 18-55) waits for a message from the client and treats the message according to its type. If the message has the type SEND then it is inserted in the queue and acknowledgement message with null body is sent to the client. Otherwise, if the message has the type READ then a message is retrieved from the queue and an acknowledgement message with the content of that message is sent to the client.

```

1. public class Session extends Thread {
2.
3.     private Socket clientSocket;
4.     private ObjectInputStream inFromClient;
5.     private ObjectOutputStream outToClient;
6.
7.     public Session(Socket cSocket) {
8.         this.clientSocket = cSocket;
9.         try {
10.             inFromClient = new ObjectInputStream(clientSocket.getInputStream());
11.             outToClient = new ObjectOutputStream(clientSocket.getOutputStream());
12.         } catch (IOException e) {
13.             e.printStackTrace();
14.         }
15.     }
16.
17.     @Override
18.     public void run() {
19.         Message messageReceived;
20.
21.         try {
22.             // Wait for message from client
23.             messageReceived = (Message) inFromClient.readObject();
24.
25.             // Treat messages according to the type of the message
26.             switch (messageReceived.getType()){
27.                 case "SEND":
28.                     try {
29.                         //insert the message into the queue
30.                         Queue.getInstance().put(messageReceived.getContent());
31.                         sendMessageToClient(new Message("ACK", null));
32.                     } catch (InterruptedException e) {
33.                         e.printStackTrace();
34.                         sendMessageToClient(new Message("ERR", null));
35.                     }
36.                     break;
37.                 case "READ":
38.                     try {
39.                         //retrieve a message from the queue
40.                         //since the underlying queue is a
41.                         //BlockingQueue, this method call will wait
42.                         //if the queue is empty
43.                         String content = Queue.getInstance().get();
44.                         sendMessageToClient(new Message("ACK", content));
45.                     } catch (InterruptedException e) {
46.                         e.printStackTrace();
47.                         sendMessageToClient(new Message("ERR", null));
48.                     }
49.                     break;
50.             }
51.         } catch (ClassNotFoundException | IOException e) {

```

```

51.         e.printStackTrace();
52.     }
53.
54.     closeAll();
55. }
56.
57. public void sendMessageToClient(Message messageToSend) {
58.     try {
59.         outToClient.writeObject(messageToSend);
60.     } catch (IOException e) {
61.         e.printStackTrace();
62.     }
63. }
64.
65. public void closeAll() {
66.     try {
67.         // Close the input stream
68.         if (inFromClient != null) {
69.             inFromClient.close();
70.         }
71.         // Close the output stream
72.         if (outToClient != null) {
73.             outToClient.close();
74.         }
75.         // Close the socket
76.         if (clientSocket != null) {
77.             clientSocket.close();
78.         }
79.     } catch (IOException e) {
80.         e.printStackTrace();
81.     } finally {
82.         inFromClient = null;
83.         outToClient = null;
84.         clientSocket = null;
85.     }
86. }
87. }

```

Figure 4.12. Session class code snippet

4.3.4. Application sequence diagram

There are two major components that are running asynchronously with regards to one another (the producer and the consumer). The queue server is the means of communication between them. Therefore, two sequence diagrams will be presented, one with the interaction of the producer client and the queue server (Figure 4.13), and one with the interaction of the consumer client and the queue server (Figure 4.14.).

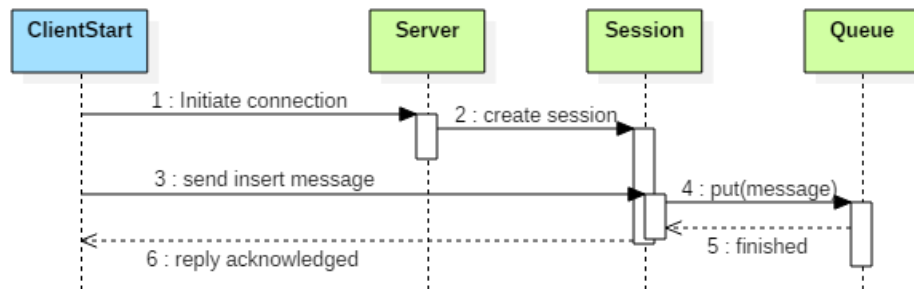


Figure 4.13. Sequence diagram for message insertion in queue (producer client)

The sequence diagram from Figure 4.13. comprises of the following steps:

1. The client initiates a connection with the queue server.
2. The server creates a session (thread) to handle the client.
3. Client sends the actual request and the message to be inserted into the queue.
4. Server inserts the message into the queue
5. Insertion is executed successfully
6. Return an acknowledged message, notifying that the operation was successful.

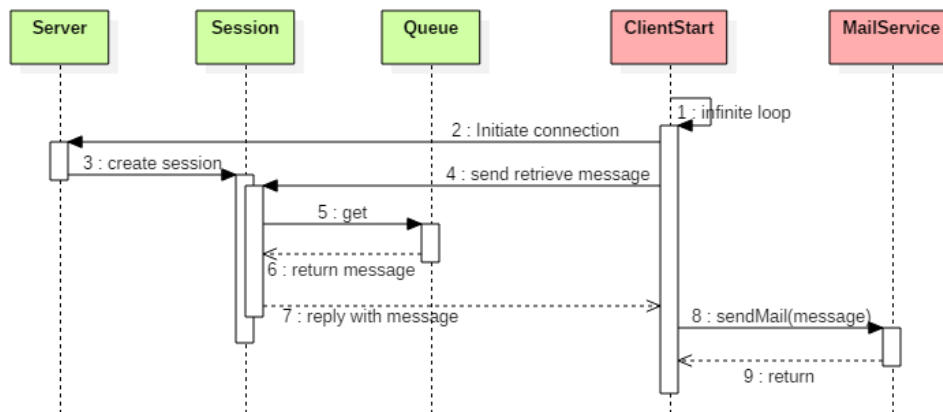


Figure 4.14. Sequence diagram for message retrieval from queue (consumer client)

The next sequence diagram refers to the flow of the consumer client from Figure 4.14.:

1. The client has an infinite loop which executes *readMessage* (steps 1-7) -> *processMessage* (steps 8-9).
2. The client initiates a connection with the queue server.
3. The server creates a session (thread) to handle the client.
4. Client sends the actual request (to retrieve a message from the queue).
5. Message is retrieved from the queue (popped, i.e. eliminated from the queue).

6. Message is returned.
7. Reply to the client with the respective message.
8. Call the *sendMail()* method, to send a mail with the message (content) got from the queue.
9. Return to the client main function.

4.4. Building and running the example

1. Setup GIT and download the project from
https://bitbucket.org/utcn_dsrl/ds.handson.assignment3.git
 - Create an empty local folder in the workspace on your computer
 - Right-click in the folder and select *Git Bash*
 - Write commands:
 - *git init*
 - *git remote add origin*
https://bitbucket.org/utcn_dsrl/ds.handson.assignment3.git
 - *git pull origin master*
2. Import the project into Eclipse: *FILE-> Import->Maven-> Existing Maven Projects->* Browse for project in the folder created at step 1
3. Run the project:
 - Run the *ServerStart* class from the queue server module, package start
 - Run the *ClientStart* class from the consumer client module, package start
 - Run the *ClientStart* class from the producer client module, package start
4. The application will perform the following: the consumer client, once it receives elements from the queue server, will process them (i.e. send mails with the messages). The producer client will send messages to the queue server to be inserted in the queue (i.e. to be processed).
5. By default, consumer client will only print to STDOUT the messages. To actually send mails, a valid Gmail username and password will need to be specified in *ClientStart* class from consumer client module. In this case access for less secure apps must be switched to on for the Gmail account: <https://www.google.com/settings/security/lesssecureapps>

4.5. Laboratory work: asynchronous distributed system application

4.5.1. Requirements

Design, implement and test a distributed system that uses MOM to create an asynchronous communication between the client (message producer) and the server (message consumer).

Functional requirements:

- The application is used by a DVD store administrator

- Each time new information about a DVD is added in the system by the administrator, the application must
 - send automatically notification e-mails to all the subscriber customers to notify them about the new item.
 - create automatically a text file and write the information about the DVD in it.

Implementation technologies:

- Use one of the following technologies:
 - For message producer and consumer use **Java** or **.NET**
 - For message queue:
 - **Java:** JMS or RabbitMQ Java API
 - **.NET:** MSMQ or RabbitMQ .NET API

4.5.2. Deliverables

- A solution description document (about 4 pages, Times New Roman, 10pt, Single Spacing) containing:
 - Conceptual architecture of the distributed system.
 - UML Deployment diagram.
 - DB design.
 - Readme file containing build and execution considerations.
- Source files. The source files will be uploaded on the personal *bitbucket* account, following the steps:
 - Create a repository on *bitbucket* with the exact name:
DS_Group_FirstName_LastName_Assignment_3
 - Push the source code and the documentation (push the code not an archive with the code or war files)
 - Share the repository with the user *utcn_dsrl*

4.5.3. Evaluation

Table 4.2. shows how grading is performed for this assignment.

Table 4.2. Asynchronous communication laboratory work grading details

Points	Requirements
5 p	<ul style="list-style-type: none">• Web page for filling information and creating new DVD• Message Sender, Message Queue and Message Receiver• Documentation
2 p	Sending email from Message Receiver

1 p	Creating a text file from Message Receiver
2p	<p>Correct answers to assignment related topics:</p> <ul style="list-style-type: none">• Types of communication: Point-to-Point vs Publish-Subscribe• MOM concepts: Message, Message Producer, Message Consumer, Queue, Topic, etc.

4.6. Bibliography

- [1] http://www.coned.utcluj.ro/~salomie/DS_Lic/
- [2] JMS, MSMQ: Lab Book: I. Salomie, T. Cioara, I. Anghel, T. Salomie, *Distributed Computing and Systems: A practical approach*, Albastra, Publish House, 2008, ISBN 978-973-650-234-7
- [3] RabbitMQ
- a. <https://www.rabbitmq.com/getstarted.html>
 - b. <https://dzone.com/articles/getting-started-rabbitmq-java>
 - c. <https://www.rabbitmq.com/install-windows.html>
- [4] Java EE tutorial, <https://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html>

5. XML based Communication and Web Services

5.1. Introduction

The web services are self-describing, platform-independent computational elements that execute a specific business task. They can be seen as the next evolutionary step in software development after distributed objects and aim to eliminate a major drawback of these, difficulties in cross platform integration. The main reason behind this problem relies in fact that the remote interface the distributed objects need to be exposed and described in a technology specific language in order to be invoked. The services address the cross-platform code heterogeneity problem by relying on a language that is interpreted in the same way in all existing technologies. Thus, on top of the distributed objects architecture, an XML layer is added as presented in Figure 5.1.

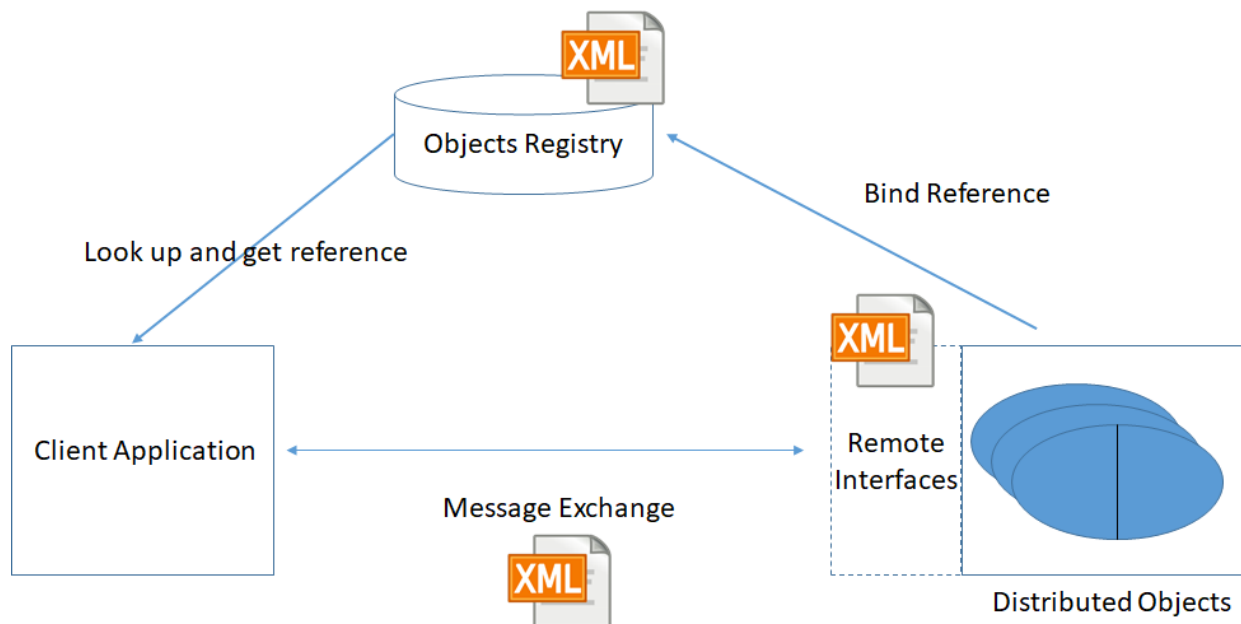


Figure 5.1. From distributed objects to services

The new emerging architecture is called SOA (Service Oriented Architecture) and it is based on three main communicating entities that are described and that collaborate using XML language (see Figure 5.2):

- **Service Consumer** whose main task is to search and discover public services using Service Brokers. The services that fulfill the requests of the consumer are used to develop new applications.

- **Service Broker** whose main task is to publish or expose the available services.
- **Service Provider** aiming to develop and publish loosely-coupled software services.

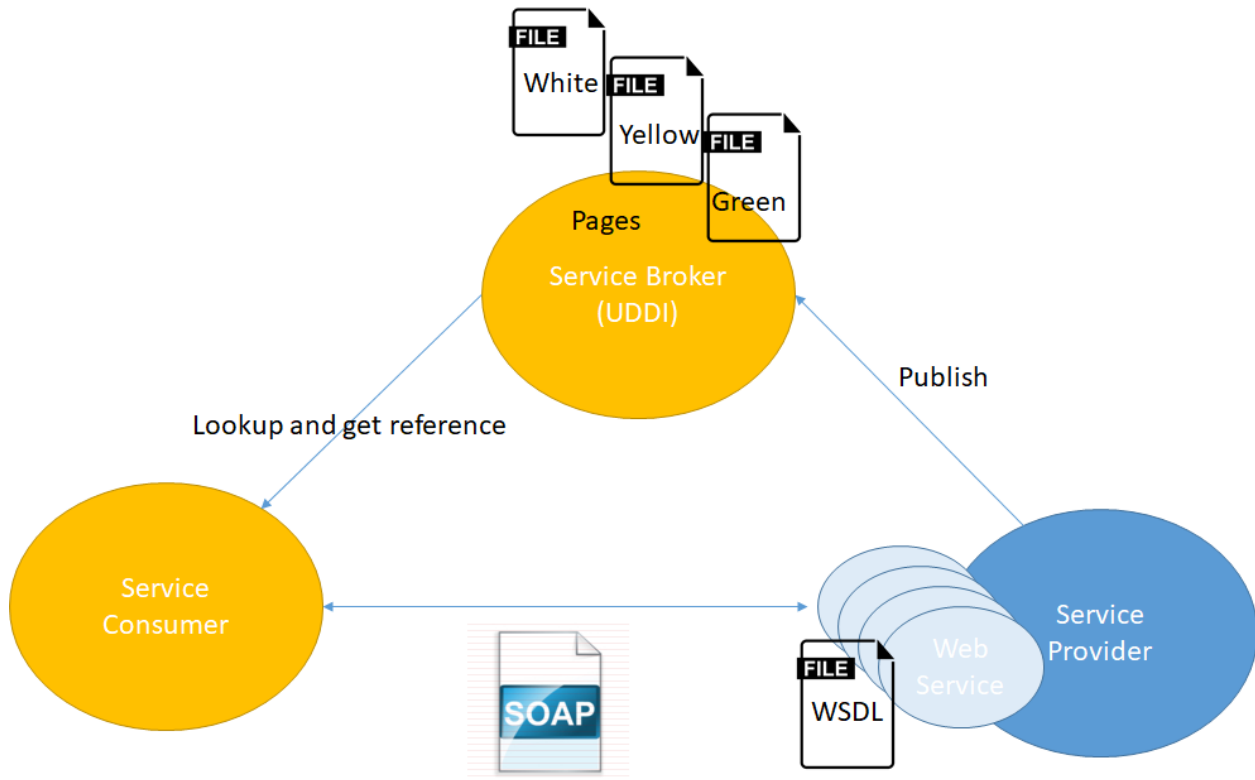


Figure 5.2. SOA architecture overview

The Service Provider creates a web service and publishes it into a Service Broker. The Service Broker is a registry that makes the published web service available to any requestor (client). The Service Consumer locates the web services in the broker registry and then binds to the Service Provider to properly invoke the web service. The interactions between these three components are made via operations like *publish*, *lookup* and *bind*.

The SOA web services are the solution to the technological heterogeneous application integration because of their openness, achieved via XML (see Figure 5.2.):

- The Web Service is described using a Web Service Description Language (WSDL) document, an XML standard format for describing the service interface;
- The Web Service is published and discovered using the Universal Discovery, Description and Integration (UDDI) registry which enables applications to find Web Services at design or run time;
- The Web Service invocation is achieved by using Simple Object Access Protocol (SOAP), an XML based standard for message exchange.

The adoption of XML language for description and communication bring many benefits to the web services such as (more detailed outlook on SOA principles is provided in Table 5.1):

- **Technology neutral** – the services must be invoked from different technologies using standardized invocation mechanisms
- **Loosely coupled** – the services shouldn't know anything about the internal structure and implementation of other services
- **Location transparent** - the information regarding the services' location or description must be stored in a registry. The clients will find and invoke the services without knowing in advance their location.

Table 5.1. The main features of SOA architecture⁷

Architectural Principle	Description
<i>Service Encapsulation</i>	The services are exposed only through their interfaces and each service implements a specific business activity that can be reused
<i>Service Loose Coupling</i>	There should be a reduced number of inter-dependencies between services and the clients / consumers invoking them.
<i>Service Contract</i>	The services must be described using XML description documents which provide an interface contract with the potential clients
<i>Service Abstraction</i>	The services should hide their complex logic. Usually behind a well-defined interface
<i>Service Reusability (Maximization of reuse)</i>	Complex business logic is divided into simple business activities that are mapped onto services with the intention of promoting reuse
<i>Service Composition</i>	Collections of simple services can be coordinated and orchestrated to form composite services
<i>Service Autonomy</i>	The services have control over the business logic they encapsulate
<i>Service Statelessness</i>	Service should be stateless and should not withhold information from one state to the other
<i>Service Discovery</i>	The services should be self-descriptive so that they can be found and accessed via available discovery mechanisms

⁷ Service Oriented Architecture Principles, <https://www.guru99.com/soa-principles.html>

5.2. WSDL

To be used and invokes the Web Services must expose information regarding their operational capabilities or functionality in form of a XML interface. More over to automatize as much as possible the software development the web services interface must also permit the reverse engineering of code generating the main end points of the service. To achieve this goal, WSDL describes the service syntactic information and its interface in XML (see Figure 5.3).

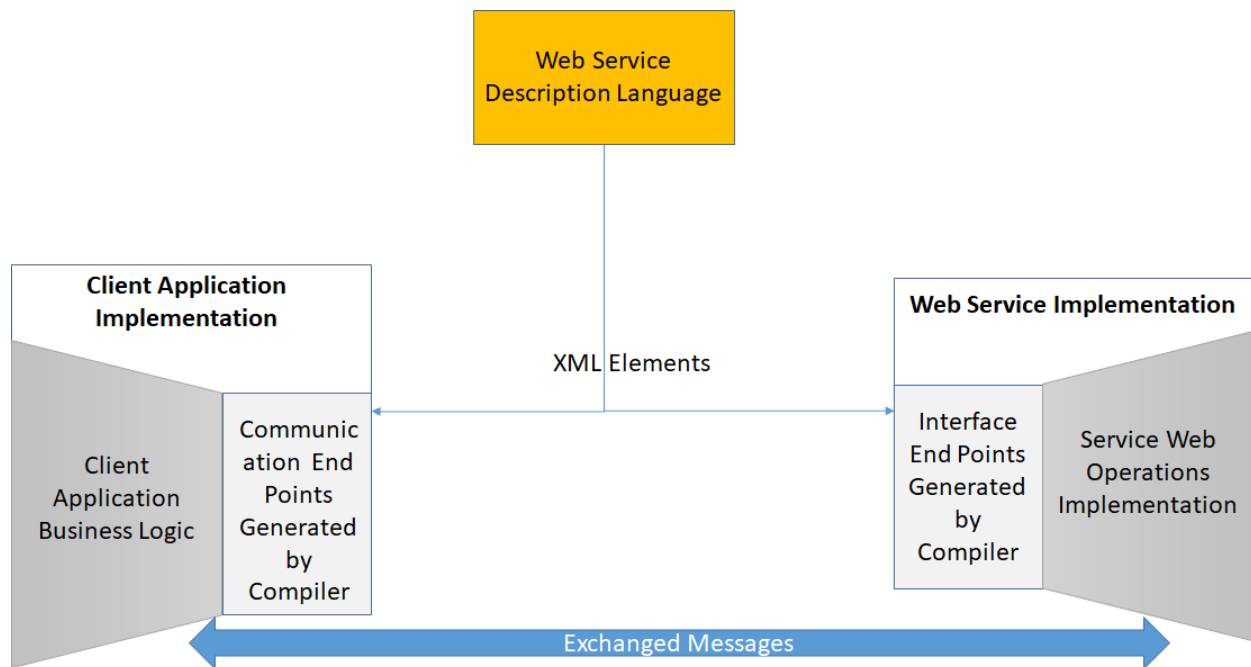


Figure 5.3. WSDL as a service Interface Description Language (IDL)

Following the WSDL specifications⁸, a WSDL document defines *services* as collections of network endpoints or *ports* (see Table 5.2). WSDL makes a separation between the abstract definition of endpoints and messages and their concrete network data format bindings. This separation makes possible to reuse abstract definitions like messages (abstract descriptions of the data being exchanged) or *port types* (abstract collections of *operations*). The concrete protocol and data format specification for a particular port type forms a reusable *binding*.

Table 5.2. WSDL XML elements

XML Element	Description
<types>	Describes the data type definitions used by the web service

⁸ <https://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/>

<code><message></code>	Data elements of each service defined operation. Describes the messages that can be sent or received by the service. The message is identified by its name and is usually defined based on an XML schema
<code><operation></code>	Web method describing the basic functionality of the service. For an operation, the input and the output messages must be defined
<code><portType></code>	Describes the set of operations that are offered by the service and the exchanged messages involved
<code><binding></code>	A concrete protocol and data format specification each particular port type

5.3. SOAP

SOAP is the XML protocol for accessing web service operations by providing a simple mechanism for information exchanging in a distributed environment based on XML. A SOAP message is an ordinary XML document containing the elements described in Table 5.3⁹.

Table 5.3. XML elements of a SOAP message

XML Element	Description
<i>soap:Envelope</i>	Identifies the XML document as a SOAP message. The SOAP Envelope element is the root element of a SOAP message and acts like a container for the information that must be delivered.
<i>xmlns:soap namespace attribute</i>	Should always have the value of: http://www.w3.org/2003/05/soap-envelope/
<i>soap:encodingStyle attribute</i>	Special SOAP construction which is used to define the data types used in the document. This construction may appear in any SOAP element presented above and applies to the same element's contents and to all its child elements. A SOAP message has no default encoding. Example: http://www.w3.org/2003/05/soap-encoding
<i>soap:Header</i>	Optional element. An extension mechanism that provides a way to pass general information to SOAP messages. If the Header element is present, it must be the first child of the Envelope element. Contains application-specific information (like authentication, payment, etc.) about the SOAP message.

⁹ Simple Object Access Protocol, <https://www.w3.org/TR/soap/>

<i>soap:Body</i>	The Body element (required) - contains the call or the response information. The SOAP Body element contains the SOAP message intended for the ultimate endpoint of the message.	
<i>soap:Fault</i>	Optional Element. Provides information about errors that occurred while processing the message. It is used to carry error and status information within a SOAP message. If present, the SOAP Fault element must appear only once within the body element.	
	<i>Faultcode – sub element</i>	A code for identifying the fault
	<i>Faultstring - sub element</i>	A human readable explanation of the fault
	<i>Faultactor - sub element</i>	Information about who caused the fault to happen
	<i>Detail - sub element</i>	application specific error information
SOAP bindings are mechanisms which allow SOAP messages to be effectively exchanged using a transport protocol such as HTTP.		

The main advantages of using SOAP over the classical RPC / XML approach are mentioned below:

- provides an easier way to communicate behind proxies and firewalls than RPC technology
- allows the use of different transport protocols (HTTP – as default, TCP, SMTP etc.).

5.3. UDDI

UDDI is a service registry used to publish and locate web services, leveraging on XML-based service descriptors. Each service descriptor contains the information needed by the service requester to find and then bind to a particular web service.

Main functionalities provided:

- **Service Publication.** UDDI defines operations that allow organizations to expose their web services.
- **Service Finding.** UDDI defines operations that allow consumers to extract information about services published in the UDDI registry.
- **Service Classification.** UDDI defines operations that permit the classification of businesses and services according to standard taxonomies.

In UDDI a service is described at different levels of abstractions using three types XML elements called pages (see Table 5.4).

Table 5.4. UDDI pages

UDDI Page	Description
<i>White Pages</i>	Contains the basic contact information for each company providing web services.
<i>Yellow Pages</i>	Contains more details about the company, and includes descriptions of the services the company can offer to potential consumers
<i>Green Pages</i>	Contains the Web Service binding information. It includes various interfaces, URL locations, discovery information and similar data required to find and invoke the Web Service

The descriptive sections of UDDI are called *Listings* and consist of the web service interface descriptions that are created from WSDL and stored into a UDDI registry. UDDI will ultimately allow registries to exchange listings with each other, so that it is possible to have the same listing replicated to many UDDI registries.

The necessary steps for using the UDDI registry in the context of SOA architecture are:

- A web service Provider publishes information about the web service (taken from the WSDL file) in the UDDI registry.
- A web service Consumer searches the UDDI to find a service that matches its business requirements.
- The UDDI sends the description of the matching service to the web service Consumer.
- The Consumer connects to the web service Provider using the SOAP protocol and the service web methods are invoked.

5.4. Laboratory work: SOA web services

5.4.1. Requirements

Design, implement and test a distributed application called “Online Tracking System” comprising of a GUI (web or desktop) and of several web services that implement the actual business logic that must be offered to the users.

Functional requirements:

- The application has two types of users: administrators and simple users (clients)
- If the user does not have an account, it can register and become a simple user
- After the login, the user is redirected to his/her corresponding page.
- The simple user can:

- List all his/her packages
 - Search specific packages
 - Check the status of a package delivery
- The administrator can:
 - Add/remove package. The package has the following characteristics: Sender (a simple user), Receiver (a simple user), Name, Description, Sender City, Destination City, Tracking (Boolean – initially false)
 - Register package for tracking. The package becomes tracked, and a route is associated to it. This route represents the path of the package to the destination, as pairs of (City, Time).
 - Package status updating. A new entry (City, Time) is introduced to the route

These functionalities will be exposed as 2 SOAP web services, each implementing the following operations:

- WS1:
 - Login and register
 - Simple client operations
- WS2
 - Administrator Operations

Implementation technologies:

- Develop one SOAP web service in .NET and the other one in JAVA
- Choose between .NET or JAVA for implementing the GUI (either web or desktop)

5.4.2. Deliverables

- A solution description document (about 4 pages, Times New Roman, 10pt, single spacing) containing:
 - Conceptual architecture of the distributed system.
 - UML Deployment diagram.
 - DB design
 - Readme file containing build and execution considerations.
- Source files. The source files will be uploaded on the personal *bitbucket* account, following the steps:
 - Create a repository on *bitbucket* with the exact name:
DS_Group_FirstName_LastName_Assignment_4
 - Push the source code and the documentation (push the code not an archive with the code or war files)
 - Share the repository with the user *utcn_dsrl*

5.4.3. Evaluation

Table 5.5. shows how grading is performed for this assignment.

Table 5.5. SOA web services laboratory work grading details

Points	Requirements
5 p	<ul style="list-style-type: none">• Simple GUI (web or desktop)• One SOAP web service (WS1 or WS2)• DB• Documentation
2 p	Save and display routes for each package in a DB table containing the pairs (City, Time)
2 p	Both web services implemented for complete functionality (WS1 and WS2)
1 p	Correct answers to assignment related topics: <ul style="list-style-type: none">• SOA architecture and its components: WSDL, UDDI, SOAP• SOAP protocol• WSDL components• UDDI components• How platform independence is assured for Web Services

5.5. Bibliography

- [1] http://www.coned.utcluj.ro/~salomie/DS_Lic/
- [2] Lab Book: I. Salomie, T. Cioara, I. Anghel, T.Salomie, *Distributed Computing and Systems: A practical approach*, Albastra, Publish House, 2008, ISBN 978-973-650-234-7
- [3] Java SOAP Web Services: <https://docs.oracle.com/javaee/6/tutorial/doc/bnayl.html>
- [4] .NET SOAP Web Services: <https://msdn.microsoft.com/en-us/library/t745kdsh%28v=vs.90%29.aspx>

6. Server-side Frameworks: Spring for Developing REST Web Services

6.1. Introduction

To be able to create an application that contains a set of micro-services, the user needs to use an appropriate architectural style. REST, or REpresentational State Transfer, is an architectural style used for providing standards between computer systems on the web, so that the communication between systems becomes easier, but also to define some constraints and properties such as performance and scalability. In this architectural style, data and functionalities are considered resources which can be accessed by URI. These resources can be accessed by a set of simple and well-defined operations.

As mentioned above, the REST architecture imposes some constraints for an application out from which the most important are:

- *Client-Server architecture* – the principle behind this constraint is the total separation of concerns. The user interface does not need to know information about data retrieve or any other information that does not interest it directly, while the server does not have any interest in knowing aspects of the interface. If this constraint is violated than the application cannot be called a REST application. This separation allows the components to evolve independently and become more portable.
- *Statelessness* – describe the fact that no information is restrained by either sender or receiver, in our case client and server. Basically, they are agnostic of the state in which the other is.
- *Cacheability* – this constraint imposes that the response of a request can be cached so that the client will not receive inappropriate data in response. This improves scalability and performance.

REST applications are considered fast and easy to use because of the following principles:

- The resources are identified by URI – a RESTful web service exposes a set of resources which identifies the goals of their interaction with their clients. After URI identifies the resources, it offers a global space for addressing and discovery of services and resources.
- Interface uniformity – the resources are manipulated by using exactly four types of operations: create, read, update and delete.
- Descriptive Messages – the resources are decoupled from their actual form so that their content can be accessed in multiple formats such as HTML, JSON, XML and others.

6.2. Hands-on application

The Spring Framework is an application framework for the Java platform. The framework was first released on the 1st October 2002 and was written by the Australian computer specialist Rod B. Johnson. Due to its continuous enhancement and development the framework became widely used in software companies nowadays. The hands-on example from this sub-section is a skeleton for a Spring application that can be used to get the information from a database using RESTfull services.

6.2.1. Application installation and configuration

To download and configure the example follow the steps described below:

1. Setup GIT and download the project from https://bitbucket.org/utcn_dsrl/spring-demo.git
 - Create an empty local folder in the workspace on your computer
 - Right-click in the folder and select *Git Bash*
 - Write the commands:
 - `git init`
 - `git remote add origin https://bitbucket.org/utcn_dsrl/spring-demo.git`
 - `git pull origin master`
2. Create an empty schema in MySQL with the name *spring-demo*
3. Import the project in Eclipse
4. Check the *application.properties* file from *src/main/resources* and fill the username and password of the local MySQL server.
5. Run *Maven Clean Install* for Eclipse as illustrated in Figures 6.1.
6. Go to MySQL Workbench and insert manually a user in the provided example DB
7. Run the application in Eclipse (see Figure 6.2.)

As an alternative you can run the application in the Apache Tomcat server installed locally. Copy the generated *spring-demo.war* from the folder *target* at the location *C:\apache-tomcat-9.0.1\webapps* and start the Apache Tomcat server using the instruction *C:\apache-tomcat-9.0.1\bin\startup.bat*.

To test the implemented REST endpoints, access the following links:

- <http://localhost:8081/spring-demo/user/all> => retrieve all the users from the database
- <http://localhost:8081/spring-demo/user/details/1> => retrieve the details of a user with the given ID

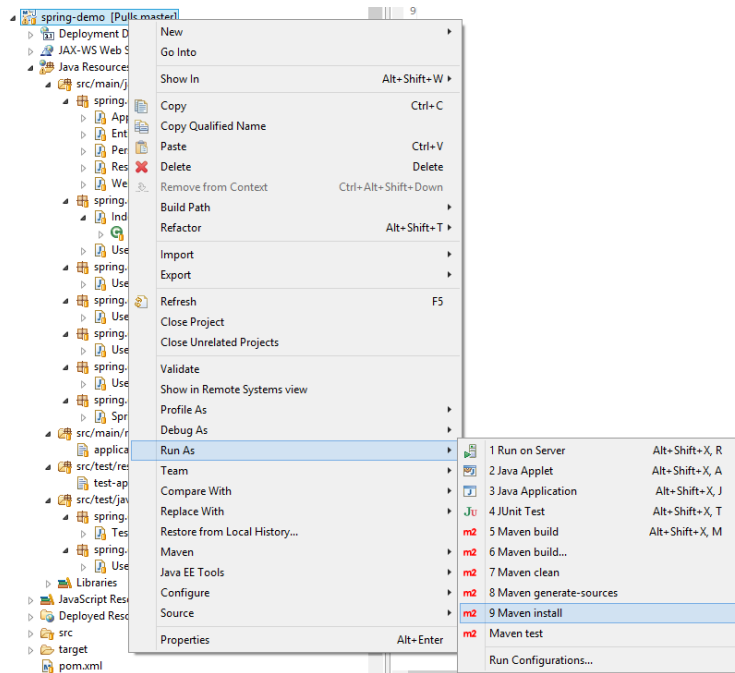


Figure 6.1. Maven install in Eclipse

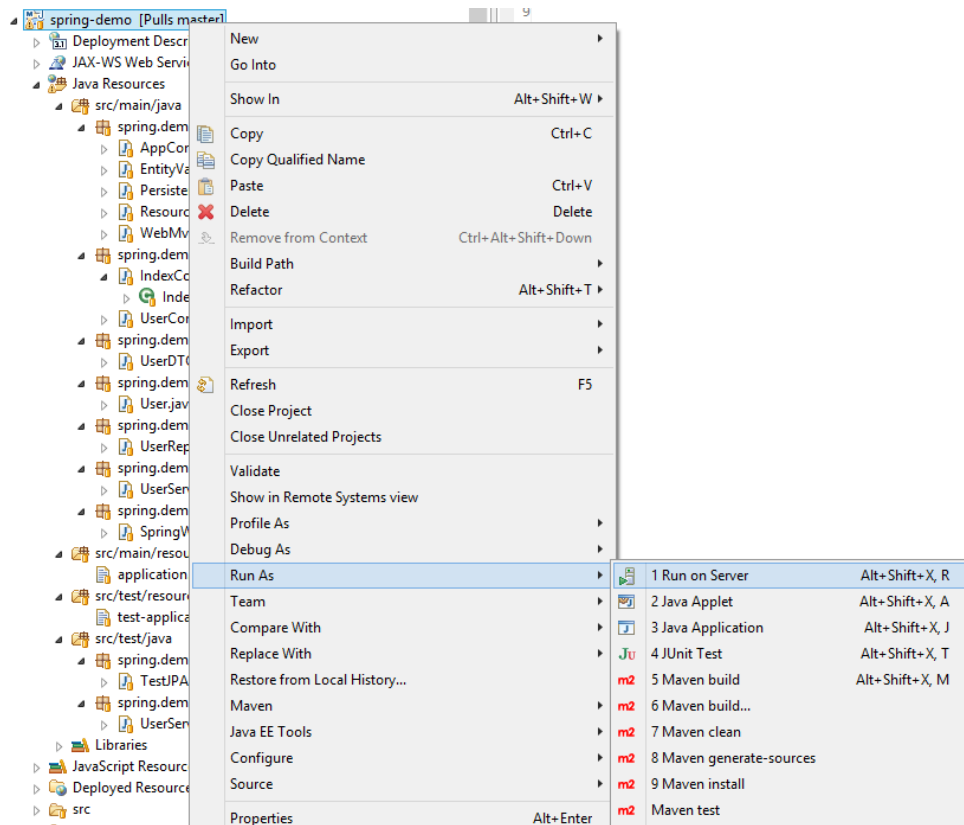


Figure 6.2. Application execution in Eclipse

The application is based on a single table database and aims at presenting the layers involved in performing CRUD operations on the user table shown in Figure 6.3. When opened in Eclipse, the project has the structure shown in Figure 6.4. All the components are detailed in next sub-section.

Column	Type
id	INT(11)
address	VARCHAR(200)
city	VARCHAR(100)
country	CHAR(2)
email	VARCHAR(200)
name	VARCHAR(100)
postcode	VARCHAR(50)
telephone	VARCHAR(50)

Figure 6.3. User table from the example project

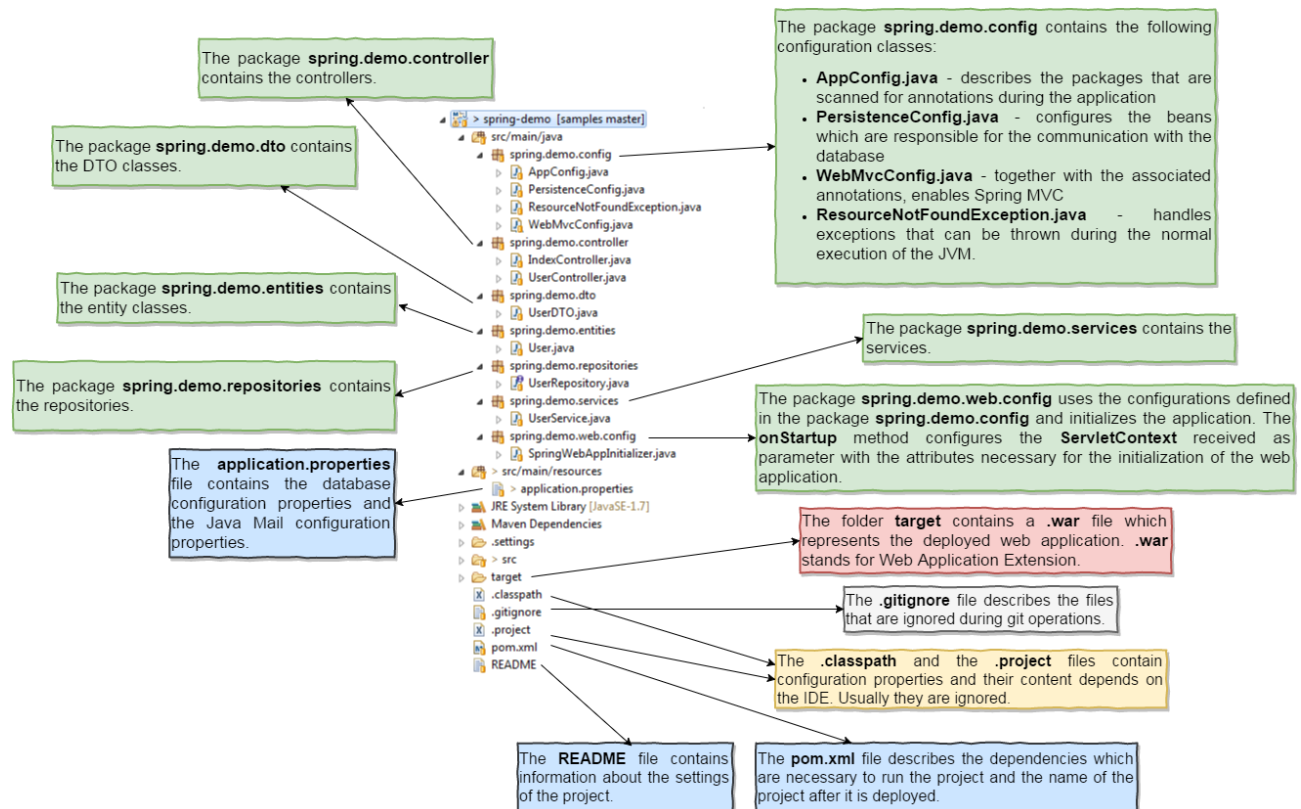


Figure 6.4. Project structure in Eclipse

6.2.2. Application conceptual architecture

The high level conceptual architecture of the system is presented in Figure 6.5.

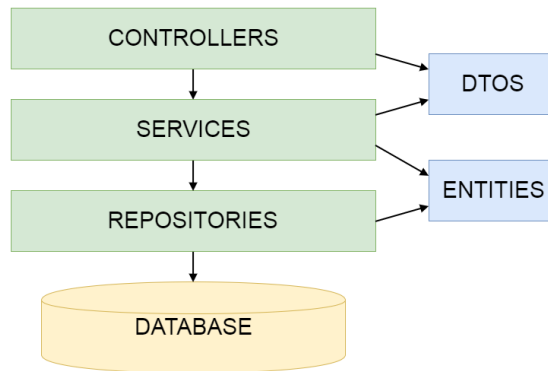


Figure 6.5. Project conceptual architecture

Table 6.1 describes each architectural component.

Table 6.1. Project Component Description

Component	Package	Description
<i>Repositories</i>	<i>Spring.demo.repositories</i>	Package that contains the repositories, classes that facilitate the DB access
<i>Entities</i>	<i>Spring.demo.entities</i>	An entity represents a table from the relational database and each instance of the entity corresponds to a row from the database
<i>Services</i>	<i>Spring.demo.services</i>	This layer represents the business logic layer of the Spring application. It translates the DTOs into entities and back, also performing more complex operations.
<i>DTOs</i>	<i>Spring.demo.dto</i>	A DTO is a special object exposed outside the application (to the UI or APIs). It contains only part of the underlying Entities.
<i>Controller</i>	<i>Spring.demo.controller</i>	The layer that exposes the application functionality as an API able to handle HTTP REST requests.

6.2.3. Application implementation details

A simple sequence diagram involving the interactions between the components for a GET operation is shown in Figure 6.6.

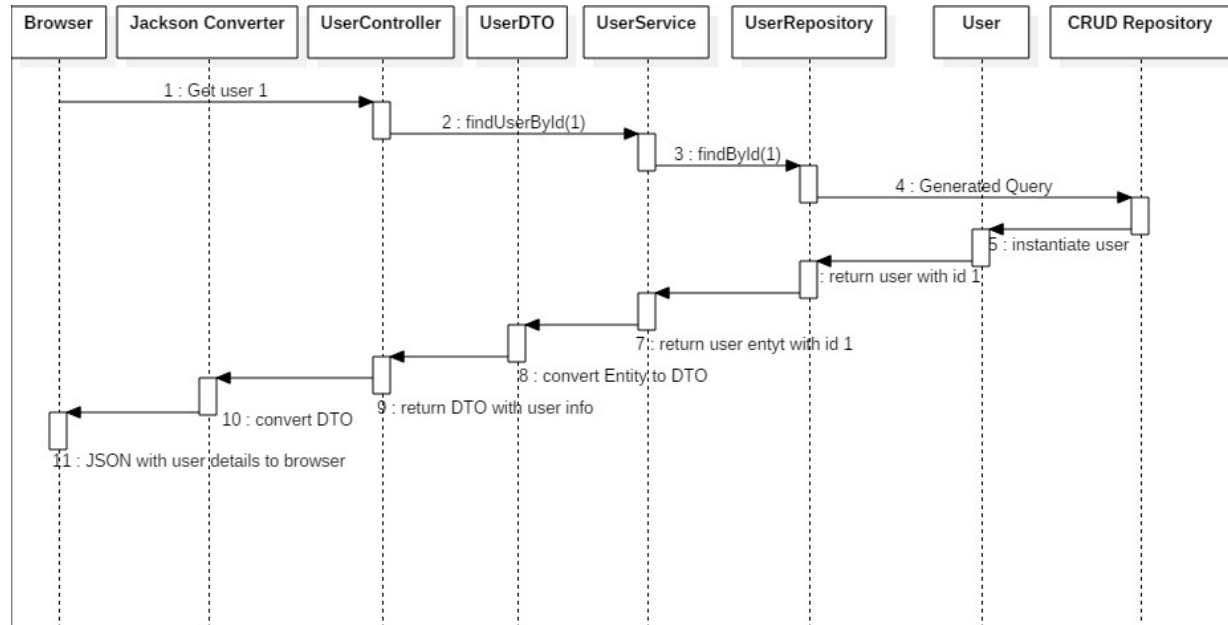


Figure 6.6. Sequence diagram for GET operation

The processing steps are described below.

1. Browser sends a HTTP request with the method GET to retrieve the user with id =1. This happens by calling the URL: <http://localhost:8081/spring-demo/user/details/1>. This URL is composed of the following parts:
 - **http**: protocol used to communicate
 - **localhost**: address of the server to communicate with. This can be either an URL resolved by DNS to an IP address, or an IP address. (localhost or 127.0.0.1 in this case).
 - **808**: the port on which the web server which will respond to the request is listening.
 - **spring-demo**: the name of the application deployed in the web server
 - **user/details/1**: The last part of the address is mapped to the resources within the application by the web server. In this case, the application exposes a REST API through its **controllers**. The mapping is done in three steps, as shown in the Figure 6.7.:
 - i. mapping to the controller at line 17,
 - ii. mapping to the method within the controller and defining the request type at line 23 and
 - iii. defining the parameters of the method at lines 23 and 24 (the name in the request must correspond to the name within the *@PathVariable* tag. Inside the method, the Java Parameter is used – *int id*).

```

15 @CrossOrigin(maxAge = 3600)
16 @RestController
17 @RequestMapping("/user")
18 public class UserController {
19
20     @Autowired
21     private UserService userService;
22
23     @RequestMapping(value = "/details/{id}", method = RequestMethod.GET)
24     public UserDTO getUserById(@PathVariable("id") int id) {
25         return userService.findUserById(id);
26     }
27

```

Diagram annotations:

- i) points to `@RequestMapping("/user")`
- ii) points to `@RequestMapping(value = "/details/{id}", method = RequestMethod.GET)`
- iii) points to `@PathVariable("id") int id`

Figure 6.7. Spring controller mapping

- The Spring controller automatically instantiates a service due to the annotation `@Autowired` (line 20). Using this `UserService` object, inside the `getUserById` method it calls the `findUserById` method, delegating the processing to the service layer.

Good to know

The **Controllers Layer** is a layer over the **Services Layer** and calls the methods which are provided by the **Services Layer**.

a) How are the controllers defined?

The controllers are defined using the annotation `@RestController`. This annotation specifies the fact that the corresponding annotated class can handle RESTful WEB Services.

b) How are the controllers mapped to the URLs?

The controllers are mapped to the URLs using the `@RequestMapping` annotation. This annotation is used in two cases:

- For annotating the entire class – in this case the value of the `@RequestMapping` (line 17) is a **prefix** for all the other URLs that are handled by the controller. In this example the controller is accessed using the following URL: `localhost:8080/spring-demo/user/all`

```

1 package spring.demo.controller;
2
3 import java.util.List;
4
15 @CrossOrigin(maxAge = 3600)
16 @RestController
17 @RequestMapping("/user")
18 public class UserController {
19
20     @Autowired
21     private UserService userService;
22

```

Figure 6.8. RequestMapping for UserController

- For annotating a specific method – in this case the value of the `@RequestMapping` is a **suffix** for the URL that corresponds to the method. The parameter **value** describes the

location while the parameter **method** describes the type of the request method. There are several types for the **RequestMethod**: GET, POST, PUT or DELETE.

```
28 @RequestMapping(value = "/all", method = RequestMethod.GET)
29 public List<UserDTO> getAllUsers() {
30     return userService.findAll();
31 }
```

Figure 6.9. RequestMapping for getting all the users

c) **How are the Services Layer instances accessed?**

The objects from the **Services Layer** are accessed using the *@Autowired* annotation.

d) **What are the most common input parameters of the methods annotated with @RequestMapping?**

- A *path variable* – in this case the variable is a part of the path specified by the *@RequestMapping* annotation

```
13 @RequestMapping("/remove/{id}")
14 public String removeUser(@PathVariable("id") int id) {
15     ...
16 }
```

Figure 6.10. PathVariable annotation example

- A *model* – an interface which defines a holder for the model attributes; the model object comes from the body of the method request

```
13 @RequestMapping("/edit/{id}")
14 public String editUser(@PathVariable("id") int id, Model model) {
15     ...
16 }
```

Figure 6.11. Model object annotation example

- A *model attribute* – the primary objective of this annotation is to bind the request parameters to a model object from the body of the method request

```
13 @RequestMapping(value = "/user/add", method = RequestMethod.POST)
14 public String addUser(@ModelAttribute("user") User u) {
15     ...
16 }
```

Figure 6.12. ModelAttribute annotation example

3. The *UserService* object is called with the method *findUserById*. It uses the *userRepository* object that was injected due to the *@Autowired* annotation to find the user in the DB.

Good to know

The **Services Layer** is an intermediary layer between the **Repositories Layer** and the **Controllers Layer**. The purpose of the **Services Layer** is to define methods that perform several operations on a database in such a way that either all the operations execute successfully or none of them is executed. In the second case the database rollbacks to the original state.

a) What are the services?

The services provide transactional operations for the business logic. A service method is the smallest atomic operation that the application can perform on a database. A service method either completes or the database rollbacks to the previous state.

b) How are the services defined?

The services are defined using the annotation `@Component` or the annotation `@Service`.

c) How are the objects defined in the Repositories Layer accessed?

The objects from the **Repositories Layer** are accessed using the annotation `@Autowired` (line 24). The purpose of this annotation is to auto wire beans.

```

1 package spring.demo.services;
2
3 import java.util.ArrayList;
17
18 @Service
19 public class UserService {
20     private static final String SPLIT_CH = " ";
21     public static final Pattern VALID_EMAIL_ADDRESS_REGEX = Pattern.compile("^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,6}$"
22         Pattern.CASE_INSENSITIVE);
23
24     @Autowired
25     private UserRepository userRepository;

```

Figure 6.13. UserService example

d) Why the Services Layer uses DTOs instead of entities?

Usually the DTOs reduce the overhead between the backend and the presentation. The optimized DTOs contain only that information which is absolutely required.

- The `UserRepository` method `findById` is called. This method uses an auto generated query by the `JpaRepository` superclass to retrieve a user by its ID.

```

7 public interface UserRepository extends JpaRepository<User, Integer> {
8
9     User findByName(String name);
10
11     User findById(int id);
12
13 }

```

Figure 6.14. UserRepository definition

Good to know

The **Repositories Layer** intermediates the communication between the **Services Layer** and the **Database**.

a) How are the repositories defined?

The repositories are defined by extending the interface *JpaRepository<T, ID extends Serializable>*. The first argument *T* describes the type of the entities used by the repositories while the second argument *ID* describes the type of the id of the entities.

```

1 package spring.demo.repositories;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5
6
7 public interface UserRepository extends JpaRepository<User, Integer> {
8
9     User findById(int id);
10
11 }

```

Figure 6.15. UserRepository interface

The *UserRepository* handles entities of the type *User* which have the id of the type *Integer*.

b) How are the repositories set in Spring?

The beans that are involved in the settings of the repositories are defined in the *PersistenceConfig* class. The class is fully depicted in the Figure 6.16. and its contents are detailed below:

- *DataSource* bean (lines 71 – 79) – is used for the configuration of the access to the relational database. It configures the following parameters:
 - *driver class name* (line 74) – the name of the class of the driver used for the communication with the database
 - *URL* (line 75) – the address of the database
 - *username* (line 76) – the username required to access the database
 - *password* (line 77) – the password required to access the database
- *DataSourceInitializer* bean (lines 81 – 90) – uses the *DataSource* bean. It can use a script to initialize the database
- *LocalContainerEntityManagerFactoryBean* bean (lines 45 – 64) – is the most powerful setup option for the JPA (Java Persistence API). It produces a container-managed *EntityManagerFactory*. It is a threadsafe object intended to be used by all the threads of the application and it is created only once on the startup of the application.


```

1 package spring.demo.config;
2
3 import java.util.Properties;
4
5 @Configuration
6 @PropertySource(value = { "classpath:/src/main/resources/application.properties" })
7 @EnableTransactionManagement
8 @EnableJpaRepositories(basePackages = "spring.demo.repositories")
9 public class PersistenceConfig {
10
11     @Autowired
12     private Environment env;
13
14     @Value("${init-db:false}")
15     private String initDatabase;
16
17     @Bean
18     public PlatformTransactionManager transactionManager() {
19         EntityManagerFactory factory = entityManagerFactory().getObject();
20         return new JpaTransactionManager(factory);
21     }
22
23     @Bean
24     public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
25         LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
26
27         HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
28         vendorAdapter.setGenerateDdl(Boolean.TRUE);
29         vendorAdapter.setShowSql(Boolean.TRUE);
30
31         factory.setDataSource(dataSource());
32         factory.setJpaVendorAdapter(vendorAdapter);
33         factory.setPackagesToScan("spring.demo.entities");
34
35         Properties jpaProperties = new Properties();
36         jpaProperties.put("hibernate.hbm2ddl.auto", env.getProperty("hibernate.hbm2ddl.auto"));
37         factory.setJpaProperties(jpaProperties);
38
39         factory.afterPropertiesSet();
40         factory.setLoadTimeWeaver(new InstrumentationLoadTimeWeaver());
41         return factory;
42     }
43
44     @Bean
45     public HibernateExceptionHandler hibernateExceptionHandler() {
46         return new HibernateExceptionHandler();
47     }
48
49     @Bean
50     public DataSource dataSource() {
51         BasicDataSource dataSource = new BasicDataSource();
52         dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
53         dataSource.setUrl(env.getProperty("jdbc.url"));
54         dataSource.setUsername(env.getProperty("jdbc.username"));
55         dataSource.setPassword(env.getProperty("jdbc.password"));
56         return dataSource;
57     }
58
59     @Bean
60     public DataSourceInitializer dataSourceInitializer(DataSource dataSource) {
61         DataSourceInitializer dataSourceInitializer = new DataSourceInitializer();
62         dataSourceInitializer.setDataSource(dataSource);
63         ResourceDatabasePopulator databasePopulator = new ResourceDatabasePopulator();
64         databasePopulator.addScript(new ClassPathResource("db.sql"));
65         dataSourceInitializer.setDatabasePopulator(databasePopulator);
66         dataSourceInitializer.setEnabled(Boolean.parseBoolean(initDatabase));
67         return dataSourceInitializer;
68     }
69 }

```

Figure 6.16. PersistenceConfig class

EntityManagerFactory has the following parameters:

- *dataSource* (line 53) – the *DataSource* bean which is used for the communication with the database

- *vendorAdapter* (line 54) – sets the Hibernate implementation for the *EntityManager*. The *EntityManager* is an interface used for the communication with the persistence context
- *packagesToScan* (line 55) – specifies the packages in which the entities are defined
- *jpaProperties* (line 59) – specifies additional properties such as the auto-generation of the database
- *PlatformTransactionManager* bean (lines 39 – 43) – uses the singleton version of the *LocalContainerEntityManagerFactoryBean* to create a *JpaTransactionManager*. The *JpaTransactionManager* is appropriate for the applications that use a single JPA *EntityManagerFactory* for the transactional data access
- *HibernateExceptionTranslator* (lines 66 – 69) – translates the *HibernateExceptions* to *DataAccessExceptions*

c) How to access the database using the Spring repositories?

There are different ways to access the database:

- Use one of the methods declared by the *JpaRepository*. The CRUD operations are implemented by default by the *JpaRepository* and it is not necessary to declare them again in the interface that extends it

```

35  /*
36   * (non-Javadoc)
37   * @see org.springframework.data.repository.CrudRepository#findAll()
38   */
39   List<T> findAll();

```

Figure 6.17. JpaRepository snippet

- Create methods based on the fields from the entity (e.g. *findById*, *findByName*, etc.). In this case the name of the method is parsed and interpreted by the Spring framework in order to execute the corresponding query. Also, there is the possibility to create queries which are more complex with filters, join and so on (for more details see the *JpaRepository* documentation)

```

1  package spring.demo.repositories;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4
5
6  public interface UserRepository extends JpaRepository<User, Integer> {
7
8      User findByName(String name);
9
10     User findById(int id);
11
12 }
13

```

Figure 6.18. UserRepository interface

- Use custom defined queries – in the case of the custom defined queries the name of the method is not parsed; the purpose of the *@Param* annotation is to specify the names of the parameters which are used in the definition of the query

```

19 @Query("SELECT c "
20        "FROM Company c "
21        "INNER JOIN c.opportunitieses op "
22        "WHERE op.id = :oppId")
23 Company findCompanyByOpportunityId(@Param("oppId") int id);

```

Figure 6.19. Custom defined queries

5. The *UserRepository* retrieves a user entity object instantiated with values from the DB.

Good to know

a) What are the entities?

An entity represents a table from the relational database and each instance of the entity corresponds to a row from the database. An example of entity is shown in Figure 6.20.

b) What are the main requirements for the creation of the entities?

- The entity class must be annotated with the annotation *@Entity*
- The table, the id and the columns are mapped using the annotations *@Table*, *@Id* and *@Column*
- The class **must** have one public/protected no-argument constructor
- In many cases the class must implement the *Serializable* interface

c) Which are the most common annotations used by the entities?

The most common annotations which are used in the mapping process are described below:

- *@Entity* – specifies the fact that the class which is annotated with this annotation is an entity
- *@Table* – specifies the table to which the entity is mapped
- *@Id* – the annotated field is an ID of the table
- *@Column* – the annotated fields are columns of the table from the database
- *@OneToOne* – maps the one-to-one relationship between two tables
- *@OneToMany* – maps the one-to-many relationship between two tables
- *@ManyToOne* – maps the many-to-one relationship between two tables
- *@ManyToMany* – maps the many-to-many relationship between two tables

```

1 package spring.demo.entities;
2
3 import static javax.persistence.GenerationType.IDENTITY;
10
11 @Entity
12 @Table(name = "user")
13 public class User implements java.io.Serializable {
14
15     private static final long serialVersionUID = 1L;
16     private Integer id;
17     private String name;
18     private String email;
19     private String address;
20     private String postcode;
21     private String city;
22     private String country;
23     private String telephone;
24
25     public User() {
26     }
27
28     public User(Integer id, String name, String email, String address, String postcode, String city, String country,
29         String telephone) {
30         super();
31         this.id = id;
32         this.name = name;
33         this.email = email;
34         this.address = address;
35         this.postcode = postcode;
36         this.city = city;
37         this.country = country;
38         this.telephone = telephone;
39     }
40
41     @Id
42     @GeneratedValue(strategy = IDENTITY)
43     @Column(name = "id", unique = true, nullable = false)
44     public Integer getId() {
45         return this.id;
46     }
47
48     public void setId(Integer id) {
49         this.id = id;
50     }
51

```

Figure 6.20. User entity

6. A user entity is returned to the *UserRepository*.
7. A user entity is returned to the *UserService*.
8. The *UserService* converts the entity object to a DTO. The BUILDER Design Pattern is used in this case to ease the adaptation between the two classes.

Good to know

a) What are the DTOs?

The DTOs (Data Transfer Objects) are objects that carry data between processes and are exposed by the application to the UI or through an API.

b) What is the relation between the DTOs and the entities?

If the database changes then the mappings used by the entities must also change, but the objects (DTOs) might remain unchanged.

c) Why are the DTOs used?

The motivation for using the DTOs is represented by the fact that they reduce the cost of communication between the processes. The DTOs aggregate in one call data that might be transferred by several calls.

d) How are the entities mapped to DTOs?

One possibility to map the entities to the DTOs is to use a static class *Builder* which contains the fields that are mapped from the entity to the DTOs. First a new instance of the *Builder* is created, then the fields which are mapped are set and finally the *Builder* object calls the *create* method which returns a DTO.

```

33     UserDTO dto = new UserDTO.Builder()
34         .firstname(names[0])
35         .surname(names[1])
36         .city(usr.getCity())
37         .address(usr.getAddress())
38         .email(usr.getEmail())
39         .telephone(usr.getTelephone())
40         .country(usr.getCountry())
41         .postcode(usr.getPostcode())
42         .create();

```

Figure 6.21. Creating a DTO

9. A *UserDTO* is returned to the controller.
10. Transparent to the programmer, the Spring framework calls the Jackson Converter to convert the retrieved user DTO to a JSON object that will be sent to the browser.

```

1.  {
2.    "id": 1,
3.    "firstname": "UTCN",
4.    "surname": "DSRL",
5.    "email": "dsrl@cs.utcluj.ro",
6.    "address": "Baritiu 26",
7.    "postcode": "305402",
8.    "city": "Cluj-Napoca",
9.    "country": "RO",
10.   "telephone": "0345823823",
11.   "iban": "3424959434594582"
12. }

```

Figure 6.22. JSON with data from the DTO object

11. The Data is then displayed in the browser:

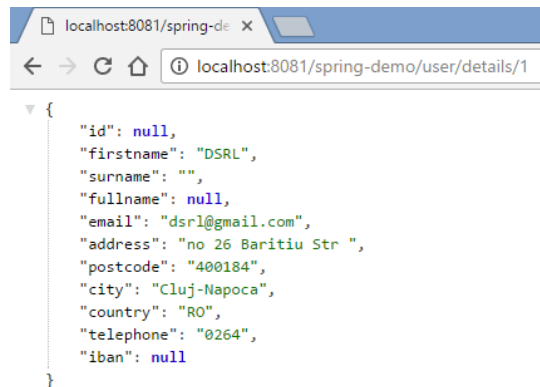


Figure 6.23. Response in the browser for a REST request

6.2.4. Testing the application

1. Insert several users in the table *user* from the *spring-demo* database
2. Configure the database properties from the *src/main/resources/application.properties* file
3. Run the project as: Click on Project > Run on Server (on the configured Tomcat Server)
4. Use a tool such as Postman¹⁰ to retrieve all the users from the database:
localhost:8080/spring-demo/user/all

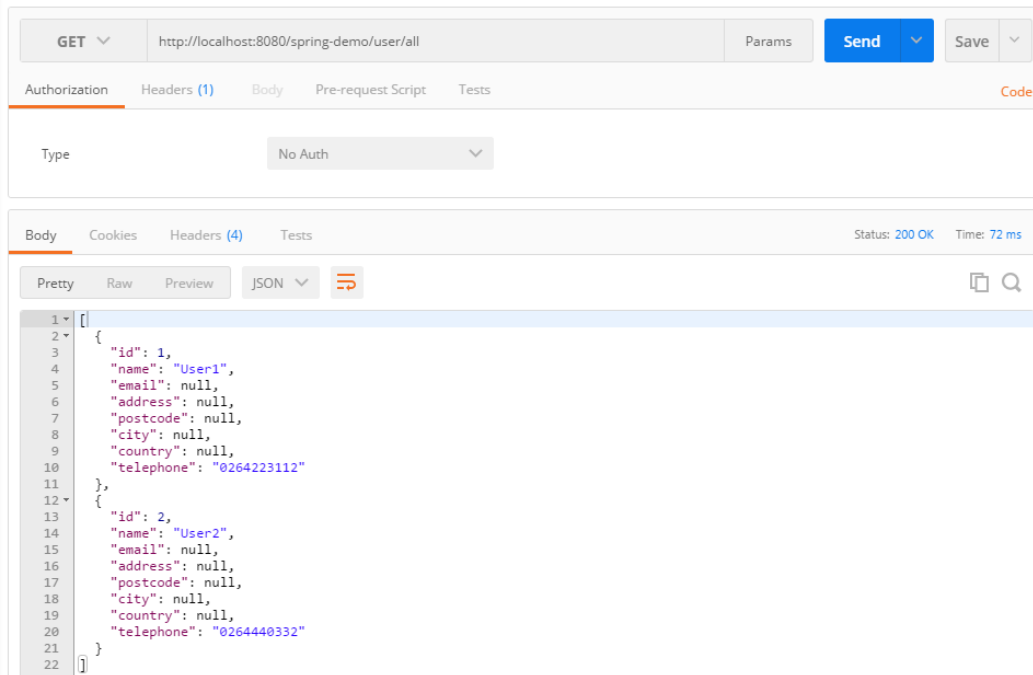


Figure 6.24. Example of using the Postman tool

¹⁰ <https://www.getpostman.com/>

6.3. Laboratory work: Spring REST backend for a distributed application

6.3.1. Requirements

Design and implement the backend services for an application with a layered architecture. The implemented functionality will be exposed as RESTful services, thus enforcing interoperability and portability of the system.

The functional requirements of the project will be defined by each student. As minimal complexity, the projects should have:

- A DB of around 3-4 tables, at least two foreign key relationships (one-to-many, many-to-many, etc.)
- CRUD operations implemented on the DB tables and exposed as REST functionality
- Login operation for specific types of users
- Two additional complex operations, such as reports with charts, mail sending to users, shopping cart for online shop, reservation system for hotel management, etc.

The technologies that must be used for implementation are: Spring REST with ORM or .NET C# WEB API with REST controllers.

6.3.2. Deliverables

Each student will deliver:

- A solution description document (about 4 pages, Times New Roman, 10pt, single spacing) containing:
 - General application description
 - Functional requirements
 - Non-functional requirements
 - Main use-case diagrams describing the complex operations functionality
 - DB design
- Source files. The Spring REST backend code that will be uploaded on the personal *bitbucket* account, following the steps:
 - Create a repository on *bitbucket* with the exact name:
DS_Group_FirstName_LastName_Spring_Backend
 - Push the source code and the documentation (push the code not an archive with the code or war files)
 - Share the repository with the user *utcn_dsrl*

The students will have to show tests for most of the CRUD functionalities (integration tests) with REST console applications (eg. Postman).

6.3.3. Evaluation

Table 6.2. shows how grading is performed for this assignment.

Table 6.2. REST backend laboratory work grading details

Points	Requirements
4 p	CRUD operations on DB tables (minimum 3 tables)
4 p	Two complex operations (rating, comments, shopping cart etc.)
1 p	Log-in with session keys
1 p	Code styling and good practices

6.4. References

- [1] Spring framework tutorials, <http://howtodoinjava.com/category/spring/spring-core/>
- [2] Spring framework documentation, <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/overview.html>
- [3] Spring Boot features, <http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-messaging.html>
- [4] Intro to WebSockets with Spring, <http://www.baeldung.com/websockets-spring>
- [5] Java EE Tutorial, <http://docs.oracle.com/javaee/5/tutorial/doc/bnbqa.html>

7. Client-side Frameworks: Angular for Developing Single-page Web GUIs

7.1. Introduction

Angular is an open-source front-end web applications framework and is based completely on JavaScript.

The framework addresses the main challenges which are encountered in the development of the web single-page applications such as duplication of code, dependency injection, business logic in JavaScript and declarative templates.

The project example from this laboratory work is a skeleton for an Angular application that can be used to communicate with the Spring hands-on application described in Chapter 6.

7.2. Hands-on application

7.2.1. Application installation and configuration

Install and configure the following resources:

- Node.js and Node Package Manager (NPM) – Install the last versions from <https://nodejs.org/> and check that Node and NPM are installed using the commands:
 - `node -v`
 - `npm -v`
- Angular CLI – install the last version from <https://cli.angular.io/>:
 - `npm install --save-dev @angular/cli@latest`
- WebStorm – download WebStorm from <https://www.jetbrains.com/webstorm/>
- NGINX – download the stable version of NGINX from <http://nginx.org/en/download.html>

To download and configure the example follow the steps described below:

1. Setup GIT and download the project from https://utcn_dsrl@bitbucket.org/utcn_dsrl/angular-demo.git
 - Create an empty local folder in the workspace on your computer
 - Right-click on the folder and select *Git Bash*
 - Write the commands:
 - `git init`
 - `git remote add origin https://utcn_dsrl@bitbucket.org/utcn_dsrl/angular-demo.git`
 - `git pull origin master`
2. In WebStorm select *File* and *Open...*
3. Search for the location of the project on the disk, enter the name of the project *angular-demo* and click *OK*.
4. When opened in WebStorm, the project has the structure from Figure 7.1.

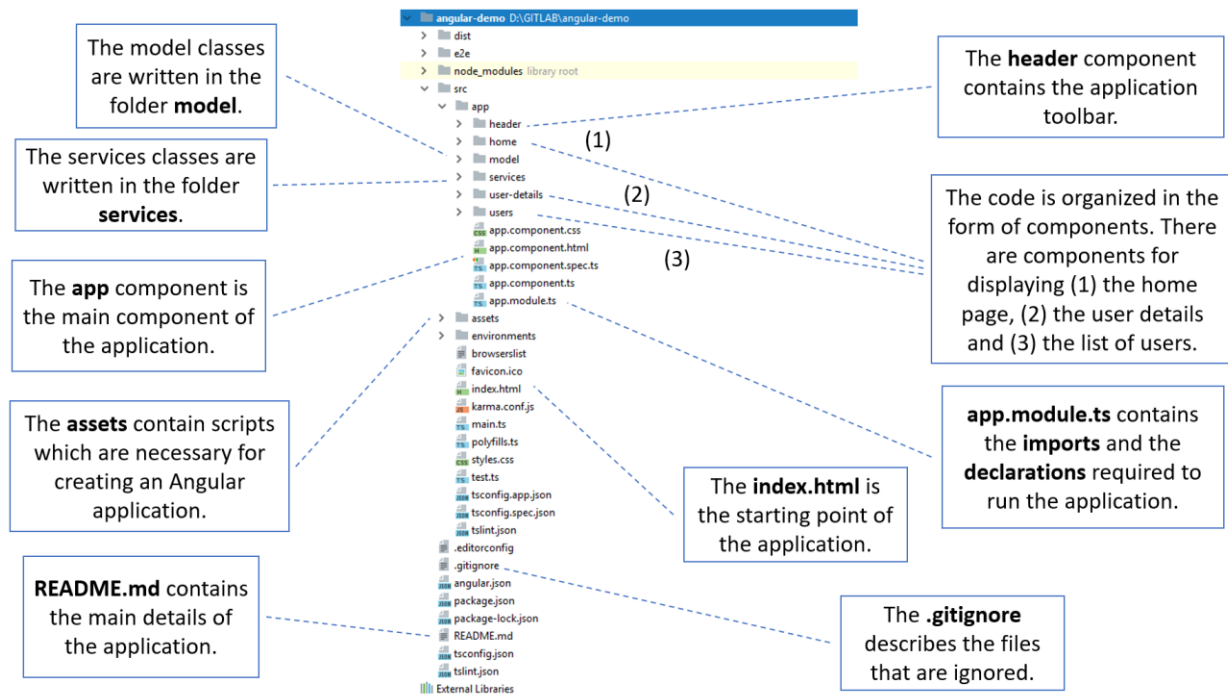


Figure 7.1. Angular project structure

To properly run the project, follow the next steps

1. From WebStorm open the terminal using *View, Tool Windows* and *Terminal*.
2. Download the packages writing the following command in the terminal:
`npm install`
3. Build the project using the following command from terminal:
`ng build --prod`
4. Run the project using one of the following two alternatives:

Alternative 1 – Run the application in WebStorm: in the WebStorm terminal write the command: `ng serve --open`

Alternative 2 – Run the application from NGINX: copy and paste the files from *angular-demo/dist/angular-demo* to *nginx-version/html* and start the NGINX server using the following command from a command line interface opened at the location *nginx-version*:
`start nginx`

5. Upon successful execution, you should access the *angular-demo* web application from browser.

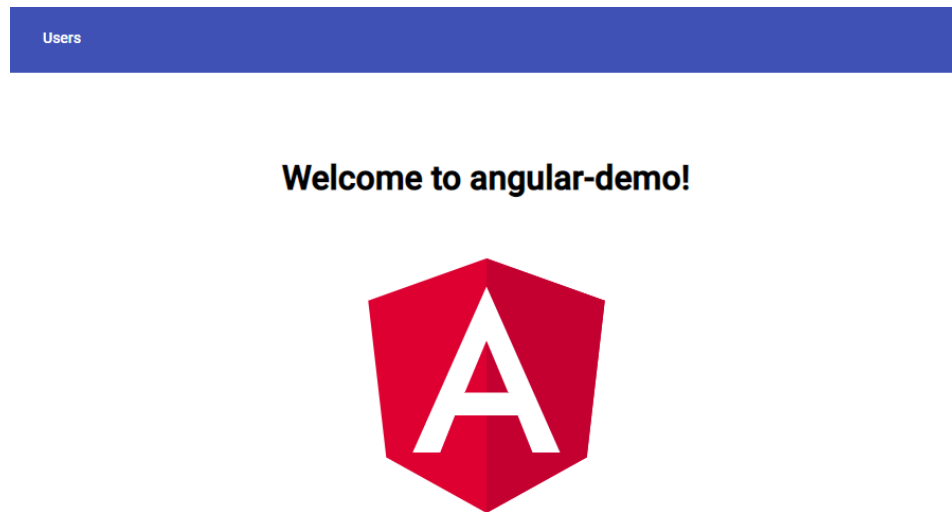


Figure 7.2. Hands-on example welcome page

7.2.2. Application conceptual architecture

The conceptual architecture of the system is presented below. The Angular application communicates with the Spring application through JSON (JavaScript Object Notation) objects. JSON is a syntax which can be used to store and exchange data. The **controllers** execute complex operations and communicate with the back-end. The **views** are html files and css files that use services to display information or to send information. The **model** classes are written in JavaScript and are used by the services and by the views. These components are detailed in Table 7.1.

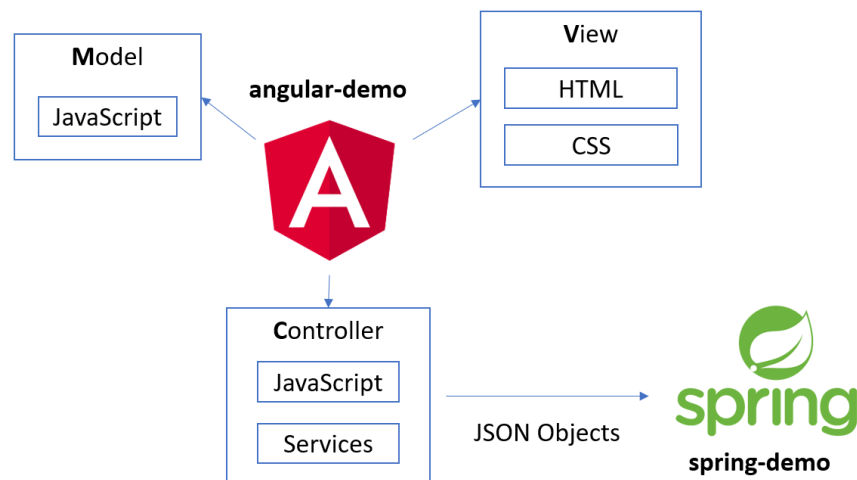


Figure 7.3. Project conceptual architecture

Table 7.1. Project component description

Architectural Dimension	Files	Description
<i>Model</i>	Model classes located in model folder.	Contains the model classes which are used for the communication between views and controllers. The classes are written in JavaScript.
<i>View</i>	Files ending in .html and .css.	Contains the .html and .css pages of the application. The JavaScript objects populate the .html files.
<i>Controller</i>	Files located in services folder and files ending in .component.ts.	The services execute complex operations, communicate with the back-end spring-demo application through JSON objects and use model classes for displaying the information in the view files.

7.2.3. Application implementation details

The flow of the operations which are required for the retrieval of all the users from the angular-demo application using the spring-demo application is presented in Figure 7.4.

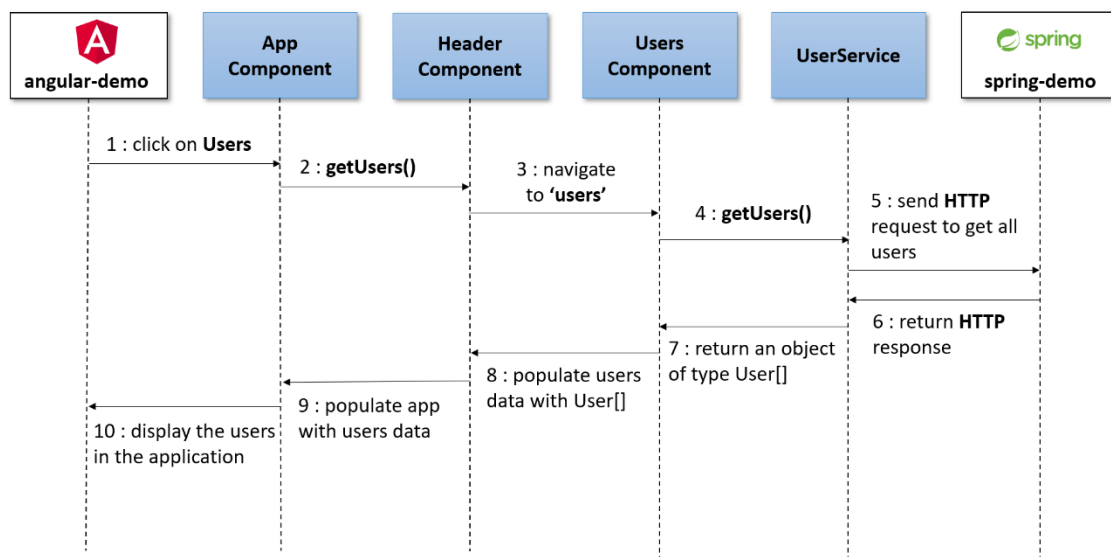


Figure 7.4. Sequence diagram for retrieving all users

The processing steps are detailed next:

1. The user clicks on the *Users* button from the top-left corner of the application.

Good to know

What is the purpose of the `<app-root>` tag?

The `<app-root>` tag is written in `index.html` and this is how Angular knows how to determine the component which corresponds to the tag, in this case the component `AppComponent`.

```
13 <body>
14   <app-root></app-root>
15 </body>
16 </html>
```

Figure 7.5. Snippet from `index.html`

2. The method which is specified by the attribute (`click`) (line 3) in the file `header.component.html` is called.

```
1 <mat-toolbar color="primary">
2   <button (click)="getUsers()" mat-flat-button color="primary">Users</button>
3 </mat-toolbar>
```

Figure 7.6. Calling the method `getUsers()`

Good to know

What are the three main parts of an Angular component?

The three main parts of an Angular component are the selector, the template and the style. The selector is a tag that is written in an html file and this is how the application knows how to interpret that an Angular component should be used. The template is an html file in which the component displays the information and the style describes how the content from the html file should be displayed.

```
4 @Component({
5   selector: 'app-header',
6   templateUrl: './header.component.html',
7   styleUrls: ['./header.component.css']
8 })
```

Figure 7.7. Snippet from `header.component.ts`

3. Navigate to `users` (line 18) which is associated with the `UsersComponent`. In the file `app.module.ts` the path `users` corresponds to `UsersComponent`.

```
9 export class HeaderComponent implements OnInit {  
10  
11     constructor(private router: Router) {  
12     }  
13  
14     ngOnInit() {  
15     }  
16  
17     getUsers() {  
18         this.router.navigate( commands: ['users']);  
19     }  
20 }
```

Figure 7.8. Snippet from HeaderComponent

Good to know

How to connect the paths and the components?

The *RouterModule* from *app.module.ts* is used for connecting the paths and the components. Each route has a *path* and a *component*. The *path* describes how the *component* can be accessed in the application.

```
31 RouterModule.forRoot( routes: [  
32     {  
33         path: 'users',  
34         component: UsersComponent  
35     },  
36     {  
37         path: 'user/:id',  
38         component: UserDetailsComponent  
39     },  
40     {  
41         path: '',  
42         component: HomeComponent  
43     }  
44 ]) )  
45 ],
```

Figure 7.9. Snippet from app.module.ts

4. Call the *getUsers()* (lines 15-17) method from *UserService*.

```
8 export class UserService {
9
10     usersURL = 'http://localhost:8080/spring-demo/user';
11
12     constructor(private http: HttpClient) {
13     }
14
15     getUsers() {
16         return this.http.get<User[]>({ url: this.usersURL + '/all' });
17     }
18
19     getUserById(id: number) {
20         return this.http.get<User>({ url: this.usersURL + '/details/' + id });
21     }
22 }
23
24
```

Figure 7.10. Snippet from UserService

5. The following HTTP request is sent to the *spring-demo* application:
http://localhost:8080/spring-demo/user/all

Good to know

How to create an HTTP request in Angular?

In Angular an HTTP request can be created using the class *HttpClient* from the library *@angular/common/http*.

```
15     getUsers() {
16         return this.http.get<User[]>({ url: this.usersURL + '/all' });
17     }
```

Figure 7.11. Snippet from user.service.ts

6. The *spring-demo* application returns the HTTP response and at this stage the objects are in JSON format.
7. The objects in JSON format populate an array of objects of type *User*: *User[]*.
8. The *users* object from *UsersComponent* is populated with *User[]* data (line 22).

```

11 export class UsersComponent implements OnInit {
12
13     users: User[] = [];
14     displayedColumns: string[] = ['name', 'telephone', 'details'];
15
16     constructor(private router: Router, private userService: UserService) {
17     }
18
19     ngOnInit() {
20         this.userService getUsers()
21             .subscribe( next data => {
22                 this.users = data;
23             });
24     }

```

Figure 7.12. Snippet from UsersComponent

9. The users' data is used for populating the table from *users.component.html*.

Good to know

What is Angular Material?

Angular Material is used for creating high-quality UI components using TypeScript and Angular.

```

25 MatToolbarModule,
26 MatMenuModule,
27 MatIconModule,
28 MatButtonModule,
29 MatTableModule,

```

Figure 7.13. Snippet from app.module.ts

10. The users table is displayed in the *angular-demo* application.

7.2.4. Testing the application

1. Insert several users in the table *user* from the *spring-demo* database.
2. Run the *spring-demo* application as presented in Chapter 6.
3. Run the *angular-demo* application as explained in the first section of this Hands-On.
4. Access the following link in the browser:
 - a. Alternative 1 (WebStorm): <http://localhost:4200/>
 - b. Alternative 2 (NGINX): <http://localhost:80/>
5. Click on the *Users* button located in the top-left corner of the web page. You should be able to visualize the users that populate the *user* table from the *spring-demo* database.

Users

Name	Telephone	Details
User1	065633422	<button>Details</button>
User2	065433211	<button>Details</button>
User3	034233420	<button>Details</button>

Figure 7.14. Angular GUI to display all the users

7.3. Laboratory work: Angular GUI for Chapter 6 Spring backend

7.3.1. Requirements

Design and implement a proper web GUI for the Section 6.3 implemented Spring REST backend services.

The design of the frontend will be defined by each student.

As minimal complexity, the frontend should interconnect with the already implemented Spring REST services to:

- login for specific users
- access the CRUD operations on the DB
- access the defined complex operations, such as reports with charts, mail sending to users, shopping cart for online shop, reservation system for hotel management, etc.

7.3.2. Deliverables

Each student will deliver:

- A solution description document (about 4 pages, Times New Roman, 10pt, single spacing) containing:
 - Conceptual architecture of the integrated distributed system.
 - Sequence diagrams describing the complex operations functionality
 - UML Deployment diagram.
 - Readme file containing build and execution considerations.
- Source files. The Angular frontend code that will be uploaded on the personal *bitbucket* account, following the steps:

- Create a repository on *bitbucket* with the exact name:
DS_Group_FirstName_LastName_Angular_Frontend
- Push the source code and the documentation (push the code not an archive with the code or war files)
- Share the repository with the user *utcn_dsrl*

The project will be executed, and the GUI functionality will be manually tested.

7.3.3. Evaluation

Table 7.2. shows how grading is performed for this assignment.

Table 7.2. Angular GUI laboratory work grading details

Points	Requirements
4 p	CRUD operations on DB tables
2 p	Appropriate GUI for the complex operations
1 p	Login functionality
1 p	Error messages and error pages
1 p	Code styling and good practices
1p	Pretty view

7.4. References

- [1] Angular framework, <https://angular.io/>
- [2] CRUD functionality using Angular 5 with Bootstrap 4
<https://medium.com/@mail.bahurudeen/simple-crud-functionality-using-angular-5-with-bootstrap-4-f7baac0d2000>
- [3] An Angular 5 tutorial: step-by-step, <https://go.tiny.cloud/blog/angular-5-tutorial-step-step-guide-first-angular-5-app/>
- [4] Angular Material, <https://material.angular.io/>