

Teodor-Traian ȘTEFĂNUȚ

USER INTERFACE DESIGN

Laboratory guide



UTPRESS

Cluj-Napoca, 2019

ISBN 978-606-737-382-0

Teodor-Traian ȘTEFĂNUȚ

USER INTERFACE DESIGN

Laboratory guide



Editura UTPRESS
Cluj-Napoca, 2019
ISBN 978-606-737-382-0



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. Câmpean

Recenzia: Prof. dr. ing. Dorian GORGAN
Și. dr. ing. Victor Ioan BĂCU

Copyright © 2019 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 978-606-737-382-0

Preface

The development of interactive web and mobile applications has increasingly become a significant area in software development activities. With multiple solutions competing over the same user needs and challenges, providing a very similar set of functionalities, the aspects of usability and improved user experience are becoming more important in the decision to use one app or another.

This publication groups together the support materials for the practical activities of the 4th year students of Computer Science Department from Technical University of Cluj-Napoca, at the User Interface Design discipline. At the same time, it aims to provide to any person interested in the basics of web and mobile application development, the introductory notions required to implement a first, basic application.

The material has been structured in two main sections: one dedicated to Web Technologies (HTML, CSS and JavaScript) and one focused on mobile development using Android technology. Each of the two sections has been organized in five topics (laboratories), that have been explored from both theoretical and practical perspectives. Implementation considerations have also been added to each guide providing debugging hints and best practices recommendations.

It is highly recommended for the beginners to follow strictly the order of the topics in each of the sections, as this will have an important role in understanding theoretical notions and in improving implementation efficiency. At the same time, more experienced developers can focus on particular topics of choice, as long as the explaining terms are familiar.

The information included in this publication has been carefully selected to provide a condensed, efficient and thorough start on the path of learning the described technologies. As searching for solutions and understanding faced challenges is a very important part of the learning process, the examples included are not complete tutorials, but rather key parts of the implementation process that guides the learner on the right path to the solution.

Table of Contents

LABORATORY 1 – HTML	2
1.1 Introduction	2
1.1.1 Laboratory objectives	2
1.2 Theoretical considerations	2
1.2.1 HTML labels	2
1.2.2 Most common attributes of HTML labels.....	5
1.3 Development considerations	6
1.4 Exercises	6
LABORATORY 2 – CSS formatting	8
2.1 Introduction	8
2.1.1 Laboratory objectives	8
2.2 Theoretical considerations	8
2.2.1 CSS selectors	8
2.2.2 Defining CSS classes.....	9
2.2.3 Rules for applying CSS specifications on HTML documents	10
2.3 Development considerations	11
2.4 Exercises	12
2.5 References	13
LABORATORY 3 – JavaScript	14
3.1 Introduction	14
3.1.1 Laboratory objectives	14
3.2 JavaScript	14
3.2.1 Including the code directly into the HTML file	14
3.2.2 Including JavaScript code from an external file	14
3.2.3 Subset of JavaScript functions	14
3.3 Development considerations	16
3.4 Exercises	17
LABORATORY 4 – jQuery library	19
4.1 Introduction	19
4.1.1 Laboratory objectives	19
4.2 Introduction to jQuery	19
4.2.1 Import jQuery library	19
4.2.2 Use jQuery functions	20
4.2.3 Manipulate HTML tags using jQuery	20
4.2.4 jQuery objects.....	21
4.2.5 Manipulate HTML elements	21
4.2.6 DOM traversal.....	24

4.2.7	DOM manipulation	25
4.2.8	Chaining function calls	26
4.3	Exercises	26
LABORATORY 5 – AJAX calls		28
5.1	Introduction	28
5.1.1	Laboratory objectives	28
5.2	AJAX calls using jquery library	28
5.2.1	Initiate server calls	28
5.2.2	Call answer management	31
5.2.3	Display received information in HTML	31
5.3	Development considerations	32
5.3.1	Server communication	32
5.3.2	Dedicated processing functions	33
5.4	Exercises	33
5.5	References	34
LABORATORY 6 – Introduction in Android development		36
6.1	Introduction	36
6.1.1	Laboratory objectives	36
6.2	Theoretical considerations	36
6.2.1	Hardware configurations	36
6.2.2	Basic notions for the development of Android applications adaptive to multiple resolutions	36
6.3	Development considerations	37
6.3.1	Create a new Android application	37
6.3.2	The generic structure of an Android application	40
6.3.3	Describing the user interface	41
6.4	Java aspects specific to Android applications	41
6.4.1	Access visual elements	42
6.4.2	Attach callbacks to user interactions	42
6.5	Exercises	42
LABORATORY 7 – Android UI and user interaction (1)		44
7.1	Introduction	44
7.1.1	Laboratory objectives	44
7.2	Theoretical considerations	44
7.2.1	Creating custom lists	44
7.2.2	Create a contextual menu and define user interactions with it	45
7.3	Development considerations	45
7.3.1	Transition to another activity	45
7.3.2	Creating an Adapter class	46
7.3.3	Connect the ListView element and the Adapter	46
7.3.4	Implementing the contextual menu	46

7.4	Exercises	47
LABORATORY 8 – Android UI and user interaction (2)		49
8.1	Introduction	49
8.1.1	Laboratory objectives	49
8.2	Theoretical considerations	49
8.2.1	Options bar	49
8.2.2	Progress bar	49
8.2.3	BACK button.....	49
8.2.4	Dialog windows.....	50
8.2.5	TOASTs	50
8.3	Development considerations	50
8.3.1	Defining a timer	50
8.3.2	Defining elements for the options bar	50
8.3.3	Populate the options bar	50
8.3.4	Identify selected option	50
8.3.5	Display a Toast message	51
8.3.6	Display a dialog window	51
8.3.7	Customize the behavior of the Back button	51
8.3.8	Pass data between activities	51
8.4	Exercises	53
LABORATORY 9 – Android UI and user interaction (3)		54
9.1	Introduction	54
9.1.1	Laboratory objectives	54
9.2	Theoretical considerations	54
9.2.1	Reuse of UI templates (elements of type View).....	54
9.2.2	ViewHolder template.....	54
9.2.3	Notify the RecyclerView.Adapter on data model updates.....	55
9.2.4	Customize list elements display	56
9.3	Development considerations	56
9.3.1	Adding required library for RecyclerView to the APP	56
9.3.2	Implement RecyclerView	56
9.3.3	Process the onItemClick event for a RecyclerView list element	58
9.3.4	Use different display templates in the same list	58
9.4	Exercises	58
LABORATORY 10 – connect to a REST API from Android		59
10.1	Introduction	59
10.1.1	Laboratory objectives.....	59
10.2	Retrofit library setup	59
10.2.1	Including required dependencies.....	59
10.2.2	Creating model classes.....	60
10.2.3	Declaring HTTP operations.....	60
10.2.4	Register the API interface with the Retrofit library	61

10.2.5	Making calls to the API endpoints.....	62
10.3	Development considerations	63
10.3.1	Adding internet access permissions.....	63
10.4	Exercises	63
10.5	References	63

WEB TECHNOLOGIES FOR USER INTERFACE DEVELOPMENT

LABORATORY 1 – HTML

1.1 Introduction

The development of professional WEB applications requires the implementation of a user interface that looks and behaves identical on each and every browser that the application's users might be familiar with. The main technologies used are HTML, CSS and JavaScript.

1.1.1 Laboratory objectives

Present basic notions on HTML description language and describe the most common HTML tags . Apply theoretical concepts in practical examples.

1.2 Theoretical considerations

HTML is the standard language for describing an organizing the content meant to be visualized through a web browser.

The structure of a valid HTML document has the following components:

1. **first line of the document** – specifies the language version used by the document
2. **header section** – contains general information about the document and needs to be fully loaded before starting the download for the document body; most of the information from the header section is not visible into the document.
3. **document body** – the actual content

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <TITLE>My first HTML document</TITLE>
  </HEAD>
  <BODY>
    <P>Hello world!</P>
  </BODY>
</HTML>
```

1.2.1 HTML labels

Used for content layout and formatting inside the document (content section) or for attaching more meta-information about the document (header section) like: external files, format specification, information about authors etc. The entire structure of the document can be represented as a tree, having <HTML> element as root. This tree representation of an HTML document is called DOM (Document Object Model) and has an important role in allowing the browser to access and manage elements, exposing them also to the various client-side scripting languages (ex. JavaScript).

According to the implicit display mode, visual HTML labels that are used to layout the content can be grouped into:

- *block labels* – they add a vertical break to the content and use the entire horizontal space available
- *inline labels* – displayed inline with the text content, they are influenced by the formatting applied to text

Most common labels encountered in an HTML document are:

HTML label	Description
TITLE	Label included into the header section, it specifies the title of the document. The text here is used by the browser as a title for the window (tab) in which the document is displayed.
LINK	Defines a link to an external document (ex. CSS, another HTML) that can be used by the browser to visually format or manage the content of the document. This label is also included into the header section.
SCRIPT	Allows the integration of programming instructions inside the HTML document. These instructions will be executed by the browser when the page is loaded or upon different user interactions. The code itself can be loaded from an external document referenced by the SCRIPT label or can be verbatim included into the original file. Most common language used today is JavaScript.
STYLE	Allows the integration of CSS formatting instructions inside the document. Best practices in web development recommend the deprecation of this label in favor of including all CSS specifications into an external file referenced in the header section of the HTML document.
H1, H2, H3, H4, H5, H6	Block type labels that can be included only in the body section of the document. They are used to structure the content into sections and sub-sections. Best practices recommend to have only one H1 label in each document.
DIV	Block type label used to group the content into a rectangular section.
P	Block type label used to mark paragraphs into the text type content.
A	Inline type label that allows the specification of an external (to another document) or internal (different section of the current document) link.
IMG	Label that has the implicit representation as an inline element and allows the integration of images into the HTML document.
SPAN	Inline type label used for specific formatting of text content.
FORM, INPUT, LABEL	Labels used to describe forms inside HTML documents: <ul style="list-style-type: none"> • FORM – block type label that groups INPUT elements • INPUT – generically describes the elements available to be included into a form, the exact type of the element being specified through the TYPE attribute • LABEL – associates a title to the elements of type INPUT
TABLE	Allows the integration of a table layout into an HTML document.

THEAD	Visual and semantic element that highlights a section of a TABLE as a header / title section.
TBODY	Visual and semantic element that groups the actual content of a table.
TR, TD	Labels that describe rows (TR) and cells (TD) into a table. The number of <TD> labels contained by a <TR> label establishes the number of columns for that specific row, that can be the same for the entire table or different for every row.
OL, UL	Labels that allow the specification of ordered (OL) or unordered (UL) lists.
LI	Describe a single element from a list, either ordered or unordered.

Labels added by the HTML 5 standard with the purpose of semantic organization of the content:

HTML label	Description
HEADER	Groups the graphical elements that are part of the visual header of the page (ex. company name and logo, motto, picture, etc.)
SECTION	Allows the partition of the content into logical sections.
FOOTER	Describes the bottom section of the HTML document which usually presents information about copyright, last page update, contact, etc.
NAV	Includes the visual HTML labels that describe the main menu of the document and other navigational information.

New content types natively embedded into HTML 5 through the definition of new labels:

HTML label	Description
VIDEO	Allows video content embedding and playback without the use of a third-party technology (ex. Silverlight, Flash, etc.)
AUDIO	Allows sound playback natively by the browser without the use of a third-party technology (ex. Silverlight, Flash, etc.)
CANVAS	Provides programmers with a bitmap formatted display zone that allows real time display of images, animations etc.

1.2.2 Most common attributes of HTML labels

HTML label	Description
id	Common attribute for most of the HTML labels. It's value uniquely identifies the HTML element in the entire document.
class	Common attribute for most of the HTML labels. It allows the specification of one or more CSS classes for the current element, deciding the visual formatting for it and for it's embedded elements.
src	Defines the path towards an external file of type script, image, etc. (depending on the type of the label fir which it has been defined).
type	Specifies the type of the INPUT label that contains this attribute.
action	Attribute specific to the FORM label, indicates the path that will receive the information gathered at form submission.
encoding	Attribute specific to the FORM label, indicates the encoding type used for the data included into the form in order to ensure correct transmission to the server.
onfocus	Allows the attachment of a JavaScript function that is called when the element receives focus.
onblur	Allows the attachment of a JavaScript function that is called when the element loses focus.
onchange	Allows the attachment of a JavaScript function that is called when the value of the element (ex. content of INPUT label) has been changed.
onload	Usually defined on the BODY element. Allows the attachment of a JavaScript function that is called when the element for which it has been defined is completely loaded and available for JavaScript access.
onkeydown	Specifies the JavaScript function that is called for each key press, if the element for which it is defined has keyboard focus.
onkeyup	Specifies the JavaScript function that is called for each key release, if the element for which it is defined has keyboard focus.
onkeypress	Specifies the JavaScript function that is called for each key stroke, if the element for which it is defined has keyboard focus.

1.3 Development considerations

In order to write HTML code any text editor will do just fine (ex. Notepad). For more advanced features (code auto-completion, code highlighting, etc.) you can use editors like Notepad++, Eclipse, Visual Studio, Netbeans, Sublime, Brackets, etc.

In order to visualize the HTML structure interpreted by the browser you can use the development tools provided by most of the current browsers, usually accessible on Windows platform by pressing F12 key.

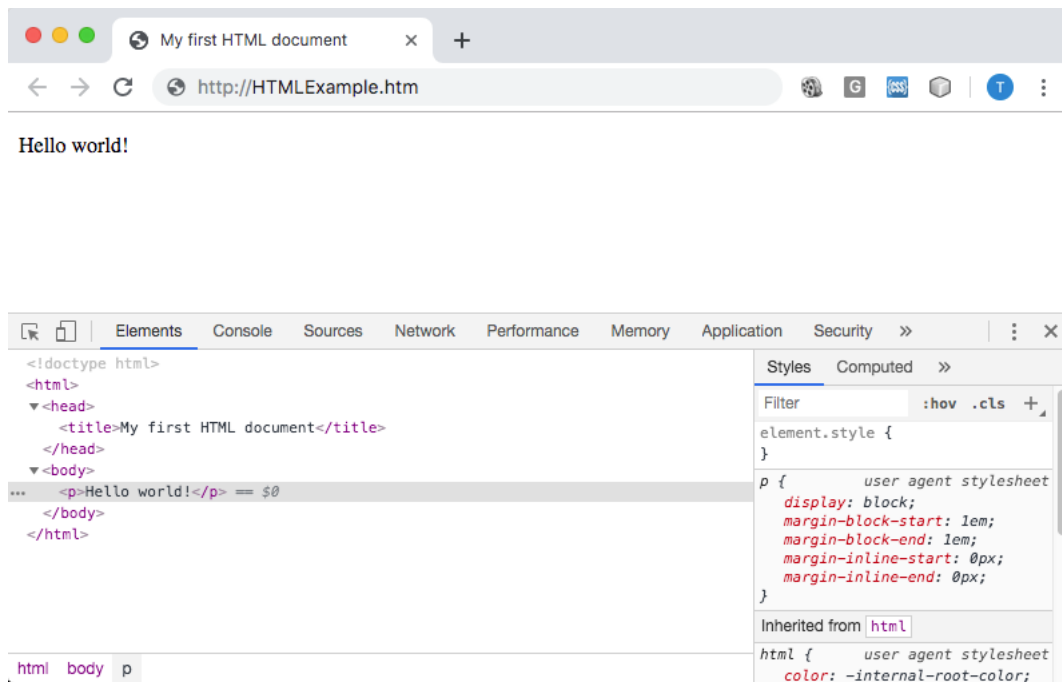


Figure 1.1: Development tools example in Chrome under Mac OS X

In order to view the result of your code, files with `.htm` or `.html` extensions can be opened into the browser directly from the local HDD. You should be able to use any browser of your choice available on the development workstation (Microsoft Edge, Mozilla Firefox, Google Chrome, Opera etc.).

1.4 Exercises

Exercise 1. Create an HTML page for a tourism agency with the following minimum elements:

- one H1 label for the title of the page
- an unordered list of hyperlinks (A labels), that will be used as the main menu of the website, having the following entries: *Home*, *Cruises*, *Hiking*, *Swimming*, *Special offers*
- a 900 x 350 px image that will have the role of a publicity banner
- one H2 label with the title *About us*
- two text paragraphs (P label) for the section *About us*

Exercise 2. Continue the development of the Web Page created at Exercise 1 by adding the following elements:

- one H2 label with the text *Special offers*, that will mark a new section into the page
- three subsections, each of them containing the following elements (please see the image below as example):

- one H3 label as a title
- one IMG label,
- one P label as containing description

Barcelona



Ut elit odio, varius vel est vel, tincidunt malesuada ex. Phasellus vulputate quam placerat, mollis sapien in, laoreet mi.

LABORATORY 2 – CSS FORMATTING

2.1 Introduction

Style sheets represent a modular approach in defining the visual formatting of HTML documents, easy to integrate directly at page or even website level. Best practices in web development recommend to use external files having .css extension to group all the visual instructions that are later applied to the HTML document through the use of LINK label.

```
<LINK rel="stylesheet" href="file_name.css" type="text/css">
```

2.1.1 Laboratory objectives

Present basic notions in defining CSS specifications, using CSS selectors (classes, id bases, DOM based) and applying CSS formatting on HTML elements.

2.2 Theoretical considerations

Style instructions are mandatory grouped into **CSS classes** that are atomically applied to the HTML elements they are connected to. It is not possible to apply only a subset of the directives from one class, but it is allowed to overwrite specific instructions with the use of another class.

A CSS class has the following generic syntax, in which **[class_name]** can be any of the selectors mentioned in section 2.3:

```
[class_name]
{
  property_1: value_1;
  property_2: value_2;
  ...
  property_n: value_n;
}
```

2.2.1 CSS selectors

The link between CSS specifications and the HTML element to which these should be applied can be defined in one of the following ways:

2.2.1.1 Using HTML label

The CSS classes that use a **class_name** an HTML label are automatically applied to all the elements of that label type identified into the document. For example, the following instructions will be applied to all DIV type elements, displaying a 1px thick, green color border around them.

```
div
{
  border: 1px solid #00FF00;
}
```

Consequently, it is recommended to use this type of selectors in order to define general display rules for HTML element types, exceptions being defined through CSS classed with different, more specific selectors.

2.2.1.2 Using CLASS attribute of the HTML label

In order to apply same formatting styles to a heterogeneous group of elements from the document it is necessary to define a type-independent CSS class, using the syntax **".className"**. For example:

```
.importantMessage
{
  color: red;
  font-style: italic;
  font-weight: bold;
}
```

The link between the HTML element and the class definition is performed through the **class** attribute:

```
<ELEMENT class="class1 [class2 class3 ...]">
...
<ELEMENT class="importantMessage">
```

2.2.1.3 Using the HTML element ID

The use of this selector is recommended only when the CSS instructions apply to only one HTML element in the entire document. In this case the element must be uniquely identified through it's ID attribute:

```
<ELEMENT id="errorMessage">
```

In CSS, the selector must be defined as **#idElement**. In our example:

```
#errorMessage
{
  color: white;
  font-weight: bold;
  background-color: red;
}
```

2.2.1.4 Using the STYLE attribute of the HTML element

This approach has top priority in overwriting the CSS visual instructions applied to the element (see section 2.5). However, best practices recommend avoiding this type of visual formatting, as the specifications here cannot be reused anywhere else into the document (or in other documents) except for the elements embedded into the formatted label.

```
<ELEMENT style="css_property_1:value_1; css_property_2:value_2; ...">
```

2.2.2 Defining CSS classes

CSS classes that have common formatting instructions can be defined together, separating the selectors through comas.

```
[class_name_1, class_name_2, class_name_3]
{
  property_1: value_1;
  property_2: value_2;
  ...
  property_n: value_n;
}
```

The definition syntax presented above can be used with any selector types, the developer having the possibility to mix them for the same definition. As mentioned above, CSS instructions of a class cannot be applied only partially. The subtle differences from classes defined together can be overridden later into the same document, redefining the new properties:

```
[class_name_2]
{
  property_1: new_value;
  new_property_1: new_value_1;
  new_property_2: new_value_2;
  ...
  new_property_n: new_value_n;
}
```

In any CSS specification, in the same file or in different files, a specific selector can be redefined multiple times. The final form of the CSS class is obtained through the overridden of the definitions encountered before, according to the positional rule: last value overrides previous values. When multiple CSS files are used, the order in which they are parsed is the same with the order in which they are included into the HTML document.

2.2.3 Rules for applying CSS specifications on HTML documents

Each of the HTML elements have multiple CSS rules applied, from different CSS classes. Some of these rules are inherited from enclosing labels while others are directly applied to the element. Similar to the CSS classes with multiple definitions, identical display instructions are overridden according to the attachment order and to the **selector specificity**.

When the CSS specifications are attached to the HTML element through the same selector types, similar to:

```
<ELEMENT class="class_name_1 class_name_2 clas_name_3">
```

the properties from **class3** will overwrite all the identical instructions from **class1** and **class2**.

In order to establish the overridden order for CSS specifications attached through different selectors (inherited from parent elements, HTML tag selectors, id based, etc.), the browser computes a number called **selector specificity**, applying the following rules:

- formatting instructions defined in **style** attribute have highest priority (1,0,0,0 points)
- for each **ID** mentioned in selector, are awarded 0,1,0,0 points
- for each **generic CSS selector** (.className) are awarded 0,0,1,0 points
- for each selector based on **HTML tags** are awarded 0,0,0,1 points

See picture below for an illustration of specificity computation (picture taken from [1])

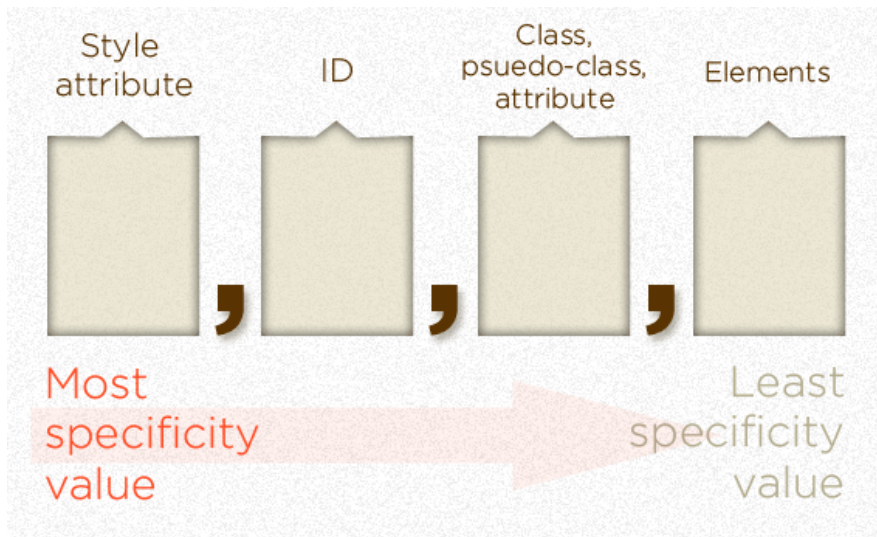


Figure 2.1: CSS rule specificity computation (taken from [1])

It is also important to mention that all the formatting instructions inherited from ancestor elements, no matter how the applying selector has been defined, have the specificity 0,0,0,1.

2.3 Development considerations

CSS description does not need to be compiled before publishing. It can be written with any text editor (ex. Notepad). For more advanced features (code auto-completion, code highlighting, etc.) you can use editors like Notepad++, Eclipse, Visual Studio, Netbeans, Sublime, Brackets, etc.

In order to view the results of a CSS formatting it is necessary to link it with an HTML file and to apply the rules on the HTML elements. When the connection between the HTML and CSS files is correct, opening the former in any browser will trigger automated loading of the latter and updates on CSS formatting for all the labels in the document. Applied styles can be easily checked in any browser using Developer Tools section:

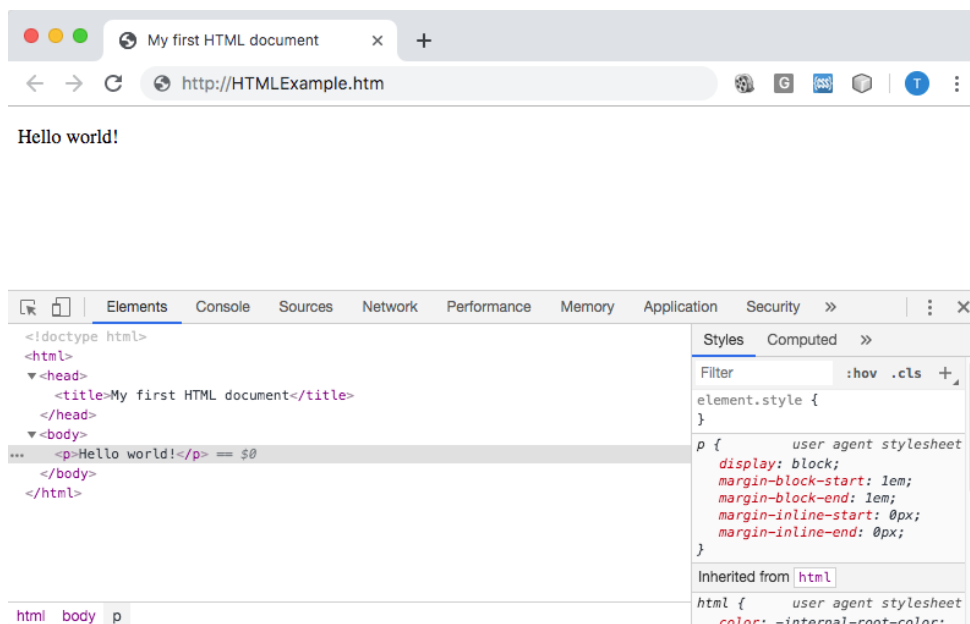
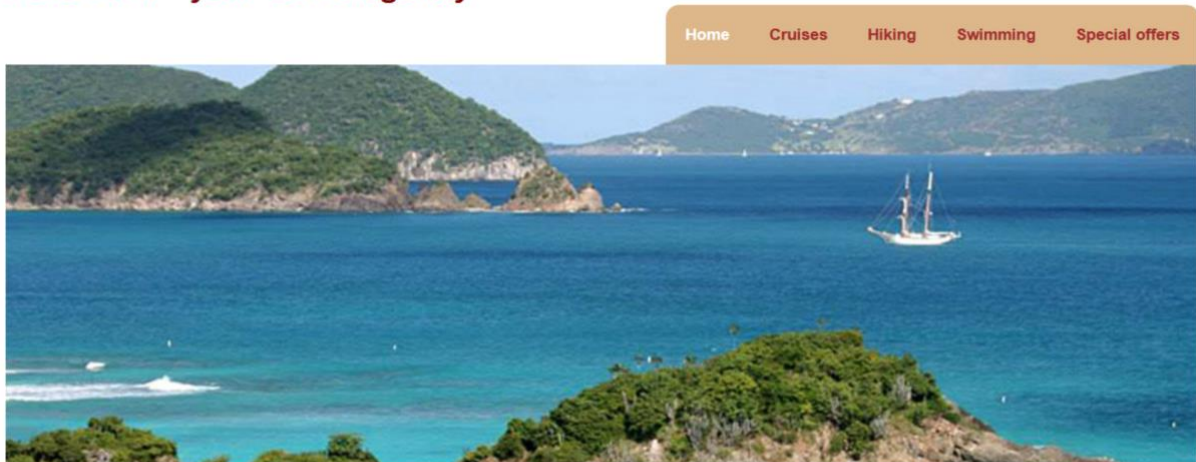


Figure 2.2: Development tools example in Chrome under Mac OS X

2.4 Exercises

Exercise 1. Implement the design below using HTML and CSS technologies:

CalaTour - your travel agency



ABOUT US

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut elit odio, varius vel est vel, tincidunt malesuada ex. Phasellus vulputate quam placerat, mollis sapien in, laoreet mi. Aliquam venenatis dui vitae quam feugiat accumsan. Aenean in aliquet libero. Mauris odio dui, placerat a efficitur et, pulvinar at quam. Phasellus auctor odio dui, ac tincidunt ipsum ornare vitae. Mauris quis sapien scelerisque, consectetur diam ac, consequat augue. Cras hendrerit lectus non urna volutpat pulvinar. Etiam a pretium nulla, ac condimentum dolor. Curabitur commodo accumsan lobortis. Duis sed justo eget mi dignissim volutpat. Donec imperdiet, odio et vulputate efficitur, purus neque malesuada massa, rutrum elementum turpis dui vel enim. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Phasellus ornare erat ipsum, sed congue felis varius eget. Fusce erat ligula, faucibus sit amet magna a, imperdiet lacinia arcu. Duis tristique faucibus justo eget maximus. Maecenas dictum pulvinar sem, sed efficitur augue interdum quis. Duis quis tellus est. Donec pretium nisi sed portitor rutrum. Integer portitor tellus quis velit convallis pellentesque. Nullam hendrerit nec ex sit amet elementum. Quisque mauris justo, rutrum ut tellus vel, placerat ultrices velit. Vestibulum porta odio eu cursus lacinia. In blandit turpis nec turpis tempus accumsan. Aenean sodales ultricies mauris id viverra.

Proin eleifend malesuada volutpat. Donec ut nibh eget est volutpat suscipit. Aliquam erat volutpat. Suspendisse semper diam ut massa semper tincidunt. Sed ligula nisi, efficitur sed velit sit amet, eleifend fringilla massa. Cras vestibulum commodo egestas. Sed sed ex dui. Ut at quam vulputate, pellentesque turpis nec, tincidunt arcu.

Implementation requirements:

- **title** – font: Arial 24px bold, color: #800000; it is also a hyperlink to the first page (Home)
- **menu** – background color: #DEB887, round corners of 10px radius
- **menu element** – are of type A, font: Arial 12px, default color: #800000, hover color: #FFFFFF
- **section title** – font: Arial 18px bold, color: #000000, margin: 10px top and bottom
- **regular text** – font: Arial 12px, color: #666666, alignment: justified; spacing between paragraphs: 10px
- when creating the HTML structure you can use only the following tags: *html, head, body, title, header, section, footer, h1, h2, h3, div, p, span, ul, li, a, img, strong, em*
- when applying styles you must use only HTML based CSS selectors
 - you should not use *style* attribute
 - you should not include CSS description in the HTML document
 - you are not allowed to use ID or CLASS attributes on the HTML tags

Exercise 2. Add a new section to the document created for Exercise 1 by implementing the design in the picture below using HTML and CSS:

LAST MINUTE

Crete - 5 days  <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec imperdiet, odio et vulputate efficitur.</p>	Barcelona  <p>Ut elit odio, varius vel est vel, tincidunt malesuada ex. Phasellus vulputate quam placerat, mollis sapien in, laoreet mi.</p>	Montenegro  <p>Donec imperdiet, odio et vulputate efficitur, purus neque malesuada massa, rutrum elementum turpis dui vel enim.</p>
--	---	--

Implementation requirements:

- **section title** – font: Arial 18px bold, color: #000000, margin: 10px top and bottom
- **subsection title** – font: Arial 14px bold, color: #0000FF, background color: #7FFFD4, padding: 5px
- **subsection** – width: 33% of display width, spacing: 10px left and right
- **image** – with: 100% of subsection width, fixed height: 300px
- **text** – font: Arial 12px, color: #666666, alignment: justified, background color: #DEB887
- **paragraph** – margin: 10px top and bottom, padding: 10px
- when creating the HTML structure you can use only the following tags: *html, head, body, title, header, section, footer, h1, h2, h3, div, p, span, ul, li, a, img, strong, em*
- when applying styles you must use only HTML based CSS selectors
 - you should not use *style* attribute
 - you should not include CSS description in the HTML document
 - you are not allowed to use ID or CLASS attributes on the HTML tags

2.5 References

[1] Chris Coyier, Specifics on CSS Specificity, 10 Mai 2010, <http://css-tricks.com/specifics-on-css-specificity/> [ONLINE, 10 May 2019]

LABORATORY 3 – JAVASCRIPT

3.1 Introduction

HTML and CSS are only descriptive languages, that can interact with the user only in a very limited and static manner. More complex interactions required in today's web applications (dynamic animations, asynchronous content update from the server, data validations, etc.) are widely implemented in JavaScript, a programming language that can be interpreted by the browser in real-time.

3.1.1 Laboratory objectives

Present basic notions on JavaScript programming: DOM manipulation and data validation. Apply theoretical concepts in practical examples.

3.2 JavaScript

Through JavaScript the programmers have access to the DOM structure of the document and can dynamically, at runtime, modify different properties of the HTML elements. It is important to remember that the DOM structure is not available from the first second of document access. It becomes accessible only after all the elements from between the <BODY> tag have been downloaded and interpreted by the browser. At that moment, the event **window.onload** is triggered, which can be used as an initiator for any JavaScript code that needs to run automatically on page load.

There are two main approaches in connecting the SCRIPT elements to an HTML document:

3.2.1 Including the code directly into the HTML file

Approach usually used when the JavaScript code is specific to the HTML file and will not occur anywhere else into the website.

```
<SCRIPT type="text/javascript">  
    // JavaScript instructions  
</SCRIPT>
```

In this case, the JavaScript code is inserted directly under the <HEAD> or <BODY> tags of the document.

3.2.2 Including JavaScript code from an external file

This is the recommended way of organizing active code for a website, as the same source code can be in this way used in multiple HTML pages, without any alterations.

```
<SCRIPT src="file_name.js" type="text/javascript"></SCRIPT>
```

In most of the cases (but not always) this use of the <SCRIPT> tag is encountered into the HEAD section of the HTML document. Into the same document, this tag can be reused as many times as necessary, allowing the inclusion of any number of external scripts.

3.2.3 Subset of JavaScript functions

In our laboratory implementation we will mostly use the following JavaScript functions:

window	Keyword that returns a pointer to the DOM entry that represents the browser window in which the current script is running.
window.onload	Event raised by the browser when the DOM construction is finished and it becomes available to the script.
document	Keyword that returns a pointer to the DOM entry that represents the <BODY> tag.
document.getElementById("id_element")	Returns the DOM reference to the HTML element that has the attribute ID = "id_element".
document.getElementsByTagName("LABEL")	Returns an array of DOM references to all the HTML elements from the document that are of the type LABEL.
this	Keyword that is a reference to the DOM element for which the current script has been called. For example:
	<pre>elem.onblur = function() { formInputBlur(this); };</pre>
	In the above script, when the anonymous function is called, this = elem .
FORM.onSubmit	Event raised by the browser when a FORM element from the current page is ready to be sent to the server. In order to cancel the submission, the attached callback function needs to return false .
ELEMENT.onblur	Event raised by the browser when ELEMENT is losing focus.
ELEMENT.onclick	Event raised by the browser when the ELEMENT has been clicked.
ELEMENT.getAttribute("attribute_name")	Returns the value of the attribute_name attribute of the ELEMENT.
ELEMENT.setAttribute("attribute_name","value")	Sets the value of the attribute_name attribute to value for the ELEMENT.

ELEMENT.innerHTML	Returns the content of the ELEMENT label in the form of HTML code. If a value is set on this property, the content of ELEMENT will be replaced
VARIABLE.length	Returns the number of elements in VARIABLE: <ul style="list-style-type: none"> • number of letters, if VARIABLE is a STRING • number of elements, if VARIABLE is an ARRAY
STRING.indexOf(string)	Returns the index (count starting from 0) on which the first occurrence of string in STRING is found, or -1 otherwise.
Date.parse(string)	Returns the number of milliseconds between the valid date in string and 1 st of January 1970. If string is not a valid date, the result will be NaN.
isNaN(object)	Returns true if object has value NaN.

In order to use regular expressions in JavaScript, one needs to first define the expression using **/expression/** construction and then use the **test** function over the verified element.

```
var re = /^[^<>()[\]\\..,;:~@"]+(\.[^<>()[\]\\..,;:~@/;
re.test( string ); // returns true if string matches the expression re
```

3.3 Development considerations

The same as for HTML and CSS, JavaScript code can be written in any text editor (ex. Notepad). For more advanced features (code auto-completion, code highlighting, etc.) you should use editors like Notepad++, Eclipse, Visual Studio, Netbeans, Sublime, Brackets, etc.

JavaScript code can be run only in a browser and only after it has been correctly integrated with an HTML file. After loading in browser, you can debug the code using the development tools (provided by most of the current browsers), usually accessible on Windows platform by pressing F12 key.

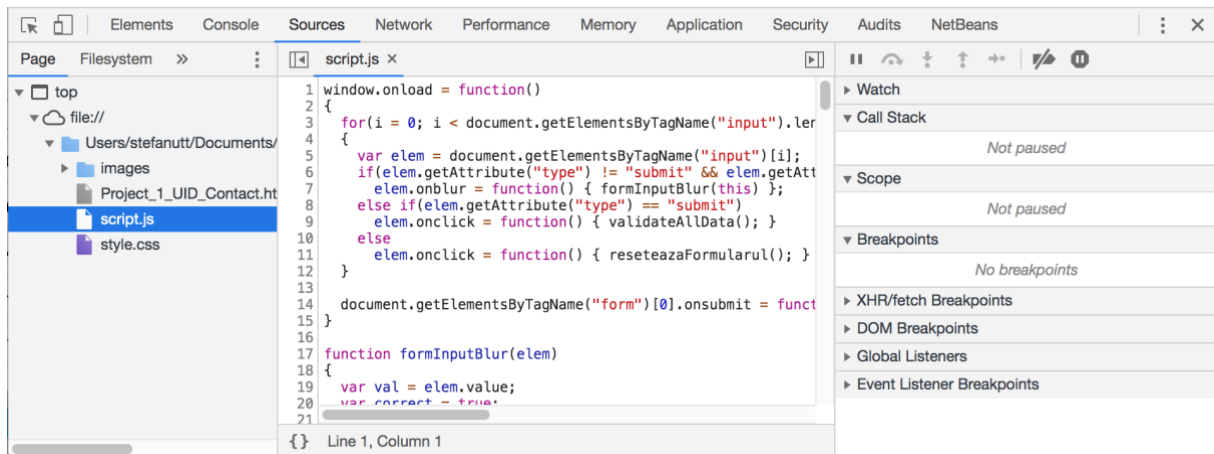


Figure 3.1: Development tools example in Chrome under Mac OS X

Related with the specifics of running JavaScript code it is important to mention that each page reload in the browser will re-initialize all JavaScript execution (the previous state will be lost). Data persistence using JavaScript can be achieved only through cookies, URL or using Local Storage functions.

3.4 Exercises

Exercise 1. Implement a contact form containing the following fields: *Surname*, *First name*, *Address*, *Birthdate*, *Phone number*, *Email*, *Favorite color*. Add two buttons named **Reset** and **Submit**. Using JavaScript, implement the following requirements:

- when the **Reset** button is pressed, all the fields in the form are reset to their initial state (no content or initial value)
- when the **Submit** button is pressed, the form will be hidden and in its place the information from the fields will be displayed as follows:

Surname: [surname_field_value]

First name: [first_name_field_value]

Address: [address_field_value]

Birthdate: [birthdate_field_value]

Phone number: [phone_numer_field_value]

Email: clickable link to the email address from the Email field

Favorite color: 200 x 20px rectangle filled in with the selected color in the form

- all the above requirement will be implemented without page reloading

Exercise 2. Using the form created at Exercise 1, implement in JavaScript the following requirements:

- when the **Submit** button is pressed
 - **Surname** must contain at least three alphanumeric characters
 - **First name** must contain at least three alphanumeric characters
 - **Address** should be at least three characters long and contain at least a digit. You should make sure that the address does not contain any of the symbols: @#\$\$%^&*
 - **birthdate** should be filled in with a valid date in the format mm/dd/yyyy
 - **phone number** field should contain only digits and one – sign, in the format nnn-nnnnnnnnn (3 - 9)

- email should also be verified to accept only syntactically valid addresses
- each time the **Submit** button is pressed the messages for all the identified errors, no matter how many there are, will be displayed to the user through a single **Alert** window.

LABORATORY 4 – JQUERY LIBRARY

4.1 Introduction

JavaScript language is, in fact, a version of ECMAScript, and present some implementation differences between browsers. Although sustained efforts are carried on with the purpose of reducing these differences, JavaScript code created for production still needs to be tested extensively on the most common browsers to ensure the same user experience.

As a response to these issues and on an attempt to reduce the testing effort, multiple cross-browser JavaScript libraries have been implemented. They ensure code compatibility and functionality consistence on all targeted browsers, providing also a higher level API for rich applications development. One of the most popular libraries today is jQuery.

4.1.1 Laboratory objectives

Present basic aspects and functionalities of the jQuery library. Focus on HTML tags interaction and DOM manipulation.

4.2 Introduction to jQuery

It is important to understand that jQuery is not a programming language, but a library of functions that is implemented in JavaScript and ensures cross-browser compatibility. In other words, it hides the differences in JavaScript code execution and provides to the users an identical behavior of the web application independent on the browser used to access it. This is achieved through the identification of available native APIs and the call of browser-specific functions whenever deemed necessary.

4.2.1 Import jQuery library

In order to access the jQuery function from your own JavaScript code, it is first necessary to include it in the HTML file. The statement is identical with the inclusion of any other external JavaScript file. As it is written in JavaScript, jQuery is actually included directly as source-code and compiled at runtime by the browser at each page load. The last available version of the library can be downloaded from: <http://jquery.com/download/>

There are two main approaches on including jQuery library in the HTML file:

- (1) download the source code on the application's web server, next to other JS files that you have, and include it in HTML as follows:

```
<script type="text/javascript" src="js/jquery.min.js"></script>
```

- (2) use a Content Delivery Network (CDN) to optimize delivery time and save bandwidth, by loading the source code directly from the source

```
<script type="text/javascript"
  src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js">
</script>
```

4.2.2 Use jQuery functions

It is very important to note that jQuery functions can be invoked only after the library has been downloaded by the browser, parsed and initialized. For this matter, any JavaScript code that relies on jQuery should run only after the page is fully loaded and DOM elements become available.

It is thus recommended to make use of the **window.onload** event for starting your own code execution only after all JavaScript resources have been downloaded, parsed and initialized. For these cases, jQuery provides the following syntax:

```
$( document ).ready( function() {  
    // code here will be executed when all DOM and JavaScript resources are available  
});
```

After jQuery library has been initialized, its functions can be accessed through two different syntactical constructions that have identical results: **jQuery** and **\$**. If there are no conflicts (ie. JavaScript code that redefines the **\$** function), the exact same functions are called through any of the two mentioned constructions. For example, the code above can be rewritten as:

```
jQuery( document ).ready( function() {  
    // code here will be executed when all DOM and JavaScript resources are available  
});
```

Using any of the **jQuery** or **\$** symbols in any other JavaScript files (or directly in HTML, in `<script>` sections) is accepted by the browser only if the reference to the jQuery library precedes these occurrences in HTML. More details can be found at: <http://learn.jquery.com/using-jquery-core/document-ready/>.

4.2.3 Manipulate HTML tags using jQuery

Identical with plain JavaScript code, the first step in interacting with HTML tags is getting a reference to the desired element's representation in the DOM. In most cases, jQuery provides to developers a broader approach than native JavaScript in searching for HTML tags, based on any valid CSS selector.

However, in all situations where jQuery selectors do not have a direct correspondence in native JavaScript calls, the search of the elements is implemented through successive DOM queries which can get time and resources consuming in the case of complex HTML documents.

In the following examples, the use of **\$** sign for the variables names is just a visual queue to indicate that the returned references are actually jQuery objects (more details in the following section) and not native DOM elements. Selectors examples:

```
// select element with a specific ID  
var $element = $( "#myId" ); // Note: IDs must be unique per HTML document.  
  
// select all elements that on execution time have a specific class attached  
var $elements = $( ".myClass" );  
  
// select all elements of a specific HTML type  
var $elements = $( "input" );  
  
// select all elements of a specific HTML type and attribute value  
var $elements = $( "input[name='first_name']" );  
  
// select all LI elements of the unordered lists with class "people"  
// that have in parents list the element with ID "contents"  
var $elements = $( "#contents ul.people li" );
```

More examples of available selectors, including jQuery specific ones, are available at: <http://learn.jquery.com/using-jquery-core/selecting-elements/> .

4.2.4 jQuery objects

The result of each of the examples above is a jQuery object that encapsulate all the DOM elements that match the selector used. Even if there is no match found, the result is not **null** but a jQuery object that has no references encapsulated. Consequently, in order to verify if a selector has returned any elements we cannot use:

```
if ( $elements == null ) { }
```

as this expression will always be **false** (the selectors are not returning **null**). The correct approach is to verify the **length** of the returned jQuery object:

```
if ( $elements.length > 0 ) { }
```

There are situations in which we need a direct reference to the DOM element and not the jQuery encapsulation. For these cases the function **[jQueryObject].get(index)** defined in jQuery can be used:

```
var jQuery_element = $( "#myElement" );
var DOM_element = document.getElementById( "myElement" );

// this will print "true"
console.log( DOM_element == jQuery_element.get( 0 ) );
```

We can thus conclude that jQuery objects are maintaining a reference to the DOM elements that have been matched by a specific selector. On these encapsulations we can apply all the functions defined in the jQuery library (more details at: <http://learn.jquery.com/using-jquery-core/jquery-object/>).

4.2.5 Manipulate HTML elements

One of the important things to remember when working with jQuery library is that a jQuery object hold the references to **zero**, **one** or **several** DOM elements. All the functions defined in jQuery that are called on the object will be applied to all the referenced DOM entities. For example, in order to add class "customized" to all the paragraphs from an HTML document, we will simply use the following line:

```
$( "p" ).addClass( "customized" );
```

If used correctly, this can be very powerful and will allow complex implementations with few lines of code. However, if used incorrectly, this can have undesired side-effects:

```
$( "p" ).html( "Content that is meant for a single paragraph" );
```

will **change the content of each paragraph from the page** to "Content that is meant for a single paragraph".

In order to complete the exercises proposed for this laboratory work you will need the following functions from the jQuery library:

.html() <https://api.jquery.com/html/>

returns the content of the DOM element as HTML

.html(htmlString)

<https://api.jquery.com/html/>

sets the content of the DOM element to **htmlString**

! this function will parse the htmlString and will automatically add to the DOM any tag found

! **this function is vulnerable to JavaScript code injection** (through <script> elements)

.text()

<https://api.jquery.com/text/>

returns the content of the DOM element as plain text

.text (plainText)

<https://api.jquery.com/text/>

sets the content of the DOM element to **plainText**

! if plainText contains valid HTML code it will be ignored and no DOM elements are created - the text will be displayed exactly as provided

.val()

<https://api.jquery.com/val/>

returns the content of an INPUT element

.val(value)

<https://api.jquery.com/val/>

sets the content of an INPUT element

.attr(attrName)

<http://api.jquery.com/attr/>

returns the value of the **attrName** attribute of the element

.attr(attrName, attrVal)

<http://api.jquery.com/attr/>

sets the value of the **attrName** attribute of the element

.addClass (className/s)

<https://api.jquery.com/addClass/>

applies to the element the CSS class (or CSS classes) from **className/s** (the parameter can be a list of classes separated through spaces)

.removeClass (className/s)

<https://api.jquery.com/removeClass/>

removes from the list of applied CSS classes the class (or classes) specified in **className/s** (the parameter can be a list of classes separated through spaces)

! if this function is called without parameters it will remove all the CSS classes applied on the element

.hide()

<https://api.jquery.com/hide/>

instantaneous hiding of a visible HTML element by setting **width** and **height** properties to 0px.

i. this function has multiple signatures enabling also an animated transition between visible and invisible states

.show()

<https://api.jquery.com/show/>

instantaneous display of an HTML element previously hidden using *.hide()*, by setting **width** and **height** properties to their initial values

i. this function has multiple signatures enabling also an animated transition between invisible and visible states

.on(eventString, handlerFunc)

<https://api.jquery.com/on/>

allows registration of **handlerFunc** to be called when the **eventString** event is raised; the handler function receives the **event** parameter.

.animate(prop, delta,
onCompletion)

<https://api.jquery.com/animate/>

animates a visible DOM element by continuously updating the properties mentioned in **prop** over the **delta** time interval, until the desired values are set

- **prop** – JavaScript object that specifies the visual properties that should be animated and the desired final values; for example, in order to animate the size of an object (width and height) from *current_values* to *250 px 300 px*:

```
prop = {  
    height: 300px,  
    width: 250px  
}
```

- **delta** – duration of animation in milliseconds
- **onCompletion** – function called when the animation is complete

.delay (delta)	https://api.jquery.com/delay/ allows the delay in execution launching of functions applied to the same HTML element in the same processing queue, for delta milliseconds <ul style="list-style-type: none"> • this is different in outcome compared to the native JavaScript function setTimeout()
.queue (queuedFunction)	https://api.jquery.com/queue/ add a new function to the current processing queue of an HTML element <p style="text-align: center; color: red; margin-top: 10px;">! when queuedFunction has finished execution, .dequeue() must be invoked or the processing queue will stall</p>
.dequeue()	https://api.jquery.com/dequeue/ launches the following function in the execution queue

4.2.6 DOM traversal

When we have a reference to a DOM element, encapsulated in a jQuery object, we can use it as a starting point for a DOM traversal in one of the following three directions: (more details and the complete list of available functions can be found on: <http://learn.jquery.com/using-jquery-core/traversing/>):

- (1) towards the root of the DOM (searching through the ancestors of the current element), using one of the following functions:
 - **.parent([selector])** – <https://api.jquery.com/parent/>
returns the parent element if it matches the **selector**
 - **.parents([selector])** – <https://api.jquery.com/parents/>
returns all the ancestor nodes (between the current element and the root of the DOM), that match the **selector**.
- (2) on the same level (HTML elements having the same parent element), using one of the following functions:
 - **.prev([selector])** – <https://api.jquery.com/prev/>
returns the previous sibling, if it matches the **selector**
 - **.prevAll([selector])** – <https://api.jquery.com/prevAll/>
returns all the previous siblings that are a match for the **selector**
 - **.next([selector])** – <https://api.jquery.com/next/>
returns the next sibling, if it matches the **selector**
 - **.nextAll([selector])** – <https://api.jquery.com/nextAll/>
returns all the next siblings that are a match for the **selector**
 - **.siblings([selector])** – <https://api.jquery.com/siblings/>
returns all the HTML elements that have the same parent (siblings) and are matching the **selector**

(3) towards the leaves of the DOM (searching through the child nodes), using one of the following functions:

- **.children([selector])** – <https://api.jquery.com/children/>
returns all the HTML elements that are direct children of the current element and match the **selector**
- **.find([selector])** – <https://api.jquery.com/find/>
searches the DOM through all the children of the current node and returns the elements that match the **selector**

4.2.7 DOM manipulation

In the Web 2.0 interactive applications it is often necessary to create, add, move or remove HTML elements on the fly, as a response to user's actions or according to data display requirements. jQuery provides a set of functions for easy HTML nodes CRUD operations:

`$el.insertAfter($target)` <http://api.jquery.com/insertAfter/>
insert all the HTML elements from **\$el** after each element from **\$target**

`$el.after($content)` <http://api.jquery.com/after/>
insert all the HTML elements from **\$content** after each element from **\$el**

`$el.insertBefore($target)` <http://api.jquery.com/insertBefore/>
insert all the HTML elements from **\$el** before each element from **\$target**

`$el.before($content)` <http://api.jquery.com/before/>
insert all the HTML elements from **\$content** before each element from **\$el**

`$el.appendTo($target)` <http://api.jquery.com/appendTo/>
insert all the HTML elements from **\$el** as the last child elements of each DOM element from **\$target**

`$el.append($content)` <http://api.jquery.com/append/>
insert all the HTML elements from **\$content** as the last child elements of each DOM element from **\$el**

`$el.prependTo()` <http://api.jquery.com/prependTo/>

insert all the HTML elements from **\$el** as the first child elements of each DOM element from **\$target**

`$el.prepend()`

<http://api.jquery.com/prepend/>

insert all the HTML elements from **\$content** as the first child elements of each DOM element from **\$el**

4.2.8 Chaining function calls

One of the most useful particularities of the jQuery library is that almost all the functions return a reference to the jQuery object they have been invoked on. This enables **function calls chaining**, which allows developers to do a lot with very little code and to optimize resources (single DOM search for multiple updates and no persistent references). For example, the following changes:


- remove class **warning**
- add class **success**
- update displayed message
- make sure the HTML element is visible

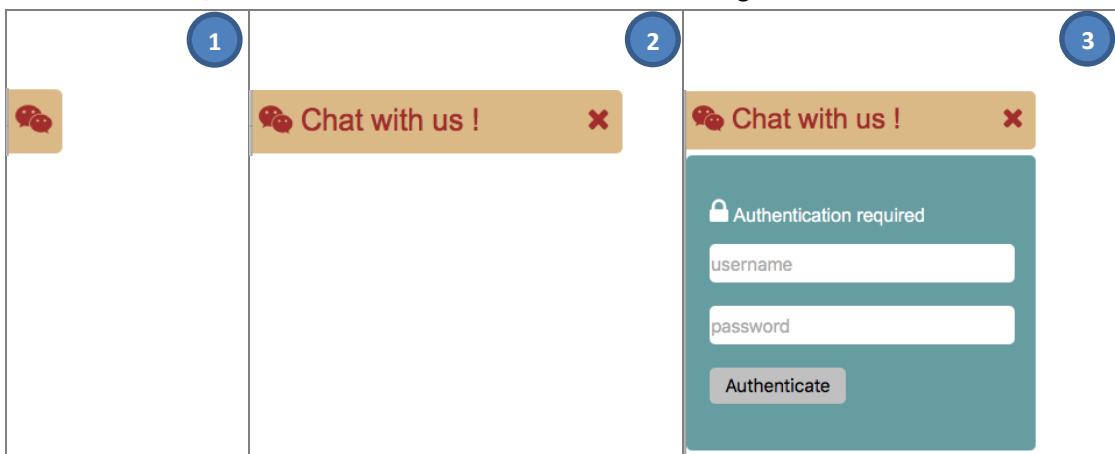
can be expressed through:

```
$("#myElement").removeClass("warning").addClass("success")  
                .text("Operation completed successfully").show();
```

4.3 Exercises

Exercise 1. Use jQuery library to create an HTML, CSS and JavaScript based component that has chat functionality and the following properties:

- when the page is loaded
 - the Chat component is hidden, only the following icon is visible: 
 - the icon has a fixed position on the left of the page, at 150px from the top
 - the size of the icon is 40 x 40 px
- when the icon is clicked, the component is displayed through a two stage animation, as illustrated below; each animation will be 500 milliseconds long:



- final display size of Chat component (step no. 3 above) is: 300 x 300 px
- when the X button is pressed, the component should be resized back to icon view through reverse animations

Exercise 2. Continue the development of the Chat component from Exercise 1 by adding the following Authentication functionality:

- when the **Authenticate** button is pressed
 - all the inputs from the form become disabled
 - a looping animation and the "Processing ..." message are displayed for 3 seconds
 - if authentication is successful, display a success message in a green color
 - if authentication is not successful (see credentials below) display an error message in red color
- the authentication result messages should be displayed above the input fields of the form
- the only accepted credentials are: admin / admin1234

LABORATORY 5 – AJAX CALLS

5.1 Introduction

Before Web 2.0, updating the information displayed in HTML pages was possible only through synchronous calls to the server. Even if the updates represented a very small part of the data, it was necessary to reload everything in order to present fresh information to the user.

Through the use of AJAX (Asynchronous JavaScript and XML) technology, which has been intensively developed when Web 2.0 concepts and increased interactivity in web applications became popular, we have the possibility to connect to the server in the background, retrieving new data without entire page reloading. AJAX is in fact a set of technologies that are working together:

- **HTML and CSS** for information display
- **DOM** for dynamic updates of the content (modify display settings of existing elements, create or remove new HTML elements on the fly)
- **JSON or XML** to encode the data exchanged between the client and the server
- **XMLHttpRequest (XHR)** – native API provided by each browser to support background asynchronous calls to the server
- **JavaScript** – the programming language that links all the above mentioned components

5.1.1 Laboratory objectives

Explain *synchronous* and *asynchronous calls* to a Web Server. Exercise these calls to an existing API using AJAX technology.

5.2 AJAX calls using jquery library

The jQuery library provides high level functions that hide the native API XMLHttpRequest provided by the browser. This approach insures cross-browser compatibility, a more succinct syntax and a better integration with other jQuery functions.

5.2.1 Initiate server calls

In order to initiate an AJAX call using jQuery functions we can use the **\$.ajax** function (<http://api.jquery.com/jquery.ajax/>):

```
$.ajax({
  url: "web_address",                // https://www.example.com/api.php
  method: "http_method",           // POST, GET ...
  contentType: "sent_data_encoding", // "json", "text"
  data: "data_to_be_sent",
  dataType: "response_data_encoding", // "json", "text"
  beforeSend: before_send_callback,
  success: success_callback,
  error: error_callback,
  complete: call_complete_callback,
  statusCode: status_codes_callbacks
})
```

In the example above the parameter meaning is the following:

web_address

The URL from which the data will be loaded. It can be provided in absolute or relative path:

- absolute: https://exemplu.com/read_news/
- relative to the calling HTML page: `read_news`
- relative to the current domain: `/read_news`

`http_method` Indicates the HTTP method [1] used to exchange information between the server and the client. In our examples we will use: **GET, POST, PUT, DELETE, OPTIONS**.

`sent_data_encoding` Indicates the encoding [2] used for the data sent by the client. In our example we will use the encoding **application/json**.

`data_to_be_sent` Represents the information that will be sent to the server when the call is performed. The data needs to be provided in the encoding indicated by the parameter **sent_data_encoding**. For example, in order to prepare a JavaScript object to be sent as a JSON encoded entity we will use:

```
JSON.stringify( JavaScript_object )
```

`response_data_encoding` Indicates the data encoding [2] expected by the client and that should be used by the server when responding to the request. For JSON encodings jQuery automates response processing by transforming the data in JavaScript objects on the fly. Because of this, jQuery functions are using non-standard [2] values for this parameter. In our example, the value should be **json**.

`before_send_callback` Allows the developer to connect a callback function to the `beforeSend` event of an AJAX call. Function signature is:

```
function ( requestReference ) { }
```

This approach is mainly used to set custom headers for the request. For example, adding a security TOKEN according to oAuth standard we should use a function similar to:

```
function ( xhr ) {
    xhr.setRequestHeader( 'Authorization', 'Bearer ' + token);
}
```

`success_callback` As the AJAX calls are asynchronous, program execution does not wait for the answer from the server. When the answer arrives, a specific event is fired by the browser according to the result of the call: **success** or **error**. In our implementation we will register a callback functions for the **success** event using the following signature:

```
function ( data_from_server ) { }
```

This function is invoked when the AJAX call has been completed successfully and data received from server is available. In our example, **data_from_server** could be:

- unprocessed answer from the server in **text** format
- a JavaScript object with the information received from the server, if the **response_data_encoding** value is set to a jQuery recognized value, like **json**.

call_complete_callback

As the AJAX calls are asynchronous, program execution does not wait for the answer from the server. When the answer arrives, a specific event is fired by the browser according to the result of the call: **success** or **error**. In our implementation we will register a callback functions for the **error** event using the following signature:

```
function ( requestReference ) { }
```

This function is invoked when there has been an error registered in the AJAX call (communication, server processing, data encoding/decoding etc.). The parameter is populated with the native JavaScript object that holds a reference to the call and allows access to:

- the error code received and its standardized message
- any content sent by the server with details on the processing error

Example of error callback function:

```
function ( xhr ) {  
    // default error message (could be the standard one)  
    var message = "Generic error message."  
    if( xhr.responseText ) {  
        // details received from server - use these  
        // response expected as json  
        message = $.parseJSON(xhr.responseText).message;  
    }  
    displayError( message );  
}
```

call_complete_callback

As the AJAX calls are asynchronous, program execution does not wait for the answer from the server. When the answer arrives, a **success** or **error** event is fired, according to the result of the call. Immediately after, a second event (**complete**) is fired (for all cases: success or error) indicating the finish of the asynchronous call. For this event we can register a callback function with the following signature:

```
function ( ) { }
```

This function will be called after the success or error callbacks.

status_codes_callbacks

When dedicated actions are necessary on the client according to the response code, dedicated callback functions can be registered for each

code. This represents a convenient method to implement a refined processing for specific **success** and **error** codes. For example:

```
{
  200: function () {
    displayMessage("Authentication successful");
  },
  403: function () {
    displayMessage("Invalid username or password");
  }
}
```

These code-specific callback functions are invoked after **error_callback** and **success_callback**.

5.2.2 Call answer management

As mentioned before, client-server communication through AJAX calls is asynchronous, all the callback functions registered for the **success**, **error** and **complete** events being invoked at some time (few milliseconds or even seconds) after the call has been initiated.

CAUTION!! When multiple asynchronous calls are fired immediately one after the other (perceived as "at the same time") the order of receiving responses is not necessarily the same with the calls initiate order.

When the callback for the **success** event is called, data received from the server in **text** format is automatically processed by jQuery and transformed according to the value set for the **datatype** parameter. For example, if expected data is in **json** format, inside the success function we can treat the answer as a regular JavaScript object:

Data example in **json** format:

```
[{
  "name" : "Ann Smith",
  "institution": "Boston University",
  "email": "ann@bouniv.edu"
},
{
  "name" : "John White",
  "institution": "MIT",
  "email": "jwhite@mit.com"
}]
```

Matching **success** function to process the response:

```
function ( data ) {
  // it will display: Ann Smith
  console.log( data[0].name );

  // it will display: jwhite@mit.com
  console.log( data[1].email );
}
```

5.2.3 Display received information in HTML

In order to display in HTML the information received through an AJAX call it is necessary to dynamically modify the DOM structure. For example, assuming that the HTML element below already exists in the page when the AJAX call is performed:

```
<div id="authors-list"> </div>
```

we can update its content with the example data from subtitle 2.2 in two different ways:

5.2.3.1 Making use of .html() function

Requires as a first step to create a **string** that contains a valid HTML description of all the HTML elements, formatting information and data that should be displayed. When applying this string as a content of an already existing DOM element, the browser will interpret the string and will automatically create all the HTML elements described in it, adding them to the DOM and showing them on the current page.

CAUTION!! This approach on data processing is vulnerable to JavaScript and HTML injection attacks, as the browser will execute any code found in the string parameter between any <script> and </script> tags.

```
function ( data ) {
    var htmlString = "";
    $( data ).each ( function () {
        htmlString += '<div><h3 class="author_name">' + this.name + '</h3>';
        htmlString += '<div class="author_affiliation">' + this.institution + '</div>';
        htmlString += '<span>' + this.email + '</span></div>';
    });
    $("#lista-autori").html( htmlString );
}
```

5.2.3.2 Making use of the DOM management capabilities

In this approach, all the required HTML elements are explicitly created through code (not implicitly by the browser from a string). The content for each new element is set through the **.text()** function, which automatically escapes all special characters, does not interpret any HTML code from the **string** parameter and thus prevents any injection attacks.

```
function ( data ) {
    $( data ).each ( function () {
        $("<div>")
            .append( $("<h3>").addClass("author_name").text( this.name ) )
            .append( $("<div>").addClass("author_affiliation").text( this.institution ) );
        .append( $("<span>").text( this.email ) );
        .appendTo("#authors-list")
    });
}
```

As can be seen in the example above, calls like: **\$("tag_html")** automatically create an HTML element of **tag_html** type (eg. div, p, td ...) which can be accessed and managed through ordinary DOM functions and then inserted into the current page's DOM.

5.3 Development considerations

5.3.1 Server communication

When calling a remotely located API from local files you will notice a supplemental call towards the server, launched automatically by the browser just before the one you initiated. This is necessary because the communication is considered to be cross-domain (from **local domain** to **server domain**) and the browser verifies on the server what cross-domain data transfer policy is in place. When connecting to ChatAPI [3], this behavior and the limit of maximum number of calls accepted from the same IP address in a one second interval can lead to errors that should be separately diagnosed and solved.

It is recommended to allow enough time between subsequent calls to the ChatAPI (for example read messages each 3 seconds) and to prevent launching parallel asynchronous calls (by waiting for the response, waiting for 3 seconds and then launching a new call).

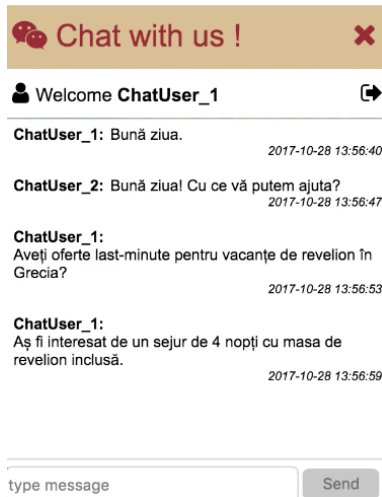
5.3.2 Dedicated processing functions

Next to the jQuery and JavaScript functions already presented in the previous laboratories, for the practical assignment today the following functions will also be necessary:

Library	Function	Description
JavaScript	JSON.parse (string)	Converts string from a valid json encoding to native JavaScript objects [4]
JavaScript	JSON.stringify (Javascript_object)	Converts Javascript_object from native format in valid json description (text format) [5]
JavaScript	setTimeout (function_ref, milisec)	Invokes function_ref after milisec have passed since setTimeout has been called[6]
JavaScript	new Date()	Creates a new Date [7] object that provides dedicated methods for the management of date-related information. As this is a constructor, it needs to be invoked with the new operator.
jQuery	\$.parseJSON (string)	Converts string from a valid json encoding in text format to a native object in JavaScript. This function encapsulates the native JavaScript function <i>JSON.parse (string)</i> [4]
jQuery	\$("html_tag")	Creates and returns a jQuery object that encapsulates a newly created DOM element of html_tag type (<i>details in section 3.3</i>)

5.4 Exercises

Exercise 1: Use the jQuery library to connect to the <https://cgisdev.utcluj.ro/moodle/chat-piu/> API [1]. Create a new account using <https://cgisdev.utcluj.ro/moodle/chat-piu/register> and use the chat interface developed in the previous laboratories to interact with the server. Implement the Authentication step using the newly created account and adjust the display of success and error messages accordingly.



Exercise 2: Continue the development of the chat component implemented in the previous laboratories by connecting to the <https://cgisdev.utcluj.ro/moodle/chat-piu/> API [1]. Update the existing UI with elements similar to the image on the left and connect them to the API using jQuery AJAX calls.

When the implementation is complete, open two different browser windows, load the chat on both of them and test the communication by sending messages from one to the other.

5.5 References

- [1] Complete list of HTTP methods: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.ht>
- [2] Complete list of content types: <https://www.iana.org/assignments/media-types/media-types.xhtml>
- [3] Teodor Ștefănuț, ChatAPI Technical Specification, [Online] https://cgisdev.utcluj.ro/moodle/chat-piu/ChatAPI_specification.pdf (last access 10.02.2019)
- [4] **JSON.parse ()** function https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse
- [5] **JSON.stringify ()** function https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify
- [6] **setTimeout ()** function <https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setTimeout>
- [7] **Date()** object https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

USER INTERFACE DEVELOPMENT IN ANDROID

LABORATORY 6 – INTRODUCTION IN ANDROID DEVELOPMENT

6.1 Introduction

The process of developing Android applications involves the description of the user interface using an XML based language that is specific to the Android platform, while the application's logic is described in Java. The main tools used in Android applications development are:

- Android SDK – for our laboratory applications we will use version: **API 26: Android 8.0 (Oreo)**
- [Android Studio](#) – the official IDE for the development of Android applications

6.1.1 Laboratory objectives

Introduction into the basic notions required for the development of an Android application, applied in the creation of a basic, very simple app.

6.2 Theoretical considerations

One of the most important aspects of the devices that use Android operating system is represented by their **diversity**, which manifests as

- hardware configuration (processor, memory, storage capacity, available sensors, etc.)
- available display resolutions
- hardware buttons number and functionality

A successful application must adapt as much as possible to most of the configurations available and provide the same functionality through the best user experience available.

6.2.1 Hardware configurations

Android operating system is currently installed on a variety of devices: TVs, mobile phones, tablets, smart watches, etc. A complete list of devices that are compatible with Google Play, updated frequently, can be viewed at <https://support.google.com/googleplay/answer/1727131?hl=en-GB>

6.2.2 Basic notions for the development of Android applications adaptive to multiple resolutions

- **screen size** – represents the physical dimension of the screen, measured on the diagonal and usually expressed in *inch*. For simplicity, android screens can be grouped in *small*, *normal*, *large* and *extra-large*.
- **pixel density** – indicates the number of physical pixels that are included in a surface unit. It is usually expressed as *dots-per-inch (dpi)*. For simplicity, Android groups pixel densities in *low*, *normal* and *high*.
- **orientation** – represents the orientation of the device relative the user
- **display resolution** – indicates the total number of pixels available into the display
- **density-independent pixel** – defines a virtual pixel used in the design process of the user interface, in order to ensure the independence next to the physical attributes of different devices.
 - $1\text{ dp} = 1\text{ px}$ for a screen with density of 160 dpi
 - $\text{px} = \text{dp} * (\text{dpi} / 160)$

6.2.2.1 Tipuri de afișaje suportate de Android

In order to simplify applications development, Android makes use of the following [generalized densities](#):

- ldpi (low) ~120dpi
- mdpi (medium) ~160dpi
- hdpi (high) ~240dpi
- xhdpi (extra-high) ~320dpi
- xxhdpi (extra-extra-high) ~480dpi
- xxxhdpi (extra-extra-extra-high) ~640dpi

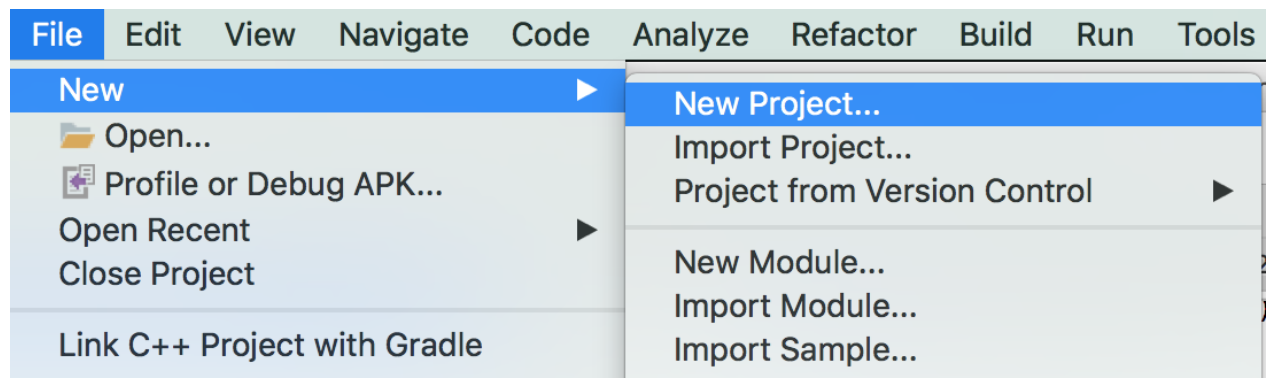
6.3 Development considerations

As mentioned before, in our laboratory activities we will use [Android Studio](#), the official IDE for creating Android applications.

6.3.1 Create a new Android application

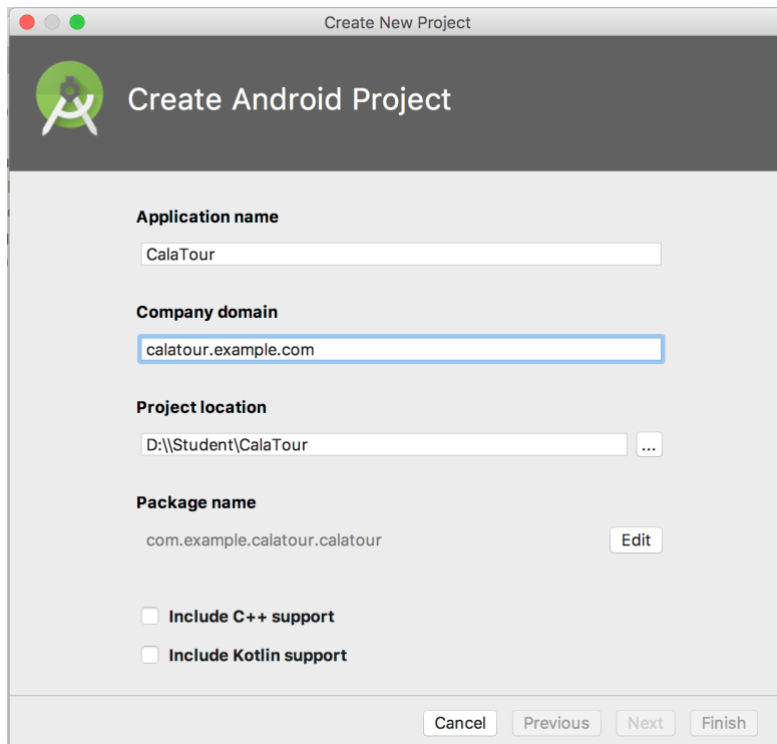
In order to create a new application please follow the steps described below:

6.3.1.1 Create a new project



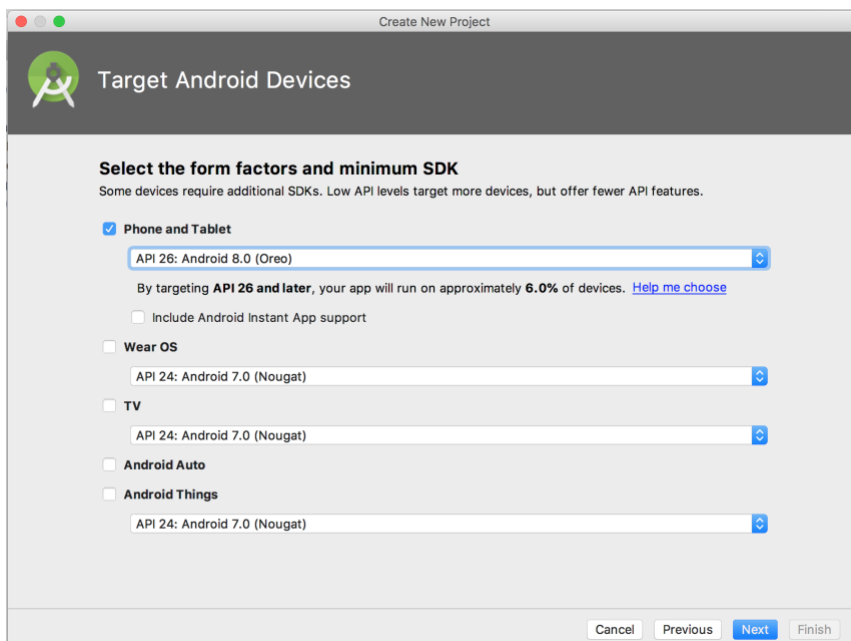
6.3.1.2 Define the directory where the APP sources should be stored and choose a name for the software package that will contain your Java classes

- **Application Name** – the title of the APP as it will be shown on the phone
- **Company Domain** – used to automatically create a unique name for the package that will group the APP's dedicated Java classes
- **Package Name** – similar to other Java applications, developed classes are organized in packages, that should have a unique name for the APP and organization. Initial value is automatically created based on the Company Domain and Application Name, but it can be further customized.



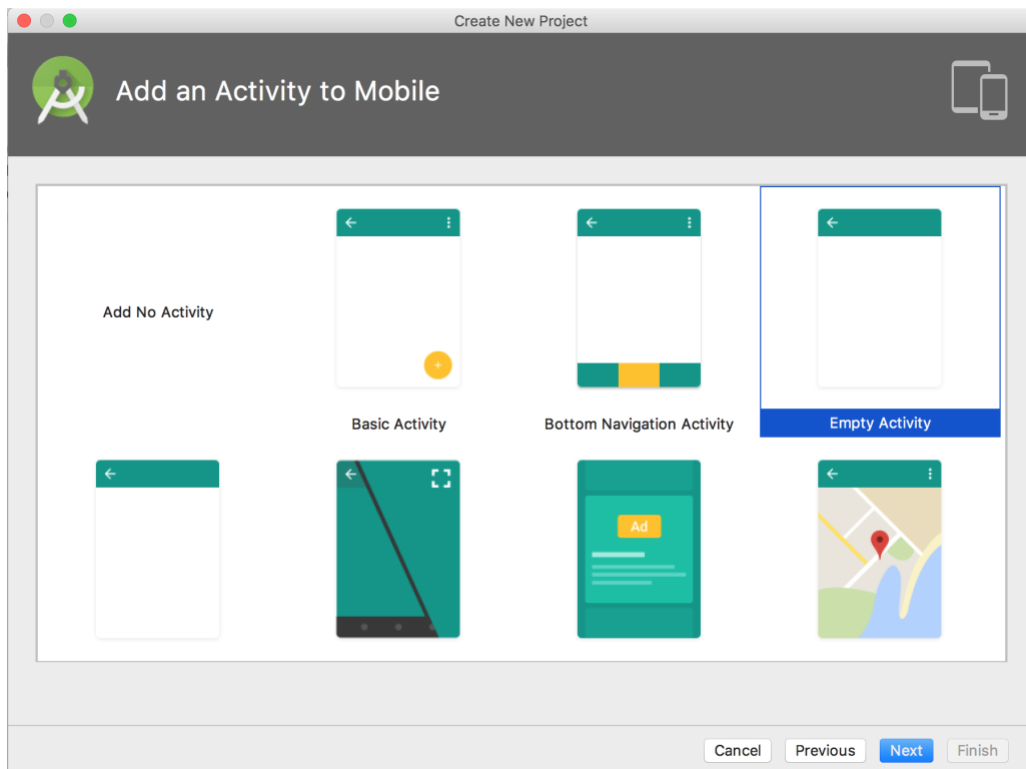
6.3.1.3 Add information about targeted devices of your APP

- **Phone and Tablet** – we will develop our app only for these devices. **Please make sure that all the other options have been unchecked.**
- **Minimum Required SDK** – indicates the minimum Android version required for the APP. When selecting this option you need to consider that the available native functions for APP development are the ones on the minimum SDK selected.

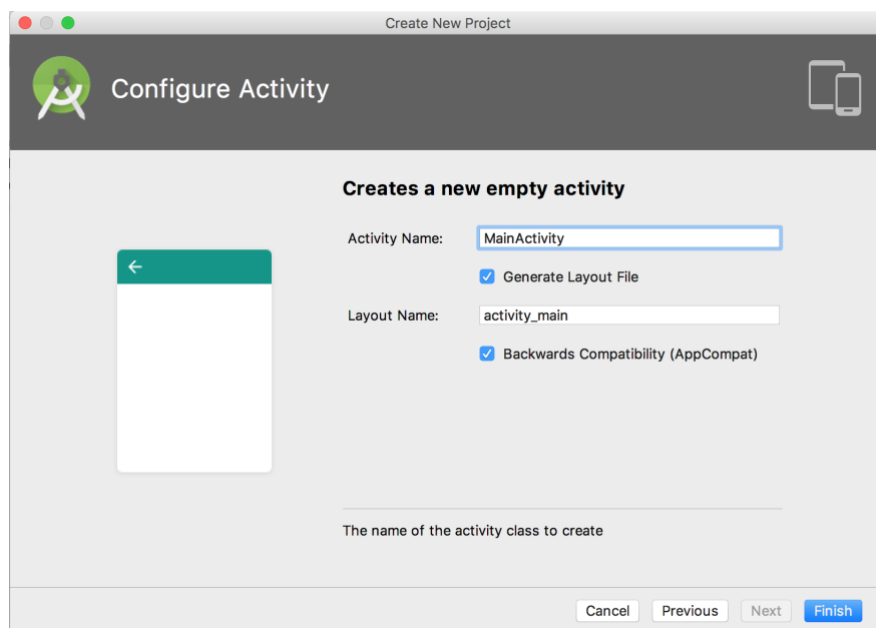


6.3.1.4 Indicate the initial activity type that should be automatically created by the tutorial

- In our example application we will use: **Empty Activity**



6.3.1.5 Provide a name for the first Activity and the connected layout file



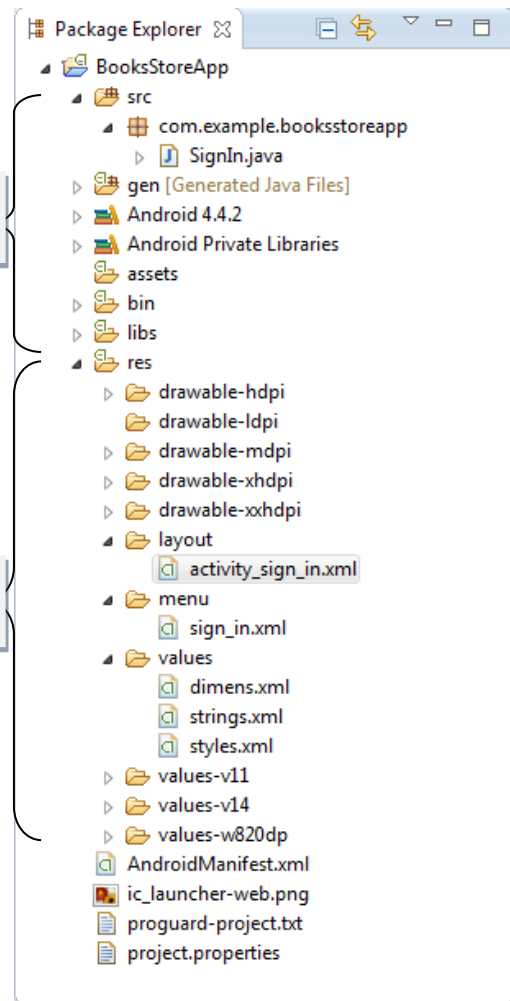
6.3.2 The generic structure of an Android application

As you can see in the image on the right, section **A** of the files and folders tree that describe the Android application is very similar to other Java applications, including:

- the **src** folder where the source code files are grouped into packages
- references to external libraries (in our example Android 4.4.2 and others)
- the **bin** folder that holds the executable version of the project

Section **B** contains a set of folders that are specific to Android applications, named in predefined patterns (that should not be modified) and having the following meaning:

- **res** – root folder of this; here, different other [types of resources](#) can be also included.
- **drawable-****** - folders that include [visual elements](#) like images, visual settings, etc. Using the predefined folder structure, different versions of the same graphic, optimized for different resolutions can be included
- **layout** – folder that includes the files that describes the user interface for the Android application, using the dedicated XML based language
- **menu** – folder that includes all the XML files that describe the structure of a menu into the app
- **values-v****** - folders that group XML files in which different static elements of the application are defined: sizes, dimensions, styles, text constants, numeric constants, etc. The way the application is using this data is the following: at each launch will use the values included in the folder that has the “v” number smaller and closest to the running Android version (on which the app is currently running). In other words:
 - **values-v14** – is the first folder searched by the app when running on Android with API ≥ 14 . If the searched values are not found here, they will be further looked in the rest of the directories, applying the same rule
 - **values-v11** – is the first folder searched by the Android versions with API < 14
 - **values** – last of the searched folders, only if the searched values have not been found in any of the previously searched folders
- **values-w820dp** – folder dedicated to store the static values for the devices that have the with display resolution $\geq 820dp$



- The file **AndroidManifest.xml** includes all the elements that describe the application and are necessary to the device in order to install it, check compatibility and grant the required access rights. Details about the information that can be included here and about the data structures can be found on: <http://developer.android.com/guide/topics/manifest/manifest-intro.html>

6.3.3 Describing the user interface

The description of the user interface is performed in the XML files placed into the folder **/src/layout/** and usually includes different visual elements stored into the **drawable** folders, menus described in **menu** folder or constant values and styles defined in **values** folders. The visual layout of all these graphic elements is described through [invisible containers](#), that define different implicit rules to layout the included visual components. According to the container type, different display attributes can also be globally attached to all the child elements using XML language (eg: `layout_alignLeft`, `layout_height`).

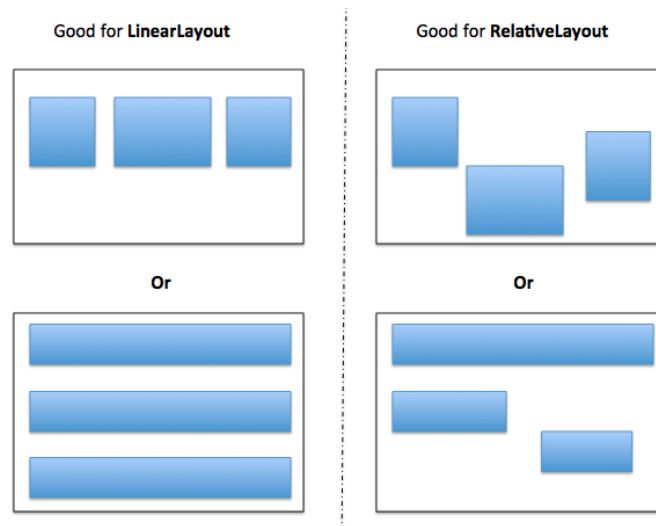
In our example application we will use mainly two types of containers:

6.3.3.1 Linear Layout

All the included visual elements are placed one after the other, either vertically or horizontally, without overlaps. A list of the specific XML attributes that can be attached to child components is available at: <http://developer.android.com/guide/topics/ui/layout/linear.html>

6.3.3.2 Relative Layout

Allows the placing of the visual child elements through relative positioning according to their neighbors. A list of the specific XML attributes that can be attached to child components is available at: <http://developer.android.com/guide/topics/ui/layout/relative.html>



6.4 Java aspects specific to Android applications

In order to create the necessary link between the data model and data display into an Android application, it is necessary to write Java code. One of the most important aspects to note is the fact that the **/res/** folder and its entire structure is automatically mapped by the Android SDK into a Java object structure named **R**, under the form of **int** resource types. Different elements are organized in categories, as follows:

- each of the IDs defined inside the XML layout files using constructs like `android:id="@+id/elemID"` are accessible through `R.id. elemID`
- each of the graphical resources included in any of the **drawable** directories can be accessed through `R.drawable.file_name_without_extension`
- each of the interface files can be accessed through `R.layout. file_name_without_extension`
- etc.

In an Android application the R component is defined in two different packages. In order to access resources included into the SDK, one will use:

```
import android.R
```

In order to access resources added to the application, one will use:

```
import com.example.booksstoreapp.R;
```

6.4.1 Access visual elements

In order to access visual elements from Java code it is first necessary to parse the XML document describing the UI and transform it into a Java object representation. This is usually achieved inside the `onCreate` function, using the Java instruction:

```
setContentView ( R.layout.activity_sign_in );
```

After the instruction is successfully executed, any visual element from the parsed XML file can be access through a command similar with:

```
ElementType variable = (ElementType) findViewById ( R.id.TVErrorPassword );
```

6.4.2 Attach callbacks to user interactions

In order to attach a callback to an `onClick` event triggered on a specific visual element, we can define in the XML file the attribute `android:onClick` for that specific element:

```
<Button
    android:id="@+id/button1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="100dp"
    android:minWidth="300dip"
    android:onClick="SignInClick"
    android:text="Sign in" />
```

In the Java classes that will extend this layout, it is necessary to define the `SignInClick` function with the following signature:

```
public void SignInClick (View view )
{
    // function body
}
```

6.5 Exercises

Exercise 1: Create a new Android project and design an authentication screen with the following elements:

- screen title – font: 20px, color: #0000FF, horizontally centered
- two input fields
 - username – provided in email format
 - password – with hidden typing
- labels for the input fields
 - displayed above each corresponding field
 - visual format – font: 14px, color: #FFFFFF, aligned to the left
- error messages for the input data
 - displayed under each field
 - visual format – font: 14px, color: #FF0000, aligned to the left
- a button
 - title: Authenticate
 - visual format – width: 400px, horizontally centered
- vertical spacing
 - 60px under the title
 - 40px just above the labels for the input fields
 - 20px over the Authenticate button

Exercise 2: When the Authenticate button is pressed:

- validate username
 - cannot be empty
 - should be a syntactically valid email address
- validate password
 - cannot be empty
 - should contain at least one upper case letter and a digit
 - should be at least 5 characters long
- if the input values fail one of the above requirements
 - under the specific field, display a custom error message for each requirement
- if the input values meet the above requirements
 - compare the values with
 - username: [user@uid.com](#)
 - password: Passw0rd
 - if the values are identical, display the message “Authentication successful” in #00FF00 color, placed under the Authenticate button
 - if the values do not match, display the message “Wrong username or password” in #FF0000 color, placed under the Authenticate button

LABORATORY 7 – ANDROID UI AND USER INTERACTION (1)

7.1 Introduction

As in most of mobile technologies, Android uses very frequently list-based controls for data display. In order to interact with elements displayed in a list, one can define a contextual menu, that is displayed by the app upon a long-press action from the user.

7.1.1 Laboratory objectives

Use and customization of lists in Android applications. Define and interact with contextual menus.

7.2 Theoretical considerations

7.2.1 Creating custom lists

In order to create a list with customized elements in Android, it is necessary to complete the following steps:

7.2.1.1 Describe the visual structure of a list element

The visual structure and appearance of each element of the list is described through an XML file that resides in the `res/layout/` folder.

7.2.1.2 Create a list adaptor

The link between the data model and the visual structure for each list element is created through a specialized class named `Adapter`. Each time the structure of the list is modified through addition or removal of new elements, the adapter needs to be updated and the list must be notified to refresh the visual display.

The development of a custom `Adapter` class requires the implementation of a specific interface, defined in the Android SDK as a virtual class. The most common approach for creating a custom `Adapter` is to inherit from `BaseAdapter` class (or any of its more specialized versions) and to implement the functions:

- | | |
|------------------|---|
| getCount | returns the number of elements in the Adapter's collection, which also represents the total number of elements that should be displayed by the list |
| getItem | return the element from the data model that can be found (is displayed) at a specific position into the list |
| getItemId | returns the ID of the element that is displayed at a specific position into the list |
| getView | creates and returns the visual representation of a list element, customized with the data from the model; if the list contains more elements that can fit onto the screen at once, one of this function's parameters will contain a reference towards already existing list elements that could be reused (have been displayed but now have become invisible – through list scroll), for a more efficient use of resources. |

In development it is common to use **ArrayAdapter<T>** instead of the **BaseAdapter** class, because its generic implementation provides list elements management and default code for *getCount*, *getItem*, *getItemId*. In most cases, only the *getView* method needs to be overwritten in order to connect the data model to the View elements included in the XML file.

7.2.2 Create a contextual menu and define user interactions with it

Creating a contextual menu, that is displayed upon long-press gesture on a list element, requires the completion of the following steps:

7.2.2.1 Describe menu elements

The elements of the contextual menu are described in a XML file that should reside in **res/menu/** directory. If the directory does not exist, create it from scratch. For each of the menu's elements you must add an **item** tag to the XML file:

```
<item
    android:id="@+id/menu_item_id"           // the unique id of the action
    android:title="@string/menu_item_title" // displayed text
/>
```

7.2.2.2 Register visual elements for the contextual menu

Visual elements that are responsive to the long-press gesture and can trigger the display of a contextual menu must be first registered using a call of the function **registerForContextMenu**.

7.2.2.3 Display and customization of the contextual menu

In the Activity's controller in which the contextual menu should be displayed, it is necessary to override the method [onCreateContextMenu](#), which is automatically called by the Android SDK when the contextual menu needs to be displayed. In this method, you can decide which contextual menu (if there are more than one available) should be displayed and which entries should be available, according to the specific attributes of the element on which the long-press gesture occurred. This is also the method responsible with loading the visual structure of the menu described into the XML file.

7.2.2.4 React when an item of the contextual menu has been selected

In order to capture and react to the selection of an item from the contextual menu, one must override the method [onContextItemSelected](#) in the controller.

7.3 Development considerations

In the following examples we will consider that the visual structures of the list elements and contextual menu have been described into the files **res/layout/activity_books_list.xml** and **res/menu/books_list_context.xml**.

7.3.1 Transition to another activity

The transition to another activity (a new screen) in Android can be implemented using the following instructions:

```
Intent intent = new Intent ( this, new_activity_class );
startActivity ( intent );
```

7.3.2 Creating an Adapter class

```
public class MyAdapter extends ArrayAdapter<T>
{
    public MyAdapter(Context context, List<T> objects)
    {
        // reference of the objects list is sent to the super class which
        // implements getCount, getItem and getItemId methods

        super(context, 0, objects);
    }
}
```

7.3.2.1 Implementing getView function of the Adapter class

```
public View getView ( int position, View convertView, ViewGroup parent )
{
    // get a reference to the LayoutInflater service
    LayoutInflater inflater =
        (LayoutInflater) context.getSystemService ( Context.LAYOUT_INFLATER_SERVICE );

    // check if we can reuse a previously defined cell which now is not visible anymore
    View myRow = (convertView == null) ?
        inflater.inflate ( R.layout.layout_name, parent, false ) : convertView;

    // get the visual elements and update them with the information from the model
    return myRow;
}
```

7.3.3 Connect the ListView element and the Adapter

Connecting the Adapter and the ListView that displays the information is usually performed into the onCreate method of the controller, using the following instructions:

```
MyAdapter myAdapter = new MyAdapter ( list_items );
listViewReference.setAdapter ( myAdapter);
```

7.3.3.1 Update list display when changes occur into the data model

When changes occur into the data model and the Adapter is updated, an automatic notification can be sent to the ListView component that will notify the need for a display update. The registration for the notification can be done using the instruction:

```
myAdapter.notifyDataSetChanged();
```

7.3.4 Implementing the contextual menu

In order to be able to trigger the display of the contextual menu, the visual elements must register for the event. Usually, the registration is performed in the onCreate method of the activity controller, using the instruction:

```
registerForContextMenu ( view_reference );
```

7.3.4.1 Customize the display of the contextual menu

When a request to display the contextual menu is received, the Android system will call the method **onCreateContextMenu** from the current activity controller. The following example demonstrates how the appearance of the contextual menu can be customized, according to the item from the list that has been selected. More detailed information about the list element for which the contextual menu

has been displayed can be accessed through the parameter of type [ContextMenuInfo](#), according to the following example.

```
public void onCreateContextMenu ( ContextMenu menu, View v, ContextMenuInfo menuInfo )
{
    super.onCreateContextMenu(menu, v, menuInfo);

    // check if the display of the contextual menu has been triggered by the list
    if ( v.getId() == R.id.id_lista_vizata )
    {
        // identify selected element from the list
        AdapterView.AdapterContextMenuInfo info =
            (AdapterView.AdapterContextMenuInfo) menuInfo;

        menu.setHeaderTitle ( String valoare_specifica_elementului );

        // load the visual structure of the contextual menu
        getMenuInflater().inflate ( R.menu.meniu_contextual, menu );
    }
}
```

7.3.4.2 Identify selected item from the contextual menu

```
public boolean onContextItemSelected ( MenuItem item )
{
    // access information attached to the contextual menu
    AdapterView.AdapterContextMenuInfo info =
        (AdapterView.AdapterContextMenuInfo) item.getContextMenuInfo();

    // identify the item from the menu that has been selected, using the predefined IDs
    if ( item.getItemId() == R.id.id_item_1 )
    {
    }
    else if ( item.getItemId() == R.id.id_item_2 )
    {
    }

    return super.onContextItemSelected ( item );
}
```

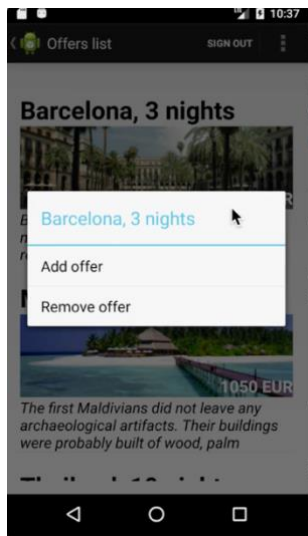
7.4 Exercises



Exercise 1. Extend the application from the previous laboratory with a new activity that displays a list of vacation offers, similar to the image on the left. The user can access this section of the APP only after a successful authentication on the first screen.

Implementation requirements:

- the information about each offer is stored in a class that plays the role of a Model in the application
- the data in the list is loaded dynamically at runtime (not added at design time)
- for the implementation you will use the ListView component



Exercise 2: When a long-press gesture is performed on any element on the list, a contextual menu will be displayed (see image on the left).

Implementation requirements:

- the contextual menu will display the title of the targeted vacation
- the menu will have two elements
 - Add offer
 - Remove offer

LABORATORY 8 – ANDROID UI AND USER INTERACTION (2)

8.1 Introduction

Android applications are using an options bar, placed on the top of the screen, in order to display general actions that can be performed on the current activity. This component must reflect at all times the options available to the user.

When time consuming activities are performed (download, processing, etc.) the application must inform the user on the progress, usually through a progress bar.

8.1.1 Laboratory objectives

Using and customizing the options bar, progress bar, toast, dialog window and Back button components.

8.2 Theoretical considerations

8.2.1 Options bar

Depending on the App theme, it could include a back button that allows the user to return to the previous activity (named *parent activity*) and a text that displays the current activity title. Both elements are usually aligned to the left of the screen.

In the available space of the options bar, different actions that are specific to the current activity can be displayed. These elements are defined into a menu resource type, and are connected to the activity through the override of **onCreateOptionsMenu** function, in the activity controller. This function is automatically called by the Android operating system when the activity is displayed. The different ways available for displaying the options into the options bar can be controlled by the developer directly from the XML file, through the parameter **app:showAsAction**.

8.2.2 Progress bar

When time or resource consuming activities are performed, it is recommended to inform the user on the progress using a progress bar. The status of the progress bar component can be updated periodically, either using a specific event provided by the performed processing (eg. download) or using the **CountDownTimer** component, that enables the insertion of UI updating instructions into the main execution thread.

8.2.3 BACK button

Most of today's Android devices provide a set of three physical buttons with predefined behaviors, used to interact with the applications. One of these buttons implements the „previous step“ functionality that allows the user to easily access a previous activity or even a previous app. However, there are situations in which it is recommended to alter the default behavior of this button in order to improve the app's behavior. The alteration can be achieved through the overridden of **onBackPressed** function in the controller of the activity.

8.2.4 Dialog windows

Dialog windows are used to ask for a confirmation from the user for actions that are destructive or might be triggered by mistake. In Android, a dialog window is usually displayed using the **AlertDialog.Builder** component.

8.2.5 TOASTs

In Android a TOAST represents a short message that is displayed to the user and automatically dismissed without user interaction. These messages are created using the **Toast** component and most of the time display informative alerts or confirmation for successfully completed actions. The message is visible on the screen for a predefined time interval and then automatically hidden by the Android operating system.

8.3 Development considerations

8.3.1 Defining a timer

```
progressTimer = new CountdownTimer ( tick_time, maximum_execution_time )
{
    @Override
    public void onTick ( long millisUntilFinished )
    {
        // code that will be executed repeatedly, after each tick_time
    }

    @Override
    public void onFinish()
    {
        // code that will be executed only once, after maximum_execution_time
    }
}.start();
```

8.3.2 Defining elements for the options bar

The list of options that will be included into the options bar will be defined in an XML file placed in the resources folder named **menu**. The usual template for describing an element is presented below:

```
<item
    android:id="@+id/id_action_1"           // the unique id of the action
    android:showAsAction="always"         // display settings
    android:title="Action 1"/>           // displayed text
```

8.3.3 Populate the options bar

```
public boolean onCreateOptionsMenu ( Menu menu )
{
    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate ( R.menu.fisier_xml_meniu, menu );
    return true;
}
```

8.3.4 Identify selected option

```
public boolean onOptionsItemSelected ( MenuItem item )
{
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so long
    // as you specify a parent activity in AndroidManifest.xml.
    int id = item.getItemId();
```

```

if ( id == R.id.id_action_1 )
{
    // code for Option1
    return true;
}
else if ( id == R.id.id_action_2 )
{
    // code for Option2
    return true;
} else if ...

return super.onOptionsItemSelected ( item );
}

```

8.3.5 Display a Toast message

```
Toast.makeText ( this, "text_to_be_displayed", display_time_milisec ).show();
```

8.3.6 Display a dialog window

In order to capture and respond to the events raised by the buttons of a dialog window it is necessary to implement the **OnClickListener** interface. If the component that will handle these events is the current activity, the interface must be added to the list of interfaces implement by the activity's controller.

```

AlertDialog.Builder myDialog = new AlertDialog.Builder ( this );
logoutConfirmation
    .setTitle ( "dialog_title" )
    .setMessage ( "dialog_message" )
    .setPositiveButton ( "button_title", reference_implement_interf_ OnClickListener )
    .setNegativeButton ( "button_title", reference_implement_interf_ OnClickListener )
    .show();

```

8.3.7 Customize the behavior of the Back button

When the physical Back button is pressed, the Android operating system will call the function **onBackPressed** on the current activity. Customizing the behavior can be achieved through the overridden of the function:

```

public void onBackPressed()
{
    // code executed when the Back button is pressed
}

```

In order to allow the implicit behavior to take place, we need to call also the function defined in the parent class: **super.onBackPressed()**.

8.3.8 Pass data between activities

In Android development there are two main approaches in dealing with data necessary for application runtime:

8.3.8.1 Defining a singleton class that acts as a data model for all the activities

Defining a singleton in Android is identical to defining a singleton in Java.

8.3.8.2 Passing the necessary data between activities

Passing data between activities could occur in two different situations:

- from the current activity to a newly created one (parent to child)
- from the current activity to the previous one (child to parent)

8.3.8.2.1 Passing data from the current activity to a newly created one (child activity)

When it is necessary to pass information from one activity to a newly created one, we can use the Intent to package the data:

```
Intent intent = new Intent ( this, new_activity_class );
intent.putExtra ( "key_name", "value" );
startActivity ( intent );
```

On one Intent we can add as many extra parameters as necessary, as long as they have unique keys.

In the new activity we read received information inside **onCreate** function by getting a reference to the Intent and extract the data based on the "key_name" values:

```
// gets the previously created intent
Intent myIntent = getIntent();

// will return the value of the "key_name" entry
String value = myIntent.getStringExtra ( "key_name" );
```

Depending on the type of data you should use specific functions for extracting values from intent: **getIntExtra**, **getDoubleExtra**, etc.

8.3.8.2.2 Passing data from the current activity to the parent activity

In the parent activity:

- use the function **startActivityForResult** to launch the new activity:

```
Intent intent = new Intent( this, new_activity_class );
intent.putExtra ( "key_name", "value" );
startActivityForResult ( intent, unique_int_identifier );
```

- override the function **onActivityResult** to process the response from the child activity:

```
protected void onActivityResult ( int requestCode, int resultCode, Intent data )
{
    if ( requestCode == unique_int_identifier )
    {
        if ( resultCode == Activity.RESULT_OK )
        {
            // extract the information from the Intent
            String result = data.getStringExtra ( "back_key_name_1" );
            ...
        }

        if ( resultCode == Activity.RESULT_CANCELED ) {
            // write your code if there's no result
        }
    }
}
```

In the child activity:

- when there is **data** to return to the parent:

```
Intent returnIntent = new Intent();
returnIntent.putExtra ( "back_key_name_1", "return_value_1" );
returnIntent.putExtra ( "back_key_name_2", "return_value_2" );
setResult ( Activity.RESULT_OK, returnIntent );
finish();
```

- when there is **no data** to return to the parent:

```
Intent returnIntent = new Intent();
setResult ( Activity.RESULT_CANCELED, returnIntent );
finish();
```

8.4 Exercises



Create a Details page that can be used to display detailed information about the offers. The page should include: the *offer title*, *full description*, *image*, *price* and the *number of visualizations* (how many times the Details page has been displayed for each offer).

In the options bar, inform the user if the offer is in his/her favorites list, by displaying one of the following options:

- ADD TO FAVORITES – if the offer is not yet in the favorites list
- REMOVE FROM FAVORITES – if the offer is already in the favorites list

When any of the above options is used, display a toast to confirm the action.

The details page should be displayed when an offer from the list is tapped.

LABORATORY 9 – ANDROID UI AND USER INTERACTION (3)

9.1 Introduction

The **RecyclerView** component is an alternative approach in displaying lists of elements, being more optimized for large lists than **ListView** component. The implementation and functionality principle is very similar in both approaches: (1) create the template of a single list element in an XML file and (2) create the adaptor class that maps the data model over the UI template. The advantages and disadvantages of using **RecyclerView** are discussed later in this laboratory.

9.1.1 Laboratory objectives

Create, display and customize a list of elements using the **RecyclerView** component and an adaptor of type **RecyclerView.Adapter**.

9.2 Theoretical considerations

RecyclerView is a component very similar with **ListView**, but requires a slightly different adapter to connect to the data model. The type of this particular adapter is: **RecyclerView.Adapter**. Consequently, in order to display some information from the model in a **RecyclerView** based list, we will need to extend and customize the **RecyclerView.Adapter**.

9.2.1 Reuse of UI templates (elements of type View)

Parsing XML template files and transforming them in Java objects is a costly process that requires a significant amount of processing power and memory, especially for large lists. In the approach of using an **ArrayAdapter** or **BaseAdapter** together with the **ListView** component, the reuse of already processed visual components that have been removed from the field of view is entirely developer's responsibility. When preparing the display of a new element from the data model, the programmer can choose to either recycle a currently unused visual template by updating the content or to create a new one from scratch, based on the XML file.

The recycling process in an **ArrayAdapter** or **BaseAdapter** class is implemented through the following line of code (more details in **Laboratory 7**):

```
View myRow = ( convertView == null )
    ? inflater.inflate ( R.layout.layout_name, parent, false )
    : convertView;
```

When creating lists based on **RecyclerView.Adapter** the recycling of visual elements on list scroll is automatically managed, without further intervention from the developer. This optimization is the main advantage that **RecyclerView** approach has over the **ListView** approach.

9.2.2 ViewHolder template

The **ViewHolder** template is a dedicated class that holds the references to all the visual elements from the parsed XML template of a list element. When the **View** is recycled, the costly search using **findViewById** of each visual item (TextView for title, ImageView for pictures etc.) is prevented by reusing the attached **ViewHolder** (which already has these references). In practice, the **ViewHolder**

also implements a dedicated method (usually named **bindViewHolder**) that maps the model data over the UI elements.

The **ViewHolder** template can be used also with the **ListView** component, but the integration must be explicitly implemented by the developer. As the required changes are not trivial (the **getView** method of the adapter needs to be significantly redesigned), this approach is rarely used and the entire **RecyclerView** paradigm is usually implemented instead.

Example of a **ViewHolder** class for a list of trip offers that displays for each offer: *title, short description, price and one image*.

```
public class TripsViewHolder {

    private TextView titleTextView;
    private TextView descriptionTextView;
    private TextView priceTextView;
    private ImageView tripImageView;

    public TripsViewHolder ( View itemView )
    {
        // when the ViewHolder is created, get & store references to the visual elements
        titleTextView = itemView.findViewById ( R.id.title );
        descriptionTextView = itemView.findViewById ( R.id.description );
        priceTextView = itemView.findViewById ( R.id.price );
        tripImageView = itemView.findViewById ( R.id.image );
    }

    public void bindViewHolder ( Trip trip )
    {
        // use the references in the recycling process to update the information
        titleTextView.setText ( trip.getTitle() );
        descriptionTextView.setText ( trip.getDescription() );
        priceTextView.setText ( trip.getPrice() );
        tripImageView.setImageResource ( trip.getImageResourceId() );
    }
}
```

9.2.3 Notify the RecyclerView.Adapter on data model updates

In **ListView** based implementation, when the data model has changed it is necessary to invoke the adapter's method **notifyDataSetChanged()** in order to request the display update. This method initiates the refresh for the entire list, no matter how many of the elements have been changes. Consequently, invoking **notifyDataSetChanged()** is very resource intensive and should occur only when the entire list (or, at least, most of it) needs to be updated. Unfortunately, **ArrayAdapter** and **BaseAdapter** do not implement other changes notification methods and require entire list update event for small changes (eg. for a single element).

The **RecyclerView** based implementation provides to the developer more flexibility through a list of more focused update methods:

- *notifyItemInserted (int index)* – invoked when a new element has been added to the list on position **index**
- *notifyItemRemoved (int index)* – invoked when the element at position **index** has been removed from the list
- *notifyItemChanged (int index)* – invoked when the element on position **index** has been updated in any way and it needs to be redrawn

- `notifyItemRangeChanged (int fromIndex, int toIndex)` – invoked when a range of grouped elements in the list has been updated and it is required to be redrawn
- `notifyDataSetChanged()` – should be invoked ONLY when all the elements from the list need to be redrawn

9.2.4 Customize list elements display

When using **RecyclerView** component to display a list it is **mandatory** to specify one of the available layouts for list items display. With **RecyclerView** the developers have more predefined layout options to organize list's elements than when using **ListView**. For example, the list can be displayed *horizontally* or *vertically*, on more than 1 *columns* or in a *grid*-like style. This is not possible out-of-the box with **ListView**, which implements only the vertical layout by default.

In order to set the display mode of a list created with **RecyclerView**, we will use a **LayoutManager** component:

- vertical display on a single column (similar with ListView)

```
LinearLayoutManager layoutManager = new LinearLayoutManager ( context );
```

- horizontal display on a single row:

```
LinearLayoutManager layoutManager = new LinearLayoutManager ( context );
layoutManager.setOrientation ( LinearLayoutManager.HORIZONTAL );
```

- vertical display on two columns:

```
LinearLayoutManager layoutManager = new GridLayoutManager ( this, 2 );
```

9.3 Development considerations

9.3.1 Adding required library for RecyclerView to the APP

RecyclerView is not one of the core elements of the Android platform. It is provided in an external support library that must be referenced as an entry to the **app/build.gradle** file in the section **dependencies**:

```
implementation 'com.android.support:recyclerview-v7:26.1.0'
```

Although any available version can be included in the APP, it is recommended that all the external libraries to have the same version.

9.3.2 Implement RecyclerView

9.3.2.1 Create the XML layout file

Adding a `RecyclerView.Adapter<ViewHolder>` component to the visual structure of an activity (or fragment) starts in the layout file:

```
<android.support.v7.widget.RecyclerView
    android:id = "@+id/idRecyclerView"
    android:layout_width = "match_parent"
    android:layout_height = "match_parent" />
```


9.3.2.2 Update the Java code

As already mentioned, when a RecyclerView component is used it is mandatory to set the LayoutManager and the Adapter. For this purpose, we first need a reference to the visual element defined in the layout file.

```
recyclerView = findViewById ( R.id.messagesRecyclerView );
recyclerView.setLayoutManager ( new LinearLayoutManager ( context ) );
recyclerView.setAdapter ( adapter );
```

9.3.2.3 Create RecyclerView.Adapter and RecyclerView.ViewHolder

In order to benefit from the performance optimization of the **RecyclerView** based approach for displaying lists, we need to create a new view holder (for example **MyViewHolder**) that inherits from **RecyclerView.ViewHolder**. Also, a dedicated adapter (in our example **MyAdapter**) will be inherited from **RecyclerView.Adapter<ViewHolder>** in order to map the model to the visual template:

```
public class MyAdapter extends RecyclerView.Adapter<MyViewHolder> {
    private List objects;
    private Context context;

    public MyAdapter(Context context, List objects)
    {
        this.context = context;
        this.objects = objects;
    }

    @Override
    public MyViewHolder onCreateViewHolder ( ViewGroup viewGroup, int position )
    {
        LayoutInflater inflater = LayoutInflater.from ( context );

        // interpret the XML file and create Java internal structure
        View layout = inflater.inflate ( R.layout.layout_name, viewGroup, false );

        // returns the view holder which is created from the view
        return new MyViewHolder ( layout );
    }

    @Override
    public void onBindViewHolder ( MyViewHolder viewHolder, int position )
    {
        // binds the view holder with the data; in that method, everything necessary
        for displaying the correct object should be set
        viewHolder.bindViewHolder ( objects.get ( position ) );
    }

    @Override
    public int getItemCount()
    {
        // returns the number of objects from the list
        return objects.size();
    }

    // override this method if you want different types of view for the elements
    of the RecyclerView depending on their position (eg. odd or even position)
    @Override
    public int getItemViewType ( int position )
    {
        // should return a code that will be used in method onCreateViewHolder
        to determine which layout to inflate for this item
    }
}
```

When creating the ViewHolder we will start from the code available in Section 2.2 of this guide, making sure that we inherit from **RecyclerView.ViewHolder** class and that in the constructor we initialize the parent by calling:

```
super ( itemView );
```

9.3.3 Process the **onItemClick** event for a **RecyclerView** list element

As a major difference from **ListView** component, where we could use the **OnItemClickListener** method of the list container to attach an **onItemClick** handler for all the list's elements at once, when using **RecyclerView** for list display we need to attach the event handler directly on the visual component of each element (for example in the method **bindViewHolder**).

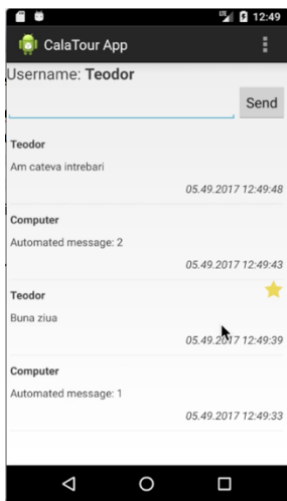
```
public void bindViewHolder ( Message message )
{
    itemView.setOnClickListener ( new View.OnClickListener()
    {
        @Override
        public void onClick ( View v )
        {
            // add here the code to be executed when a List element is tapped
        }
    });
}
```

9.3.4 Use different display templates in the same list

As exemplified in subsection 3.2.3 of this guide, the **RecyclerView.Adapter** enables the developers to tap into the process of creating view holders and to customize even further the design of the list. For example, if some elements should have a different design depending on the information displayed, the developers can prepare the required number of different templates and then, by overwriting the **getItemViewType** method, can decide (using a specific algorithm) to use for each element one or another of the designs previously prepared.

The method **getItemViewType** should be explicitly invoked from the **onCreateViewHolder** method before the layout is inflated.

9.4 Exercises



Create a chat interface with the design showcased in the left image.

Implementation requirements:

- messages list is implemented using RecyclerView component
- new messages will be displayed at the top of the list
- every 3 seconds an automated message is generated and added to the list
- when a message is tapped
 - it will be marked as favorite and will showcase a yellow star on the top-right side
 - a second time, it will be removed from the favorite list and the star will be hidden

LABORATORY 10 – CONNECT TO A REST API FROM ANDROID

10.1 Introduction

Making requests to a REST Web API from Android can be implemented using different approaches:

- AsyncTask and HttpURLConnection: included by default in the Android framework
- Volley library - <https://github.com/google/volley>
- Retrofit library - <http://square.github.io/retrofit/>

Out of these we will concentrate in this laboratory on the Retrofit based approach, as it is one of the most popular solutions currently used by developers.

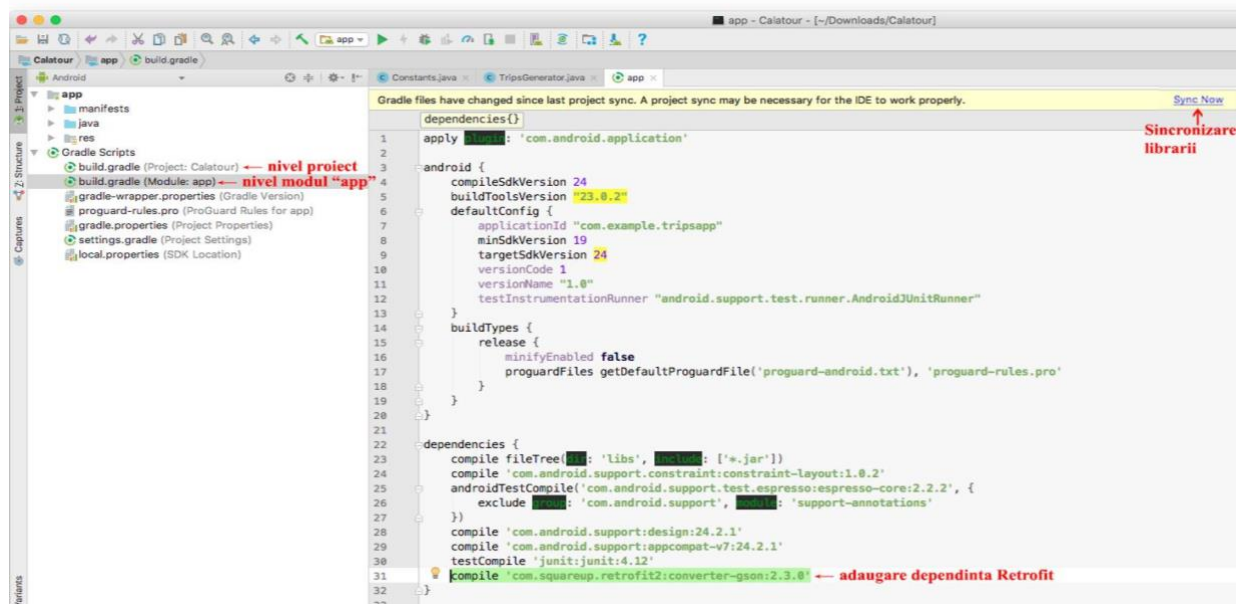
10.1.1 Laboratory objectives

Present the necessary steps for integrating Retrofit library into an Android application and the methods used to connect to a REST API.

10.2 Retrofit library setup

10.2.1 Including required dependencies

As it is not delivered with the Android SDK, in order to use the Retrofit library in an Android application it is necessary to include a reference to it in the **app/build.gradle file**, section **dependencies**. Before compiling, please make sure to use the “Sync now” feature after any update to the **build.gradle** file.



```
dependencies {
    .....
    compile 'com.squareup.retrofit2:converter-gson:2.3.0'
}
```

10.2.2 Creating model classes

When making calls to a REST API using the Retrofit library it is necessary to convert sent information and received data to local models. Depending on the content encoding used it is necessary to convert the sent and received information to local models. Specialized libraries can be used for these operations, like **Gson** for **json** content or **Simple XML** for **XML** encoded content.

The conversion will not work appropriately if the local defined models are not mapped correctly to the server's response. Consequently, for each of the called API it is necessary to analyze the response and to define a Java class that can be automatically mapped to it, similar to the example below:

```
{
  "value1": "some string value",
  "value2": 2
}
```

```
public class ClaseModel
{
  @SerializedName("value1")
  private String value1;

  @SerializedName("value2")
  private int value2;
}
```

10.2.3 Declaring HTTP operations

Retrofit library requires that each of the API endpoints is declared as a method of a public interface, where all the necessary information for creating the call is included (except actual data): the *URL of the endpoint*, *request headers* (eg. Authorization), *data encoding* information (also as a header) and *data mapping rules* (what data is encoded in the URL and what data is sent in the body of the request).

```
public interface UserService
{
  // describe the HTTP method type (other values: @POST, @PUT, @DELETE)
  // and the path of the endpoint relative to the root address of the API
  @GET ( "users/{id}" )

  // set the static header that specifies the encoding of the data transfer
  @Header ( "Content-Type: application/json" )

  // add information about the data mapping and return type
  Call<User> getUser ( @Header ( "Authorization" ) String auth, @Path ( "id" ) int id );
}
```

10.2.3.1 Server answer type

The type of the answer expected from the server is specified as the return type of the end-point method (**Call<User>** in our example). In this case, **Retrofit** will convert the **json** response from the server to a **User** object, making use of the convertor provided by the developer at instantiation time (please see section 2.4 for details).

10.2.3.2 Set request headers

The request headers, defined through the **@Header** notation, can be **static** (with predefined value for all the calls) or **dynamic** (which are instantiated with runtime values). In our example, the **Content-Type** header is static (as all the data transfer will use **json** encoding) while the **Authorization** header is dynamic (because the token is received only after user authentication). Static headers should be defined as *method annotations* while dynamic headers as *method parameters*.

10.2.3.3 Configure request body

For the HTTP methods POST and PUT most of the information from the client to the server is transferred through the body of the request. In the API interface, this can be defined as follows:

```
@PUT ( "end_point_URL" )  
Call<Void> EndPointName ( @Body ModelClass body );
```

When the call to the **EndPointName** is initiated, the content of the **body** parameter will be automatically convert from **ModelClass** to **json** or **XML** representation, according to the Retrofit settings for this specific API.

10.2.3.4 Define query parameters

Query parameters represent data included in the URL after the ? sign, encoded as **key=value** pairs and separated through **&** signs. Example of URL with query parameters:

```
http://example.com/users?page=3&order=ascending
```

In Retrofit, these parameters can be declared using the **@Query** notation, as method parameters. For example:

```
@GET("users/")  
Call<List<User>> getUsers(@Query("page") String page, @Query("order") String order);
```

10.2.3.5 Map path parameters

In REST APIs is very common to include some of the necessary information for the call in the URL structure, as a **virtual path element** and not as a **query parameter**. For example, the following URL retrieves the offer with ID=3 created by the user with ID=1:

```
http://example.com/user/1/offer/3/
```

In order to configure Retrofit library correctly, it is necessary to use the **@Path** notation for the correspondent parameters of the endpoint description method:

```
@GET ( "offers/{id}" )  
Call<Offer> getOffer ( @Header ( "Authorization" ) String auth, @Path ( "id" ) int id );
```

10.2.4 Register the API interface with the Retrofit library

When all the support classes are ready (data models, API description interface), we need to create a Retrofit instance providing as parameters:

- the root URL address of the API
- the data encoding convertor, that will be used for request data as well as for response data

```
Retrofit retrofit = new Retrofit.Builder()  
    .baseUrl ( "base_url" )  
    .addConverterFactory ( GsonConverterFactory.create() )  
    .build();
```

The next step is to register the interface describing the API with this newly created Retrofit instance:

```
UserService userService = retrofit.create ( UserService.class );
```

10.2.5 Making calls to the API endpoints

All the Retrofit calls to the server are asynchronous. The response (on success) or errors (on failure) are processed through callback functions that will be invoked by the operating system when the state of the connection changes. In the functions description, we will pass the expected data model to be received and a secondary parameter that represents a reference to the raw response (in case of success) or to the thrown error (in case of failure). The headers of the functions are described in the **Callback<T>** interface.

For the **UserService** examples describe above, a call can be performed as follows:

```
userService.getUser("auth_token", 1).enqueue(new Callback<User>()
{
    @Override
    public void onResponse(Call<User> call, Response<User> response)
    {
        // code to be executed on successful response from the server
    }

    @Override
    public void onFailure(Call<User> call, Throwable t)
    {
        // code to be executed on connection or processing failure
    }
});
```

10.2.5.1 onResponse

This method is invoked when a connection with the server has been successfully established and information has been exchanged. The developers need to process here all server codes, no matter if they indicate **success** (status code 200) or a processing **error** (status codes 400, 404, 500 ...).

response.isSuccessful() method can be used to easily test the outcome of the server's processing. If the return value is **false**, we can usually extract more information using the **response.errorBody()** method.

```
@Override
public void onResponse(Call<T> call, Response<T> response)
{
    if (response.isSuccessful())
    {
        // request is successful and response is mapped in responseBody
        T responseBody = response.body();
    }
    else
    {
        // request is not successful (some error occurred on the server)
        int errorStatusCode = response.code();
        String errorMessage;
        // verify if the server has sent more details about the error
        try
        {
            errorMessage = response.errorBody().string();
        }
        catch (IOException e)
        {
            errorMessage = "Error message cannot be obtained!";
            e.printStackTrace();
        }
    }
}
```

10.2.5.2 onFailure

This method is called in one of the following situations:

- internet error connection (cannot connect to the server) – this can be caused by
 - missing internet connection on the device
 - missing application permissions to access the internet
 - server not responding
- error when converting request data from model to designated encoding
- error when converting the server response to the local model

10.3 Development considerations

10.3.1 Adding internet access permissions

In Android devices the access of the applications to the internet is subject to permissions granted by the user. In order to register for these permissions, it is necessary to update the **AndroidManifest.xml** file by adding a **uses-permission** tag:

```
<manifest package="com.app.example"
  xmlns:android="http://schemas.android.com/apk/res/android">

  <uses-permission android:name="android.permission.INTERNET" />
  <application ... />

</manifest>
```

10.4 Exercises

Exercise 1: Implement the models and interfaces necessary to connect from an Android application to the <https://cgisdev.utcluj.ro/moodle/chat-piu/> API [1]. When the implementation is complete, use the <https://cgisdev.utcluj.ro/moodle/chat-piu/register> and create a new account, then verify if you can authenticate from the Android APP.

Exercise 2: Implement the models and interfaces necessary to connect from an Android application to the <https://cgisdev.utcluj.ro/moodle/chat-piu/> API [1]. Test your implementation by running your application on two emulators (or a device and an emulator) and sending messages from one instance to the other.

10.5 References

- [1] Teodor Ștefănuț, ChatAPI Technical Specification, [Online]
https://cgisdev.utcluj.ro/moodle/chat-piu/ChatAPI_specification.pdf (last access 10.02.2019)