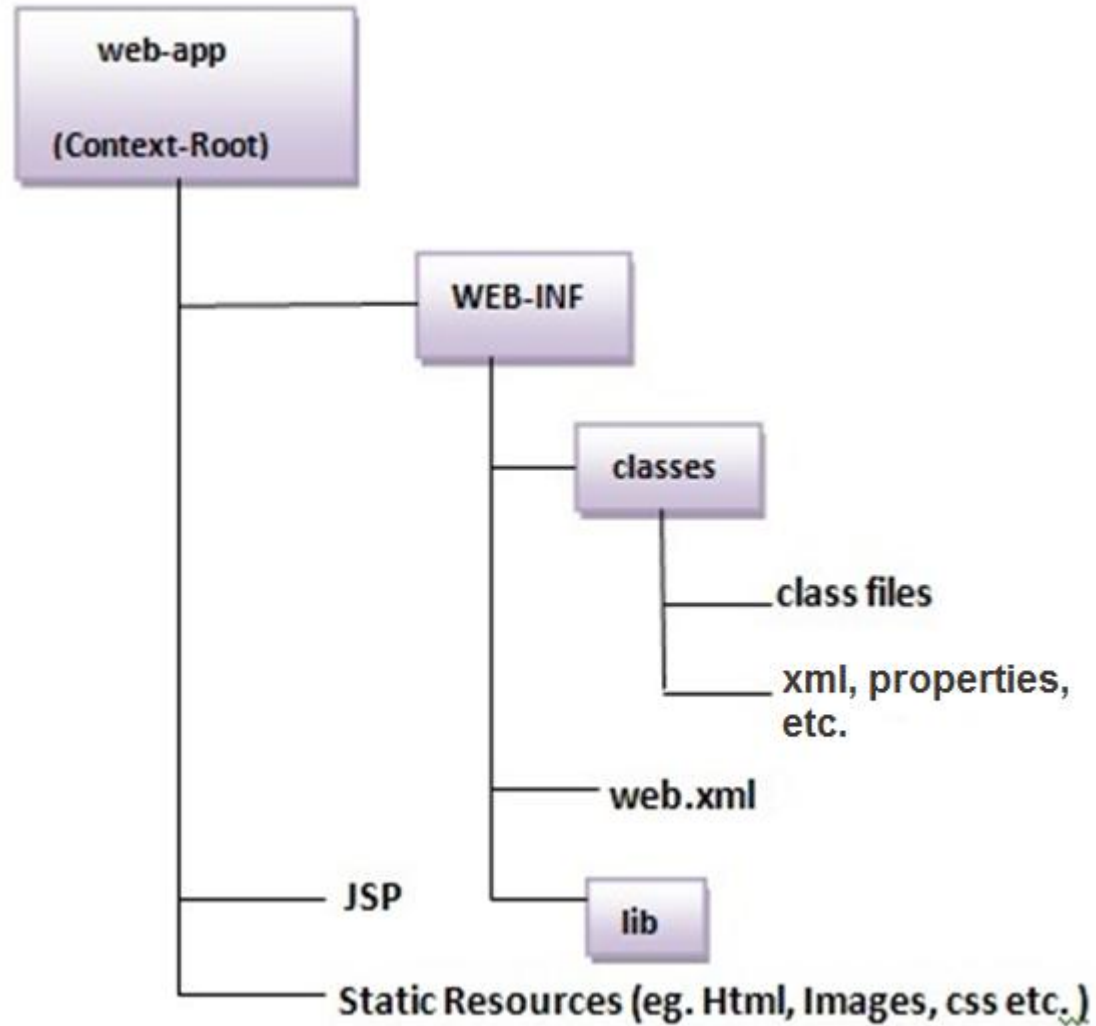


Intracom Telecom

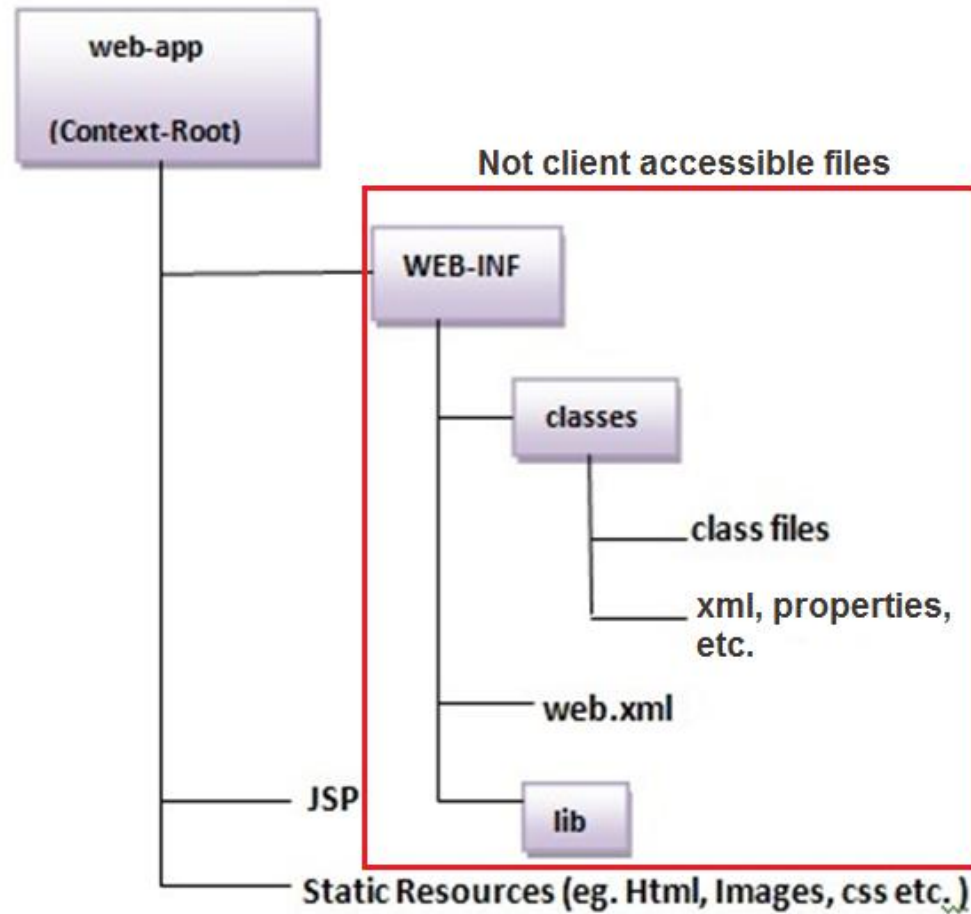
Java SE / EE Workshop

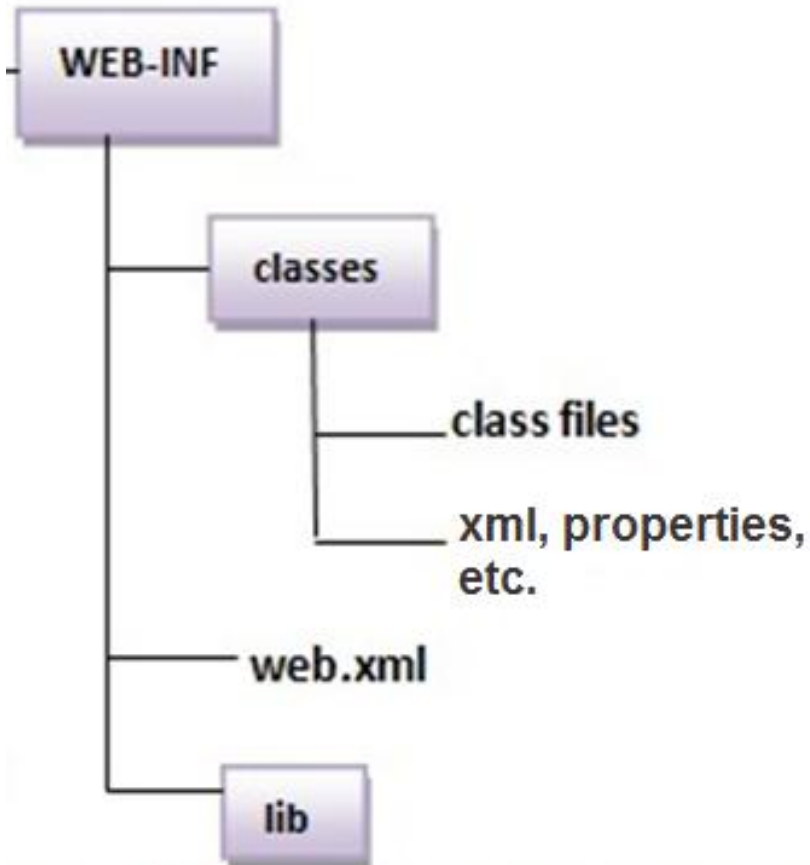
Hands-on

- Aim of this part of Workshop is to examine a number of Java EE technologies in depth and make usage of them in order to build a Web Application.
- Java EE (Java Platform Enterprise Edition) is a collection of Enterprise oriented Java Technologies. Each Technology is defined by its own Specification.
- Enterprise: Scalable and Distributable.
- Web Application is a client-server program in which the communication protocol between the two parts is the HTTP/HTTPS and client side runs in a Web Browser. In practice the same term covers a dynamic Website also.
- A Java based Web Application runs either on a standalone Web Container (i.e. Tomcat) or on a Java Application Server which implements the Java EE standard. According to the standard, an Application Server should implement (among other things) a Web Container for running Web Applications.
- Such a program is packaged in a special file format which is called WAR (**W**eb **A**pplication **R**esource or **W**eb application **A**Rchive). Actually WAR (as JAR) is a ZIP file with a certain file structure.



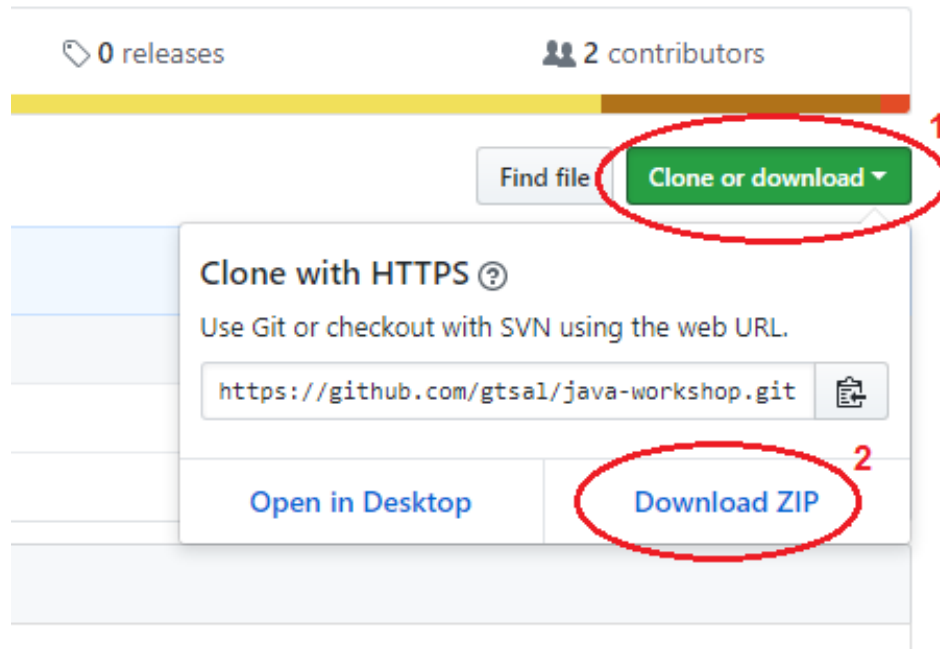
- The action of installing a Java based Web Application on its Server Program is called **deployment**.
- After the successful deployment of a WAR file, the Web Container gives access through HTTP/HTTPS to all static and dynamic Resources of Web Application. It achieves that by usually adding a **Context Path** at its listening URI.
- **Context Path** is always the first part in a request URI. It starts from the first “/” character after the domain name of the server, and ends at the next one, without out included it.
- Context Path may be defined somewhere inside WAR file, or it be passed during deployment.
- So, if the base URI of our Web Server is **http://www.mysite.org** and the Context Path is “/allwith1euro”, we can access the main page of Web Application by sending an HTTP GET at **http://www.mysite.org/allwith1euro/**.
- All static Resources of WAR file can be accessed by adding the file path of the Resource at the above URI. So if WAR contains a car.jpg file in directory “/images/items/”, we can download it by sending an HTTP GET operation at **http://www.mysite.org/allwith1euro/images/items/car.jpg**.
- Actually what ever is in WAR, apart from the items under **WEB-INF/** (and META-INF/ if it exists), it can be retrieved from the client.



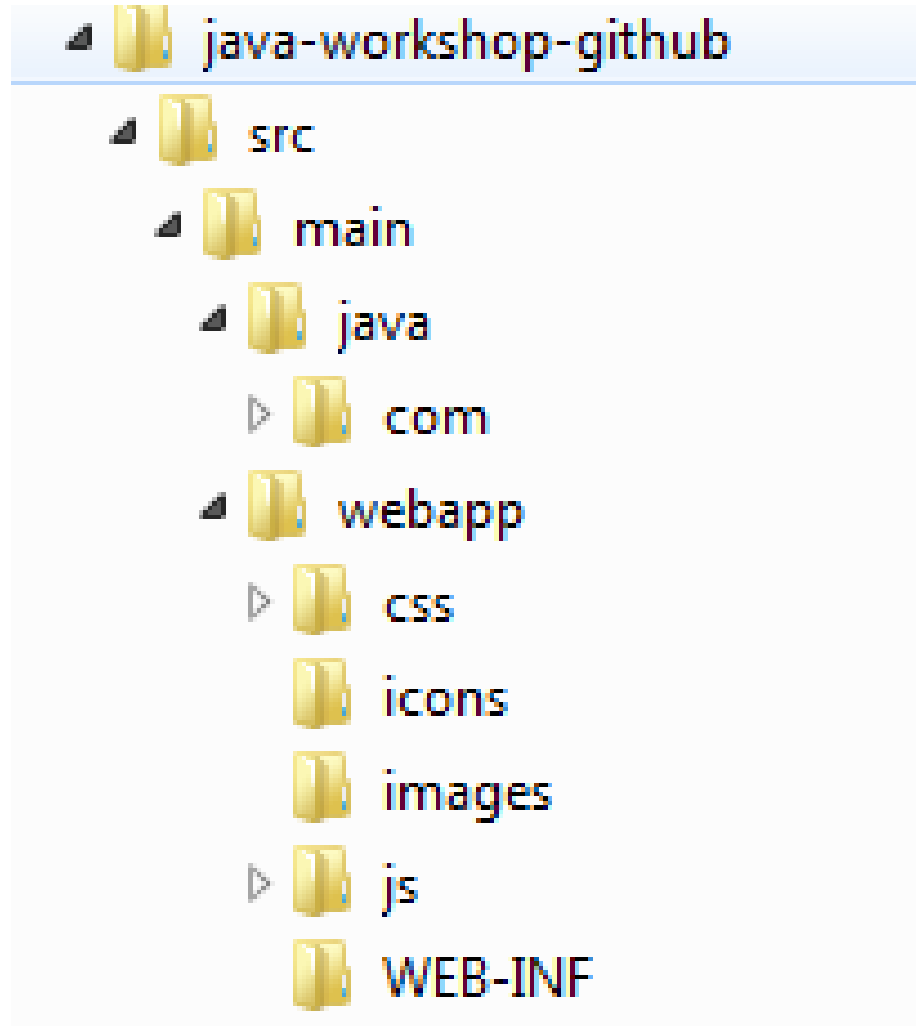


- **web.xml** is the heart of a Java Web Application. It defines which Servlet serves each URL part, describes the configuration of the application, the main page, which resources can be accessed only by using HTTPS, e.t.c.
- **classes/** directory contains the “**.class**” compiled files that implement the server side functionality. It can also contain “**.xml**”, “**.properties**”, or any other kind of data files that our code depends on.
- **lib/** directory contains jar files of third party libraries that our applications makes use of.

- Access URL
<https://github.com/gtsal/java-workshop>



- Press the “Clone or download” button (1).
- Select the option “Download ZIP” (2).
- Unzip file in a directory on your hard disk.



- You can easily build the WAR file and deploy it to a local WildFly server with maven. You can also install a new version of your WAR file, without having to stop WildFly, or remove any file. This action is called **Redeploy**.
- At first, start your WildFly 10.1 server.
- Next, open a Command Prompt window and go to the directory you have unzip the github code.
- From the main directory of the project (the directory containing the **pom.xml** file) give the following command
 - `PATH_TO_MAVEN/bin/mvn clean package -P standalone`
- If everything goes as expected, at the end of the execution of the above command, your war file will have be deployed in wildfly.
- To verify that, start your favorite Web Browser and give the following URL:
 - <http://localhost:8080/workshop/>

You should see the following page

Intracom Telecom S.A. Java SE/EE Workshop

- It does not seem so impressive, but it is a start.
- How does wildfly know which page to show? It checks the **WEB-INF/web.xml** it found in deployed WAR.

```
<welcome-file-list>  
  <welcome-file>index.html</welcome-file>  
</welcome-file-list>
```

This is actually the only (not commented for now) thing the file contains.

- **WEB-INF/** directory also contains a file named **jboss-web.xml**. It is a jboss/wildfly specific file and it contains the **Context Path** of our application.

```
<context-root>/workshop</context-root>
```

So in order to access our application we had to add the above path in the URL.

1st Exercise: Display the map

- Now let's start adding some functionality to our application.
- The first thing that we add is a geographical map of the area we are now.
- Edit your **src/main/webapp/index.html** file, find the string "1st Exercise" and remove the comments around **<script>** to see the real page:

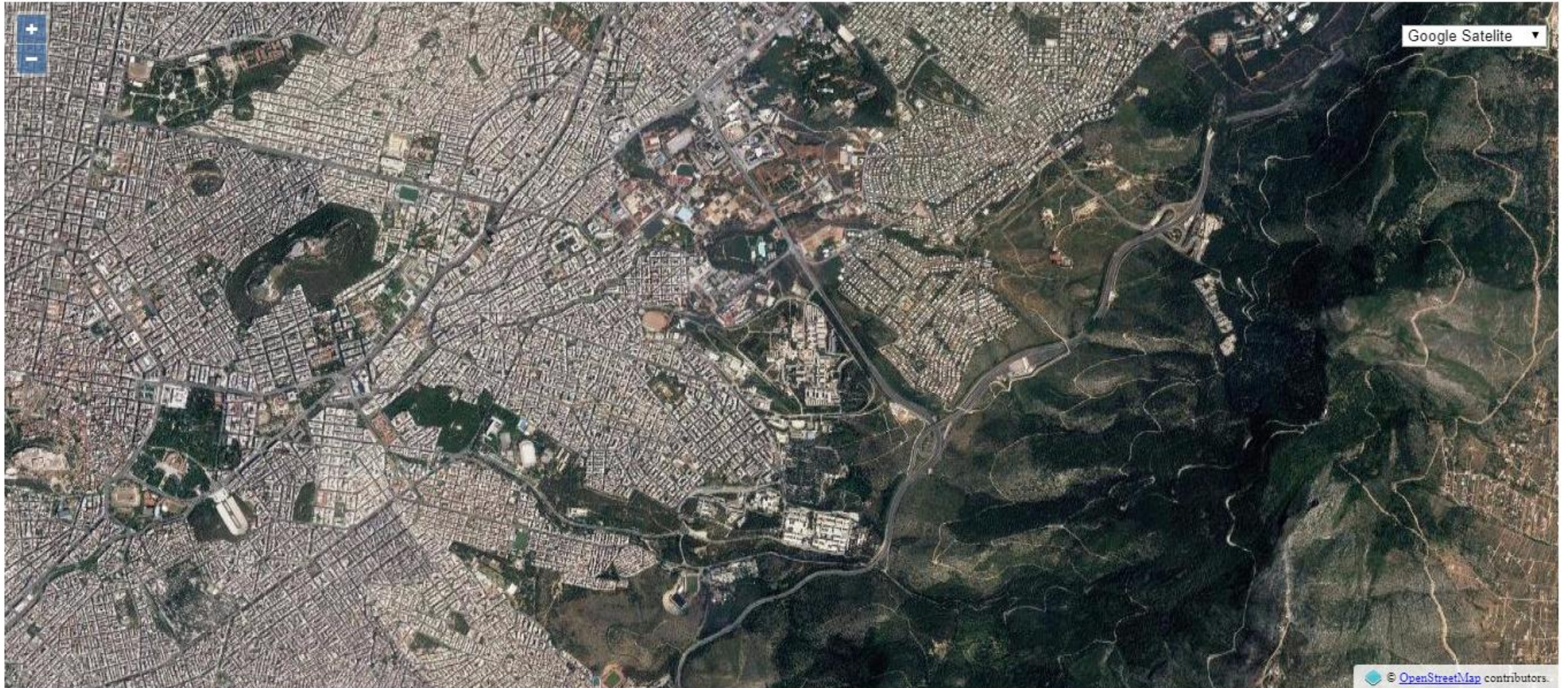
```
<!-- 1st Exercise: Display the map.
```

```
Uncomment the following script to see the page with the map.
```

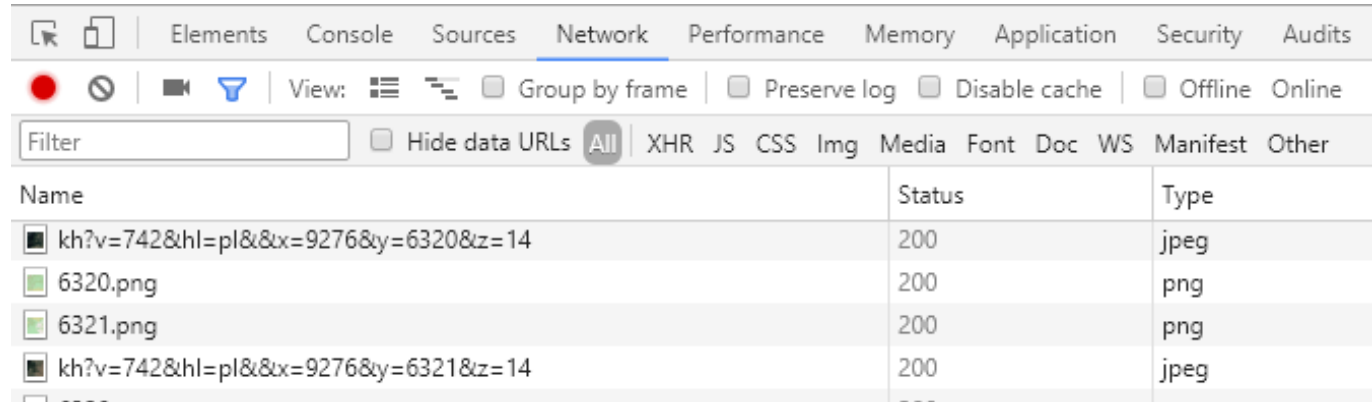
```
<script src="js/require.js" data-main="js/app"></script>
-->
```

- Save the change in file and run the maven command to redeploy the application.
- At the end, refresh the page on your web browser. You should see the following page.

1st Exercise: Display the map

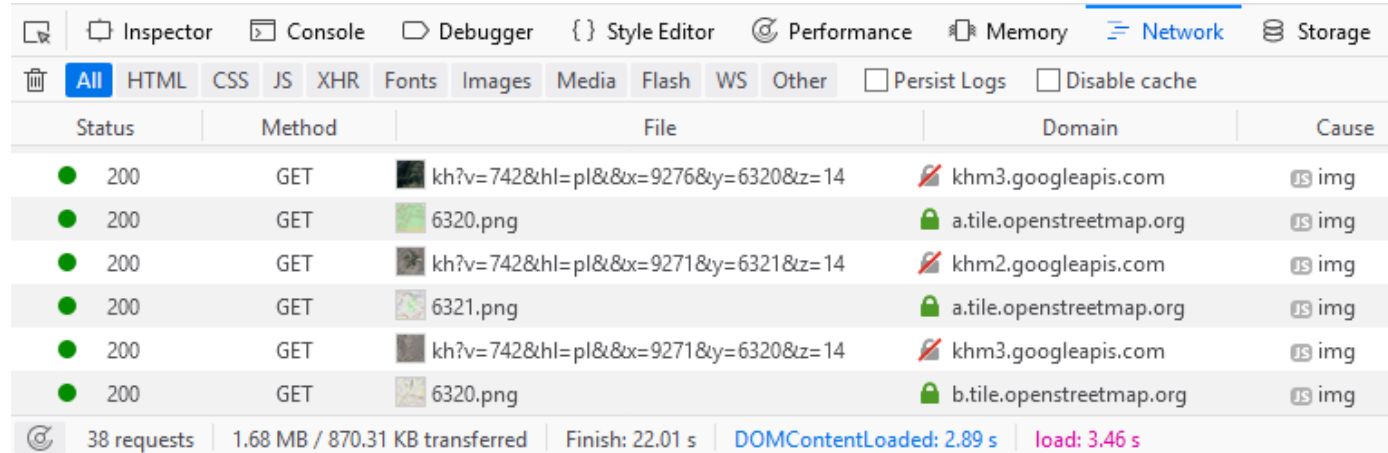


- In case were the result was not the expected, we could use the “Developer Tools” each modern Web Browser has.
- On most of them, we can open “Developer Tools” by pressing the F12 key. Below you can see screen shots from “Developer Tools” of Google Chrome and Mozilla Firefox.



Network tab showing a list of requests. The table has columns: Name, Status, and Type.

Name	Status	Type
kh?v=742&hl=pl&&x=9276&y=6320&z=14	200	jpeg
6320.png	200	png
6321.png	200	png
kh?v=742&hl=pl&&x=9276&y=6321&z=14	200	jpeg



Network tab showing a list of requests. The table has columns: Status, Method, File, Domain, and Cause.

Status	Method	File	Domain	Cause
200	GET	kh?v=742&hl=pl&&x=9276&y=6320&z=14	khm3.googleapis.com	img
200	GET	6320.png	a.tile.openstreetmap.org	img
200	GET	kh?v=742&hl=pl&&x=9271&y=6321&z=14	khm2.googleapis.com	img
200	GET	6321.png	a.tile.openstreetmap.org	img
200	GET	kh?v=742&hl=pl&&x=9271&y=6320&z=14	khm3.googleapis.com	img
200	GET	6320.png	b.tile.openstreetmap.org	img

Summary: 38 requests | 1.68 MB / 870.31 KB transferred | Finish: 22.01 s | DOMContentLoaded: 2.89 s | load: 3.46 s

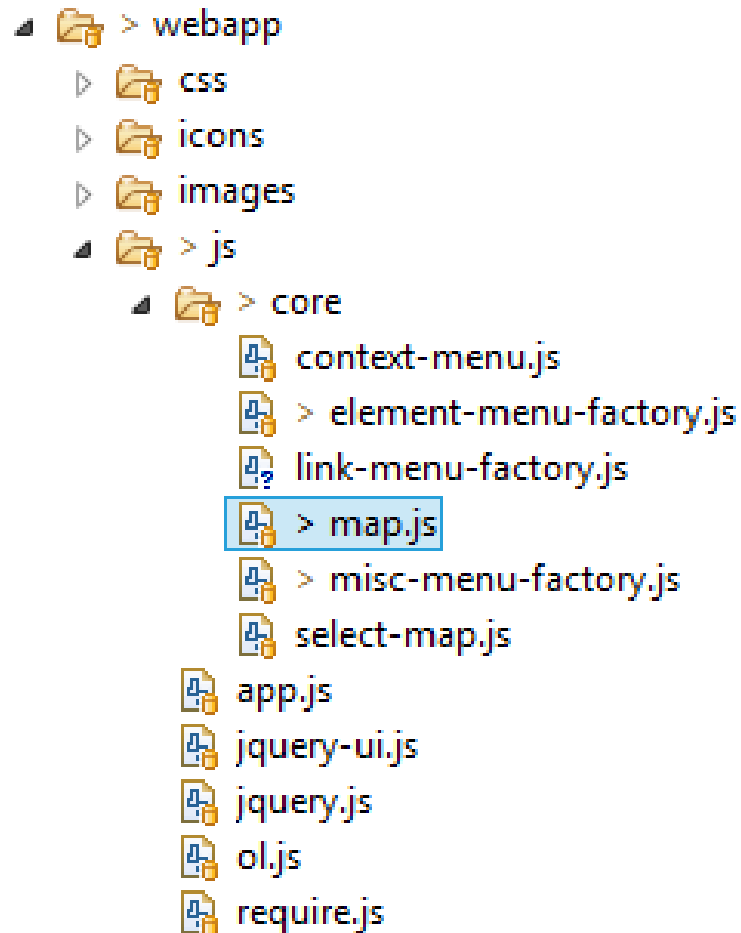
- What was the script we loaded in index.html?
- RequireJS is a javascript library which allows to break our javascript code into logical sections (modules) and keep it in separated files.
- You can request the loading of a module from some file with the following function:

```
require(["jquery", "ol"], function($, ol) {  
    /* you can now safely access $ jquery global variable and ol OpenLayers global  
       variable */  
})
```
- The first argument of `require()` is an array containing the symbolic names of all modules (libraries) that should be loaded before running the function defined in the second parameter.

- You can define your own module by using the `define()` function:

```
define(["jquery"], function($) {  
    /* initialization code */  
    return {  
        message: "Hello world",  
        foo: function() {  
            },  
        ...  
    }  
})
```

Again the first argument is the list with the dependencies. The second argument is again a function, only that now it has to return a JSON Object. That object exposes the functionality of the module.



- Directory **js/** hosts all JavaScript code, ours and third party code (jQuery, OpenLayers, RequireJS).
- Our code resides under **js/code/** directory.
- The only application java script in js/ is app.js which defines the RequireJS dependencies. So we will not be needed, because all dependencies are already defined.

- OpenLayers is a JavaScript library for displaying map data in a Web Browser and it provides an API in order to create applications similar to **Google Maps**.
- Code for creating a map (check code in **js/core/map.js**)

- We have to create at least one Layer.

```
var openStreetMapLayer = new ol.layer.Tile({
    source: new ol.source.OSM(),
    zIndex: 0
});
```
- We define a vector layer for our Features

```
var vectorSource = new ol.source.Vector({});
var vectorLayer = new ol.layer.Vector({
    source: vectorSource,
    zIndex: 2
});
```

- We create the map

```
var map = new ol.Map({
  layers: [openStreetMapLayer, vectorLayer],
  target: "map",
  controls: ol.control.defaults({
    attributionOptions: /** @type {olx.control.AttributionOptions} */ ({
      collapsible: false
    })
  }),
  view: new ol.View({
    center: ol.proj.fromLonLat([23.781975, 37.978253]),
    zoom: 14
  })
});
```

Property **map.layers** holds the layers we created before. Property **map.target** keeps a reference to HTML **<div>** element with id “**map**”. Look at index.html. At last we define the initial view of our map with the **view** property.

- The next thing we will add in our application is a context menu when we right click on map.
- File **js/core/context-menu.js** contains the code for creating a context-menu.
- In file **js/core/map.js** we define an event listener for that action and we assign it to the ViewPort of Map object.

```
map.getViewport().addEventListener("contextmenu", function (event) {  
    event.preventDefault();  
    var point = map.getEventPixel(event);  
    var coord3857 = map.getEventCoordinate(event);  
    var coord4326 = ol.proj.transform(coord3857, "EPSG:3857", "EPSG:4326");  
    var context = {  
        point: map.getEventPixel(event),  
        coordinate: coord4326  
    }  
    contextMenu.showMenu(event, context);  
}
```

- With the above code we just listen for right click events and block the default action, which is to display the browser's context menu.
`event.preventDefault();`

- The real happens in file **src/main/webapp/js/core/context-menu.js**:

```
$("#cMenu").menu({
    select: function(event, ui) {
        ...
    }
});
$("#cMenu").on({
    contextmenu: function(evt) {
        return false;
    }
});
```

- The above jQuery/jQueryUI code obtains a reference to a **** section created in **src/main/webapp/index.html** with a id="cMenu" and it transforms it to a vertical menu. The **** sub-elements of that **** will be the menu items.
- The second thing this code does is to disable the context menu when a right click is performed on a menu item.
- Function **select()** is called by jQueryUI when user selects one of the menu items. Parameter ui represents the selected menu item.

2nd Exercise: Display an Item on context menu

- Since `` does not have any child `` nothing is displayed when we right click on our map. In our application the `` subelements are created at runtime according to where we click. We can create new HTML tag elements by using the jQuery:

```
var helpItem = $("<li><div>Help</div></li>").attr({  
    funcName: "help"  
});
```

The above code snippet not only creates a new element but assigns it and one attribute. Of course that element should be appended to `` to has a meaning. This is done in another place.

- In order to see a simple context menu with one item we will edit file **src/main/webapp/js/core/misc-menu-factory.js**. Search for string “2nd Exercise” and remove the comments to enable the line in between.

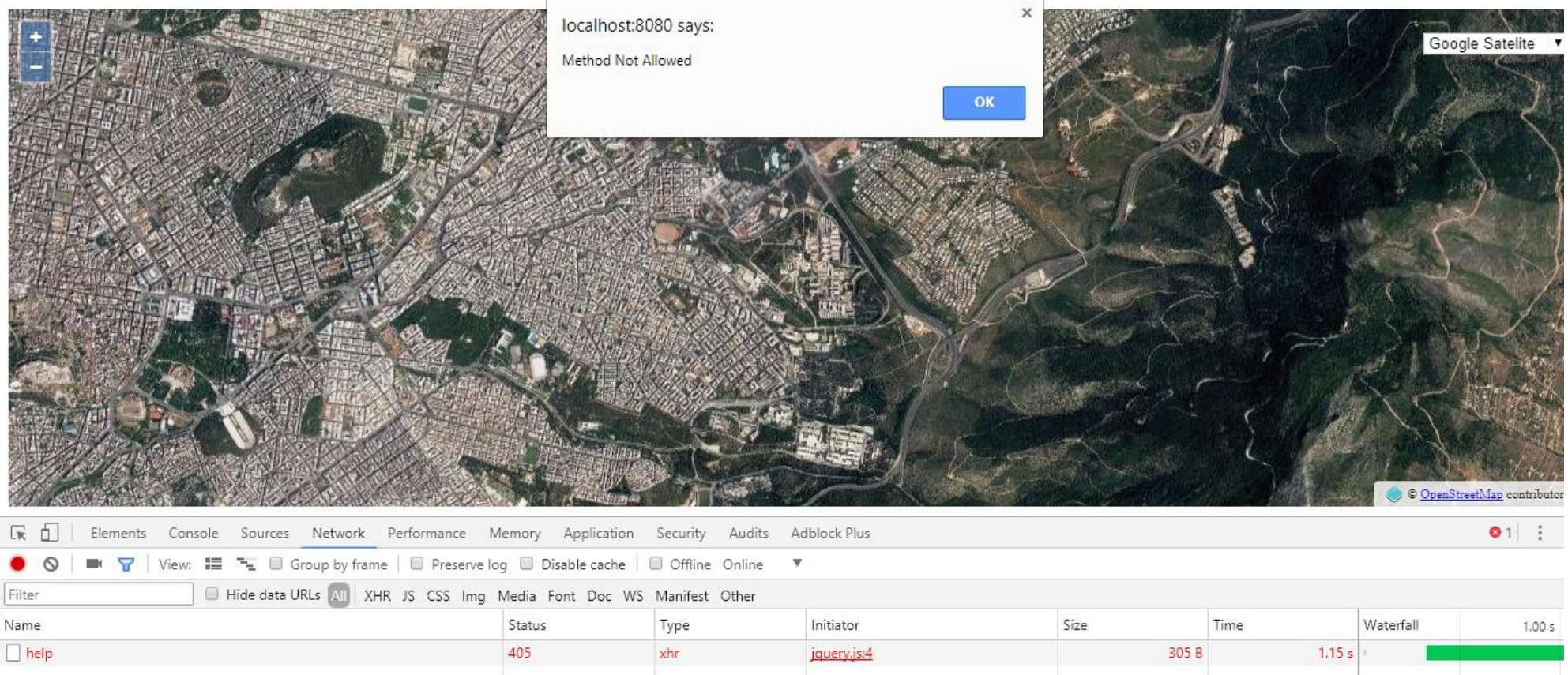
```
/* 2nd Exercise: Display a context menu.  
 * Uncomment the following line to enable the Help command.  
contextMenu.registerItemFactory(factory, 3);  
 */
```

- Save the change and run the maven command to build and deploy the war.

2nd Exercise: Display an item on context menu

- Refresh page on your browser.
- After the end of loading, right click anywhere on the map. Do you see the “Help” menu item?
- Select the item. A dialog box should open.
- Press the F12 key and go to the “Network” tab. After that, input your first and last name and press enter.
- Browser reports an Error Message. It is OK. We have to write code to server in order to be served.
- Check Developer Tools, where a request to server has been recorded.

Network Tab of Developer Tools



localhost:8080 says:
Method Not Allowed

OK


Google Satellite

OpenStreetMap contributor

Elements Console Sources **Network** Performance Memory Application Security Audits Adblock Plus

View: [Icons] Group by frame Preserve log Disable cache Offline Online

Filter [] Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other

Name	Status	Type	Initiator	Size	Time	Waterfall	
help	405	xhr	jquery.js:4	305 B	1.15 s		1.00 s

- This request was an Ajax call.

- Ajax (**A**synchronous **J**avascript and **X**ML) is a technology allowing to emit asynchronous HTTP/HTTPS requests to the server and get the response without interfering with the display and behavior of the current page. Response can be used to update dynamically the page through JavaScript commands without having to reload the entire page.
- In practice, the XML representation of the data has been replaced by JSON.
- Library jQuery provides a convenient wrapper for Ajax technology. See next slide.

```
$.ajax({
    method: "PUT",
    url: window.location.href + "help",
    contentType: "text/plain; charset=UTF-8",
    data: "Intracom Telecom S.A.",
    headers: {
        "Accept": "text/plain"
    }
}).done(function(data) {
    alert(data);
}).fail(function(jqXHR, textStatus, errorThrown) {
    alert(errorThrown);
});
```

- As we have seen with Developer Tools, our first Ajax call has failed. The reason is that there is no entity that can serve it.
- In the next section we will examine the Servlet technology that we allow us receive client calls and respond to them.

- A Servlet is a component of Java Web Application, which serves HTTP requests and it can dynamically create responses.
- To create a Servlet implementation class a programmer has to extend `javax.servlet.http.HttpServlet` abstract class and to override at least one of the following methods:
`doGet`, if the servlet supports HTTP GET requests
`doPost`, for HTTP POST requests
`doPut`, for HTTP PUT requests
`doDelete`, for HTTP DELETE requests
- Each of the above methods accepts two arguments: the first argument contains the information about the request (`HttpServletRequest`) and the second about the response (`HttpServletResponse`).
- Argument `HttpServletRequest` provides with information about: the request URL, the Headers the client provided, the Content-Type the client can accept, the Character Set of the request's body e.t.c.
- Argument `HttpServletResponse` is used to construct the Response that will be sent back.

- `Cookie[] getCookies();`
- `String getHeader(java.lang.String name);`
- `Enumeration<String> getHeaders(java.lang.String name);`
- `Enumeration<String> getHeaderNames();`
- `String getQueryString();`
- `Session getSession();`
- `String getContentType();`
- `ServletInputStream getInputStream();`
- `Reader getReader();`

- `void setHeader(String name, String value);`
- `void setCharacterEncoding(String charset);`
- `void setContentType(String type);`
- `void setStatus(int sc);`
- `void sendError(int sc, String msg);`
- `ServletOutputStream getOutputStream();`
- `PrintWriter getWriter();`

- A Servlet should be defined in web.xml in order Web Container to know about it and which URL path serves.

```
<servlet>
    <servlet-name>ServletName</servlet-name>
    <servlet-class>Servlet implementation class</servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>ServletName</servlet-name>
    <url-pattern>URL path of Requests that the servlet serves</url-pattern>
</servlet-mapping>
```

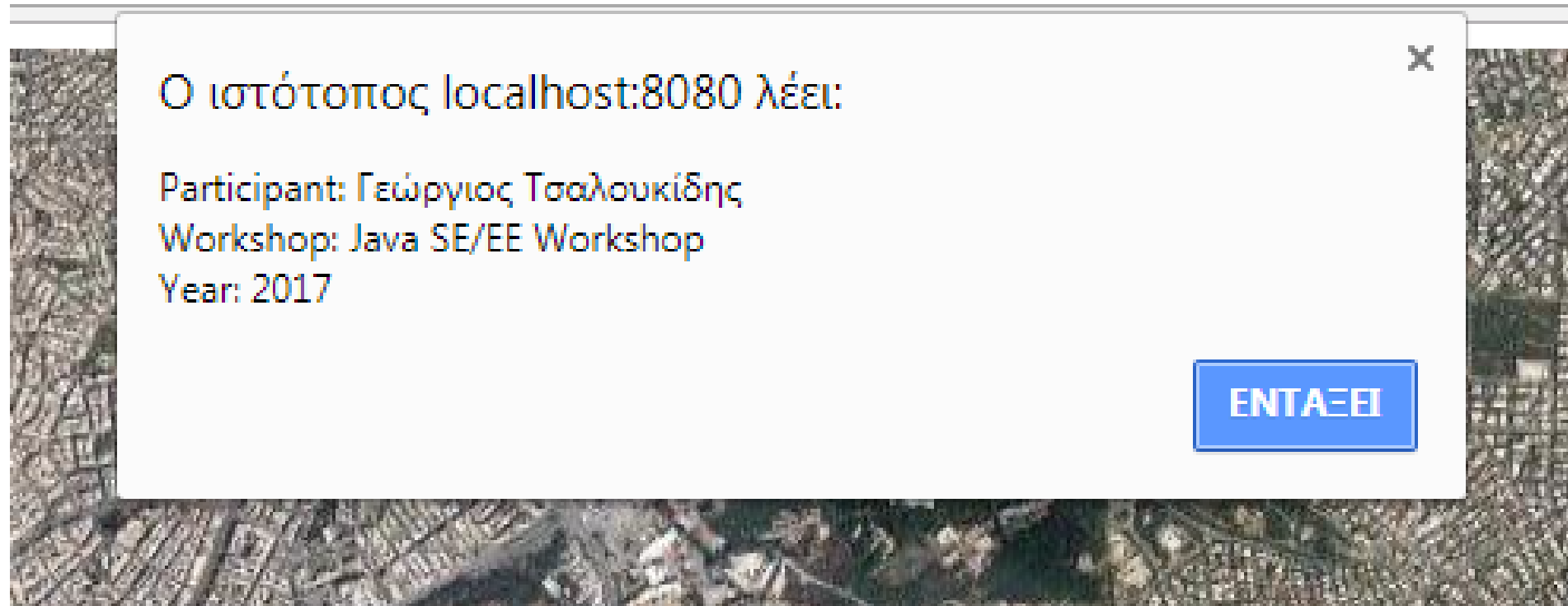
- The **<servlet>** element defines a logical name for the servlet and its implementation class. The logical name is used as a reference to the **<servlet-mapping>**. That element defines the URL path after the Context Path, that this servlet can handle. The **<url-pattern>** should start with a “/” and it can contain the wildcard pattern “/*” at its end. So a url-pattern “/*” means that all requests will be served by that servlet.

Third Exercise: Create your First Servlet

- The time for writing some code has come.
- Edit file **src/main/java/com/intracom/ems/workshop/HelpServlet.java**, and search for string “3rd Exercise”. Your aim is to write the code that will return the response in “text/plain” format. The response should have the following form:

Participant: THE NAME OF Participant
Workshop: Java SE/EE Workshop
Year: 2017
- Inside that file, you will find a set of static final String variables, that will help you to create the above string.

```
private static final String PARTICIPANT_FIELD = "participant";  
private static final String WORKSHOP_FIELD = "workshop";  
private static final String YEAR_FIELD = "year";  
private static final String WORKSHOP_VALUE = "Java SE/EE Workshop";  
private static final int YEAR_VALUE = 2017;
```
- Edit file **src/main/webapp/WEB-INF/web.xml**, and search again for string “3rd Exercise”. Uncomment definition of Servlet and fill in the missing/wrong information.
- Deploy the changes and execute the Help command by providing your full name, in the dialog box. Try one time using English characters and one more with Greek. The result is shown in the next slide.



- At first we write a method that will change the first character from lower to upper case:

```
private String capitalizeFirstCharacter(String value) {  
    StringBuilder builder = new StringBuilder(value.substring(0, 1).toUpperCase());  
    builder.append(value.substring(1));  
    return builder.toString();  
}
```

- Next we can write our response:

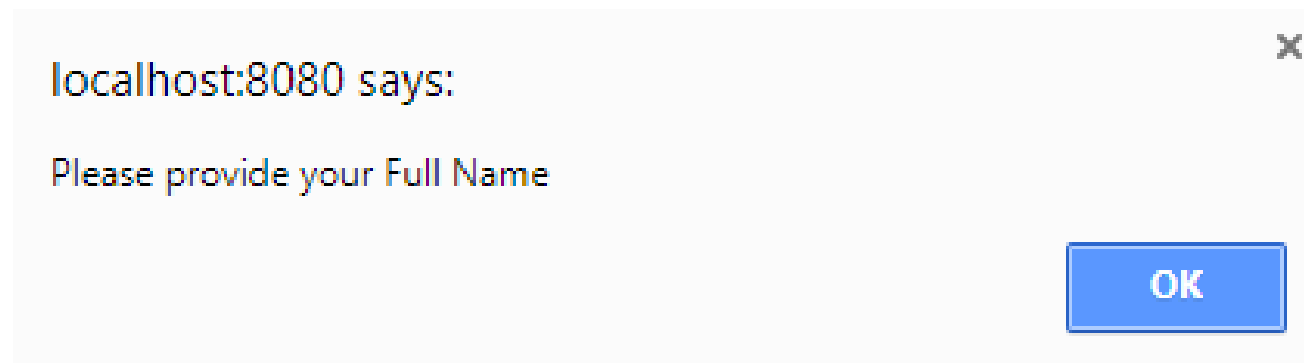
```
resp.setCharacterEncoding("UTF-8");  
try {  
    pw = resp.getWriter();  
} catch (IOException exc) {  
    resp.setStatus(500);  
    return;  
}
```

Continuing

```
resp.setContentType("text/plain");  
pw.println(capitalizeFirstCharacter(PARTICIPANT_FIELD) + ": " + participantName);  
pw.println(capitalizeFirstCharacter(WORKSHOP_FIELD) + ": " + WORKSHOP_VALUE);  
pw.println(capitalizeFirstCharacter(YEAR_FIELD) + ": " + YEAR_VALUE);  
// should we commit the resp object?
```

Forth Exercise: Examine the Response status on client

- Re-execute the Help Command giving only your first Name in the dialog box.
- See the error dialog that opens. Not very explaining, don't you think?
- We have to write code that will check the special status (599) of Response and extract the string from its body.
- Edit file **src/main/webapp/js/core/misc-menu-factory.js**, find the string “4th Exercise”, and write your code in order to show the error message that server sent.
- Deploy the changes and check the Error Box.
- It should be something like:



```
if(jqXHR.status === 599) {  
    alert(jqXHR.responseText);  
}  
else {  
    alert(errorThrown);  
}
```

- What about if we want our response to contain structured information?
- JSON or **J**ava**S**cript **O**bject **N**otation is a standard format used for transmitting information as a human-readable text.
- It is based on a subset of the JavaScript Programming Language, but due to its nature is language independent.
- JSON example:

```
{  
  "fullName": "John Smith",  
  "age": 27,  
  "address": { "streetAddress": "21 2nd Street", "city": "New York", "state": "NY",  
    "postalCode": "10021-3100" },  
  "phoneNumbers": [ { "type": "home", "number": "212 555-1234" }, { "type": "office",  
    "number": "646 555-4567" }, { "type": "mobile", "number": "123 456-7890" } ],  
  "children": null  
}
```
- In canonical form all keys are double quoted strings. Notice the comma (,) symbol that separates each key/value pair. Spaces do not have any meaning.

- JavaScript natively supports JSON.

```
var manager = {  
    fullName: "John Smith",  
    "age": 37,  
    foo: function() {  
        alert("foo() was called");  
    }  
}
```

- In JavaScript a property name can be double quoted or not. However that difference affects the way we access those properties. For example:

```
alert(manager.fullName);  
alert(manager.age);  
alert(manager["age"]);  
manager.foo();  
manager["foo"].call();
```

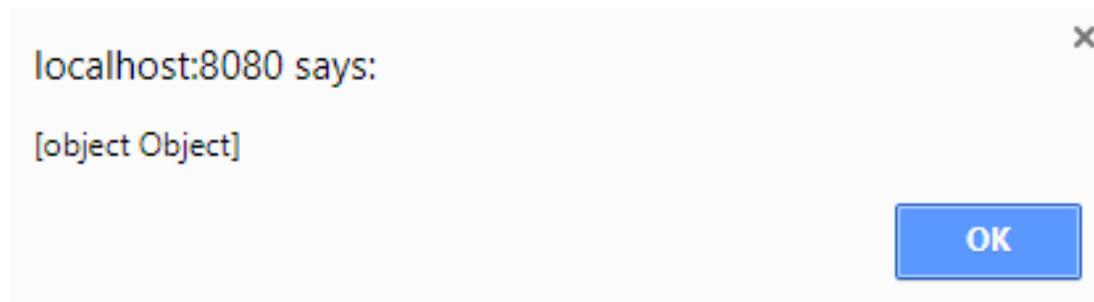
- A JSON object can be presented as string (in order to be transmitted to the server) by using the `JSON.stringify(manager);` expression.
- If we try to print the result of `stringify()` for the manager object we will get: `{"fullName":"John Smith","age":37}`.
- As you can see foo function seems that does not exist.
- Please try it on <https://jsfiddle.net/>
- Also a JSON string can be parsed to a JavaScript object using the expression `var p = JSON.parse("{\"name\": \"John\", \"age\": 35 }");`

Fifth Exercise: Send response in JSON format

- Edit file **src/main/webapp/js/core/misc-menu-factory.js**, search string “Change for 5th Exercise”, and comment out value **"text/plain"** for header **"Accept"** and uncomment **"application/json"**.

```
/* Change for 5th Exercise
 * Comment out value "text/plain" for header "Accept"
 * and uncomment "application/json"
 */
"Accept": "text/plain" /* "application/json" */
```

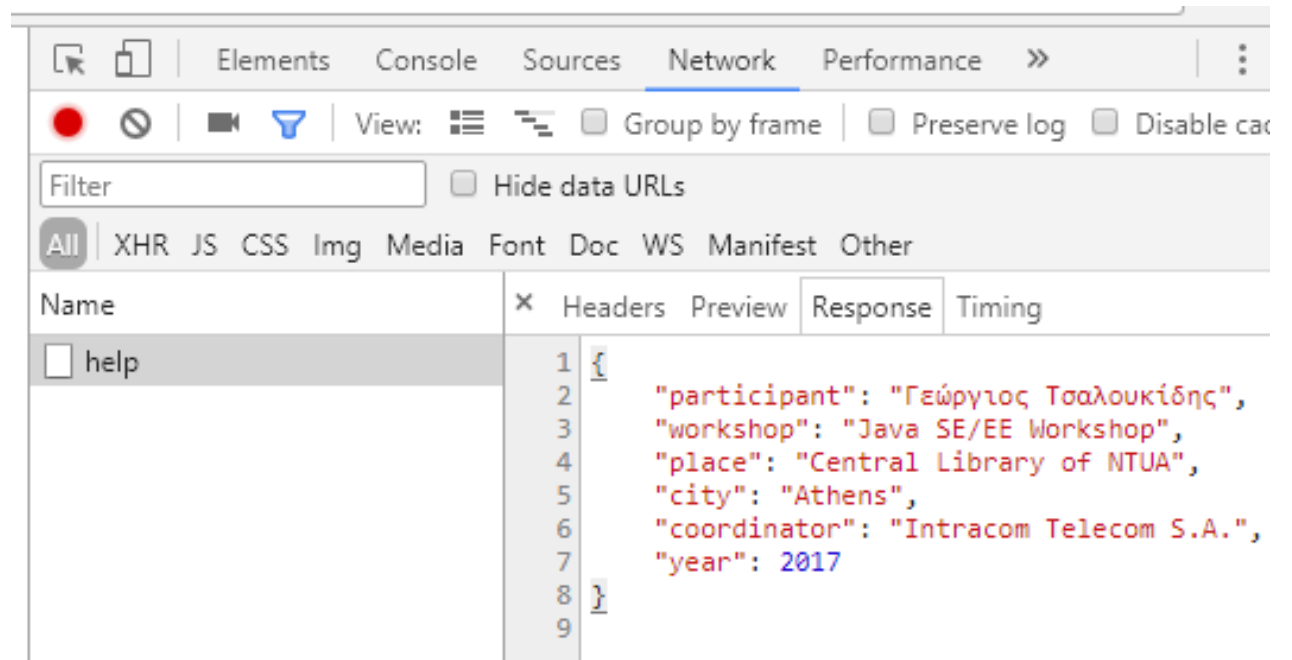
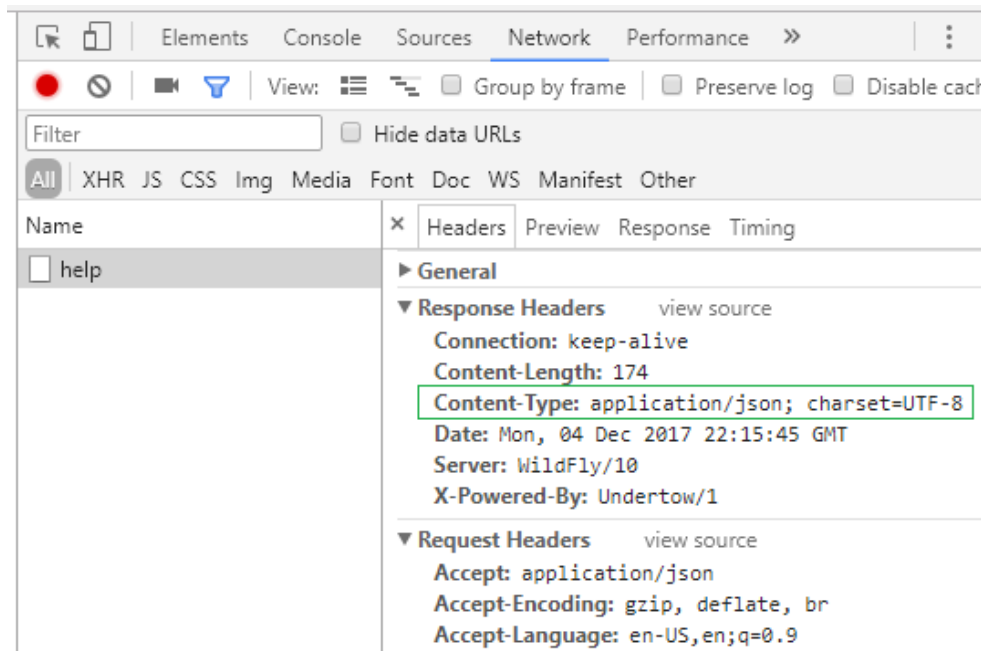
- Edit file **src/main/java/com/intracom/ems/workshop/HelpServlet.java**, find string “5th Exercise” and write your code for returning back a string in JSON format. The string should contain the following fields: “participant”, “workshop”, and “year”.
- Deploy your changes, reload the page and execute the Help action.



- This is definitely not the outcome we expected.

Fifth Exercise: Send response in JSON format

- However If we check the Content-Type and the body of Response from Developers Tools we see that server did what it had to do.



- The problem is that jQuery recognized the Content-Type of response and transform it to a JavaScript Native object. In the next exercise we will see how to extract the information.

- We create a convenient method that will quote all fields and all string values:

```
private String quoteStr(String value) {  
    return "\"" + value + "\"";  
}
```

- We write the main code:

```
PrintWriter pw = null;  
try {  
    resp.setCharacterEncoding("UTF-8");  
    pw = resp.getWriter();  
} catch (IOException exc) {  
    resp.setStatus(500);  
    return;  
}  
resp.setContentType("application/json");
```

Continuing

```
pw.println("{");
pw.println("\t" + quoteStr(PARTICIPANT_FIELD) + ": " + quoteStr(participantName) + ",");
pw.println("\t" + quoteStr(WORKSHOP_FIELD) + ": " + quoteStr(WORKSHOP_VALUE) + ",");
pw.println("\t" + quoteStr(PLACE_FIELD) + ": " + quoteStr(PLACE_VALUE) + ",");
pw.println("\t" + quoteStr(CITY_FIELD) + ": " + quoteStr(CITY_VALUE) + ",");
pw.println("\t" + quoteStr(COORDINATOR_FIELD) + ": " + quoteStr(COORDINATOR_VALUE) + ",");
pw.println("\t" + quoteStr(YEAR_FIELD) + ": " + YEAR_VALUE);
pw.println("}");
```

- The new line character after each field/value pair is optional.
- Do not forget the comma “,” separator between each field/value pair.

Sixth Exercise: Handle JSON response

- Now is time to handle the JSON response.
- Open file **src/main/webapp/js/core/misc-menu-factory.js**, find string “6th Exercise” and write the code that will access the fields of the JSON object. Remember to check the Context-Type of the Response header to decide if the new code should run.
- Redeploy your changes and check if the displayed message is the same as before.

- Create a new javascript function that will capitalize the first character.

```
function capitalizeFirstChar(value) {  
    return value.charAt(0).toUpperCase() + value.slice(1);  
}
```

- Now write the main code:

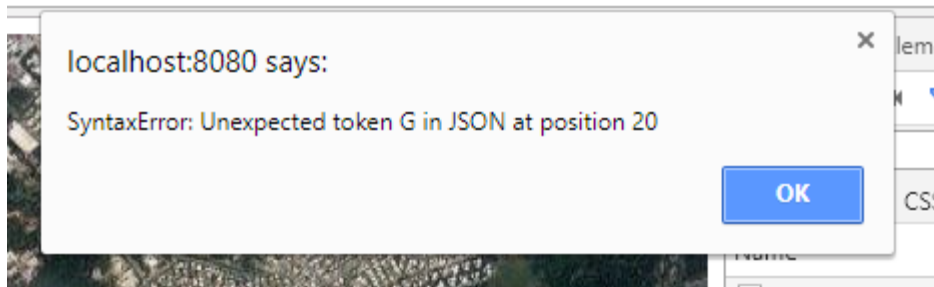
```
var fields = ["participant", "workshop", "year"];  
var outputFromJsonString = "";  
if(contentType.indexOf("text/plain") !== -1) {  
    alert(data);  
}
```

Continuing

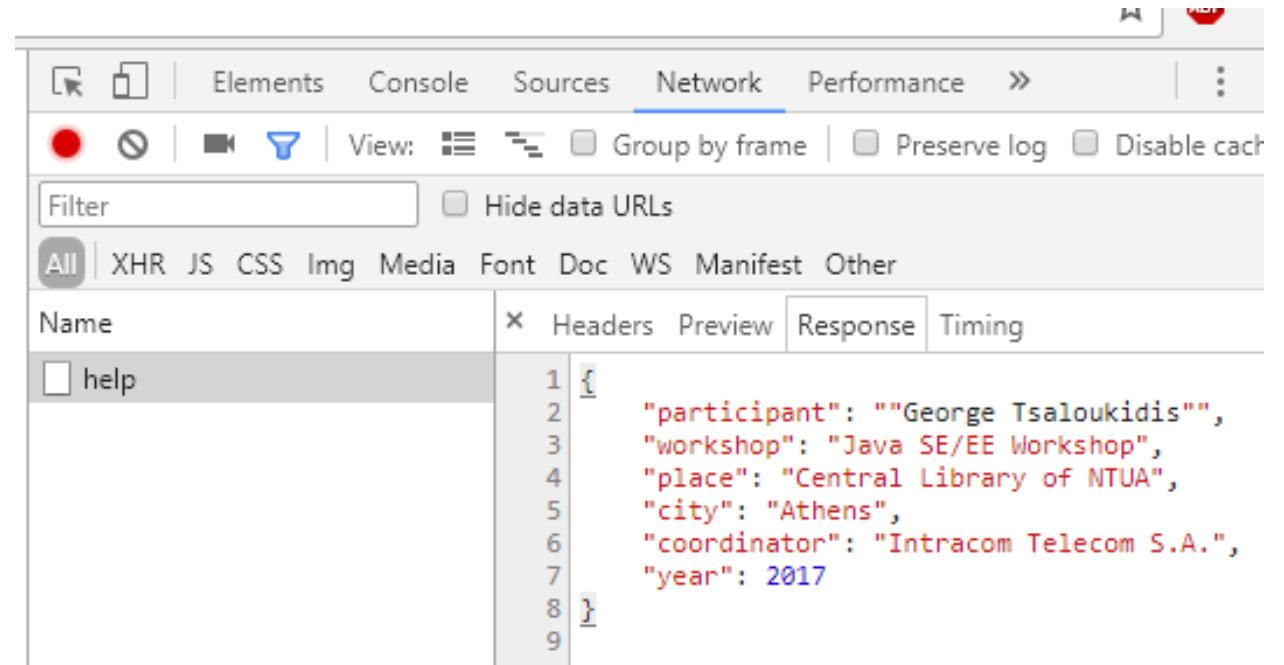
```
else if(contentType.indexOf("application/json") !== -1) {
    fields.forEach(function(value) {
        if(outputFromJsonString !== "") {
            outputFromJsonString += "\n";
        }
        outputFromJsonString += capitalizeFirstChar(value) + ": " + data[value];
    });
    alert(outputFromJsonString);
}
else {
    alert("Not expected Content-Type: " + contentType);
}
```

Common error when we use JSON or XML inputs.

- Execute the Help command and write into input box, your full name in double quotes (“), i.e. “John Smith”.
- We get a message that our JSON string is invalid.



- Notice the value of “participant” field. The “ character should have been escaped.
- The lesson from the above problem is that we should not write JSON or XML strings without using a well tested library.



- Jackson is a java library that help us to create well-formed JSON strings. It can also serialize a POJO to JSON and the reverse. Below you can find some examples:
- Create a JSON string piece by piece:

```
ObjectMapper mapper = new ObjectMapper();
ObjectNode rootObj = mapper.createObjectNode();
rootObj.put("firstName", "John");
rootObj.put("lastName", "Smith");
rootObj.put("age", 45);
ObjectNode wifeObj = rootObj.putObject("wife");
wifeObj.put("age", 38);
ArrayNode childrenArray = rootObj.putArray("children");
ObjectNode child1Obj = childrenArray.addObject();
ObjectNode child2Obj = childrenArray.addObject();
String jsonString = rootObj.toString();
```

- Parse a String, InputStream, e.t.c. and access its fields:

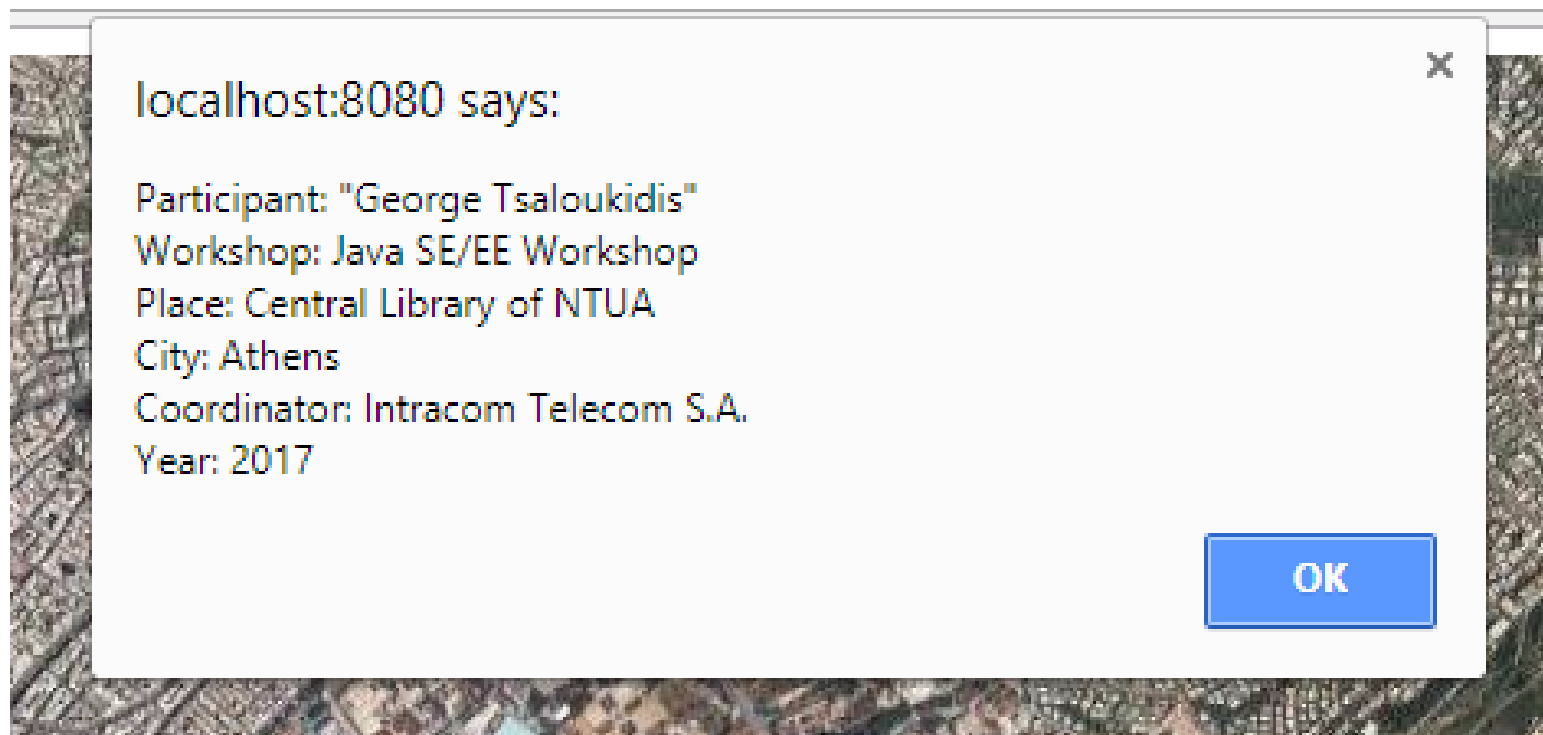
```
ObjectMapper mapper = new ObjectMapper();
ObjectNode rootObj = (ObjectNode)mapper.readTree(jsonString);
String firstName = rootObj.path("firstName").asText();
int age = rootObj.path("age").asInt();
ArrayNode childrenArray = (ArrayNode)rootObj.path("children");
```

- Serialize a POJO object:

```
class X { String name; }
X x = new X();
x.name = "George";
ObjectMapper mapper = new ObjectMapper();
JsonNode jsonNode = mapper.valueToTree(x);
String jsonString = jsonNode.toString();
```

Seventh Exercise: Use Jackson library

- Now we will re-factor code we have written for 5th Exercise in order to make use of Jackson library. Code for Exercise 5 was in **src/main/java/com/intracom/ems/workshop/HelpServlet.java** file.
- Deploy your changes and re-execute the test with “ characters. This time no error.



```
ObjectMapper mapper = new ObjectMapper();
ObjectNode rootObj = mapper.createObjectNode();
rootObj.put(PARTICIPANT_FIELD, participantName);
rootObj.put(WORKSHOP_FIELD, WORKSHOP_VALUE);
rootObj.put(YEAR_FIELD, YEAR_VALUE);
pw.print(rootObj.toString());
```

- The next thing that we will add in our application is a backend (database) to hold the data that we will create.
- Since we have some limitations from the environment we will use a in-memory database that WildFly already contains.
- We will create the tables we need and put the initial data by using a ServletContextListener.
- It is an object that receives notifications about the start up or shutdown of a Web Application.
- We can register an instance of ServletContextListener to WebContainer by either:
 - define its existence in **WEB-INF/web.xml**,
 - use the **@WebListener** annotation, or
 - call the addListener method of ServletContext.
- Open file **src/main/java/com/intracom/ems/workshop/AppContextListener** and study the code.

- The first thing we can see is the following definition:

```
@Resource(lookup="java:jboss/datasources/ExampleDS")  
private DataSource dataSource;
```

- @Resource is an annotation for binding values to variables.
- The **lookup** value is a JNDI lookup name. In a nutshell JNDI is a repository which keeps references on objects and their associated lookup name.
- DataSource is a component of an Application Server that handles database connections.
- Annotation is a feature introduced to Java Platform, Standard Edition 5 and since then it has been used about anything. It is a special kind of information (metadata) that can be accessed at compile time and/or running time. This is control by a special attribute the Annotation type contains.
- Annotations can be put on definition of classes, methods, variables, method parameters and packages.

- Something else we notice is that strange **try** block.

```
try (Connection con = dataSource.getConnection()) {  
    ...  
}
```

It is a feature of Java Platform Standard Edition version 7. That statement instructs JVM to call the `close()` method of `java.sql.Connection` object we create inside, at the end of try block. Otherwise we would have to call that function on our one.

- It is good to make usage of that format because if we forget to call that function, we will have leak of resources.
- Another point for that format is that allows us to write prettier code. Compare the following two blocks of code:

Things to notice in that file: Special try block

```
Connection con = null;
Statement stmt = null;
try {
    con = getConnection();
    stmt = con.createStatement();
    // Use stmt
}
finally {
    if(stmt != null) {
        try {
            stmt.close();
        } catch(SQLException exc) {
        }
    }
    if(con != null) {
        try {
            con.close();
        } catch(SQLException exc) {
        }
    }
}
```


Things to notice in that file: Special try block

```
try(Connection con = getConnection(); Statement stmt = con.createStatement();) {  
    // Use stmt  
}
```

- Which piece of code do you prefer?
- Only variables of `java.lang.AutoCloseable` type can be defined there.

- Notice the usage of `javax.transaction.TransactionManager`.
- This is also part of Java EE specification. Application Servers should provide an implementation.
- It allows us to group multiple actions as one entity that will succeed or fail as a whole. That means that any steps executed successfully before a failed one, should be rolled-back (canceled).
- Since we execute multiple sql commands we want either all to be executed or none of them.
- A transaction begins by calling the `start()` method of `TransactionManager` and ends by calling either its `commit()` or `rollback()` method.

- In this exercise we will simply enable an already written ServletContextListener.
- Open file **src/main/java/com/intracom/ems/workshop/AppContextListener.java** and study its code.
- Add a `System.out.println("*****");` at the beginning of `contextInitialized()` method.
- Deploy your change on wildfly. No message is printed.
- Open file **src/main/webapp/WEB-INF/web.xml**, find string "8th Exercise" and uncomment the **<listener>**.

```
<listener>  
    <listener-class>com.intracom.ems.workshop.AppContextListener</listener-class>  
</listener>
```
- Deploy your app once more. Check the logs of WildFly on its running console. You should see that line.

- As we saw earlier, when we write a Servlet method we have to set a number of different headers. With the advent of annotations that need has gone.
- JAX-RS technology we allow us to write our code in good old POJO classes and annotate them in order to indicate to Web Container their special use.
- JAX-RS is not only that. It has been created to provide a way to implement Web Services. However we will not examine that functionality on our Workshop.
- It defines the following Annotations:
 - `@Path` specifies the relative path for a resource class or method.
 - `@GET`, `@PUT`, `@POST`, `@DELETE` and `@HEAD` can be put only on methods and specify the HTTP request type the resource (method) handles.
 - `@Produces` specifies the response Internet Media Types.
 - `@Consumes` specifies the accepted request Internet media types.
- The above annotations allows us to have many methods for the same requested URL.

- For example

```
@Path("/help")  
@GET  
public String getHelp1() {}
```

```
@Path("/help")  
@POST  
@Produces("plain/text")  
public String getHelp2() {}
```

```
@Path("/car")  
@POST  
@Produces("application/json")  
public String getHelp3() {}
```

- Does the above methods ring any bell?

- A Web Application has to request support for JAX-RS from Container. This can be done with many ways.
- We chose to use the good **WEB-INF/web.xml**.

```
<servlet-mapping>  
    <servlet-name>javax.ws.rs.core.Application</servlet-name>  
    <url-pattern>/rest/*</url-pattern>  
</servlet-mapping>
```

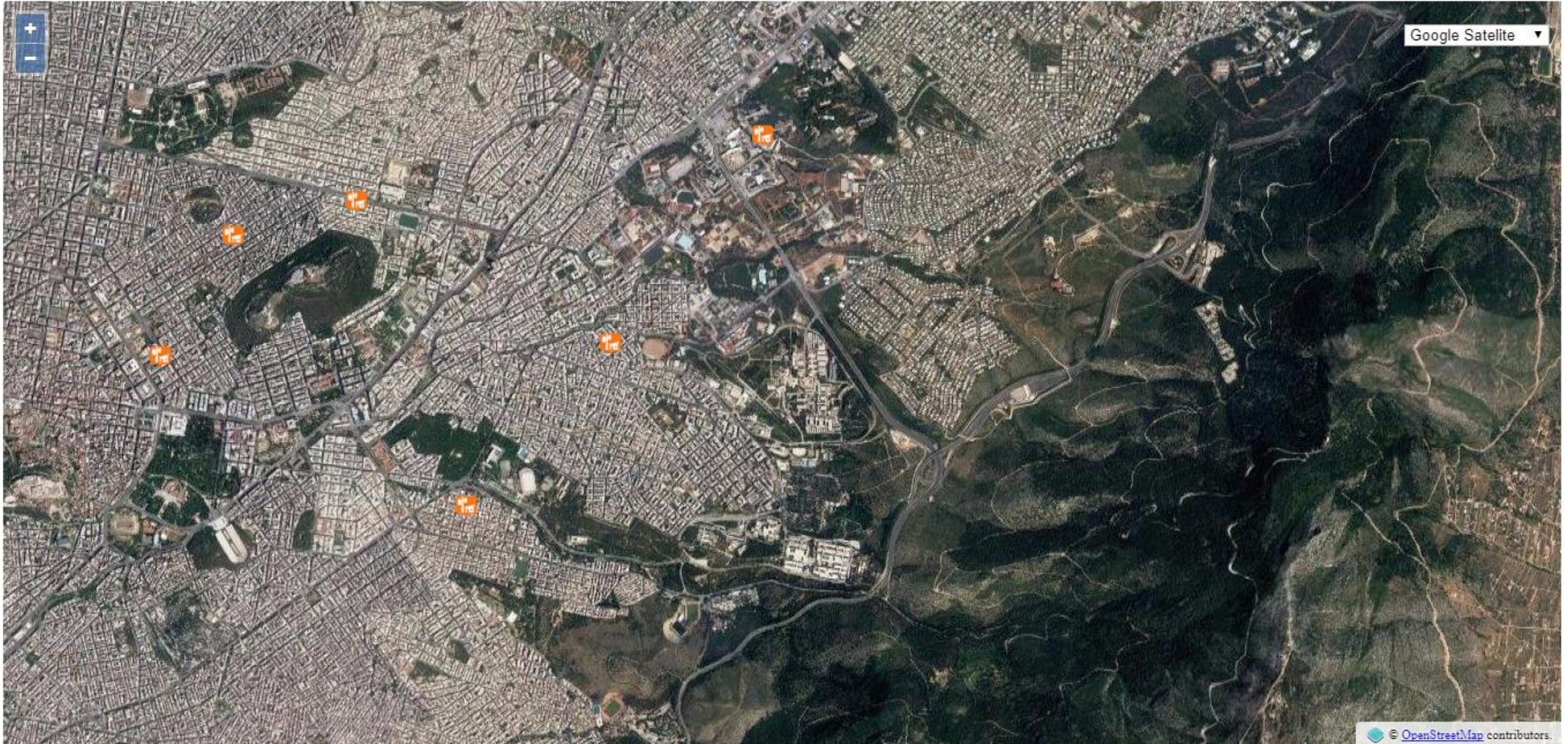
- In this exercise we will enable the JAX-RS specific servlet.
- :

```
<servlet-mapping>  
  <servlet-name>javax.ws.rs.core.Application</servlet-name>  
  <url-pattern>/rest/*</url-pattern>  
</servlet-mapping>
```
- Save change and deploy WAR on wildfly.

Ninth Exercise: Perform the first JAX-RS call

- Since our database has some data, it is the time to retrieve them by emitting JAX-RS calls.
- But before starting writing some code it is good to enable JAX-RS. So open file **src/main/webapp/WEB-INF/web.xml**, find string “9th Exercise” and uncomment the servlet mapping for it.
- Open file **src/main/webapp/js/core/map.js**, find string “9th Exercise” and remove the comment block in order to enable the ajax() call. Check the calling URL.
- Open file **src/main/java/com/intracom/ems/workshop/ElementResource** and study its getElements() method.
- Pay attention to Annotations and the usage of Jackson library.
- Deploy your change and reload page on Browser. What do you see?

Ninth Exercise: Result



Tenth Exercise: Set Ports in the Ne objects

- Now, you will add code in `ElementResource.getNes()` method, to retrieve the ports for each Ne.
- Open file **`src/main/java/com/intracom/ems/workshop/ElementResource`**, find string "10th Exercise" and complete the code. Retrieve all Ports for each Ne from the database and assign them to their Ne.
- Deploy your code and reload the browser page. The elements for the icons should have been changed.
- Study function `loadFeatures` in **`src/main/webapp/js/core/map.js`**. Could you find the code that changed the icons?

Tenth Exercise: Result




```
private List<Ne> getNes() throws SQLException {
    DataSource dataSource = WorkshopToolkit.inst().getDataSource();
    try(Connection con = dataSource.getConnection();
        Statement stmt = con.createStatement();
        ResultSet res = stmt.executeQuery(
            "SELECT element.ip, lon, lat, port FROM element, port "
            + "WHERE element.ip=port.ip ORDER BY element.ip, port")) {
        List<Ne> list = new ArrayList<>();
        Map<String, Ne> map = new HashMap<>();
        while(res.next()) {
            String ip = res.getString(1);
            Ne ne = map.get(ip);
```

Continuing

```
        if(ne == null) {
            double lon = res.getDouble(2);
            double lat = res.getDouble(3);
            ne = new Ne();
            map.put(ip, ne);
            ne.setIp(ip);
            ne.setLon(lon);
            ne.setLat(lat);
            ne.setPorts(new LinkedList<String>());
        }
        String port = res.getString(4);
        ne.getPorts().add(port);
        list.add(ne);
    }
    return list;
}
```

- In this exercise we will load the links in order to show them on the map.
- Open file **src/main/webapp/js/core/map.js**, find string “11th Exercise” and write the code for function loadLinks().
- That function should requests the links from server and add them on the map. Pay attention on what information we need to retrieve.
- You have also to define a new ol.style.Style.
- IMPORTANT: this method is not called from anywhere. You have to find the appropriate place from where you will call it. You may have to change slightly function loadElements(). That method accepts as argument the array of loaded elements. Can you figure why?
- Now open file **src/main/java/com/intracom/ems/workshop/LinkResource** and implement the server method.



Twelfth Exercise: Enable element specific actions

- Here, we will add element specific actions on our context menu.
- Open file **src/main/webapp/js/core/element-menu-factory.js**, find string “12th Exercise and uncomment the following line of code:
`contextMenu.registerItemFactory(factory, 1);`
- Deploy your change, reload page and open context menu on map. Do you see another option?



- The new menu item will be used to add a new Element in our system. It will use a certain ip and emit an AJAX request to the server. In case of a successful addition it will refresh the elements and links on the map.
- Before we execute that action we have to write the server side code.
- The first part of the server that will receive the request will be a JAX-RS resource method. That could be the only one too. However since we have to write to database we have to ensure that we will not break the integrity of our data. Since we have to insert data in two tables we have to include insertions to a single transaction.
- One way to achieve that is the usage of TransactionManager as we saw in file **`src/main/java/com/intracom/ems/workshop/AppContextListener`**.
- The other way is to use Enterprise Java Beans.

- In the previous exercise we added another one menu item on GUI. However we should write the corresponding server code that will execute that action.
- EJB specification is one of the first technologies of Java EE. It was used to accept calls from remote clients, in the old days where web application was not fashion. Of course nowadays we have Web Applications. However one of their features is that they are Transaction aware objects.
- There are three different kinds of EJBs
 - Session Beans
 - Entity Beans
 - Message Driven Beans
- In this WorkShop we will examine only the Session Beans and specifically the Stateless Session Beans.
- A Session Bean is a special entity which lives in the EJB Container, which controls its lifecycle.
- EJB specification 3.0 allows the definition of an EJB with annotations.

- In order to create a Session Bean we have to create a class that will extend a specific java **interface**. When an entity, i.e. for example a JAX-RS resource, want to make usage of an EJB it will obtain a reference to a Java Proxy Object of that interface. This is done through JNDI.
- The implementation class has to be annotated with either @Stateless or @Stateful annotation. It can also annotated with the @Remote or @Local annotation. This first one is used when we want to access the EJB from a remote client, while the second when we don't want to expose its functionality to the outside world.
- Lets see an example. Open file **src/main/java/com/intracom/ems/workshop/ElementSessionBean.java**.

@Stateless

@Local

```
public class ElementSessionBean implements ElementSession {  
}
```

So we see that is a Stateless Session bean that provides implementation for the methods of ElementSession in the local entities.

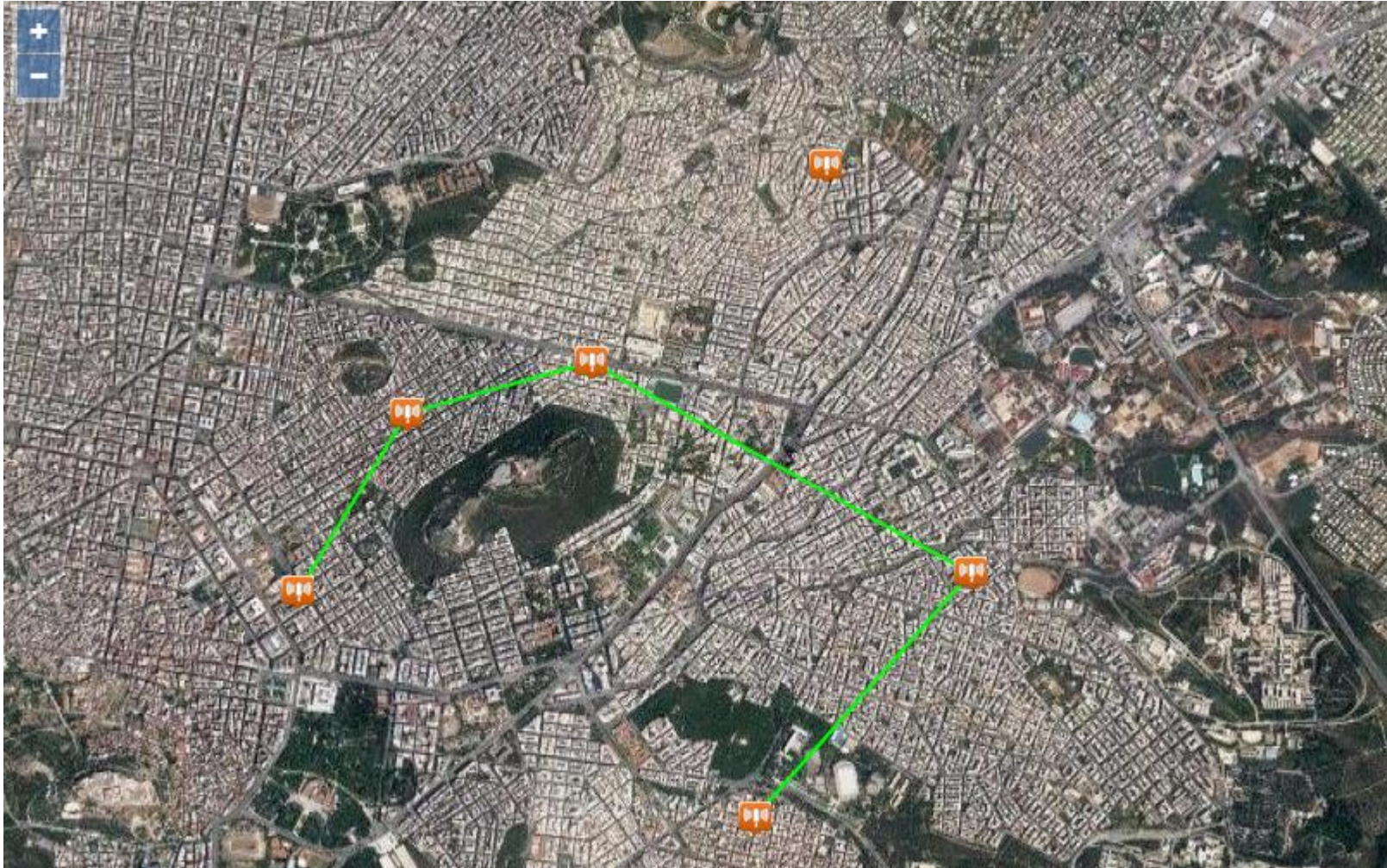
- Each method that is implemented from interface, is a special method, namely it is executed in a special environment.

- Due to that environment, EJB methods can handle transactions.
- Spec defines the *javax.ejb.TransactionAttribute* annotation that can accept a number of values and it controls the Transactional behavior of the EJB.
- Those values are:
 - *TransactionAttributeType.MANDATORY*
 - *TransactionAttributeType.REQUIRED*
 - *TransactionAttributeType.REQUIRES_NEW*
 - *TransactionAttributeType.SUPPORTS*
 - *TransactionAttributeType.NOT_SUPPORTED*
 - *TransactionAttributeType.NEVER*
- If a method is not annotated with *TransactionAttribute*, the *REQUIRED* policy is enforced.
- Only that methods that are defined in the interface are executed in the EJB environment.

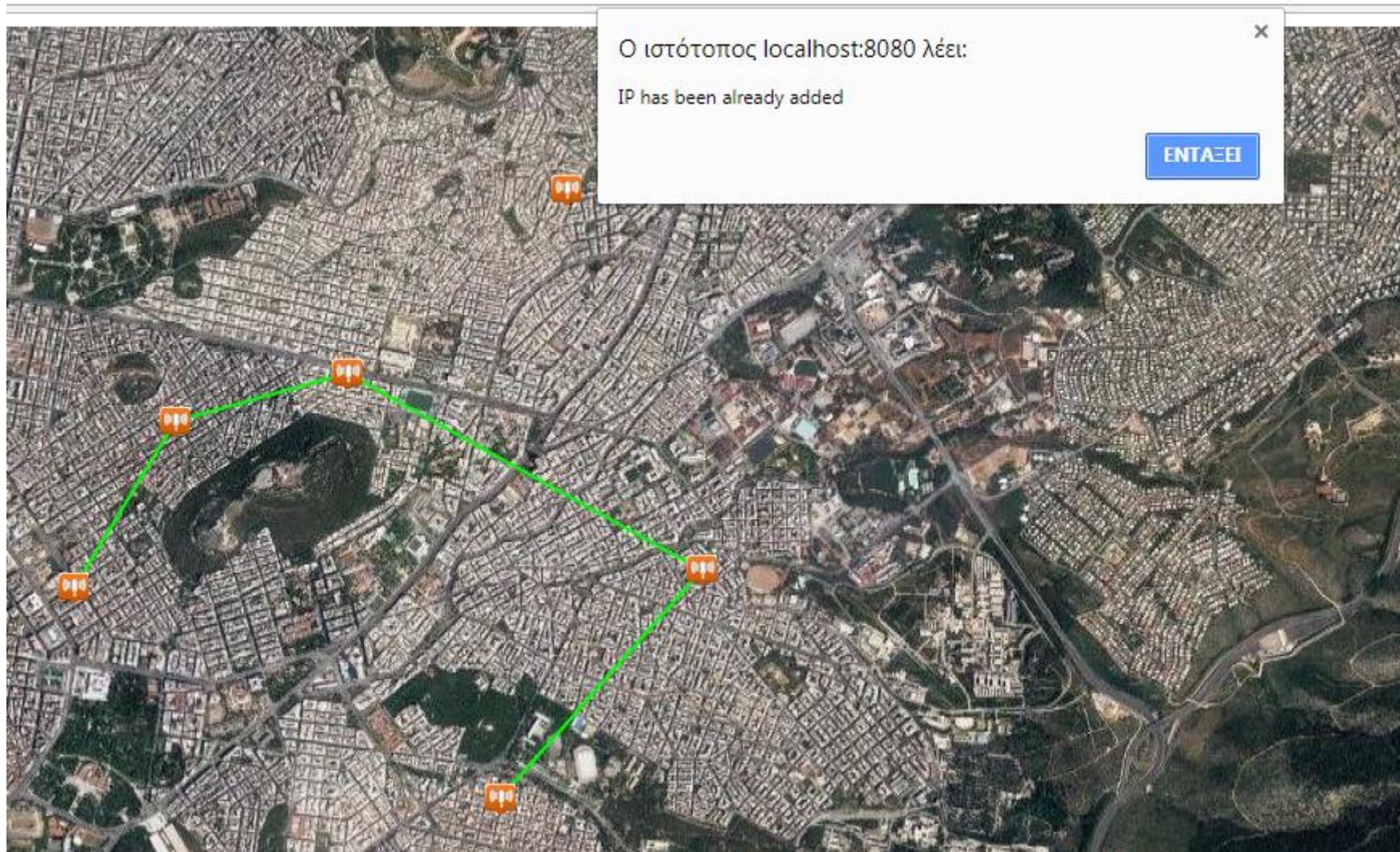
Thirteenth Exercise: Add an element in database

- At this exercise we will complete the code of method `addElement()` which is defined in `src/main/java/com/intracom/ems/workshop/ElementSessionBean.java`
- You have to write code that:
 - check if the same IP already exists in the database. In that case you will throw a `AppLogicException()` with the appropriate message.
 - Otherwise, code has to execute all the necessary insertions for that element.
- Redeploy your code and execute the “Add Element” action. The element will be added on the map in place you right click.
- Execute the “Add Element” once more. In that case you must see a message that element is already added.

Thirteenth Exercise: Executing “Add Element” for first time



Thirteenth Exercise: Executing “Add Element” once more




```
ResultSet res = stmt.executeQuery(
    "SELECT ip from element WHERE ip='" + ip + "'");) {
if(res.next()) {
    throw new AppLogicException("IP has been already added");
}
stmt.execute("INSERT INTO element(ip, lon, lat) VALUES ("
    + "'" + ip + "', " + lon + ", " + lat + ")");
stmt.execute("INSERT INTO port(ip, port) VALUES ("
    + "'" + ip + "', 'ODU 1')");
stmt.execute("INSERT INTO port(ip, port) VALUES ("
    + "'" + ip + "', 'ODU 2')");
```

thank you

For more information, visit
www.intracom-telecom.com



INTRACOM
TELECOM

