



# EoT Assignment Java Mashup Documentation

Visualising Tweets in Google Earth  
Dynamic Digital Mapping

Link to our repository on GitLab: <https://git.sbg.ac.at/s1080384/EoT/-/blob/main/README.md>

Link to our Wiki Page: <https://git.sbg.ac.at/s1080384/EoT/-/wikis/End-of-Term-Assignment>

Practice: Software Development

Assoz. Prof. Dr. Bernd Resch

SS 2021

16 July 2021

Edah Sahinovic

edah.sahinovic@stud.sbg.ac.at

12047186

Katharina Wöhs

katharina.woehs@stud.sbg.ac.at

01423067

Christina Zorenböhmer

christina.zorenboehmer@stud.sbg  
.ac.at

12040947

# General Introduction

For both practical and presentation purposes, we decided to manage and document our work on GitLab in addition to this documentation. GitLab allowed us to view and discuss our code and see how the project came together. Please take a look at our GitLab project and wiki where you can also find all .java files, code snippets, and console outputs: <https://git.sbg.ac.at/s1080384/EoT/-/blob/main/README.md>

For this assignment we agreed on two aims, in addition to the assignment instructions:

## ➤ Aim 1: Modularity

Since this assignment contains a variety of steps, it would lead to an extremely long and complex code if we designed it so that all the code was contained in a single class. Instead, after spending some time discussing our plan of action, we quickly opted for a more modular approach. This way, we can easily and intuitively dissect the programme into its sub-components. This division of sub-tasks made it much easier for us to keep a clear structure in the programme and to easily jump into any of the sub-components if we needed to work on that part. In this manner, we created one main executable class (GoogleEarthTweetMapper.java) that will bring all the individual steps together.

## ➤ Aim 2: User Input and Experience

Although it was not technically required in the assignment instructions, we wanted to allow the user to interact with the programme in a few key steps. With this, we also had the aim of enhancing the user experience. We tried to come up with simple and friendly texts that would inform the user of what is happening. For example, the programme starts with a greeting and an explanation of what will happen.

Specifically, we wanted to allow user input for the following decisions:

1. Allow the user to either accept the default directory or set a new one
2. Ask the user if they would like to see additional information on the WMS while the programme is running
3. Ask the user if they would like to proceed to launch Google Earth

## Concept

Figure 1 visualises our concept for this project. As you can see, we have one main, executable class called **GoogleEarthTweetMapper** and several non-executable classes whose methods are called from GoogleEarthTweetMapper. This division allowed us to work in a modular fashion, as stated in our aims. The arrows indicate what the program interacts with while calling on the specific non-executable classes and its methods (e. g. local tweet csv file).

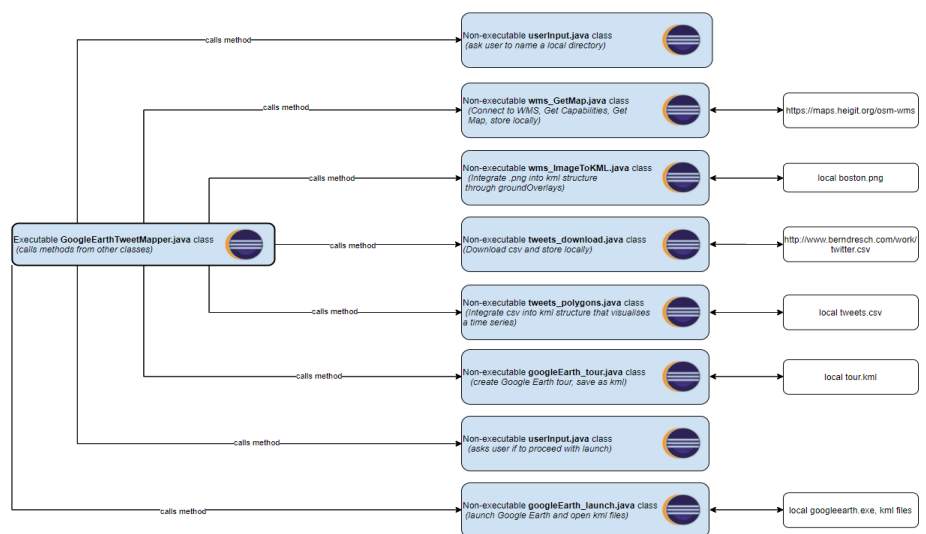
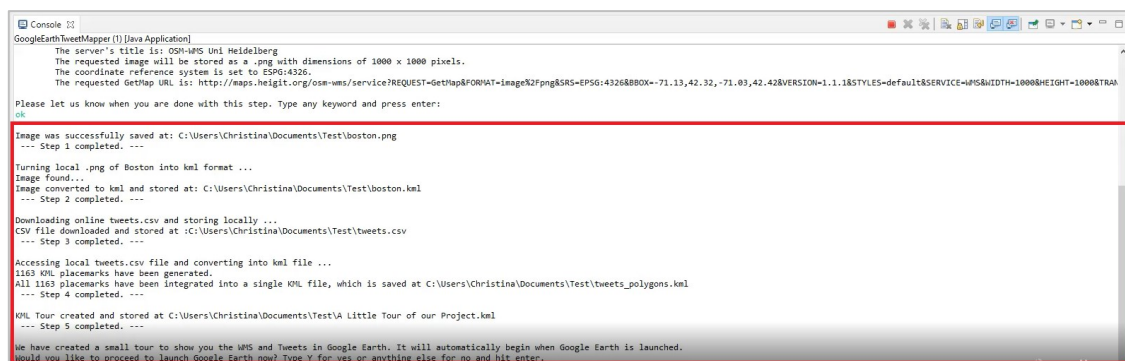


Figure 1: Project Concept

# Main class

## 1. GoogleEarthTweetMapper.java

Although the main class could not be completed until we had created all other classes, we will begin with this class since it provides a neat summary of how all components for our project come together. As stated above, we aimed to make the whole programme run smoothly without the user having to figure out which classes to execute and in which order. We hope that it may demonstrate how we applied our newly obtained knowledge about modularity. The main class calls all the other classes and their methods. Step by step, we see how the programme is proceeding and we can easily see errors in the console. Since every successful execution ends with a printed line stating "Step x completed." in the console, we know exactly where we are in the programme and where to look for possible errors or mistakes in the code. This control mechanism and the modular classes can therefore help us work out problems effectively and efficiently since we know where to start looking. Also, in the main class, we declare the main variables, which are used across all classes - this means that in other classes we call these variables. Overall, for the main class we strove for neatness and clarity, which you can see in the main variables listed at the top and the calls listed for each class (in the following referred to as 'sub-classes') and its respective methods in the body of the main method in the main class. That's about it for the main class - all the other things happen in the sub-classes in the background. The user only interacts with the main class and its console, though, to keep it simple and user-friendly.



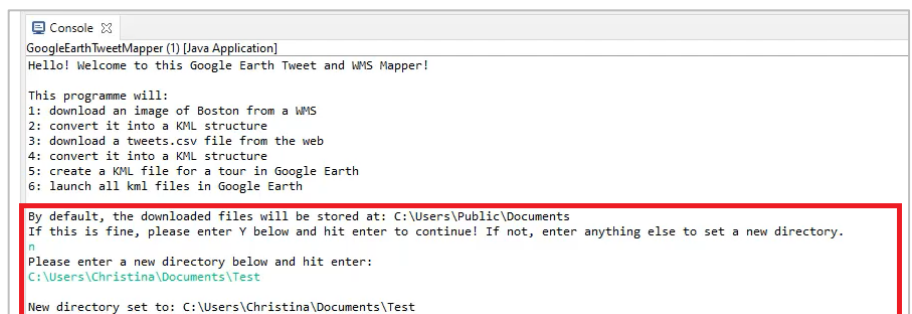
```
GoogleEarthTweetMapper (1) [Java Application]
The server's title is: OSM-WMS Uni Heidelberg
The requested image will be stored as a .png with dimensions of 1000 x 1000 pixels.
The coordinate reference system is set to EPSG:4326.
The requested GetMap URL is: http://wms.heigit.org/osm-wms/service?REQUEST=GetMap&FORMAT=image%2Fpng&SRS=EPSG:4326&BBOX=-71.13,42.32,-71.03,42.42&VERSION=1.1.1&STYLES=default&SERVICE=WMS&WIDTH=1000&HEIGHT=1000&TRANSPARENT=true
Please let us know when you are done with this step. Type any keyword and press enter:
n
Image was successfully saved at: C:\Users\Christina\Documents\Test\boston.png
--- Step 1 completed. ---
Turning local .png of Boston into kml format ...
Image found ...
Image converted to kml and stored at: C:\Users\Christina\Documents\Test\boston.kml
--- Step 2 completed. ---
Downloading online tweets.csv and storing locally ...
CSV file downloaded and stored at: C:\Users\Christina\Documents\Test\tweets.csv
--- Step 3 completed. ---
Accessing local tweets.csv file and converting into kml file ...
1163 KML placemarks have been generated.
All 1163 placemarks have been integrated into a single KML file, which is saved at C:\Users\Christina\Documents\Test\tweets_polygons.kml
--- Step 4 completed. ---
KML Tour created and stored at C:\Users\Christina\Documents\Test\A Little Tour of our Project.kml
--- Step 5 completed. ---
We have created a small tour to show you the WMS and Tweets in Google Earth. It will automatically begin when Google Earth is launched.
Would you like to proceed to launch Google Earth now? Type Y for yes or anything else for no and hit enter.
n
```

Figure 2: Main Class, GoogleEarthTweetMapper, console output

## Sub-classes and methods

### 2. userInput.java - .askUser1()

The .askUser1() method is called at the very beginning of the main class to set the directory. As we found out (learning-by-doing), the directory needs to be declared at the very beginning since multiple other variables incorporate the directory, and all the following classes and methods build upon it. In this method, we included a default directory, to make the programme works either way. We set the default directory to "C:/Users/Public/Documents", which is given on any PC. Nonetheless, we wanted to let the user specify a different directory if they prefer that. Now, the user can communicate with the program and tell it where to store all the files. To deal with any spaces in file paths, we set the scanner method to .nextLine(), and to check if the entered file path exists we added a control mechanism in the form of an if-else-loop that confirms whether the given directory is a real one, i. e. whether it exists, or else it asks the user again to set a new directory.



```
GoogleEarthTweetMapper (1) [Java Application]
Hello! Welcome to this Google Earth Tweet and WMS Mapper!

This programme will:
1: download an image of Boston from a WMS
2: convert it into a KML structure
3: download a tweets.csv file from the web
4: convert it into a KML structure
5: create a KML file for a tour in Google Earth
6: launch all kml files in Google Earth

By default, the downloaded files will be stored at: C:\Users\Public\Documents
If this is fine, please enter Y below and hit enter to continue! If not, enter anything else to set a new directory.
n
Please enter a new directory below and hit enter:
C:\Users\Christina\Documents\Test
New directory set to: C:\Users\Christina\Documents\Test
```

Figure 3: Prompting the User to either accept the default directory or set a new one.

### 3. wms\_GetMap.java

The first major part of the project was to connect to a web map service in order to retrieve the map we later on want to integrate into the KML structure and in the following into Google Earth. To do so, we created the non-executable class `wms_GetMap.java` in which we create a `WebMapServer` object and provide a URL that points at a WMS capabilities document. To check whether the URL is valid, we create a try-catch-block for exception handling. For the `WebMapServer` object, we use the following approach for exception handling: a try-catch-block in which we check several exceptions, i. e. one if the server doesn't respond for any reason (maybe it's down) and one in case we get a `ServiceException`. In case the connection works, we included a line printing "No errors in communicating with the server" as feedback for us/the user that we know everything worked out.

The next step was to get to our map. First, we ask the client to create a `GetMapRequest` object and we configure the request object in our code by setting all the required parameters as needed, e. g. format, bounding box etc. After having set the parameters, we allow the user to view additional information about the server (see Figure 4). Here we made sure that several versions of "yes" work in the console, in case the user wants more information. We ask the user to give feedback as soon as he/she is done with reading the additional information to enhance user-friendliness. To store the image, we included file writer-methods (`Image.IO` read and `Image.IO` write). If we are successful, the programme prints the line "Image was successfully saved at: ...", otherwise the `IOException` throws as line "There was an error with writing the image" into the console. In case everything works out, we receive the line "--- Step 1 completed. ---" (see main class console screenshot, Figure 2). The image is now stored locally on the computer and we can continue with the next step: Integrating it into a KML structure.



```
GoogleEarthTweetMapper (1) [Java Application]
Hello! Welcome to this Google Earth Tweet and WMS Mapper!

This programme will:
1: download an image of Boston from a WMS
2: convert it into a KML structure
3: download a tweets.csv file from the web
4: convert it into a KML structure
5: create a KML file for a tour in Google Earth
6: launch all kml files in Google Earth

By default, the downloaded files will be stored at: C:\Users\Public\Documents
If this is fine, please enter Y below and hit enter to continue! If not, enter anything else to set a new directory.
Please enter a new directory below and hit enter:
C:\Users\Christina\Documents\Test

New directory set to: C:\Users\Christina\Documents\Test

Testing Heigit's wms service URL... No errors in communicating with the server.
Would you like additional information on the WMS service while the programme runs? Type Y for yes or anything else for no and hit enter.
yes
The WMS capabilities are being retrieved from a server called: OGC:WMS
The server's title is: OSM-WMS Uni Heidelberg
The requested image will be stored as a .png with dimensions of 1000 x 1000 pixels.
The coordinate reference system is set to EPSG:4326.
The requested GetMap URL is: http://maps.heigit.org/osm-wms/service?REQUEST=GetMap&FORMAT=image/png&SRS=EPSG:4326&BBOX=-71.13,42.32,-71.03,42.42&VERSION=1.1.1&STYLES=default&SERVICE=WMS&WIDTH=1000&HEIGHT=1000&TRANSPARENT=true&version=1.1.1&styles=default&layers=osm auto:all

Please let us know when you are done with this step. Type any keyword and press enter:
```

Figure 4: Offering the User additional information about the WMS.

The result of the WMS Connector is the following .png image of Boston:

<http://maps.heigit.org/osm-wms/service?request=GetMap&srs=EPSG:4326&bbox=-71.13,42.32,-71.03,42.42&width=1000&height=1000&format=image/png&service=wms&transparent=true&version=1.1.1&styles=default&layers=osm auto:all>

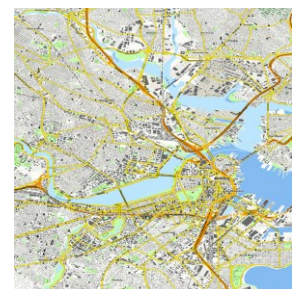


Figure 5: Boston.png

### 4. wms.ImageToKML.java

Here, we created a non-executable class `wms_ImageToKML` which contains a method `.wmsToKml()` that finds the local `boston.png` image and writes a `kml` file (containing the image) and stores it locally. First, the program informs the user what's about to happen ("Turning local .png of Boston into kml format."). Then the program tries to find the image. By using an if-else-loop, the programme checks whether the image is found, else it prints "Image not found". If found, it incorporates it into a KML format. To do so, we create a `kmlArray` which serves as a body, that we now fill with the parameters. Finally, the programme writes the file and stores it locally using a `FileWriter`. For exception handling, we again included a try-catch-block to check whether it worked out or not. In case everything works out, the line "--- Step 2 completed. ---" is printed (see screenshot of main class console, Figure 2).

## 5. tweets\_download.java

Although this step was not technically included in the assignment instructions, we decided it would be nice for the user to not worry about having to download the .csv file themselves. Therefore we created a non-executable tweets\_download.java class that contains the methods saveFileFromUrlWithCommonsIO() and downloadCSV() that access and download the .csv file from this web-URL: <http://www.berndresch.com/work/twitter.csv>. Within a try-catch-block, the program calls the method saveFileFromUrlWithCommonsIO() with the respective parameters. In case no exceptions pop up, the console prints "CSV file downloaded and stored at: ...". If successful we receive the line "--- Step 3 completed. ---" (see screenshot of main class console, Figure 2).

## 6. tweets\_polygons.java

This step proved to be the most complex one. In the end we used two methods within this non-executable tweets\_polygons.java class. The first method dynamically determines a colour-code for the placemarks depending on the time of the tweet. The second method creates a kml file that contains 3D polygons for the point locations, with pop-up boxes, time-stamps, and a colour-coded visualisation of the time of the tweet.

To extract all the relevant information from the CSV, we used a buffered reader and a while-loop that iteratively reads through the lines of the CSV file. We split each line by a semi-colon and temporarily stored the individual column values in a String Array, so that we could work with those values, e.g. to insert them into the kml placemark. We also created a kml placemark template which we filled with the column values.

A few of the key challenges in this step were:

**Creating the Time Stamp:** After some research into typical kml-conform time stamps and comparing them to "created\_at" column in the tweets.csv file, we identified a few simple changes that had to be made in order for the time to be kml conform. We needed to add in a "T" where the blank space is and add ":00" at the end. We split the content of "column[6]" of the String array, which is the "created\_at" column by " " (a blank space) and inserted a "T" in its place, and added the ending ":00".

**Creating Polygons from Points:** To solve this, we performed some simple math calculations on the latitude and long coordinates of the point locations. This required a few conversions of String to Double and then back into String, to insert the new coordinates into the kml body.

**Colouring the Polygons according to the Time of Tweet:** Here we decided to work with the difference between the first tweet's time and every other tweet's time to dynamically adjust the green colour value in a hexadecimal colour code for each placemark to give us a colour-ramp from yellow (= earliest) to red (= latest). The value for green was calculated with the two values "secondsFirstDate" (= the time of the first tweet in seconds) and "seconds" (= the time of the current tweet in seconds). By parsing the time, we are initially given milliseconds. We divided these values by 1000 to arrive at seconds, and then again by 6 to ensure all values are within a range of 256, which is required for the colour values. We then simply calculated the difference between the first and the current tweet and used that difference to set the green colour value.

## 7. googleEarth\_tour.java

Although not technically required, we thought it would improve the user experience to be guided through the results in Google Earth. We therefore, created our own KML tour and integrated it into a non-executable googleEarth\_Tour.java class that stores the tour.kml locally. We created a 1 minute 22 second long tour that automatically starts upon the launch of Google Earth: it first zooms to Boston and ends in New York City where the user can then interact with the time slider or click on the polygons for the pop-up boxes. To do so, we created a new kml file for the tour, using the FileWriter to write and store it locally. The try-catch-block tells us whether this step is successful ("KML Tour created and stored at: ...") or not ("Error writing KML file."). Finally, we get "--- Step 5 completed. ---" (see screenshot of main class console).

## 8. user\_input.java - .askUser2()

Here, we call the class user\_input.java again, but we call a different method than the one in step 2 (then: .askUser1(), now: .askUser2()). Before launching the Google Earth application, we wanted to ask the user to confirm that they want to launch the programme. For this, we simply used a scanner to prompt the user to enter some form of "yes" to launch the programme. If something else is entered, the programme ends (see Figure 6).

## 9. googleEarth\_launch.java

In this final step, we merely use a non-executable class "googleEarth\_launch" with the method .launchGoogleEarth() that launches google earth and opens all .kml files: wms\_kml, tweets\_kml and the tour. The programme again informs the user of what's about to happen. Within the method, we use the .getRuntime().exec() method to launch Google Earth and to open the kml files. If no errors appear, the line "Launch complete." and "---Step 6 completed. ---" are printed to the console (see Figure 6).

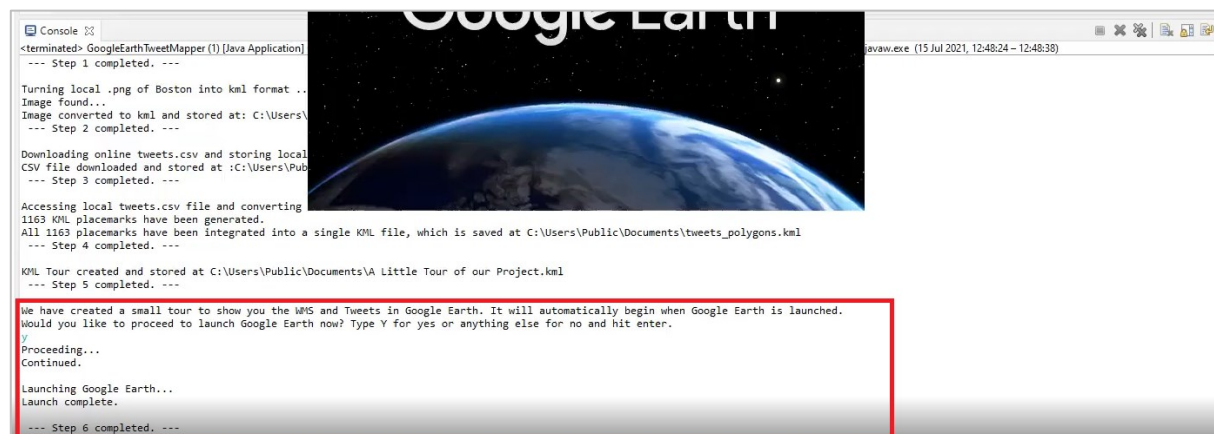


Figure 6: Asking the User if they would like to proceed with the launch of Google Earth.

## Conclusion

This project work has prompted us to apply all of our newly gained java skills, to explore ways of coordinating work in a group (GitLab was extremely useful for this), and to experience the joys of solving problems. Bringing this project together has clearly demonstrated the value of modularity in a programming environment (and in general). The flexibility we had when working on this project allowed us to get creative and incorporate our own ideas, such as the user inputs and the Google Earth tour, which we hope you will enjoy as much as we have. At the same time, the given requirements and instructions prompted us to deal with challenges that we would otherwise perhaps have avoided. In sum, this has been a challenging task, but also an extremely fun assignment.

## References:

Google Developers (2021): Touring in KML. <https://developers.google.com/kml/documentation/touring>

Google Developers (2021): KML Reference. <https://developers.google.com/kml/documentation/kmlreference>

Google Earth KML examples. <https://renenyffenegger.ch/notes/tools/Google-Earth/kml/index>