

Tommi Laukkanen


DEVELOPING WEB-SERVICES WITH GWT

Bachelor's Thesis
Information Technology

MAY 2016



DESCRIPTION

		Date of the bachelor's thesis Spring 2016
Author(s) Tommi Laukkanen	Degree programme and option Information Technology	
Name of the bachelor's thesis Developing Web-Services with GWT		
Abstract The objective of the thesis work was to further develop pedestrian counting system iGator. The development was aimed on physical cameras, counting system's infrastructure, backend servers and client-side services. The background of the project is provided in detail to explain starting point, various design decisions and the end goals. The main part of the project, the new end-customer user interface, was developed using Java, GWT and various other web technologies. The focus was on developing map based web service which shows the relative pedestrian counts in easily understood way. Used technologies and programming methods are explained in brief detail. The project was concluded in the intended timeline while achieving the main goals it was set to achieve. While minor goals changed as the project developed because of increasing knowledge of what was really needed, the main goals were kept unchanged. As the project is far from perfect, various potential development paths are explored and suggestions presented for the future developments.		
Subject headings, (keywords) Web Development, Java, GWT		
Pages 32	Language English	URN
Remarks, notes on appendices		
Tutor Reijo Vuohelainen	Bachelor's thesis assigned by Observis Oy	

CONTENTS

ABBREVIATIONS

1	INTRODUCTION	1
2	BACKGROUND	2
3	GOALS	3
4	TOOLS AND RESOURCES	5
4.1	Client Side Technologies	5
4.2	Server Side Technologies.....	7
4.3	Camera Control System	9
4.4	Otos Service	11
5	PROJECT STRUCTURE.....	12
5.1	Preparing and Installing Cameras	12
5.1.1	Gathering Camera Information.....	14
5.2	Database and DBProvider	14
5.3	Old iGator.....	16
5.4	New iGator	17
5.4.1	Client Side	18
5.4.2	Server Side.....	20
6	CONCLUSIONS	22
7	FUTURE DEVELOPMENTS.....	25
7.1	Camera Control System Changes	26
7.2	Further Data Analysis Options	27
8	BIBLIOGRAPHY	28

ABBREVIATIONS

CSS	Cascading Style Sheet
DOM	Document Object Model
GWT	Google Web Toolkit
HTML	Hyper Text Markup Language
JS	JavaScript
NTP	Network Time Protocol
ORDBMS	Object-Relational Database Management System
SQL	Structured Query Language

1 INTRODUCTION

Modern world gathers more data than ever before. Data is gathered at such rates that static representations will be obsolete the moment they get published. Also, the more data there is, less self-explanatory it tends to be, requiring good analysis and presentation to be understandable.

In this particular case the situation centres around constantly counted pedestrian traffic data. There are two major angles on this: First of all, the raw data is not very useful. Users will want to compare traffic between holidays and workdays, from specific periods and so on. The second problem is in representation: Numbers are boring, users prefer visual representation in a format that is easy to comprehend. Comparisons between crossroads must be specifically easy to draw.

The aim of this study was not to develop new solutions for these problems, but to upgrade the existing solution, a system called iGator, to modern standards and to develop a concurrent new system to run alongside the old. A company called Observis Oy, small Mikkeli based IT firm, assigned this thesis work, as Observis was also the older system's original developer and its current owner. Writing this thesis was an afterthought for already started project, which is one of the main reasons why the project does not follow normal thesis process so closely.

The next chapter deals with the background of the project, as the history of the system defined the provided infrastructure, used tools, chosen decisions, and various other details around the whole project. Chapter 3, Goals, then expands on this to explain what the project was specifically to achieve.

Chapter 4 contains various tools that were used, explaining server and client side technologies, and the infrastructure the system was to run on. The next chapter details how the tools were used and expands upon coding techniques that were chosen for the project. Various pitfalls the project faced are also revealed.

Finally, there are Chapters 5 and 6 which contain achieved results and proposed future developments. While the thesis work part of the project has been done, the project still continues. Even with all the achieved results, the system still needs maintenance and future development. And there is also potential to expanding the system to new regions, which should keep iGator alive for years to come.

2 BACKGROUND

The project was very much defined by the older version of the iGator system, which Observis was partially updating and partially replacing. The system (a.) uses video camera data to count the number of pedestrians moving on the video, (b.) stores this data in a database for further use and (c.) shows the data in an easily understandable format.

The (a.) part of the system is of older breed, and I had no need or even possibility to modify the counting algorithm, or how it sends the data to the central database. This proved problematic later, as the settings of the counting algorithm are not very self-explanatory and there was no person available that completely understood how the system worked.

While the existing (b.) part was reasonably good for what it does, there was a need to centralise the data parsing, optimize the database structure and (in the future) allow centralised source for adding new camera controllers without meddling with the project's code every time. The old interface, the part (c.), showed its age particularly badly and was not exactly the best possible tool for general overview of the data. Thus a new interface was needed. This would become the main part of this work, requiring by the far the most hours to complete.

The main difference between old and new systems was, and is, their scope. The old system had fewer cameras and each camera had less pedestrian traffic to count. The new cameras, all six of them, had far more traffic to deal with and were located neighbouring intersections. This led to greater burden for the backbone and allowed new possibilities in the interface design.

3 GOALS

The project was divided into four distinct but still rather abstract goals. As the full requirements of the system were not known before the project was well in making, these goals are described from the position of perfect hindsight, as detailing the evolving sub-goals would be exercise in futility.

Preparation: The image from the existing cameras had to be modified for the new controllers. The main addition stemmed from the fact that static mobile IP addresses are really expensive to lease. This issue was fixed by making control boxes call back home every ten minutes, thus providing their current IP address in reasonable timeframe. There was also a minor issue of installing an automatic time synchronisation system to minimize the quantity of maintenance the controllers would need later.

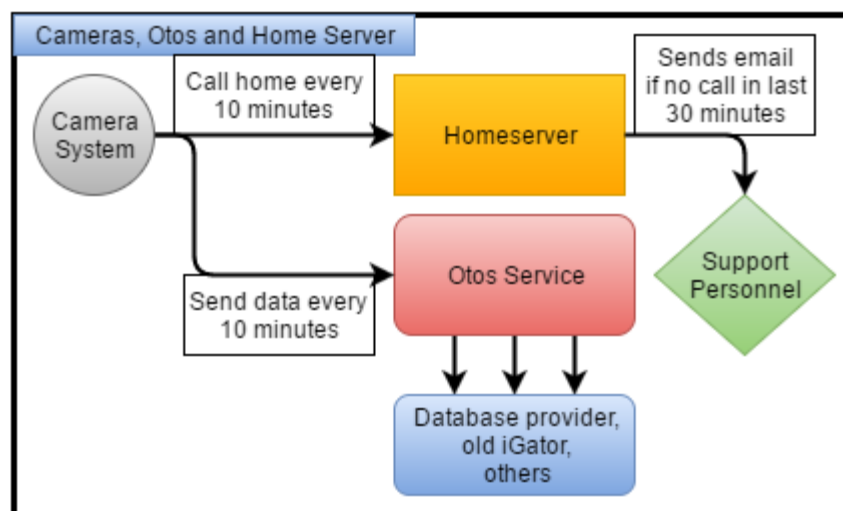


FIGURE 1. All planned callbacks

Fixing the old: The old system had various issues in it, the major one being that all earlier camera controllers were hardcoded into the system, which also led to the new cameras being hardcoded, as there was no time to make a more flexible way to add new cameras to the system. Various different user interface problems were also fixed, but the main focus on was making the old system work with new cameras, as there were plans to replace the system completely at later date.

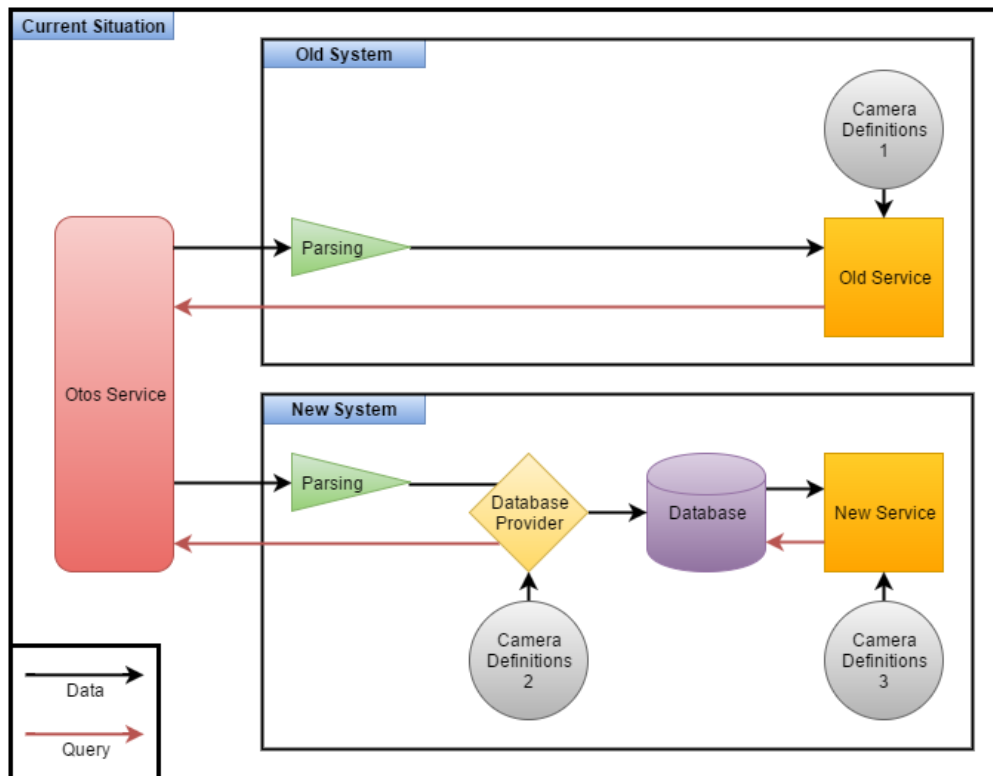


FIGURE 2. Two concurrent systems

The new database: As the old and the new systems were designed to work side-by-side, there was need to centralize the data parsing and saving the results on the Observis' own database server. The idea was to (a) make sure that the data will be constant on both systems, (b) allow performance improvements for particularly long data queries and, (c) provide a centralised controller addition which would ease future growth.

The new interface: The long term plan was to allow the old system to die in peace. Thus, there was need to develop a new system. However, as there were no resources to completely replace the old system in such a short timeline, the focus of the new system was to complement the old system instead. This was to be done by providing good overview of the data by showing the crossroads and the pedestrian data on a map.

4 TOOLS AND RESOURCES

The main programming language used was Java, with HTML, CSS and JS in a supporting role. Various libraries were also used. The main reason for choosing these technologies were that I was either familiar with the tools used in the project, or they were simple enough to be learnt quickly without causing distraction from the main task at hand.

In the terms of human resources, most of the development in the project was done by me, but I also received advice and help from other Observis employees during the project. The graphical design of the interface was done by actual graphical designer, and I got code consultation in database queries and Linux.

4.1 Client Side Technologies

The trinity of HTML, CSS and JS is the cornerstone of modern client-side web design. These three languages allow developers to define the structure, style and (client-side) scripting of the website. There are no widespread counterparts for these technologies, making them almost a mandatory part of the modern websites.

HTML (Hyper Text Markup Language) describes the contents of the document. Outside the extremely barebone websites it is used on every website. While developers can choose not to use CSS and JS, not using HTML would be like having a house without walls.

CSS (Cascading Style Sheet) is used to describe the presentation of a document written in a markup language. In most situations CSS is paired with HTML, but CSS enthusiasts have brought their own language to other settings as well. To continue with the house example, CSS would describe what the walls would actually look like (size, colour etc).

JS (JavaScript) is a scripting language that was developed for web usage. The word Java is a historical remnant at this point, as the two languages have very little to do with each other nowadays. In the house example, JS would be the button that turns on the lights.

jQuery is a library that extends JavaScript for the purpose of simplifying client side development. JavaScript is infamous for missing many useful functions and being such a widespread technology, it is slow to update as well. This has provided breeding ground for libraries like

jQuery, which make developers' tasks easier. jQuery is one of the more popular JS libraries, with 71% of the Web's top 100 000 sites using it. It is so ubiquitous that its extension libraries have their own libraries.

Leaflet is an open-source JavaScript map library. It has features like layers, zoom-levels, no need of any special server-side infrastructure and modularity through plugin support, which makes it good default JS map library for most non-trivial purposes.

Leaflet Hotline is a plugin for Leaflet which adds hotline functionality to Leaflet's native lines. Hotline is a line that shows relative quantities of the line's points with colours, and it is very useful, for example showing flows of pedestrians between intersections.

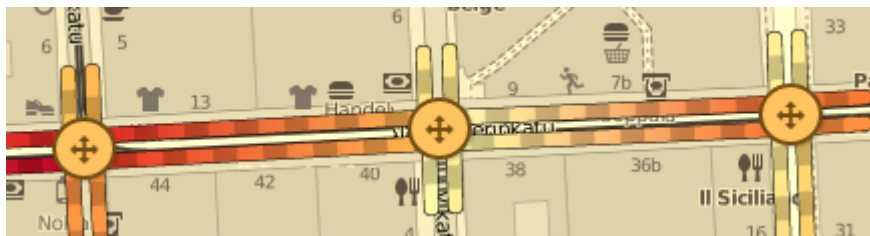


FIGURE 3. Heatlines in action: Red is more, white is less.

Chart.js is an open source HTML5-based JavaScript library that provides basic chart functionality on the client side. Being based on HTML5 instead of SVG images, Flash or other technologies it has great support on any reasonably modern browser without requiring any external plugins.

Bootstrap is a web framework for developing responsive mobile websites. It is a large collection of various useful tools, particularly for mobile development where precise cursor control and ease of keyboard writing are not facts. Of all the tools Bootstrap provides, this project only used the Timepicker function.

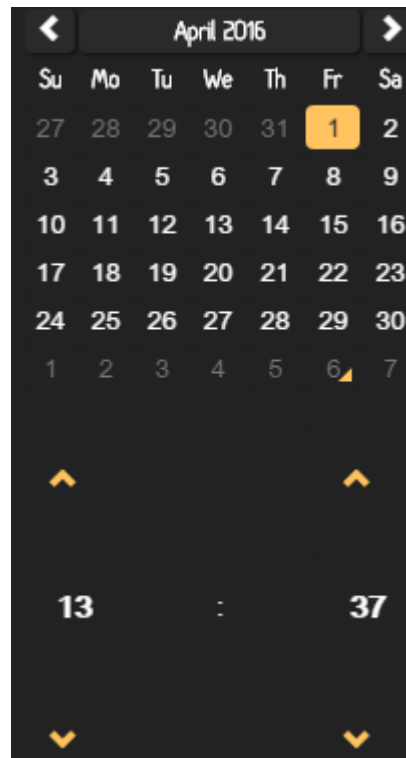


FIGURE 4. *Bootstrap timepicker*

4.2 Server Side Technologies

Java is and has been one of the more popular programming languages for years (TIOBE 2016). Java 8 design document defines the language as "...a general-purpose, concurrent, classbased, object-oriented language (Java 8 Design Document 2016)". That is, Java can be used to develop all kinds of programs which can have simultaneous instructions and the program is organised around data instead of logic.

The main reason why Java was chosen as the programming language for this project was the slight coincidence that Java was used for the older system as well. Observis also works mainly on Java, and it is also the programming language the developers were familiar with, making it a natural choice.

Google Web Toolkit (GWT) is a toolkit that provides conversion from Java to JavaScript for the purpose of frontend development, while still having the server side run on native Java. GWT has been open source since 2013, when Google finished transforming it from an internal Google project to a Google led open-source project. The most defining feature of the GWT, and the original goal it was made to provide for, is the ability to code on Java on both the client and the server side. The client side Java will be converted to JavaScript while the server side will run

on Java. This eases data transfer, as GWT can automatically handle transferring Java objects without developers needing to consider things such as encoding, different data formats or platform dependencies. This combined with strong-typing of Java provides more easily maintained code, as there is no need to handle data transformation on the client and the server sides.

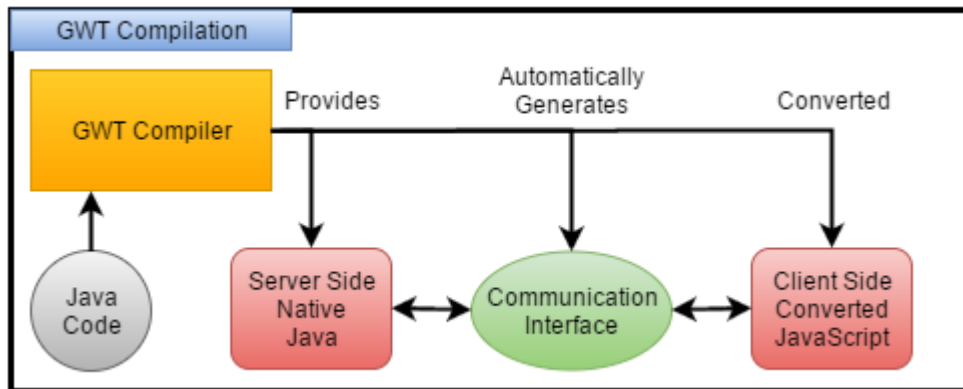


FIGURE 5. GWT in nutshell

Hibernate is a Java framework that allows object relational mapping (ORM) between Java objects and relational databases. It provides an extra abstraction layer, sparing the developers from writing specific database queries and from handling connection pools or similar things. Simply said: it provides automatic solution for a commonly faced task.

PostgreSQL is a mature Object-Relational Database Management System (ORDBMS). That is, it is a database system that implements SQL (Structured Query Language), which is the standard for databases. The main reason why it was chosen over other SQL implementations is that Observis was already running it on the production server, and the developers were familiar with it.

Spark Framework is a very minimal Java web framework. It provides basic toolset for web services, such as routing, request and response handling, and connection management. Its strengths are in quick development and deployment, which is the reason why it was chosen to be used in this project.

Apache Tomcat is mature open-source web server by Apache Software Foundation. It provides a platform for running Java Servlets and related technologies alongside basic website services. Observis had it running on the production servers making it obvious choice for the web service.

4.3 Camera Control System

The camera control system can be divided into two major parts. The first part is the camera itself, which can be any reasonably modern camera that can provide output to specific URL without requiring login credentials. The second part could be any Linux supporting, reasonably fast computer which is then used to run the processes that are needed for transforming raw video-data into movement data, which is then converted to actual numbers which are sent to the central server.

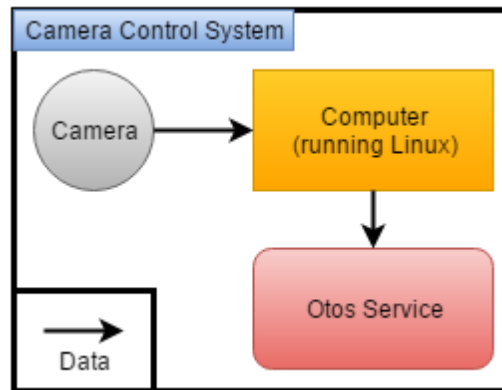


FIGURE 6. *Camera Control System*

The counting algorithm was originally developed by a company that has since stopped doing business and the algorithm was bought by another company, which did not hold the original developers in the house. These two events left that part of the project as unmodifiable, undocumented black box for this work's purposes. This proved to be problematic, as Big O notation of the algorithm was unknown and there was fears that the computer couldn't handle pedestrian masses during the peak times.

The algorithm that the camera control system uses is based on tracking moving objects in the video feed and noting when these objects cross counting lines. The algorithm then considers if the object is of a correct size: Too large and too small objects are not counted by the algorithm. There might be other variables on this, but these are not currently known.

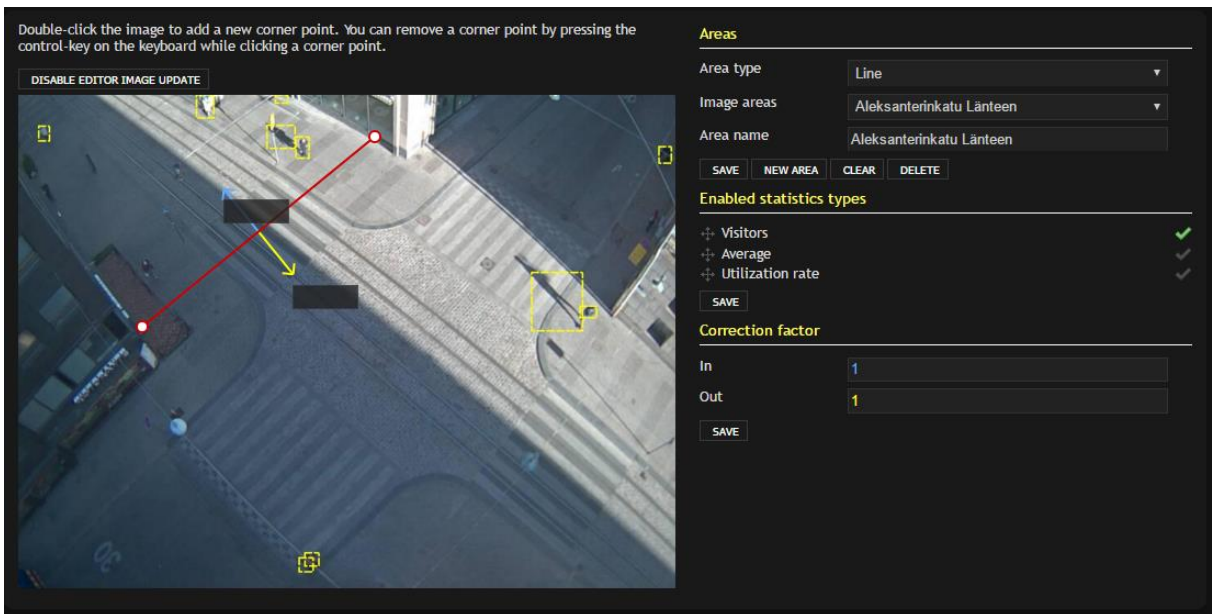


FIGURE 7. System's setup screen: Yellow rectangles are moving objects being tracked. Red line is the counting line. Note the large amount of false results.

The strengths of this algorithm (compared to laser-based counters) is in not needing specialised infrastructure for running it. The algorithm can work from any reasonable resolution (depending on the distance) camera and can be used to count from existing cameras or stored footage. Laser counters, in comparison, would require installing the counters at the street level where they are vulnerable to weather, damage and vandalism. It is also unlikely that one pair of gates would be capable of accurately counting over a whole street.

The general weakness of the algorithm is in inaccuracies and how fiddly it is. There are 19 different settings to modify, and the documentation on how these settings interact with each other is almost non-existent.

Use image trigger	<input checked="" type="checkbox"/>	Trigger interval	500
TRACKER			
Initial threshold (px)	35	Prediction threshold (px)	20
Maximum stop time (ms)	1000	Maximum prediction length (fr)	2
Minimum trajectory length (fr)	3	Allow merging of routes	<input checked="" type="checkbox"/>
BACKGROUND EXTRACTOR			
Threshold (px)	25	Alpha1 (double)	0.1
Alpha2 (double)	0.01	Maximum still time (fr)	5
Valid factor (double)	0.8		
OBJECT EXTRACTOR			
Dilate width (px)	6	Dilate height (px)	6
Erode width (px)	2	Erode height (px)	2
Area limit (px)	100	Color histogram threshold (double)	0.1

FIGURE 8. Algorithm's settings screen

4.4 Otos Service

Otos Service is the system between camera controllers and the iGator service. Being a third-party service it is very much of a black box for us, as cameras automatically send their data to the specific URL, and another specific URL provides the output data in a specific format. The documentation and internal access are non-existent, making any modifications to the system close to impossible to do.

5 PROJECT STRUCTURE

The project's different parts were often dependent on the earlier parts which led to rather natural flow through the project. That is, in theory at least, in practice, however, the different parts needed attention at different times because of various limitations in the development. Thankfully, I was the only person working (almost) full time on this, with other contributors having other projects going on, so any delays with one part did not grind the whole project into the halt.

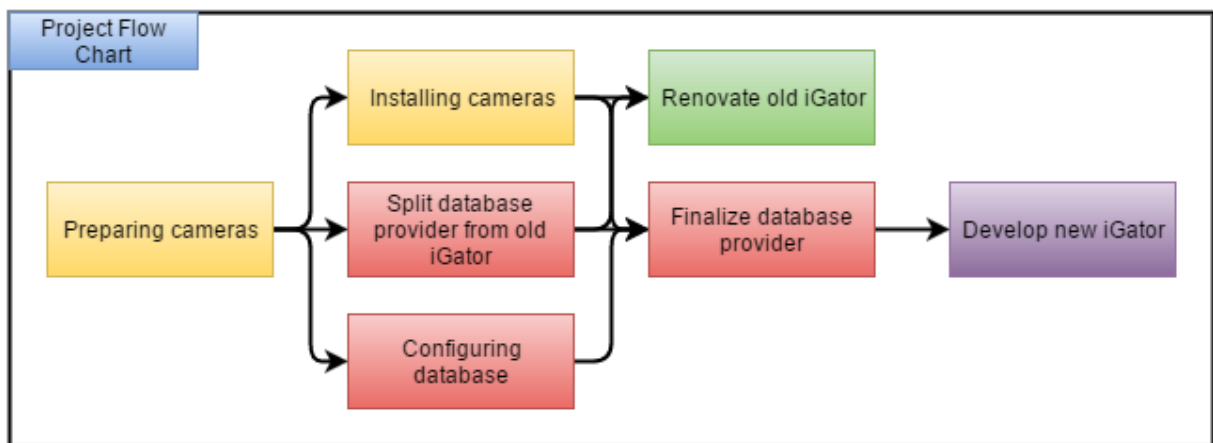


FIGURE 9. *Approximate flowchart of the project*

5.1 Preparing and Installing Cameras

The camera control system disk image was cloned from an existing working system. This image was then modified to fill a few different circumstances that the old systems did not have. Script (shown in Figure 11.) was written to call current IP home every 10 minutes. This was necessary, as new systems were to use 3G mobile connections instead of static wired connections. Static IPs for mobile connections are expensive (about tripling the monthly payment compared to non-static IP). Thus this was a financially motivated change to make.

The script is run by Crontab every ten minutes. The script parses current mac address from Linux's internal commands and queries OpenDNS.com for the system's current IP. Querying external server is the easiest and most reliable way to get the IP address as long as there is Internet connection (and, of course, if there is no connection, the IP address does not matter). If IP address is obtained, program called wget (very simple web downloading tool for Linux) is used to contact the home server's web address. As home server only requires the request,

wget is used in spider mode (the tool only checks if the page exists instead of downloading it as well) to minimize the traffic needed.

```
#!/bin/sh

URL="[redacted]"
mac=`ip link | awk '/ether/ {print $2}' | awk 'NR==1{print $1; exit}'`

ip=`dig +short myip.opendns.com @resolver1.opendns.com`
while [[ -z "$ip" ]]
do
    ip=`dig +short myip.opendns.com @resolver1.opendns.com`
    if [[ -z "$ip" ]]
    then
        sleep 5
    fi
done

ver='5'
type="start"
wget --spider -q "$URL?id=$mac&ip=$ip&ver=$ver&type=$type"
exit 0
```

FIGURE 10. Script that sends device's current IP to home server

Older cameras had trouble with summer time, causing clocks to run out of sync, if not manually adjusted. This was fixed on newer systems by making them update their time from the NTP service. Also, for the sake of easier camera configurations, a tunnel was opened through the control system to the camera. As the camera controller system's had pre-existing web-service contactable from port 80, it was necessary to open the tunnel to the camera to the port 81.

```
#!/bin/sh

echo 1 > /proc/sys/net/ipv4/ip_forward

iptables -F
iptables -t nat -F
iptables -X

iptables -t nat -A PREROUTING -p tcp --dport 81 -j DNAT --to-destination
192.168.1.2:80
iptables -t nat -A POSTROUTING -p tcp -d 192.168.1.1 --dport 81 -j SNAT --to-
source 192.168.1.2
```

FIGURE 11. Tunneling script

I did not contribute much on the physical camera installation, as the process was planned and organised by the more senior employees of Observis. I only went to help once with the actual installation. I did, however, support the installation process from the office the few times. The installation process itself is reasonably simple: The camera is installed in a good location, configured and then focused properly. The power over Ethernet cable is drawn from the camera to

the control system, which is connected to the mains current for power and to the 3G network for the Internet connection.

5.1.1 Gathering Camera Information

Lack of proper APIs made gathering cameras' data difficult. The details needed were: camera keys, counting line identification numbers and the order counting lines are provided by the service. The solution used was to manually parse Otos website's list for the data.

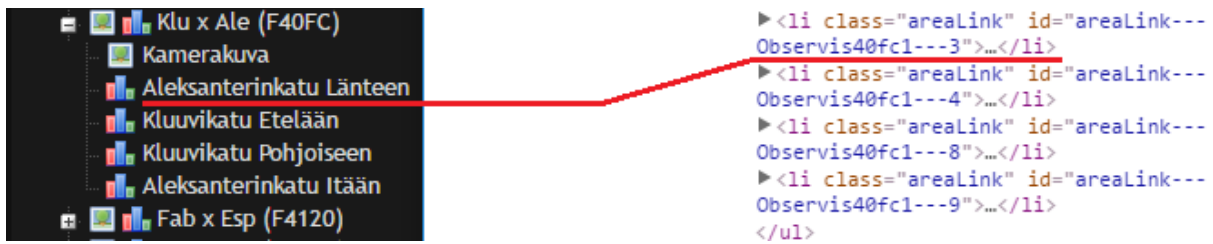


FIGURE 12. Process of manually parsing Otos webpage for camera line numbers

In practice the only tool this solution needed was the browser's element inspector, which was used to check the element's id, which contains the both camera key and the counting line's identification number. At the same time I also noted the order of the counting lines, as it is critical information for parsing the data.

5.2 Database and DBProvider

A PostgreSQL database was set up on the server. Originally there was only to be one table with data in ten minute increments. Later, it was decided to add another table, one containing the data in one day long sections. This would drop the size of more than day long queries considerably.

$$\begin{aligned}
 6 \text{ cameras} * 4 \frac{\text{lines}}{\text{camera}} * 1 \frac{\text{rows}}{\text{line}} * \frac{1}{10 \text{ minutes}} &= 24 \frac{\text{rows}}{10 \text{ minutes}} \\
 24 \frac{\text{rows}}{10 \text{ minutes}} * 6 \frac{10 \text{ minutes}}{\text{hour}} * 24 \frac{\text{hours}}{\text{day}} &= 3\,456 \frac{\text{rows}}{\text{day}} \\
 3\,456 \frac{\text{rows}}{\text{day}} * 365 \frac{\text{days}}{\text{year}} &= 1\,261\,440 \frac{\text{rows}}{\text{year}}
 \end{aligned}$$

The data bloat has always been bit of a concern, as there is need to store even the smallest incremental units for months at a time. As any reasonably powerful database server can handle

this kinds of loads, the only bottleneck was transferring the data. The fix for this was to query the day incremental table as much as possible, as this cuts the amount of transferred rows greatly.

Name ^	Rows	Size	Type
igator.igator	216 964	40,1 MiB	Table
igator.igatorday	1 970	424,0 KiB	Table
igator.igatorten	281 388	50,1 MiB	Table

FIGURE 13. *iGator database: Tables igatorday and igatorten are in use. Table igator is serving as a backup.*

▲ id	👉 cam_id	▲ date	👉 line_id	blue	yellow
2 765 719	Observis423c1	2016-01-31 22:30:00	3	4	0
2 765 725	Observis423c1	2016-01-31 22:30:00	5	32	11
2 765 612	Observis41201	2016-01-31 22:40:00	2	0	0
2 765 618	Observis41201	2016-01-31 22:40:00	4	0	18
2 765 624	Observis41201	2016-01-31 22:40:00	3	5	0
2 765 630	Observis41201	2016-01-31 22:40:00	5	1	5
2 765 636	Observis7b241	2016-01-31 22:40:00	6	0	0
2 765 642	Observis7b241	2016-01-31 22:40:00	8	0	0
2 765 648	Observis7b241	2016-01-31 22:40:00	7	0	0
2 765 654	Observis7b241	2016-01-31 22:40:00	9	0	0
2 765 660	Observis41ac1	2016-01-31 22:40:00	2	0	0
2 765 666	Observis41ac1	2016-01-31 22:40:00	4	0	0

FIGURE 14. *Example rows from igatorten table: Note non-descriptive blue and yellow columns.*

DBProvider was coded as a Java library which was then used both as independent process for filling the database and as a dependency for the new iGator. The original code base for DBProvider (it was split from the old iGator) was in dire need of refactoring if it was to be properly understood. Thus, not much refactoring was done, as there was no need to touch something that was already working. The refactoring that was done concentrated on making the functions more modular for the purpose of supporting the daily table.

```

for(String s : entry.getValue()) {
    Calendar cal = Calendar.getInstance();
    cal.setTime(idDateMap.get(entry.getKey()));
    s = s.replace(" ", "");
    String[] s1 = s.split(",");
    String[] s2 = s1[0].split("-");

    String[] baseTime = s2[0].split(":");

    Integer hr = Integer.parseInt(baseTime[0]);
    Integer min = Integer.parseInt(baseTime[1]);
    Integer blue = Integer.parseInt(s1[1]);
    Integer yellow = Integer.parseInt(s1[2]);
    cal.set(
        cal.get(Calendar.YEAR),
        cal.get(Calendar.MONTH),
        cal.get(Calendar.DAY_OF_MONTH),
        hr,
        min,
        0
    );
    cal.set(Calendar.MILLISECOND, 0);

    list.add(createIGatorDB(
        cameraInterface.getCameraId(entry.getKey()),
        areaIdMap.get(entry.getKey()), cal.getTime(), blue, yellow)
    );
}

```

FIGURE 15. *Otos parsing code: As the data is just comma separated numbers, only way to parse the data is to know the order counting lines are in.*

5.3 Old iGator

While in long term the old iGator was to be retiring from use, in short term, it needed maintenance and modifications for supporting the new cameras. Implementing support for the new cameras was an adventure in refactoring, as the original system had four different camera implementation classes, one for each wave of original cameras. The best features of each implementation were adopted to the new generic camera implementation. Other changes, such as removing hardcoded street names, adding labels on the charts and fixing smaller interface problems were done as well.

```

boolean isHistoryQuery = isHistoryQuery(period, date);
int threadsNeeded = calculateThreads(isHistoryQuery, cameraInterface);

clearMaps(threadsNeeded);
createAndRunThreads(threadsNeeded, cameraInterface, isHistoryQuery, period,
    date);
cullData(threadsNeeded, isHistoryQuery, period);
List<IGatorDBTen> results = parseResults(cameraInterface);

switch (period) {
    case YEAR:
    case MONTH:
    case WEEK:
    case DAY:
        saveResults(results, EPeriod.DAY);
        break;
    case HOUR:
        saveResults(results, EPeriod.HOUR);
        break;
    default:
        break;
}

```

FIGURE 16. Refactored Otos query function

5.4 New iGator

The new interface for iGator (internally called iGatorGWT, even though old version also uses GWT) was built from the scratch and then tied to DBProvider service. This part of the project needed most attention, being the largest and most innovating part of the process.

In general, all GWT codebases can be divided into two parts: The client side code, which is converted by GWT to JavaScript and provided to the client's web browser, and the server side, which is run on server side as pure Java. There is also shared code which is provided for both client and server side, as JavaScript and Java respectively. This division was also used in this project.

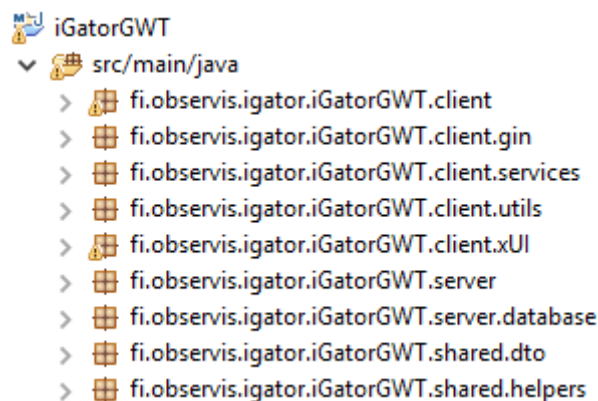


FIGURE 17. New iGator's code structure: Note client, server and shared packages which denote on what side the code is provided for.

5.4.1 Client Side

Usually GWT projects client side code only deals with user interface, and how the interface interacts with the server side. The client side code can be divided into three rough parts. There are also the web elements (HTML, CSS, image and so on) which are provided in the standard format, as GWT applications are almost always run in some kind of browser.

```

public void onModuleLoad() {
    UncaughtExceptionHandler uncaughtExceptionHandler =
        new UncaughtExceptionHandler() {
            public void onUncaughtException(Throwable e) {
                Logger.logAll(e.getMessage());
                e.printStackTrace();
            }
        };

    GWT.setUncaughtExceptionHandler(uncaughtExceptionHandler);
    xEntryPoint entryPoint = GWT.create(xEntryPoint.class);
    factoryManager = entryPoint.getFactoryManager();
    InitUI.InitUIFactory initUIFactory = factoryManager.getFactory(InitUI.class);

    InitializingPage initializingPage =
        initUIFactory.create().getInitializingPage();
    initializingPage.initialize();
}

```

FIGURE 18. *Entry point method: The method from where GWT starts the compilation process, and the point where the program itself starts.*

Client-to-server code which deals with sending and receiving information from the server. GWT deals with the details, such as transferring and parsing data, on the background. The client or the server side can simply call its counterpart's Java functions in the code, and the GWT deals with the rest. This makes transferring Java objects a particularly easy task, and allows hard-typed coding in both ends of the system.

Client side Java code which is converted by GWT to JavaScript. Most of the code deals with this part. Most of the DOM manipulation is done by this code and so is almost all of interface functionalities. This is the core of the whole system, and everything else connects to this part.

```
@ViewField protected Button closeViewButton;

[Some redacted lines here]

closeViewButton.addClassName("close-panel-button");
closeViewButton.html("<span class='fa fa-close'></span>");
```

FIGURE 19. *@ViewField in action*

To ease DOM manipulation, GWT supports @ViewField attribute in Java side (see above). When this attribute is used, GWT automatically finds the corresponding DOM element from the .xt file (basically HTML file with different name and some extra attributes in the elements) and links them together. This way the developer can link to the HTML element without requiring any manual code writing.

```
private final native void addDatePicker() /*-{
    var options = {
        format : "DD.MM.YYYY HH.mm",
        minDate : "2016-02-01 00.00",
        sideBySide : true,
        locale : "en"
    }

    $wnd.$("#inputDateStart").datetimepicker(options);
    $wnd.$("#inputDateEnd").datetimepicker(options);
}-*/;
```

FIGURE 20. *Calling JavaScript code in Java. \$wnd would be the standard window scope in normal JavaScript.*

As many **JavaScript libraries** were used, and most of them did not have existing GWT variants, there was also need to call pure JavaScript from Java code and vice-versa. This is generally one of the more confusing things to do when using GWT, as the way this is coded is rather unique.

```
private native void initializeCallbacks() /*-{
    $wnd.selectCamera = $entry(
        @fi.observis.igator.iGatorGWT.client.InitializingPage::selectCameraById(I)
    );
}-*/;
```

FIGURE 21. *Initialising JavaScript callback function: When this has been called, the Java function selectCameraById can be called from the JS side by calling the selectCamera function.*

5.4.2 Server Side

The server side was there to offer a platform for providing the client side data (both JavaScript and other web files) for the users, being the connector between clients and the database, and doing any necessary calculations with the data.

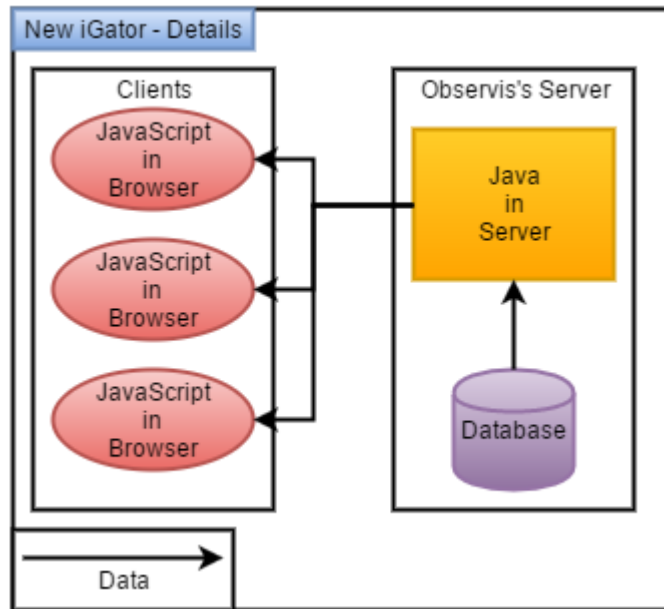


FIGURE 22. *New iGator*

While most of file provision was done autonomously by the underlying Apache Tomcat, there was still need to write some functionality related to this on the Java side. Most of this code was related to securing files to be provided only to certain IP addresses.


```

public static List<IGatorDB> doMultiQuery(Date start, Date end) {
    List<IGatorDB> list = new ArrayList<IGatorDB>();

    Date date1 = start;
    Date date2 = getNextMidnight(start);
    Date date3 = getLastMidnight(end);
    Date date4 = end;

    if (date1.getTime() < date2.getTime()) {
        list.addAll(EJBTEN.selectAllBetween(date1, date2));
    }

    List<IGatorDBDay> tempList = EJBDAY.selectAllBetween(date2, date3);
    if (tempList.isEmpty()) {
        list.addAll(EJBTEN.selectAllBetween(date2, date3));
    } else {
        list.addAll(tempList);
    }

    if (date3.getTime() < date4.getTime()) {
        list.addAll(EJBTEN.selectAllBetween(date3, date4));
    }

    return list;
}

```

FIGURE 23. *A database query: It minimizes the number of returned database rows by getting results from the day-table instead of the ten-minute-table.*

The largest amount of code was related to the database queries, and the calculations related to those. Various specialised queries were developed, such as averages by weekday and averages by hour. From both the performance and bandwidth viewpoint it was necessary to make most calculations on server side, as transferring potentially tens of thousands of rows of data to the client side is a bit much.

6 CONCLUSIONS

The project achieved the major goals it set out to achieve, which leaves this chapter rather brief. Various smaller goals were changed as project continued, and they will not be described here.

On the physical side, seven camera control systems were prepared and six were installed in real locations. One camera was kept as a spare and as a development platform. At the time of writing this thesis, two of installed cameras have problems with uptime, and extra work is needed to solve this problem. It is very likely that ongoing building work at the camera's installation site is cutting the power to one of these cameras, causing downtime. The other downtime cause is unknown.

Name	Mac	Updated	Started	Intended State	Misses / Calls
Klu x Ale	00:90:0b:2f:40:fc	2016.05.03 09.42	2015.11.16 12.23	Up and Counting	667 / 11180
Mik x Esp	00:90:0b:2f:41:ec	2016.05.03 09.42	2015.11.16 12.27	Up and Counting	5 / 9186
Mik x Ale	00:90:0b:2f:42:3c	2016.05.03 09.42	2015.11.16 12.32	Up and Counting	0 / 9212
Fab x Esp	00:90:0b:2f:41:20	2016.05.03 09.42	2015.12.02 14.33	Up and Counting	15 / 9165
Fab x Ale	00:90:0b:2f:41:2c	2016.05.03 09.42	2015.12.03 14.32	Up and Counting	2 / 9206
Kes x Ale	00:90:0b:3a:7b:24	2016.05.01 15.52	2016.02.26 09.22	Up and Counting	716 / 6949

FIGURE 24. Camera info table: Some columns have been omitted. The last row is red because the camera was down when the picture was taken.

The old iGator was changed to support new six cameras and various interface problems were fixed. Labels were added to the charts and date picking tool's bug was fixed, but most of the work was done under the hood, and as such these changes are not really visible.

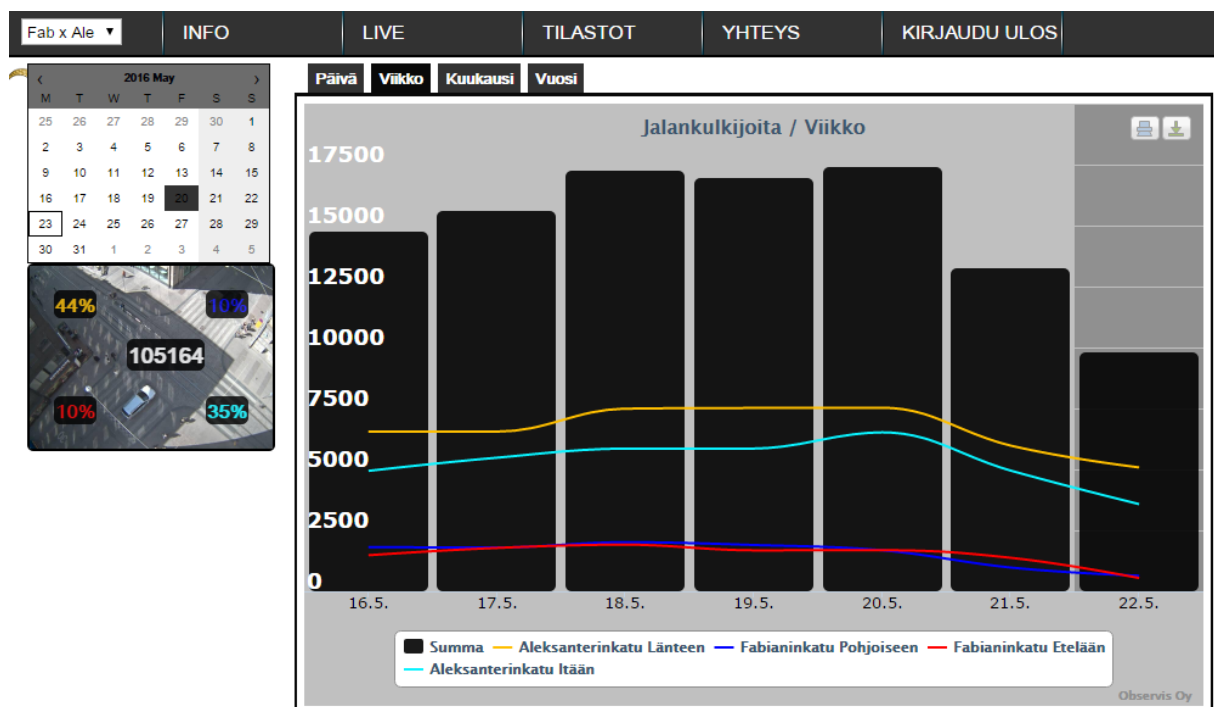


FIGURE 25. User interface of old iGator

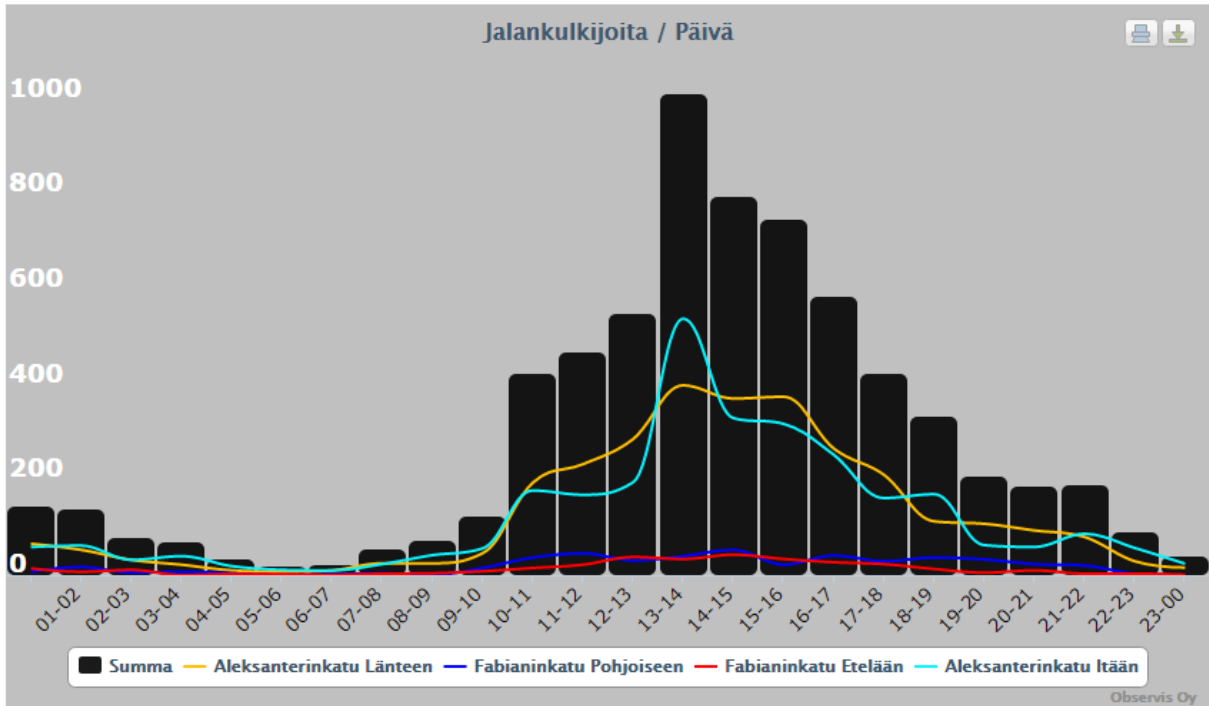


FIGURE 26. Sunday 24.4.2016 traffic as shown in the updated old iGator

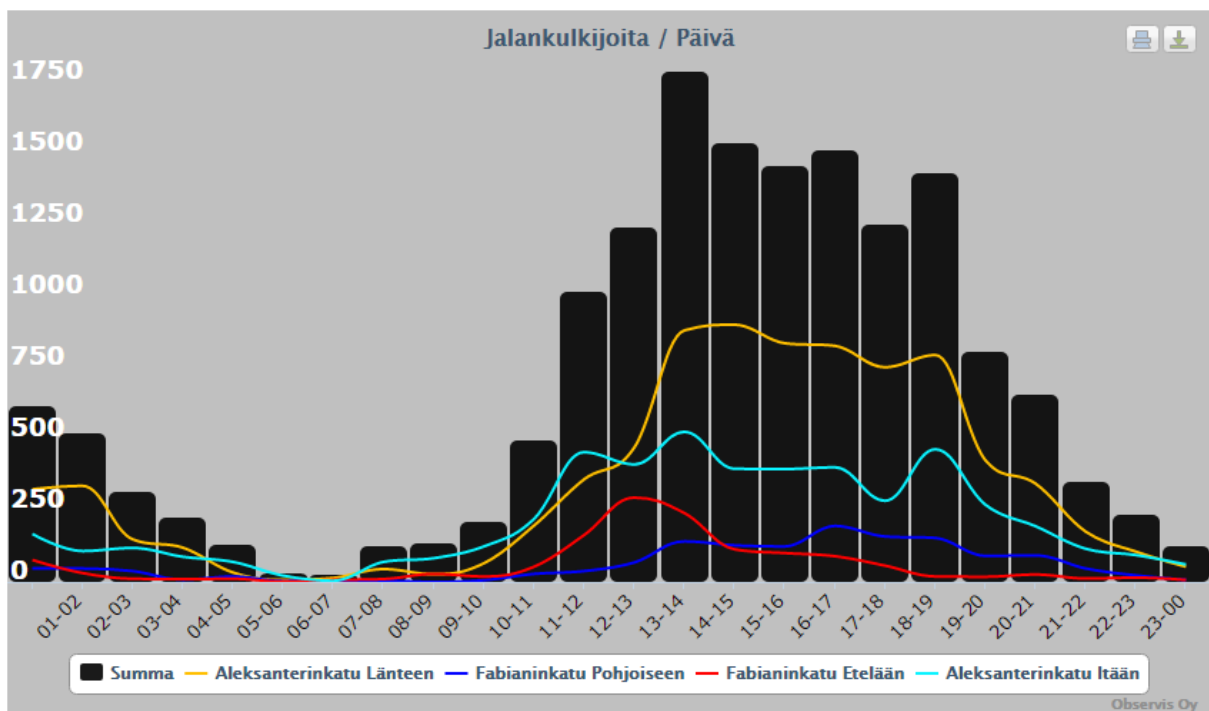


FIGURE 27. Sunday 1.5.2016 traffic as shown in the updated old iGator

New iGator was released on the Observis production server. The service has been in use by the users without serious downtime or other major problems on the service side.

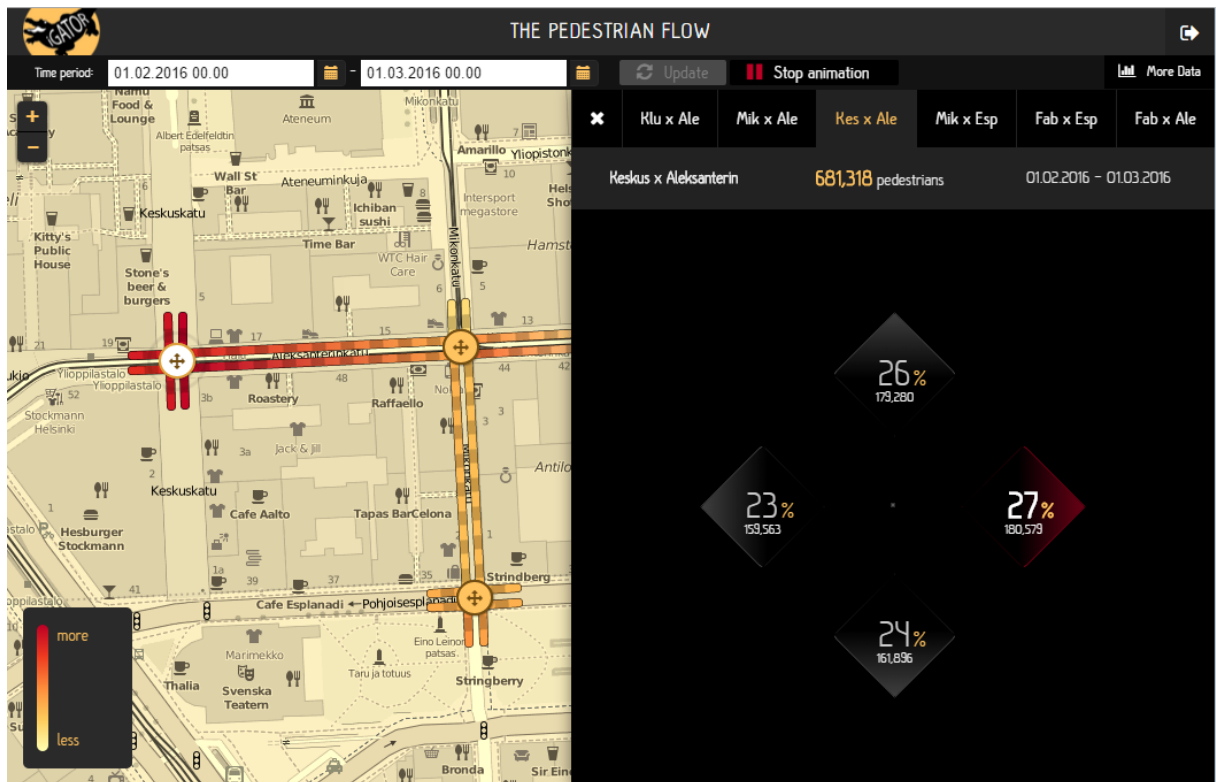


FIGURE 28. The new iGator's interface: Heat map on the left, singular crossroad's data on the right.

The results support the idea that using Java and GWT is perfectly viable, often even superior web development tool than more common pure JavaScript. I found Java's hard typing and superior structure much more pleasant than JavaScript's soft typing and more flexible structure.

While the project's programming language was defined by the existing codebase, I would likely have gone with Java and GWT even if all options had been open. Even if pure JS had been used on the frontend, the backend would have likely required some other programming language, which most likely would have been Java. Thus, using Java on both client and server side would have been reasonable choice to make.

7 FUTURE DEVELOPMENTS

While the project achieved its main goals, it still has quite many problems that make it impractical in the long term. Expanding the system requires whole lot of manual work, as there is no automation for installing system image into new desks or for adding new camera IDs into the system.

The finished software suffers from the hardness of maintenance. There are three different places where cameras are listed, and details changing (such as someone removing and re-adding a calculation line) requires changes in multiple places. This is both cumbersome, requiring modifications in the code, compiling the projects and re-uploading them on the servers. This is also error prone process which can cause extra downtime.

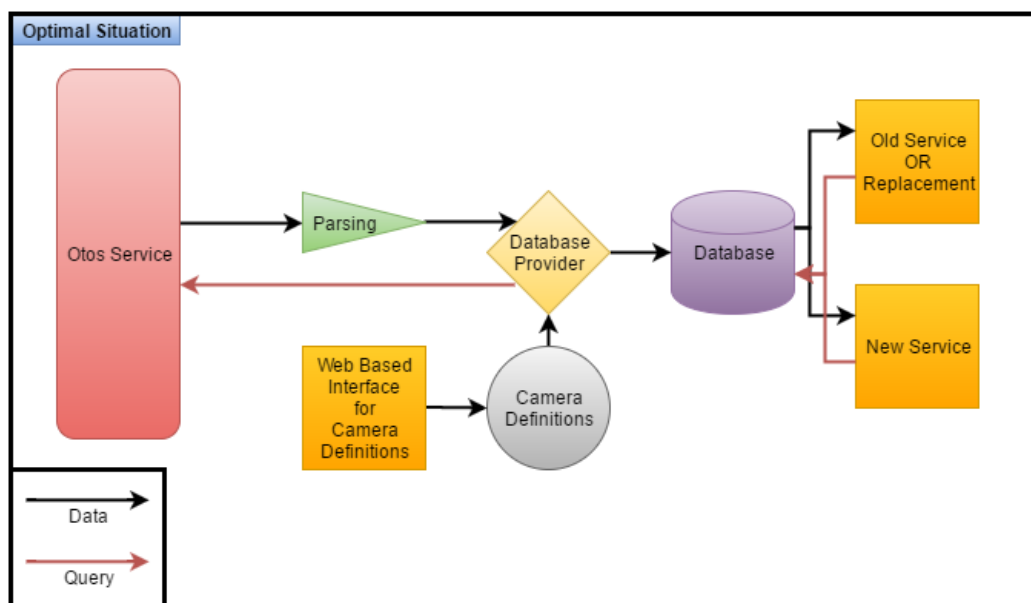


FIGURE 29. Envisioned optimal situation

In an ideal world some kind of central database would query Otos's servers for the camera meta-data, but lack of proper API makes this difficult. It would definitely be possible to make a program that would parse the website's HTML code for the data, but this would still be missing details, making the whole exercise rather pointless.

The second best option would be having a centralised database for all this and a web based tool for modifying it. Preferably the tool's interface would be so simple that it wouldn't require programmer's attention to use it.

7.1 Camera Control System Changes

Currently updating all camera controllers is a marathon of manual busywork. The update scripts need to be planned, each camera must be connected individually and the same tasks need to be repeated on each camera. Even with only six cameras, this is a tedious and error-prone process. It would be reasonably simple to setup each camera with timed update script that seeks new updates from a central server. This would greatly speed up the process of releasing any updates, saving precious coder time. Of course, it would also make it possible to a publish system destroying script on all production cameras in a single instant, but this is small price to pay for shorter publishing phases.



FIGURE 30. Computer unit: Measurements: 20cm x 14.5 cm x 4.5 cm when closed. Features SSD storage and internal SIM card for 3G mobile connection.

The system has also had, and might have in future, problems with failing for various reasons. One of these sources of failure is power cuts, which (as the power switch defaults to off state after power has been out) currently require human's attention to be fixed. When a human notices a camera being down (system sends email after half an hour) there is need to send text message to boot the system again. A better system would autonomously send text messages to try to fix the situation, and would only alert technicians, if the camera failed to get back on within a reasonable timeframe. This would greatly speed up the rebooting process, when the system fails at an inopportune time (during weekends or non-office hours) and would spare technical support resources.

Other physical problems are related to the layout of the controller box. The computer unit is screwed bottom up to the box, which, when physical access to the inside of the computer is needed for any reason (changing hard-drive or SIM-card, for example) it needs to be unscrewed from the unit. Similarly, as the box is rather cramped, connecting cables (USB and VGA in particular) is difficult, which makes onsite connection difficult. There is three reasonably easy fixes for the cable problem: Re-think whole layout of the box, add some helpful connection cables for each box or provide maintainers with cables that have 90 degree corner at the end.

7.2 Further Data Analysis Options

At the moment, deeper data-analysis is tedious to do, as, for example, to compare Sundays on different weeks, the user must do it manually by looking at different Sundays day by day. The simplest fix for this would be developing some more specialised functions to deal with these needs. Having an option to see, for example, trend changes, certain weekdays' values in longer term and so on would make the system much better for quick data lookups.

Producing system that is capable of making all possibly interesting charts and still have simple enough interface to use without higher math degree would most likely be impossible with the current resources. On the other hand, having some handpicked functions developed would definitely be possible.

8 BIBLIOGRAPHY

Bootstrap info page 2016. Bootstrap team. WWW document.

<http://getbootstrap.com/about/> No update information. Referred 6.5.2016.

GWT Project 2016. GWT Open Source Project. WWW document.

<http://www.gwtproject.org/> No update information. Referred 1.5.2016.

Hibernate Community 2016. Hibernate Community. WWW document.

<http://hibernate.org/orm/> No update information. Referred 1.5.2016.

Java 8 Design Document 2015. Oracle America. WWW document.

<https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf> Updated 13.2.2015. Referred 3.5.2016.

Leaflet Open-Source Project's website 2016. Leaflet Project. WWW document.

<http://leafletjs.com/> Updated 2015. Referred 9.5.2016.

jQuery 2016. jQuery Foundation. WWW document.

<https://jquery.com/> Updated 2016. Referred 9.5.2016.

jQuery Statistics 2016. BuildWith Pty Ltd. WWW document.

<http://trends.builtwith.com/javascript/jquery> Updated May 2016. Referred 4.5.2016.

SparkJava project's website 2016. SparkJava Team. WWW document.

<http://sparkjava.com/> No update information. Referred 22.5.2016.

TIOBE Index 2016. TIOBE software BV. WWW document.

http://www.tiobe.com/tiobe_index Updated April 2016. Referred 2.5.2016.